A Lightweight Specification Language for Bounded Program Verification

by

Kuat T. Yessenov

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

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

Certified by . . . .
Daniel N. Jackson
Professor
Thesis Supervisor

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

A Lightweight Specification Language for Bounded Program Verification

by

Kuat T. Yessenov

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

Abstract

This thesis presents a new light-weight specification language called JForge Specification Language (JFSL) for object-oriented languages such as Java. The language is amenable to bounded verification analysis by a tool called JForge that interprets JFSL specifications, fully integrates with a mainstream development environment, and assists programmers in examining counter example traces and debugging specifications. JFSL attempts to address challenges of specification languages such as inheritance, frame conditions, dynamic dispatch, and method calls inside specifications in the context of bounded verification. A collection of verification tasks illustrates the expressiveness and conciseness of JForge specifications and demonstrates effectiveness of the bounded verification technique.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

I would like to thank Daniel Jackson for all the advice, enthusiasm, and encouragement that I received throughout my years at MIT. His excellent lectures in 6.170 course sparked my interest in the subject, and led me to discovering what computer science research is really like. I am looking forward to working with him in my future graduate studies.

Thanks to all the past and present members of the Software Design Group: Emina Torlak, Greg Dennis, Derek Rayside, Felix Chang, Rob Seater, Jonathan Edwards, Eunsuk Kang, Rishabh Singh, Joe Near, and Aleks Milicevic. Much of what I know now is what I learned from them. They were always eager to discuss with me and share their thoughts and experiences. They also made SDG a very pleasant environment to work in. Special thanks to Greg for being an excellent UROP supervisor. Many of the ideas in this thesis appeared in the long hours of discussions with him. Without his tool Forge, this work would not be possible.

Thanks to all my friends at MIT and my housemates at Hardwick that made my past year stay at MIT fun and memorable. Thanks to Aizana for coming into my life.

Contents

List of Figures

Listings

Chapter 1

Introduction

Formal specifications are integral to automated reasoning about strong properties of software systems. They provide the necessary formalism to express the required conditions on the system that the designer has in mind. They are written in a rigorous language that has less ambiguities that plague natural language documentation. Lastly but no less important, they facilitate formal analysis of the software. The analysis can be either manual (performed by experts), completely automated, or achieved by a combination of tools with human assistance.

Despite all the advantages of writing specifications, average programmers still make little investment in providing formal specifications. We can think of with several reasons for that. First, formal specifications require as much work as informal specifications if not more in understanding what the system actually does. For certain parts of the system, such as public interfaces, specifications are certainly necessary, but for other, internal parts, programmers are willing to take code as its documentation. Second, formal languages are often less accessible to programmers because they require mathematical training, use semantics that is too remote from the semantics of the programming language, or are simply too verbose. Unless there is some other good justification, a formal specification that is longer than the specified code and operates on complex mathematical concepts is simply too costly for average programmers. Finally, programmers expect to see a support from tools that employ the formal specification in useful ways that is worth the effort of writing them. For example, checking whether code matches its specifications is crucial to verifying programs and can justify large investments in writing specifications.

In this thesis, we attempt to tackle some of these challenges in designing a specification language called JForge. We call the language “light-weight” since its tool support is built on top of Alloy light-weight verification methodology [13], and it attempts to be easy to use and accessible to programmers. The JForge specification language (JFSL) aims to complement code written in Java. We also developed a tool that comes in the form of a plug-in for Eclipse, a mainstream integrated development environment for Java, that fully interprets JFSL and is able to perform bounded code verification [9].

1.1 Motivation

There has been a considerable research effort recently in specification languages for high-level programming languages. The Java Modeling Language (JML) is a behavioral specification language for Java that has a wide variety of supporting tools [7]. Spec# is a specification language for API contracts designed as an extension to C# language [6]. While being a behavioral interface specification language in the same vein as JML and Spec#, JFSL is distinct in its approach to code verification since its semantics views heap as a relational structure.

In an experimental study of KOA vote tallying software annotated with JML, we showed viability of modular static analysis based on the bounded code verification technique [9]. We identified a great potential of the underlying Alloy logic in its ability to express strong heap properties in a concise form that was not exploited by the front-end specification language to its fullest. For example, transitive closure, which is necessary to talk about trees and reachability in graphs, is a complex construction in JML that has support from specialized tools only [14]. Alloy's logic has transitive closure, that could be easily included in the front-end specification language. On the other hand, underspecified semantics of treatment of method calls in specifications presented a problem for JMLForge since it required higher-order quantification and resulted in complex verification conditions. Static invariants were abundant in the experimental code but did not work well with modular verification approach. Lastly, the complexity of model types in JML as well as lack of consistent and verified specifications for them limit the use of specification fields in JML.

JFSL attempts to address directly the issues we encountered in JMLForge case study. JFSL introduces relations as first-class citizens in the language and provides navigation operators that facilitate the use of model fields. JFSL was *designed to be amenable* to automated bounded analysis. JFSL takes advantage of the vast experience in writing declarative models in Alloy to find the right balance between expressiveness of the language and full automated support by tools[13]. Finally, we provide a collection of sample JFSL specifications to demonstrate its use in annotating programs.

We have identified the following challenges in designing a specification language that we address in this work:

1. *Specifications are hard to write, verbose, and prone to errors.* JFSL uses highly expressive Alloy operators for describing heap properties. JForge facilitates debugging specifications and provides visual feedback for validating specifications. Debugging specifications is especially important for modular program verification because of the dangers of over- or under-approximation of code by its specification [9].
2. *Specification languages are complicated and plagued with inconsistent semantics.* JFSL is based upon relational first-order logic with transitive closure but is consistent with substantial fragment of Java language semantics as well.
3. *Specification languages are either not designed for mainstream programming lan-*

guages or fall behind the progress of programming languages. JFSL is designed to complement Java, a popular object-oriented programming language, and uses its novel features such as annotations [11]. JForge tool operates only on compiled byte code, which in theory is forward compatible with future versions of Java virtual machine. P. Chalin is leading a similar line of work for updating software infrastructure for JML [18].

4. *Programmers are reluctant to use specification languages and tools unless they are integrated into development environment*. JForge tool is integrated into Eclipse development environment via a set of plug-ins for code verification, interactive counter example trace stepping, and heap snapshot visualization. Related work is in progress for other specification languages. Visual Studio now offers good support for Spec# and the Boogie theorem prover as well as code contracts, a different technology from Microsoft for run-time and limited static checking [1].
5. *Verification tools should be accessible to programmers and not require expert knowledge in program verification.* JForge tool uses bounded verification which renders the logic decidable, and thus allows full automation. It provides a simple push-button interface that produces counter example traces that are converted to initial heap configurations and, thus, can be understood by programmers. This provides an additional benefit of easy validation of counter examples since JForge makes some unsound approximations of program behavior. We believe that the light-weight approach to verification is desirable as it can give meaningful results with little investment in verification; in particular, it provides an ability to verify parts of the system in a modular fashion.

1.2 Binary Search Example

We illustrate our proposed solution with a famous bug in binary search routine that stayed unnoticed for years in the Java API [4]. Listing 1.1 presents the method together with its JFSL specification. Given an array `a` and a key `key`, this method either returns an index for the array entry that equals to the key, or a negative number if the array has entries equal to the key. The bug is triggered only for arrays of length sufficiently large to cause an integer overflow in expression `(low + high) / 2`. The correct implementation should instead use `(low + high) >>> 1`.

The specification of this method consists of two parts. The pre-condition specifies that the input is a sorted array. The post-condition specifies that if the key is in array, then an index to corresponding entry is returned non-deterministically; otherwise, a negative number is returned. As you can see, specifications are relatively short and the expression language is similar to Java with elements of Alloy for quantifications.

JForge can analyze this method for violations of refinement of code and specification. If we perform bounded verification in the scope of at least two loop unrollings, JForge automatically finds a counterexample such as the one seen in

Listing 1.1: Buggy binary search from Java API

```java
@Requires({"a != null",
"all i:int, j:int | 0<=i&&i<j&&j<a.length => a[i]<=a[j]"})
@Ensures("(some i : int | a[i] = key) ? a[return] = key : return < 0")
int binarySearch(int[] a, int key) {
        int low = 0;
        int high = a.length - 1;
        while (low <= high) {
                int mid = (low + high) / 2;
                int midVal = a[mid];
                if (midVal < key)
                        low = mid + 1;
                else if (midVal > key)
                        high = mid - 1;
                else
                        return mid; // key found
        }
        return -(low + 1);  // key not found.
}
```

Figure 1-1: Counter example for the buggy binary search implementation

```
initial state:
key = 1
a = [-4, 0, 2]
...
  low = 0
  high = 2
$i0 := (low plus high) goto Stmt10
  $i0 = 2
  mid = 1
...
  low = 2
$i0 := (low plus high) goto Stmt10
  $i0 = -4
  mid = -2
...
final state:
  binarySearch_throw = java.lang.ArrayIndexOutOfBoundsException$Lit
```

figure 1-1. Since the analysis is bounded, the bit width is approximated to 3 which means values of integers span the range $\{-4, -3, \ldots, 3\}$. In the counter example, overflow happens at the second loop iteration in evaluating 2+2 which results in -4. In the end, method queries an array for a negative index, which in Java always fails with an exception and failure to satisfy the post-condition of the method.

While the approximation is unsound, the bug is *real* since the same overflow may happen in Java but for larger numbers. Moreover, with a couple lines of specifications from the programmer, JForge completely automatically discovers a non-trivial counter example that would be much harder to find with testing. Much of empirical evidence that has been collected for counter examples to models and code suggests that the limited scope of analysis may in fact discover many more if not all bugs [5] [9]. This claim is known as *the small scope hypothesis.* While the counter example traces might not be valid in Java because of unsoundness of analysis, they may correspond directly with real bugs just like in this example.

1.3 Thesis Outline

In section 2 we explain the framework of bounded verification that stems from research on Alloy modeling language and light-weight formal methods, and present an overview of the tool chain.

In section 3 we define JFSL language formally and describe its syntax via grammar production rules. The challenge lies in incorporating elements of Alloy language and Java language and resolving inconsistencies in the interpretation.

In section 4 we explain the semantics of the language and translation to bounded verification logic. We explain how we solve the problem of invariant relevancy using the notion of a *scene*. We show how specification fields fit naturally into the abstract data type pattern in Java. Frame conditions on the specification fields solve the problem of sub-typing and specification inheritance. We explain how JForge solves the challenge of method calls inside specifications via a simple substitution mechanism.

In section 5 we give an overview of the static analysis tool built as an Eclipse plug-in and show its functionality. We also present an overview of the architecture of the tool chain. The challenge here is to bring an intricate verification technology to the hands of a non-expert programmer. We discuss the kinds of automated analysis that JForge supports, and JForge solves the usability problem of verification tools.

In section 6 we demonstrate a collection of examples of using our code. We start with a Red-Black tree implementation that shows the power of JForge in describing complex heap properties.

Finally, in section 7 we conclude and present directions for the future work.

Chapter 2

Methodology

JFSL is designed to work well with bounded verification approach although it is not coupled with it. Because of that reason, much of semantics of JFSL is derived from Forge and its intermediate representation (FIR) [10]. In this chapter, we describe this intermediate representation and present an overview of JForge.

2.1 Bounded Verification

FIR is a common medium in which both specifications and code reside. It has both procedures and imperative control flow graphs as well as specification statements for declarative constraints. This way specifications and code are treated uniformly. One may think of JForge as a translator from JFSL into FIR. Specifications are translated into FIR expressions, which are then inserted into Forge control flow graphs. Forge has relations, transitive closure, integer arithmetic, set operations, relational join and override, and conditional expressions (see figure 2-1.)

Forge accepts a Forge procedure and a specification as its input. Once the bounds are specified, Forge attempts to find a counter example which is a trace of the procedure that violates the specification. The bounds specify the number of atoms per each domain, the bit width, and the number of loop un-rollings. If no counter example is found then one can be confident that there is no initial heap configuration with the number of atoms per domain within the bounds and integers with the given bit width, such that the procedure executes with no more than the given number of loop iterations and completes in a state, in which specification fails. For that reason, this approach is called bounded verification.

2.1.1 Java-to-FIR Translation

JForge includes a translator from Java byte code to Forge intermediate representation based on SOOT framework [20]. This component has undergone evolution from its use in JMLForge [9] and now is substantially more mature. The translator operates only on the byte code representation. It does not handle several features

```
Expr ::=
varId | litId | domID | ∅                     leaf
| Expr ⊆ Expr                                 subset
| Expr {= | ≠} Expr                           (in)equality
| {some | one | lone | no} Expr               multiplicity
| Expr {∪ | ∩ | \} Expr                       set operations
| Expr . Expr                                 join
| Expr → Expr                                 cross product
| Expr ⊕ Expr                                 override
|ˆExpr                                        transitive closure
| ∼ Expr                                      transpose
| π(Expr, IntegerLiteral*)                    projection
| Expr ? Expr : Expr                          conditional
| {varId* | Expr}                             comprehension
| ¬ Expr                                      boolean negation
| Expr {∧ | ∨} Expr                           boolean operations
| {∀ | ∃} varId* | Expr                       quantification
| Σ varId* | Expr                             summation
| Expr {+ | − | × | ÷ | mod} Expr             arithmetic
| Expr {> | < | ≥ | ≤} Expr                   integer inequality
| Expr {| | & |ˆ| ≪ | ≫ | ⋙ } Expr            bitwise operations
| # Expr                                      cardinality
| varId_old                                   pre-state variable
```

Figure 2-1: FIR expression grammar

that are not supported by our bounded verification technique such as real arithmetic or synchronization mechanisms. However, there is no reason apart from the required engineering effort, that bounded verification can in theory support synchronization commands.

A Java program is converted to Forge procedures as follows. First, the scope on types is reduced to a finite set of Java types relevant to the program. Then each of these types is converted to its own instance domain, a set of atoms corresponding to instances of the actual type (or unknown sub-type.) The presumption here is that the principle of behavioral inheritance holds for sub-typing relation in Java and the unknown sub-types obey specifications of their super types. Then the set of atoms of any given declared type is the union of the instance domains for all reflexive transitive sub-types and the special *null* domain for the single **null** atom. Fields are represented as relations from the declaring type to the value type. Arrays are represented as relations from the array type to integers to the base type. More subtle details such as handling of dynamic dispatch is explained in greater detail elsewhere [10].

Translation from Java is not fully accurate and some sources of unsoundness are introduced. As we have seen before in binary search example, change of bit-width triggers bugs that do not actually appear in real Java executions but may manifest themselves in larger scope. In addition to that, JForge takes liberty in introducing some unsound approximations such as interning all string literals and restricting all exception class domain to be singletons.

2.1.2 JFSL-to-FIR Translation

JForge uses the ANTLR parser generator to parse, type check, and translate specifications in JFSL. ANTLR grammar follows extended the Backus-Naum form of

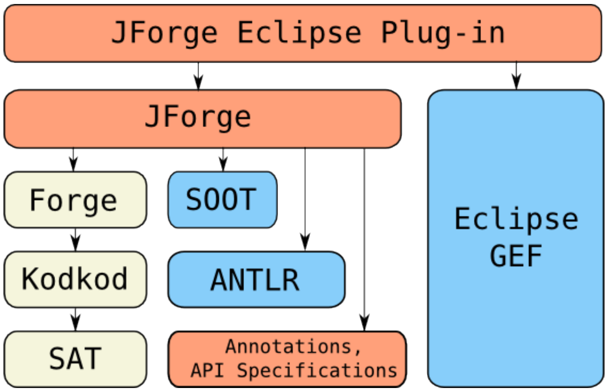


Figure 2-2: JForge toolkit module dependency diagram

JFSL (see 3-1) very closely and was initially derived from ANTLR grammar for Java [2]. The translation proceeds in phases from building an abstract syntax tree for JFSL, type checking using SOOT-loaded class signatures, and translating into FIR.

JFSL considers the heap as a set of atoms for each type, and a collection of relations representing the fields of the types. For any type T, expression T stands for the set of all non-null aton of type T; for interfaces it includes atoms for all classes implementing the interfaces, and for classes it includes atoms for all subclasses as well. For example, expression int stands for the bounded subset of all primitive integers $\{\ldots, -1, 0, 1, \ldots\}$.

2.2 Toolkit Overview

The JForge toolkit consists of the following components organized as shown in figure 2-2:

1. *JForge Eclipse Plug-in* is a plug-in for Eclipse that integrates into the development environment, executes JForge analyses and presents results to the user;
2. *JForge* is the main tool that exports public interfaces to perform analysis on methods annotated with JFSL specifications;
3. *Eclipse* represent a collection of dependency plug-ins such as Graphical Editing Framework (GEF) for visualizing heap snapshots;
4. *Forge* by Greg Dennis [10], *Kodkod* by Emina Torlak [19], and *SAT* provide the back-end of bounded verification analysis;
5. *SOOT* is a byte code optimization framework that is used for translation from Java to FIR [20];
6. *ANTLR* is a run time library for parsing JFSL expressions given production rules for the grammar [2];
7. *Annotations* is a shared library of JFSL Java annotations;

8. *API Specifications* is a library of specifications for commonly used classes from Java API, mostly from collections framework.

JForge takes as its input a compiled Java class and a method identifier. Then it extracts all relevant specifications from the compiled classes, library of API specifications, and refinement JFG files, constructs abstract syntax trees using ANTLR, and translates them into FIR. It translates code into FIR using SOOT starting from the raw byte code and applying SOOT optimizations to obtain static single assignment (SSA) form. The output of JForge is either a trace of a counter example (which specifies the initial pre-state, sequence of steps, and the final post-state heap configuration), or coverage information (only available if the back-end SAT solver supports unsatisfiable core extraction.)

The JForge Eclipse plug-in presents this information within the environment itself and facilitates interactive validation of the counter example as described later in chapter 5.

Chapter 3

JFSL Reference Manual

This chapter presents the JFSL reference manual – the annotations facility, JFG file format, and the syntax of the language.

3.1 Java 5 Annotations

JFSL introduces a collection of Java 5 annotations that provide the means of writing the specifications directly in Java source code. The annotations are stored as strings in Java byte code, and they are understood by the JForge tools. These annotations belong to the package edu.mit.csail.sdg.annotations:

1. **@Invariant** type annotation specifies type invariants (see 3.5.1)
2. **@Helper** method annotation tags a method as a helper method (see 3.5.1)
3. **@SpecField** type annotation is a declaration of specification fields (see 3.5.2)
4. **@Requires** method annotation specifies a light-weight pre-condition (see 3.6.1)
5. **@Ensures** and **@Effects** method annotations specify light-weight post-conditions for normal executions (see 3.6.2)
6. **@Returns** method annotation is a syntactic sugar for a light-weight post-condition for normal executions (see 3.6.2)
7. **@Throws** method annotation specifies a light-weight exceptional post-condition (see 3.6.4)
8. **@Modifies** method annotation specifies a light-weight frame condition (see 3.6.3)
9. **@Pure** method annotation is a syntactic sugar for an empty frame condition (see 3.6.3)
10. **@Specification** method annotation specifies a heavy-weight specification (see 3.6.5)

Each of these annotations takes a string value that contains a JFSL expression. It is possible with most of them to supply an array of strings instead of a single string. The meaning of an array of strings complies to the natural interpretation, i.e. multiple strings in specification field declaration would provide multiple declarations of distinct fields.

3.2 Formal Grammar

The following section describes the grammar of JFSL in Extended Backus-Naur form. The operators used in the production rules are Kleene star (*), choice operator (|), and option ([. . .].)

3.2.1 Design Considerations

The grammar of JFSL matches the grammar of Java 5 language closely. That means most side-effect free expressions in Java 5 belong to JFSL and carry the same semantic meaning. However, most of the operators in JFSL are overloaded with more flexible semantics of Alloy operators. The main semantic difference between JFSL and Java 5 is that every expression in JFSL evaluates to a set of tuples like in Alloy. Scalars are treated as singleton sets. In particular, JFSL lacks any notion of a higher-order set. Both x and $\{x\}$ are equivalent in JFSL. The lexical analyser rules of JFSL are identical to Java 5. Decimal integers, floating point numbers, and string literals are written in the same way as Java 5. The lexical rule for the identifier is borrowed from Java 5:

id ::= Letter (Letter | JavaIDDigit)*

where Letter represents any non-digit Unicode character. The following Java expressions are *permitted* in JFSL:

1. conditional expressions (expr ? expr : expr);
2. boolean operators for "or" (||) and "and" (&&);
3. bit operators (|, ˆ, &, >>>, <<, and >>);
4. equality and comparison operators (==, !=, <, >, <=, and >=);
5. arithmetic operators (+, -, *, /, and %);
6. boolean negation operator (! unary operator);
7. instanceof expressions.

Non-side effect free expressions are not permitted in JFSL. These include variable assignment (=), and integer postfix and prefix operators (++ and −.) The operator = can be used in JFSL but its meaning is equivalent to operator ==. Another significant difference between Java and JFSL is that this is required in any field dereference expression since fields are interpreted as binary relations. Thus, there is no implicit this in JFSL.

3.2.2 JFSL Expressions

This section describes JFSL expression language in full detail. The entire grammar is presented in figure 3-1 in extended Backus-Naur form. The order of precedence and associativity of operators in the grammar are from low to high.These rules are consistent with Java 5 language specification. The set operator precedence and associativity is generally consistent with Alloy. However, their combination slightly deviates from Alloy.

```
expression ::= conditionalExpression
conditionalExpression ::= quantifiedExpression [? expression : expression]
quantifiedExpression ::= setQuantOp decls '|' expression | logicalExpression
setQuantOp ::= setUnaryOp | # | all
logicalExpression ::= conditionalOrExpression [<=> | <! > | => conditionalOrExpression]
conditionalOrExpression ::= conditionalAndExpression (|| conditionalAndExpression)*
conditionalAndExpression ::= inclusiveOrExpression (&& inclusiveOrExpression)*
inclusiveOrExpression ::= exclusiveOrExpression (| exclusiveOrExpression)*
exclusiveOrExpression ::= andExpression (ˆ andExpression)*
andExpression ::= equalityExpression (& equalityExpression)*
equalityExpression ::= instanceOfExpression (== | ! = | =) instanceOfExpression)*
instanceOfExpression ::= relationalExpression [instanceof type]
relationalExpression ::= setUnaryExpression ( <= | >= | < | > | in | !in)
setUnaryExpression)*
setUnaryExpression ::= setUnaryOp joinExpression | shiftExpression
setUnaryOp ::= setDeclOp | no | sum
setDeclOp ::= one | some | lone
shiftExpression ::= additiveExpression ((<< | >>> | >>) additiveExpression)*
additiveExpression ::= sizeExpression (( + | -) sizeExpression)*
sizeExpression ::= (# joinExpression) | multiplicativeExpression
multiplicativeExpression ::= setAdditiveExpression ((⋆ | / | %) setAdditiveExpression)*
setAdditiveExpression ::= overrideExpression (@+ | @-) overrideExpression)*
overrideExpression ::= intersectionExpression (++ intersectionExpression)*
intersectionExpression ::= composeExpression (@& composeExpression)*
composeExpression ::= unaryExpression (− > unaryExpression)*
unaryExpression ::= (+ | - | ∼ | !) unaryExpression | unaryExpressionNotPlusMinus
unaryExpressionNotPlusMinus ::= castExpression | joinExpression
joinExpression ::= primary selector*
castExpression ::= ( primitiveType ) unaryExpression
decls ::= id (in | :) additiveExpression (, id (in | :) additiveExpression)*
common :: = ( expression )
        | return | throw | this | super
        | @old ( expression )
        | @arg ( integerLiteral )
        | relationalUnaryExpression
        | { decls | expression }
relationalUnaryExpression ::= specUnaryOp ( expression ) | specUnaryOp id
specUnaryOp ::= @∧ | ∧ | @∼ | ∼
primary ::= common | literal | typeDisambiguous |
        | id (. id)*
        | typeName @ id
selector ::= . id arguments |
        . id | . common |
        . ⋆ id | . ⋆ common |
        '[' expression ']'
```

Figure 3-1: JFSL expression language EBNF

3.2.3 Set Expressions

Set and tuple operations in JFSL are:

1. + (and @+) is set union;
2. - (and @-) is set difference;
3. & (and @&) is set intersection;
4. ++ is the relational override;
5. in and !in are subset relationship tests (and negated test);
6. = and == are interchangeable set equality tests;
7. $->$ is the Cartesian product.

Java operators (+, -, and &) acquire semantics of set operations instead of arithmetic operations whenever Java specification allows that(see 3.2.6.)

3.2.4 Primary Expressions

The following Java primary expressions are allowed in JFSL:

1. this literal;
2. boolean and integer literals;
3. fully-qualified type names and ambiguous identifier names (not that '.' carries different semantic meaning in package names and relational join chains);
4. parenthesised expressions.

JFSL introduces several new primary expressions:

1. return refers to the return value of the method (if it is available);
2. throw refers to the thrown exception (or null if none is thrown);
3. @old evaluates the sub-expression in the pre-state;
4. @arg refers to the method parameter by index (Java does not preserve argument names in interfaces and abstract classes.);
5. navigational expressions such as ˆ id which is a closure of the relation of id
6. set comprehension expressions;
7. qualified field references type @ id that refer to the field id in type type as a relation.

3.2.5 Relational Expressions

JFSL provides several operators for navigating relations. Together with the relational join they offer a concise and flexible way to describing complex conditions. These operators are:

1. . is the relational join operator.
2. .$\star$ is the reflexive transitive closure join operator;
3. ˆ (and @ˆ) is the transitive closure unary operator;

4. $\sim$ (and @$\sim$) is the relational transpose unary operator;

The use of @ in operator names is not mandated since the type inference automatically converts bitwise exclusive "or" (ˆ), number multiplication ($\star$), and bitwise complement ($\sim$) operators into relational operators if it detects that Java operators cannot be applied to the inferred types of the arguments (see 3.2.6.)

3.2.6 Operator Conversion

The meaning of arithmetic operators +, -, and & depends on the types of their sub-expressions which are derived using type inference rules (see 3.3.) If at least one of the sub-expressions has a non-numeric type, the operator is automatically converted to a set operator (@+, @-, and @&, respectively.) A numeric type is a type of arity 1 corresponding to a numeric primitive type in Java.

For example, expression 1+2 evaluates to 3 but expression 1 + "2" evaluates to {1, "2"}.

Note that while operator conversion does not interfere with expressions involving only values of numeric types, it conflicts with Java's rule of string promotion for +.

Similar mechanism resolves the meaning of the ambiguous $\sim$ operator that could either be a bit-wise not operator or a relational transpose operator using the type of the sub-expression. Other ambiguous operators can be resolved from the context without type information.

3.2.7 Set Comprehension

Set comprehension expressions are written with curly braces:

```
{ decls | expression }
```

The elements in the set are tuples of variables from the declaration decls that satisfy the predicate expression. For example, the set of integer pairs $\langle x, y \rangle$ such that $x^2 + y^2 = 25$ is written in JFSL as:

```
{x:int, y:int | x⋆x + y⋆y = 25}
```

3.2.8 Quantification Expressions

There are two types of quantification expressions in JFSL. One form has the following grammar:

```
quantifiedExpression ::= setQuantOp decls '|' expression | logicalExpression
decls ::= id (in | :) additiveExpression (, id (in | :) additiveExpression)*
```

The expression evaluates to either a boolean or an integer scalar based on the kind of the quantification operator. The interpretation of the quantification expressions is the following:

- one : there is *exactly one* tuple of declared variables from their respective domains that satisfies the predicate logicalExpression
- some : there is *at least one* such tuple;
- lone : there is *at most one* such tuple;
- no : there are *no* such tuples;
- all : for any tuple of declared variables in their respective domains, the predicate is true;
- sum : the value is the sum of values of the formula expression for all tuples of variables in the domain;
- # : the value is the total number of tuples of variables in the domain that satisfy the predicate logicalExpression.

Another kind of quantification expressions omits the predicate altogether and is described by the following grammar:

```
relationalUnaryExpression ::= specUnaryOp ( expression ) | specUnaryOp id
```

The expression has the following meaning depending on the type of the quantifier:

- one : there is *exactly one* element in the set that sub-expression evaluates to;
- some : there is *at least one* element in the set;
- lone : there is *at most one* element in the set;
- no : there are *no* elements in the set;
- sum : the value is the sum of integer elements in the set;
- # : the value is the number of elements in the set.

3.2.9 Logic Expressions

JFSL has operators for logical equivalence (<=>), negation of logical equivalence (<! >), and logical inference (=>.)

3.2.10 JFG File Format

While the annotation facility of Java 5 is convenient to use in practice, there is a need for an alternative way to provide specifications. I designed a specification file format that serves this purpose. The grammar for the file is shown in figure 3-2. The files of this format are automatically loaded by our tools if they are present in the class path. The specifications from multiple sources (JFG files and annotations) are unified together according to the following rules:

- invariants from both sources are logically conjoined;
- specification field declarations from both sources are combined;
- method specification cases are combined;
- for a given method, the resulting specification is pure if at least one of the sources is pure, and helper if at least one of the sources is helper.

```
compilationUnit ::=
        package packageName ;
        (import importName)*
        typeDeclaration ;*
typeDeclaration ::=
        (class | interface) id [typeParameters]
        { typeBodyDeclaration* };
typeBodyDeclaration ::= ; | typeDeclaration
        | @Invariant ( expression )
        | @SpecField ( declaration)
        | id ( methodParameters) { specCase (’;’ specCase)* ;* }
specCase ::=
        [@Requires ( expression )]
        [@Ensures ( expression ) | @Throws ( expression ) ]
        [@Modifies ( frame )]
        | @Helper
        | @Pure
```

Figure 3-2: JFG file format

Specifications within JFG specification case sections are identical to specifications within annotation specifications.

The mechanism of JFG file format provides a means to *refine* pre-existing specifications access to which is limited, due to unavailability of the source code for example.

3.3 Type Inference

The type system of JFSL is based on the notion of a *tuple type*. A tuple type is a vector of Java types:

$$T_1 \rightarrow T_2 \rightarrow \ldots T_k$$

where each T_i is a Java type. We call k the arity of the type.

Type inference for tuple types of arity 1 works exactly like in Java. For example, arithmetic operators or field dereferences.

For tuple types of arity 2 set operations require new inference rules which are presented in figure 3-3. These rules use the function $\mathsf{LCA}(T, U)$ that gives the least common super-type of T and U in the single inheritance hierarchy. Traditionally, we consider arrays and interfaces as descendants of java.lang.Object. Primitive types fall out of the object hierarchy but they have their own inheritance tree [11].

While Forge has its own type inference rules, they use the fact that the set of types is finitized before type checking, i.e. the universe is reduced to a finite set of types. In that case, the types may be considered as simply sets of elements and set operations directly applied to these set representations. JFSL type inference cannot make an assumptions about unknown sub-types. Therefore, the rules for JFSL only use super-type information and make crude estimations of the values.

UNION

$$\frac{a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow \ldots \rightarrow U_k}{a + b : \mathsf{LCA}(T_1, U_1) \rightarrow \ldots \rightarrow \mathsf{LCA}(T_k, U_k)}$$

DIFFERENCE

$$\frac{a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow \ldots \rightarrow U_k}{a - b : T_1 \rightarrow \ldots \rightarrow T_k}$$

INTERSECTION

$$\frac{a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow \ldots \rightarrow U_k}{a \& b : \mathsf{LCA}(T_1, U_1) \rightarrow \ldots \rightarrow \mathsf{LCA}(T_k, U_k)}$$

JOIN

$$\frac{k \geq 1 \qquad l \geq 1 \qquad a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow \ldots U_l}{a.b : T_1 \rightarrow \ldots \rightarrow T_{k-1} \rightarrow U_2 \rightarrow \ldots \rightarrow U_l}$$

REFLEXIVE TRANSITIVE JOIN

$$\frac{k \geq 1 \qquad a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow U_2}{a. \star b : T_1 \rightarrow \ldots \rightarrow T_{k-1} \rightarrow \mathsf{LCA}(T_k, U_2)}$$

PRODUCT

$$\frac{a : T_1 \rightarrow \ldots \rightarrow T_k \qquad b : U_1 \rightarrow \ldots \rightarrow U_l}{a \rightarrow b : T_1 \rightarrow \ldots \rightarrow T_k \rightarrow U_1 \rightarrow \ldots \rightarrow U_l}$$

Figure 3-3: JFSL type inference rules

A challenging aspect of the type system is handling the transitive reflexive closure. Initially, closure was considered a unary operator. That posed the question what the type of the identity relation should be. The identity relation is a binary relation that maps every atom to itself. In our type system it would acquire the type Object→Object which is useless for resolve field dereferences on the right hand side. Due to this challenge, we decided that reflexive transitive closure deserves a special binary operator.

Type inference in JFSL produces a bound on the exact type of sub-expressions. For Java expressions the inference rules derive the compile-time type of the expression. This inference is a modular static analysis that only requires knowledge of all super types of relevant types.

Type information does not capture information about whether an expression is a singleton or not. Therefore, a set of elements has the same type as the elements themselves. Despite that, type checking can possibly detect many type violations inside boolean predicates and integer expressions.

3.3.1 Name Resolution

The identifiers in the expressions are resolved using the context of the expression. The context includes the stack of local variables such as the receiver and method parameters as well as global variables in the declaring type.

Resolution of a selector expression uses the context of the preceding selector or primary expression. The context brings forward the name space corresponding to the right-most type in the inferred type of the preceding expression. This name space includes all the declared visible members of the type. This rule ensures that all valid Java field dereference expressions are correctly parsed and interpreted in JFSL.

Evaluation of the selector expressions proceeds from left to right starting from the primary expression. The production rules for the selector expressions include almost all of possible primary expression types. These expressions are joined together to the *right-side* of the primary expression. The only additional selector expression is the array bracket expression.

The meaning of the expression depends on the type of the previously evaluated left-hand side expression. If it has an array type, then Java array dereference is performed. Otherwise, the expression is treated as a *left-side join*. For example, expression $a[b]$ is equivalent to $b.a$ if the type of a is not an array type.

There is an additional explicit field of arrays elems that evaluates to the binary relation from the indexes to the elements in the array.

3.4 Undefinedness

JFSL is a declarative language so the only way it can handle undefined expressions is by choosing and assigning a special value for all sub-expressions that are not well-defined. The convention of the language is that the undefined value is *an empty set*. The result is that field dereferences of null and illegal arithmetic expressions, such as division by zero, are evaluated to $\varnothing$.

3.5 Type Specifications

Type annotations belong to data type definitions in Java, i.e. interfaces and classes declarations. While they can be attached to individual members of types, there is no special treatment of this relationship. Type specifications are inherited via sub-classing and interfaces.

3.5.1 Invariants

Invariant expressions are simply expressions in JFSL that evaluate to boolean value according to the type inference rules. The predicate for expression expression is expression = true. The expressions are evaluated in the context that contains the declaring type and the literal this of the declared type.

A special annotation is used for methods that do neither need to assume nor guarantee invariants. Methods marked with **@Helper** tag serve as *helper methods*.

3.5.2 Specification Fields

The grammar of a field declaration is the following:

```
declaration ::= id : [mult] additiveExpression [from frame [ | expression]]
mult ::= setDeclOp | set | seq
```

The identifier id is the name of a specification field. All specification fields are *instance fields* and share the same name space as Java fields belonging to the declaring type. We call additiveExpression the *upper bound* of the field id, frame the *frame* of the field id, and expression the *abstraction function* of id.

The specification fields are *public*, i.e. they can be accessed within any specification expressions. They can only be updated via the abstraction function or explicitly via a post-condition.

The multiplicity factor of a specification field is interpreted as follows. Let U be the inferred type of the upper bound in the declaration in a type T. Then the type of the field relation is:

$$T \rightarrow U$$

unless the multiplicity modifier is seq, in which case it is

$$T \rightarrow int \rightarrow U$$

The multiplicity keyword also imposes a condition on the value of the field for a given instance $t \in T$:

1. one - the field has a singleton value similar to Java fields;
2. some - the value is a non-empty set;
3. lone - the value is either an empty-set or a singleton;
4. set - the value is an arbitrary set;
5. seq - the value is a sequence.

Similar to Alloy, if U has arity 1, then the default (unspecified) multiplicity is one; otherwise, it is set.

These constraints are considered a part of the abstraction function of the field.

3.6 Method Specifications

Specification cases are written within method-only annotations. There are three kinds of specification cases provided by annotations:

1. lightweight specification case is formed from the top-level method annotations for the pre-condition, post-condition, and so on;
2. exceptional specification cases are extracted from a special method annotations;
3. heavyweight specification cases are written inside a structured annotation.

3.6.1 Pre-condition

A normal pre-condition is written inside **@Requires** method annotation or inside the heavyweight specification. Syntactically, the pre-condition expressions satisfy the following conditions:

1. the local variables are method arguments and, if the method is non-static, this;
2. access to members of other types is limited only by the type context, i.e. all private members of the declaring type are accessible but not those of other top-level types;
3. @old expression is prohibited.

The pre-condition is meant to be evaluated in the pre-state of the method.

3.6.2 Post-condition

A post-condition for the light-weight specification case is written inside either **@Ensures** or **@Returns** method annotation. The expression inside **@Ensures** satisfies the following conditions:

1. the local variables are method arguments holding their pre-state values, this (if the method is non-static), and return (if the return type of the method is not **void**.)
2. access to members of other types is limited only by type context

@Returns annotation serves as a syntactic sugar. The condition **@Returns**(”expression ”) is equivalent to **@Ensures**(”return = expression”).

The post-condition is meant to be evaluated in the post-state.

3.6.3 Frame Condition

A frame condition for the light-weight specification case is written inside **@Modifies** annotation. The grammar for frame expression is the following:

```
frame ::= [storeRef (; storeRef)*]
storeRef ::= storePrimary storeSelectors
storePrimary ::= id | common
storeSelectors ::= selector⋆ .⋆ | selector+
```

All expressions in the frame condition are evaluated in the pre-state, and so @old expression is prohibited.

For every storeRef expression, the selector fields are resolved using type inference rules using the storePrimary as the left-most expression. The special keyword ⋆ is resolved to all the fields of the type (as determined by type inference.) The fields include both concrete Java fields and specification fields.

A special tag annotation **@Pure** attached to a method serves as a shortcut for adding a specification case with the trivial pre-condition and post-condition (which is **true**) and an empty frame condition. This specification says that the program may not modify the existing heap state under any circumstances.

3.6.4 Exceptional Case

@Throws annotation provides exceptional specification cases. The grammar for the condition inside the annotation is the following:

throwsCondition ::= typeName : expression

This is a short way of adding a specification case with the pre-condition expression, empty frame condition, and the post-condition:

```
throw in ExceptionType
```

where ExceptionType is the resolved type of typeName.

For example, the exceptional behavior triggered by parameter node being equal to **null** can be written as:

```
@Throws("NullPointerException : node = null")
```

The post-condition here is that the NullPointerException is thrown and no changes to the heap are made.

3.6.5 Heavy-weight Specification Cases

For more complicated specifications, an alternative form of specification cases is provided:

```
@Specification( {
        @Case(requires = {"no x in this.nodes | x.key < node.key", "node
             in this.nodes"}, ensures = "return = null"),
        @Case(requires = "node = null", ensures = "throw in
            NullPointerException", type = Type.EXCEPTIONAL) })
```

The specification cases are written as fields of an annotation type but the same rules apply to the expressions as in light-weight specification case. The only difference concerns **throw** variable which can be directly referenced in the post-condition of an exceptional specification case.

Heavy-weight specifications may provide multiple specifications for normal execution unlike the light-weight specifications. They also use slightly more verbose syntax. If both heavy-weight and light-weight specification cases are present, they are combined together.

3.6.6 Default Values

If any of the annotations **@Requires**, **@Ensures**, **@Modifies** is present then the light-weight specification case is constructed. The default values for pre-conditions and post-conditions are **true**. The default frame condition is empty. The same default values hold for omitted heavy-weight specification case fields.

There are no default heavy-weight or exceptional specification cases.

Chapter 4

Semantics of JFSL

In this chapter, I explain the interpretation of JFSL specifications in the intermediate representation. The representation is based on the Forge framework and uses first-order logic with relations and transitive closure to express the specification conditions.

4.1 Invariants

Invariants provide an intuitively appealing and light-weight means of formalizing behavior of all objects of the given type. However, designing a modular semantics for invariants meets substantial challenges. The basic problem is understanding the dependence of method specifications on the invariants across types as well as dependence of invariants on other invariants, which is frequent in layered object structures. Another challenge occurs in combination of invariants and object type hierarchies because of disparity between the exact and the inferred type of objects.

One approach to this problem is the *explicit* validity predicate on every object. Spec# uses the notions of packing and unpacking which allow explicit guarantees and assumptions of object invariants. However, this introduces a major language change on the level of C#, the base source code language. It is rather heavy and complicated in its usage.

The approach that JForge uses is much more lightweight. However, it has its own drawbacks that may deem it less modular or tied to the particular tool I developed. Central to this approach is the notion of a *scene*. A scene is a tuple $\langle T, F, M \rangle$ of types T, fields F, and methods M. There is a natural containment relation on scenes induced by subset containment. Every scene satisfies the following conditions:

1. T is closed under "extends" and "implements" relations, i.e. every type in T has all its super-types in T;
2. declaring class of every field in F is in T;
3. declaring class of every method as well as the declaration types of its arguments in M are in T;

4. scene of the specifications of methods in M is contained in the scene;
5. scene of every abstract field in F is contained in the scene;
6. scene of every invariant of every type in T is contained in the scene;
7. scene of any expression that contains a method call includes the scene for the method.

Scenes define the notion of *relevancy* of Java elements to JFSL expressions (not only basic expressions but also declarations, predicates, and frames). A type, a field, or a method is relevant to JFSL expression if and only if it belongs to the smallest scene containing the expression. One can think of the scene as the minimal knowledge one must collect to reason about the expression in a modular fashion. The scenes for non-basic JFSL expressions are defined as follows:

1. the scene of a specification is the union of the scenes for its pre-conditions, post-conditions, and frame conditions;
2. the scene of a specification field declared in a type T is the union of the scene of its declaration, frame, and abstraction function in declaration in T.

We can now formalize the notion of invariants in JFSL by saying that the invariant of a type is the conjunction of invariants of types in the smallest scene containing the type. More generally speaking, the invariant to be used in an analysis that requires a scene $\langle T, F, M \rangle$ is the conjunction of invariants of types in T.

JFSL only allows invariants on instances but not on types. The motivation for that is that static invariants go against modularity of specifications. In addition to that, the non-determinism in Java virtual machine class loading mechanism make inference of relevant static invariants hard.

The immediate benefit of this design is that invariants are implicitly inferred and the inferred invariants are only the ones that are relevant to the system. Below are the points of criticism that we should address.

Dependence on the form of expressions The scenes for different methods may differ even though the methods may belong to the same class. Scenes may also change depending on how the expression is expressed. A convoluted example is when an invariant of some type is unsatisfiable but the type itself is only relevant to some of the methods but not others. This situation will make specifications of all methods relevant to this type have pre-conditions false. Moreover, changing the scene of an expression by referencing another type or a field may inflict changes in the invariants of a specification. We see this as a trade-off in favor of modularity of the approach. It is not feasible to consider all invariants of a system in any modular analysis, and it is also not desirable to have explicit validity predicates that would tell whether the object is valid or not as in Spec#.

No explicit control for layered objects If a type has multiple components then there is no explicit control on saying that the invariant of type depends on the invariants of its components. The situation becomes worse if the mutator methods may break invariants of components selectively while assuming invariants of other components. While it is possible to express that the method is a helper, there is no fine-grained control of validity of the object and its components. Our approach applies to the methods of the top-level class but may not work well for specifications of components within a top-level class.

4.2 Specification Fields

Specification fields are very useful in modeling the abstract state of Java objects [12]. They are the primary means of abstraction in verification process reducing complexity of multiple implementations of an abstract data type to a single model. In our experience, specifications fields simplify description of operations on a data type. Since Java facilitates design of abstract data types via interfaces and inheritance, JFSL uses similar mechanics of fields and types for declaration of specification fields.

For exposition we will use a directed graph as an example. Abstractly, a graph consists of a set of nodes and edges. Edges are ordered pairs of nodes in the graph. In JFSL this definition would look like the following code snippet:

```
@SpecField({"nodes : set Node", "edges : set Node -> Node"})
abstract class Graph {
        class Node { ...
```

In this example, class Graph declares two specification fields: nodes of type Graph → Node, a relation from a graph to its nodes, and edges of type Graph → Node → Node, a relation from a graph to its directed edges. These fields specify neither frame, nor abstraction function. The specification of this class may freely talk about these fields and update their values. For example, we may have the following method that adds a node to the graph:

```
@Requires(node != null)
@Ensures("this.nodes = @old(this.nodes) + node")
@Modifies("this.nodes")
void add(Node node) { ... }
```

Now let us give a concrete implementation of our abstract class. This implementation will necessarily use some representation for its nodes and edges. It is desirable to provide an abstraction function that ties together the abstract state (values of the specification field) and the concrete state (the representation in the concrete class.) Let us represent them as arrays:

```
@SpecField({"nodes: set Node from this.n | this.nodes = this.n[int]",
"edges: set Node -> Node from this.e | this.edges = {a : Node, b : Node
    | some edge : this.e[int] | edge.from = a && edge.to = b}"})
class ConcreteGraph {
        private Node[] n;
        private Edge[] e;
        class Edge {Node from; Node to; ...
```

Declaration of a specification field f in class C results in a *frame* $\mathcal{F}_C(f)$, an *upper bound* and an *abstraction function* (see 3.5.2.) Specification field is declared only once but may be *refined* in sub-classes. Therefore, specification fields may have multiple abstraction functions, multiple frames, and upper bound constraints. For example, field nodes is given an abstraction function in class ConcreteGraph but none in Graph.

Abstraction function binds the value of the field to the concrete state under the assumption that the object state is valid, i.e. all its representation invariants hold. Despite being called a function, it takes a form of an arbitrary constraint. Whether or not the constraint defines a function from the valid states to abstract states is an important condition subject to bounded verification by JForge.

Upper bound is an additional constraint that is conjoined to the abstraction function. It also defines the type of the values of the specification field. We will use notation $\mathcal{A}_C(f)$ to refer to the abstraction constraint of f in declaration in class C. This constraint combines the abstraction function, the upper bound constraint, and the multiplicity constraint.

Frame of a field specifies dependencies of the abstract state on the concrete state; this is essentially the data group of the field [15]. In our example, the value of nodes field is bound to a set of nodes. It depends on the concrete field this.n which means that any frame condition that allows a change to nodes field should also allow a change n field as well. On the other hand, any change to the field n might trigger an update on the corresponding abstract value of nodes.

Either frame or abstraction function or both may not be given for a specification field. In that case, the field may not be bound to the concrete state at all. This might be the case for Java interfaces which do not have any representation but are integral to abstract data type definition.

4.2.1 Inheritance and Data Groups

Inheritance provides a mechanism for refining an abstraction function. The abstraction function for a specification field f with top-most declaration in class C in a scene $\langle T, F, M\rangle$ is:

$$\mathcal{A}(f) = \bigwedge_{S \in T | S <: C} \forall \textsf{ this } \in S \mid \mathcal{A}_S(f)$$

If the sub-type S is missing both upper bound and abstraction function, we assume $\mathcal{A}_S(f) = \textsf{true}$. Note that this constraint does not truly specify a function. We use this as a handy exception to the rule that the $\mathcal{A}(f)$ has to specify a function.

Similarly, the frame of the field f in a scene $\langle T, F, M\rangle$ is:

$$\mathcal{F}(f) = \textsf{this}.f \cup \bigcup_{S \in T | S <: C} \mathcal{F}_C(f)$$

The frames of all sub-types in the scene are combined together to form a set of locations all starting from this that the specification field depends upon. These dependencies are required to be only on the concrete fields and objects, i.e. specification fields are not allowed to list each other in their frames to prevent recursive frame constraints. Whenever a specification field appears in a frame condition of a specification, it is automatically unrolled to its complete frame in the current scene. This mechanism allows method specifications to refer indirectly to the unknown concrete dependencies of a specification field in sub-types of the declaring class.

4.3 Method Specifications

Method specifications are the behavioral contracts stating the assumptions and guarantees that the method execution has to respect as in design-by-contract paradigm.

The specification consists of *specification cases* targeting different scenarios, which can be broadly grouped into normal executions and exceptional executions. The specification cases are all publicly visible and inherited from the super classes and implemented interfaces. Each specification case consists of a pre-condition pre, a frame condition frame, and a post-condition post. Such a specification says:

> only when the heap satisfies pre, the method applies and the post-condition post is established in the post-state by making changes to the pre-state heap according to frame condition and potentially adding new objects to the heap.

The pre-condition of the entire specification is the *disjunction* of pre-conditions of specification cases, and it represents the condition that the client has to satisfy in order to call the method. Failure to satisfy this condition results in unconstrained execution and arbitrary changes to the heap.

Expressions inside method specifications are evaluated in one of the two states:

- *pre-state* is the state of the program before entering the body of the method and after evaluating all the method parameters;
- *post-state* is the state of the program immediately after returning from the method body.

4.3.1 Pre-condition

The pre-condition is the a condition that the client has satisfy. It is always evaluated in the pre-state. For non-helper methods, the pre-condition is the conjunction of the stated pre-conditions for all specification cases and scene invariants.

4.3.2 Post-condition

The post-condition is a condition, imposed on the method being specified, on the heap in the post-state. For specification cases with pre-conditions pre_i, post-conditions $post_i$, and frame conditions $frame_i$, the entire specification is interpreted as formula:

$$\bigwedge_i pre_i \Rightarrow (post_i \wedge frame_i)$$

which says that for every pre-condition both its corresponding post-condition and the frame condition are implied. The post-condition of a non-helper method includes the scene invariants as well.

4.3.3 Frame Condition

A frame condition consists of a set of store references. Each store references $x.f$ can be decomposed into a set of locations x and a relation f that is either a field or array elements relation elems. If f is a specification field, then its frame is implicitly included in the frame for locations in x via substitution of **this** by x (see 4.2.)

The interpretation of a frame $\{x_1.f_1, \ldots, x_i.f_i\}$ is the following. First, all locations are grouped by their relations, i.e. if f_1 and f_2 refer to the same relation, then their frames grouped into $(x_1 + x_2).f_1$. Assuming all relations in the store references are distinct, the frame condition is the conjunction of frame conditions for every relation $x.f$:

> for any instance t in the domain of f in the pre-state, the value of $t.f$ is the same unless $t \in$ @old(x); this applies to both concrete and abstract fields.

Simply put, it says that nothing else in pre-state domain of f can be changed except for x and new instances. Note that the frame condition does not say anything about the fresh instances that might be created during the execution of the method.

For example, the following condition says that left, right, color, parent fields can be changed for any Node in the pre-state, but root field can only be changed for this instance and any newly created instance:

```
@Modifies("Node.left, Node.right, Node.color, Node.parent, this.root")
```

4.3.4 Constructors and Static Methods

Static methods do not have a receiver but that is practically the only difference they have from instance methods. Constructor calls internally in byte code are represented as methods called on fresh instances in the order of the sub-typing. We can still reason about constructors and invariants as long as the constructor specification does not assume type invariants on the instance being created.

4.4 Method Calls inside Specifications

JFSL allows pure method calls inside specifications. However, reasoning about specifications that allow unrestricted method calls inside specifications is challenging and subject to further investigation [8]. There are several competing approaches:

1. *Treat method calls as uninterpreted partial functions.* In the context of bounded verification, every method call would require an additional relation that is constrained to satisfy method specifications. While this approach is sound and elegant, the arity of the relation is equal to the number of arguments plus two which, as of this day, easily exceeds capabilities of SAT solvers that are used in the back-end of bounded verification.
2. *Solve for return values of the method calls.* The return values are constrained by the method specification with arguments substituted by the concrete inputs. If these constraints are conjoined to the predicate passed to the SAT solver, both would be solved at the same time. JMLForge successfully employed this procedure for method calls inside JML specifications [9]. However, from our experience, resulting specifications turned out to be overly complex. In addition, special considerations must be applied to handle non-deterministic method calls and failures to satisfy pre-conditions of the called method.
3. *Encourage use of specification fields in lieu of method calls inside specifications. Allow only a restricted form of method specifications to be called.* This approach works most effectively with bounded verification and is adopted by JForge.

The specification for a called method must provide a specification that specifies the return value explicitly as a function from the pre-state. This specification case would have the predicate:

$$\textsf{return} = expr$$

as one of the conjuncts in the post-condition. Method call is then replaced by a conditional expression:

$$pre \ ? \ expr : \{\}$$

where pre is the pre-condition of the specification case and $expr$ has all its inputs substituted by the actual arguments. Notice that we handle client failures using an empty set expression, for the sake of consistency with our notion of undefined expression.

If there are multiple specification cases of the required form, then one of them is chosen at random. In addition to existence of such a specification case, the method must necessarily be pure.

Chapter 5

JForge Tool and Eclipse Plug-in

JForge toolkit includes an Eclipse plug-in that enhances integrated development environment of Eclipse with JFSL annotation support, checking and simulation of Java methods, debugging of specifications, and counter example trace presentations. Via integration of all of these services into a single language and a single tool, JForge offers a friendly environment to encourage developers to use lightweight formal methods in their coding practice. The tools generally require only an understanding of JFSL and basic knowledge of the logic of the intermediate representation. The entire tool chain is packaged and available for public use [3].

5.1 Automated Analysis

JForge is able to analyze specifications written in JFSL. Most importantly, it is able to verify compliance of code against its specification (even though in the bounded setting.) In this section, I describe the exact conditions that JFSL uses and present an overview of other possible automated checks that can be done using specifications. We will assume that a control flow graph for procedure code P is handed in to us in FIR as well as the specification S and its scene.

There are three distinct ways method specifications may be used in the analysis:

1. *checking* attempts to find a counter example of the specification by exhaustively searching for the pre-state and the post-state that admit execution of P and satisfy $\neg S$;
2. *simulation* attempts to find an initial state satisfying the pre-condition of S (if the method is non-helper, the pre-condition includes the invariants as well) in the pre-state and then execute P in it;
3. *inlining* attempts to find a post-state from the given initial state using only method specifications.

Checking and simulation can be performed on a per specification case basis. In fact, JForge encourages breaking down specifications into cases for performance reasons. We believe that tracking the progress of verification (how many cases succeeded) as well as having partial verification (if some cases take too long to complete) are very important from the end-user perspective.

If checking produces a trace, then that means there is a candidate for a bug in the code or the specification. This trace certifies a failure of the specification for a concrete symbolic execution. If simulation produces a trace, then that means there are no over-constraints in the pre-condition and invariants of the specification. The simulated trace is a normal trace of execution that can be examined in detail to ensure correctness of the symbolic execution. Thus, finding a trace means two opposite things for checking and simulation. Correct implementation would produce simulation traces but no checking traces.

5.1.1 Checking

Given a specification with pre-condition pre in the pre-state, post-condition $post$, and abstraction function $\mathcal{A}$, checking attempts to find a trace of P that satisfies the formula:

$$\mathsf{@old}\,(pre \wedge \mathcal{A} \wedge \mathcal{C}) \wedge \neg post$$

where $\mathcal{C}$ is the Java pre-state well-formedness constraint (specifying that inputs are singletons and fields are functions amidst other things.) Since $post$ might use abstract values in the pre-state, the procedure P is modified to include *specification statements* that update the values of abstract functions. This statement is inserted at the end of the control flow graph for P (see section 5.1.4.)

5.1.2 Simulation

Simulation of a specification simply tries to find any trace from a valid state of the heap. It tries to find a trace of P that satisfies:

$$\mathsf{@old}\,(pre \wedge \mathcal{A} \wedge \mathcal{C})$$

Note that this condition is strictly weaker that checking condition. The benefit of this analysis is detection of over constrained invariants or pre-conditions in the pre-state that may make checking vacuously not find any counter examples. This notion of simulation is slightly different from the one used in JMLForge, which also adds $post$ as a conjunct[9].

5.1.3 Inlining

In order to support modular analysis, the method calls in P should favor inlined specifications over inlined code. To inline a specification, we construct a control flow graph that performs execution of the specification using Forge specification statements (see algorithm 1.) Here C means Java post-state well-formedness constraint, specifying that all concrete fields are functions, all arrays have correct

Algorithm 1 Control flow graph for the in inlined specification with pre-condition pre, post-condition $post$, and frame $frame$

```
update all specification fields $f$ at the same time such that $\bigwedge_f \mathcal{A}(f)$
if $pre$ then
  update $frame$ variables, return, throw such that $post \wedge frame \wedge \bigwedge_f \mathcal{A}(f) \wedge \mathcal{C}$
else
  throw $\leftarrow$ ClientFailureException
  return $\leftarrow$ default value
end if
```

index ranges, output variables are singletons, and so on. It uses a special error variable instance ClientFailureException to indicate a failure to satisfy the pre-condition of the specification. This error can propagate back to the caller of the method and provide a meaningful counter example of the client failing to satisfy the pre-condition of the called method. We find that early detection of client failures and presenting traces that pinpoint the location of a client failure is important for debugging. Simply allowing arbitrary changes to the heap would produce traces that are harder to understand and may in fact satisfy trivial specification.

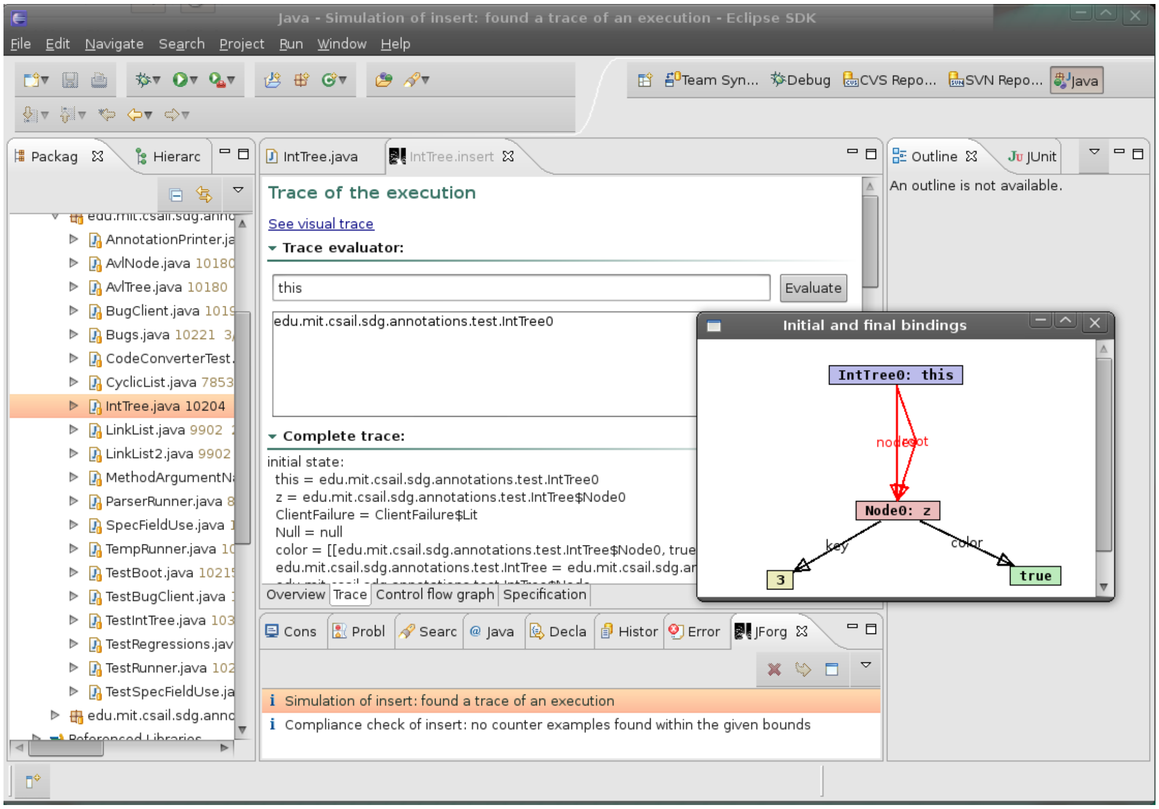
5.1.4 Updates to the Abstract State

Specification fields are updated by changes to their dependency concrete fields. The changes to specification fields are only visible at the method boundaries, and only if they are referenced in the specifications. Therefore, control flow graphs in FIR are modified as follows to include updates to the abstract state:

1. procedure P under analysis ends in a specification statement that updates the values of specification fields right before the exit;
2. inlined code is left untouched;
3. inlined specifications start with an update to specification fields as described in subsection 5.1.3.

When should the specification field be updated? Because of presence of abstraction function constraints that simply say true for interfaces, a field should only be updated only if its abstraction constraint is violated. Therefore, the specification statement that updates the values of specification fields should only modify values for instances that fail to satisfy the abstraction constraint. If the specification field is completely abstract, i.e. not tied to any representation, its values are not non-deterministically changed at every method boundary.

Our methodology of handling specification field updates is similar to packing and unpacking in Spec# [6] applied to every method. While it is not as flexible as packing, it is less intrusive into the programming language and programming practice and satisfies common needs.



```
this
```

```
edu.mit.csail.sdg.annotations.test.IntTree0
```

```
initial state:
  this = edu.mit.csail.sdg.annotations.test.IntTree0
  z = edu.mit.csail.sdg.annotations.test.IntTree$Node0
  ClientFailure = ClientFailure$Lit
  Null = null
  color = [[edu.mit.csail.sdg.annotations.test.IntTree$Node0, true
  edu.mit.csail.sdg.annotations.test.IntTree = edu.mit.csail.sdg.ar
```

```
Simulation of insert: found a trace of an execution
Compliance check of insert: no counter examples found within the given bounds
```

Figure 5-1: JForge graphical user interface

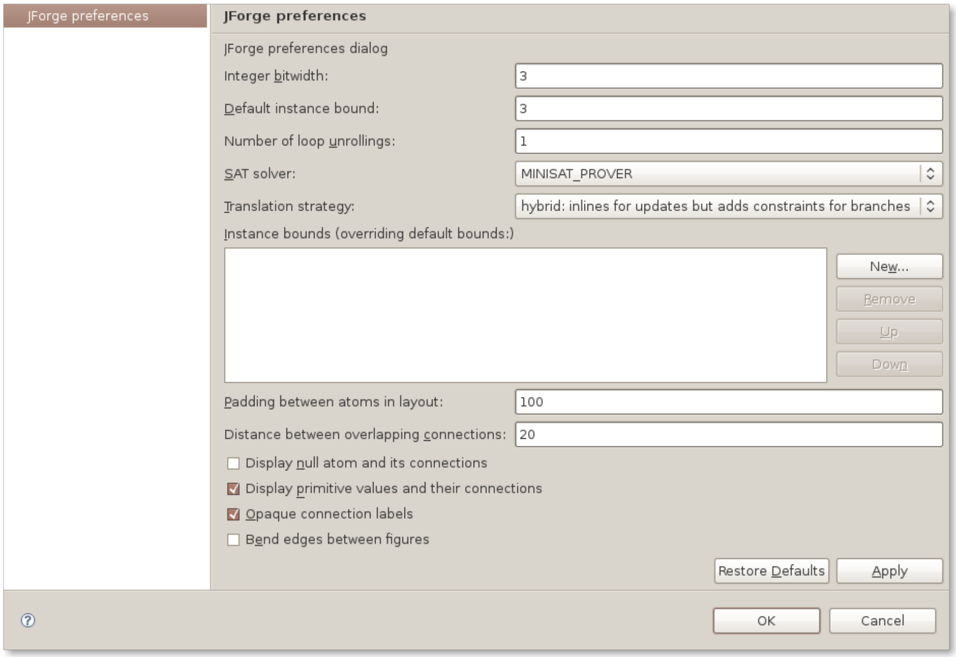
5.1.5 Consistency of Abstraction Functions

There are a few unstated assumptions that checking and simulation take advantage of. For example, the abstraction function constraint should describe a function from the valid concrete states to abstract values. This assumption is not checked every time checking or simulation is performed, but can nevertheless be verified for all instances. A counter example to this assumption is a valid state for which the abstraction constraint has either no solution or at least two solutions. While JForge does not at the moment implement this functionality, it can easily be added to the tool. Another interesting case is the use of specification fields in helper methods. These methods may not guarantee representation invariants and, therefore, cannot guarantee existence of abstract values for the specification fields.

5.2 User Interface

JForge includes an Eclipse plug-in (see figure 5-1) that extends it via several extension points:

1. it provides context menus for methods and classes to check or simulate or do both in aggregate; it also provides context menu on projects to toggle



JForge preferences

JForge preferences dialog

Integer bitwidth: 3

Default instance bound: 3

Number of loop unrollings: 1

SAT solver: MINISAT_PROVER

Translation strategy: hybrid: inlines for updates but adds constraints for branches

Instance bounds (overriding default bounds:)

Padding between atoms in layout: 100

Distance between overlapping connections: 20

□ Display null atom and its connections
☑ Display primitive values and their connections
☑ Opaque connection labels
□ Bend edges between figures

Figure 5-2: JForge preferences dialog

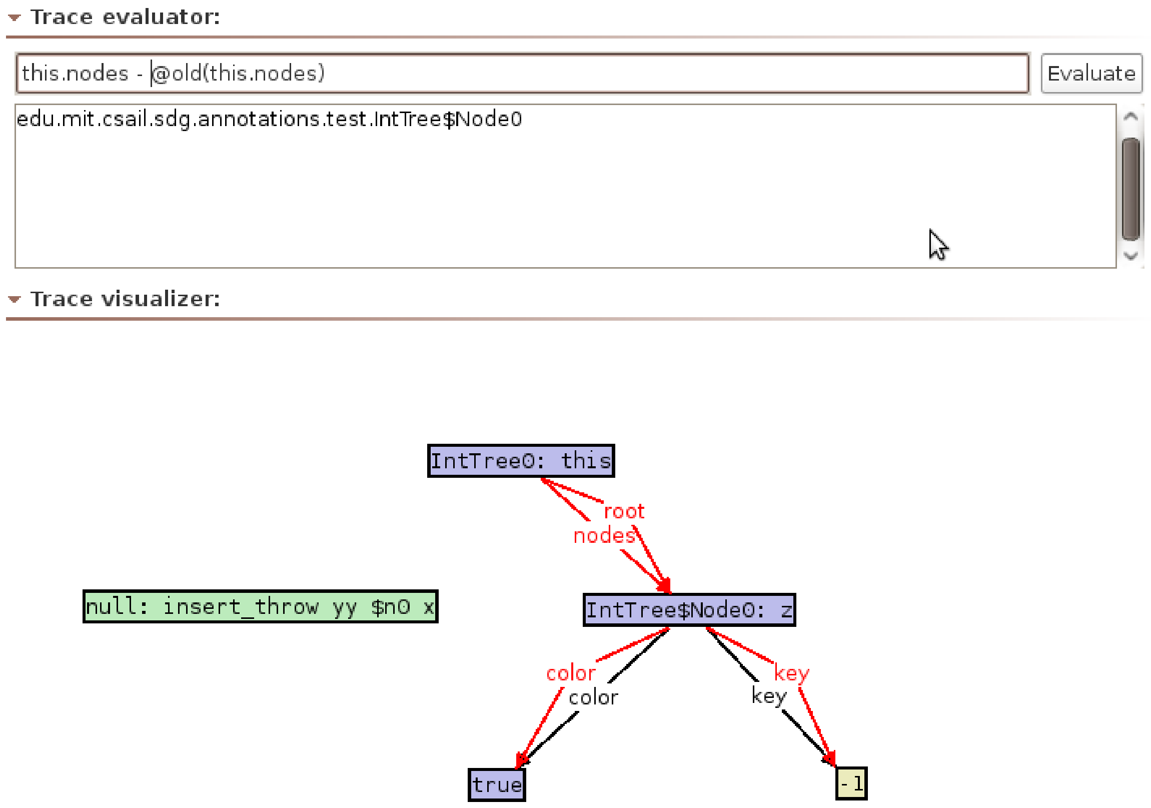
JForge nature and modify class path to include annotation and specification libraries;

2. it adds a JForge report view that displays results of completed analyses together with succinct description of the outcome;
3. a preference dialog (see figure 5-2) for tweaking Forge settings (SAT solver, coverage analysis), and bounds settings that extends standard Eclipse type dialog to select a type and the custom bounds for it;
4. analysis outcome pages that display very detailed information about the textual traces, graphical traces, control flow graphs in the intermediate representation, specifications in the intermediate representation, and a query box to evaluate JFSL expressions dynamically on the counterexample trace;
5. progress dialog using Eclipse jobs infrastructure that reports on the progress of translation and solving, and lets the user terminate the process if it takes longer than expected.

5.2.1 Usability of Verification Tools

The verification tools are notorious for the high expertise that is required to actually use them. In this work, we attempted to make intricate verification technology accessible to a programmer. In particular, our user interface strives to be minimalistic and intuitive to a non-expert in Alloy logic by using diagrams to show the object relationship graph instead of textual relational algebra.

A challenge in formal methods that is not commonly addressed is that speci-



Trace evaluator:

this.nodes - @old(this.nodes)

edu.mit.csail.sdg.annotations.test.IntTree$Node0

Trace visualizer:

Figure 5-3: Trace evaluator and trace visualizer

fications may also contain bugs, and a specialized set of tools is needed to debug specifications as well. For example, over constrained pre-conditions and invariants may give a false impression of successful verification since the verification conditions are vacuously satisfied. JForge tries to address this problem via a number of ways. For example, it offers a simulation analysis aimed exactly at discovering over constraints in the assumptions.

If the unsatisfied formula is of the form $a \wedge b$,
    then the root cause is in either a or b by the de Morgan's law.
If the unsatisfied formula is of the form $a \Rightarrow b$,
    then the root cause is in b.

Figure 5-4: Rules for the root clause discovery analysis

JForge provides several mechanisms to assist the programmer in understanding counter example traces:

1. *Trace visualization.* Figure 5-3 shows the initial(black) and final(red) states of

the heap for a tree insertion method. We can see values of both concrete fields such as key and specification fields such as nodes. Schematic description of graphs is much more effective in describing differences between them. Since Java object heap is essentially a graph, visual presentation of the transition of the heap from the initial state to the final state is very useful.

2. *Trace evaluation.* Figure 5-3 also includes a trace evaluator as part of the detailed information about a trace. It dynamically evaluates expressions using the trace as the context of evaluation – all the instances and values of the fields are extracted from the trace under analysis. This interactive dialog is effective in learning JFSL and investigation of complex heap transitions.
3. *Root cause discovery.* Whenever the specification is very complex, and a counter example is found, it is a non-trivial task to figure out what caused the failure of the specification. It can be any or a combination of a post-condition, a frame condition, or an invariant violation. JForge offers a very simple analysis that looks at the reasons why specification is not satisfied by breaking it down into a set of clauses (as shown in figure 5-4) and presenting them to the user. Each such clause individually makes the entire specification formula false. This simple analysis pin points the exact condition in the specification that is violated.

Chapter 6

Case Studies and Evaluation

We have used JForge to verify a large number of small examples. That way we could be the users of our own tool and evaluate its performance in both usability respect and as a bug-finding tool.

6.1 Red-Black Tree

A very common task in verification is formally specifying and verifying a data structure that represents a common data type such as a sorted tree. A red-black tree is an efficient implementation of a balanced binary search tree [16]. We have used an existing implementation by Emina Torlak that is used internally in Kodkod relational engine [19]. This implementation has been heavily tested since it sits at the core of SAT translation technology. In addition, it was well documented using standard @specfield annotations. We have formally specified it and verified code against the specification. We have discovered a few cases where necessary assumptions were not stated in the Javadoc specification which would count as over-approximation in specifications. Declaration of the data structures and its invariants are shown in listing 6.1.

This data structure is challenging since it uses transitive closure operator to state tree invariants and uses a number of interconnected nodes internally. This is an excellent example of highly linked data structure that requires complex constraints on the topology of the heap.

An important consideration for declarative languages, and in particular, for languages used in verification, is what amount of specification is needed to verify correctness of implementation. We have calculated number of lines of code and specifications excluding comments per method in IntTree (see table 6-1.) Method specifications are all within 7 lines and together with invariants on the data type amount only to $\frac{1}{9}$ of the total number of lines. For some methods such as insert or delete, declarative specification is much more concise than the implementation itself. For some other helper methods such as rotateLeft orrotateRight we expect to see little difference in sizes of the specification and the actual code.

Listing 6.1: IntTree specification

```
@SpecField("nodes : set Node from this.root, this.nodes.left, this.nodes
    .right, this.nodes.parent, this.nodes.color | "+
        "this.nodes = this.root.*(left+right) - null")
public final class IntTree {
private static final boolean BLACK = true;
private static final boolean RED = false;
@Invariant( {
/* root */ "this.root.parent in null",
/* distinct */ "this.root != null => one root.(this.root)"})
private @Nullable Node root;

@Invariant( {
        /* parent left */"this.left != null => this.left.parent = this",
        /* parent right */"this.right != null => this.right.parent =
            this",
        /* parent */"this.parent != null => this in this.parent.(left +
            right)",
        /* form a tree */"this !in this.^parent",
        /* left sorted */"all x : this.left.^(left + right) + this.left
            - null | x.key < this.key",
        /* right sorted */"all x : this.right.^(left + right) + this.
            right - null | x.key > this.key",
        /* no red node has a red parent */" this.color = false && this.
            parent != null => this.parent.color = true"})
        public static class Node {
                public @Nullable
                Node parent, left, right;
                public boolean color;
                protected int key;
                Node(int key) {
                        this.parent = this.left = this.right = null;
                        this.color = BLACK;
                        this.key = key;
                }
        }
}
```

method	# lines of code	# lines of specs
IntTree	1	2
clear	1	2
search	10	1
searchGTE	18	2
searchLTE	18	4
predecessor	11	7
successor	11	6
minAll	7	1
maxAll	1	1
min	6	3
max	6	3
replace	18	0
insert	96	5
delete	42	3
deleteFixUp	93	0
rotateLeft	13	0
rotateRight	13	0
entire class	436	56

Figure 6-1: Number of lines of code and specifications in IntTree; some methods are helper methods and, therefore, have no specifications; entire class also contains invariants and specification field declarations

6.2 Collections Framework

We were able to specify parts of Java collections framework using JFSL. This is important not for verification of the framework itself but for the clients of the collections framework. Using abstract models of collections defined with specification fields rather than implementation allows JForge to scale better and benefits the writer of the specifications for the client code.

Listings 6.2 6.3 show our specifications for Hashtable and List using JFG format (see 3.2.10.) The content of the data structure Hashtable is modeled using an unconstrained specification field that has type Object → Object. The specification provides neither concrete data structure for entries in the table nor the abstraction functions.

Listing 6.2: java.util.Hashtable specification

```
package java.util;
import java.util;

class Hashtable {
        @SpecField(data : Object -> Object)

        Hashtable() {
                @Ensures(no this.data)
                @Modifies(this.data);
        }

        get(Object key) {
                @Requires(key = null)
                @Throws(throw in NullPointerException)
                @Modifies();

                @Requires(key != null)
                @Ensures(return = some this.data[key] ? this.data[key] :
                    null)
                @Modifies();
        }

        put(Object key, Object value) {
                @Requires(key = null || value = null)
                @Throws(throw in NullPointerException)
                @Modifies();

                @Requires(key != null && value != null)
                @Ensures(return = @old(some this.data[key] ? this.data[
                   key] : null) &&
                        this.data = @old(this.data) ++ key -> value)
                @Modifies(this.data);
        }
}
```

Listing 6.3: java.util.List specification

```
package java.util;
import java.util;

interface List {
        @SpecField(data : seq Object)

        size() {
                @Ensures(return = # this.data);
                @Pure; @Helper
        }

        isEmpty() {
                @Ensures(return <=> (this.size() = 0));
                @Pure; @Helper
        }

        contains(Object o) {
                @Ensures(return <=> (@arg(0) in this.data[int]));
                @Pure; @Helper
        }

        add(Object e) {
                @Ensures(
                        this.size() = @old(this.size() + 1)&&
                        this.data = @old(this.data ++ (this.size() ->
                            @arg(0))) &&
                        return = true
                )
                @Modifies(this.data);
                @Helper
        }

        clear() {
                @Ensures(no this.data)
                @Modifies(this.data);
                @Helper
        }

        get(int index) {
                @Requires(@arg(0) >= 0 && @arg(0) < this.size())
                @Ensures(return = this.data[@arg(0)]);
                @Requires(@arg(0) < 0 || @arg(0) >= this.size())
                @Throws(throw in IndexOutOfBoundsException);
                @Pure;
                @Helper
        }
}
```

6.3 Interplay of Interfaces and Object Inheritance

The following example was used for the Squander project [17]. Listing 6.4 shows a specification of an address book that uses a specification field data to model the content of the address book. A concrete class is given in listing 6.5. This class redeclares the specification field and provides both the frame (this.entries) and the abstraction function for data. As you can see, the abstraction function uses the concrete representation of the entries to define the constraint that binds the abstract and the concrete states together. Also notice, that we did not have to refine the specifications unless we want to avoid non-determinism in the structure of the concrete representation that produces the given abstract state.

Listing 6.4: IAddressBook specification

```
@SpecField("data : String -> String")
@Invariant("all x:String | lone this.data[x]")
public interface IAddressBook {

        @Requires("@arg(0) != null && @arg(1) != null")
        @Ensures("this.data = @old(this.data) ++ @arg(0) -> @arg(1)")
        @Modifies("this.data")
        void setEmailAddress(String name, String email);

        @Ensures("return - null = this.data[@arg(0)]")
        String getEmailAddress(String name);

        @Returns("some this.data[@arg(0)]")
        boolean contains(String name);
}
```

6.4 Arithmetic Programs

JFSL supports specifications involving arithmetic expressions. An example often used in verification field is that of the greatest common divisor algorithm. Listing 6.6 shows the code and the specification for this algorithm. JForge is able to check it without finding any counter examples for the bitwidth of 5 and 5 loop unrollings withing a few minutes on a modern machine.

6.5 Cyclic List

Listing 6.7 presents the specification and the partial code of a linked list data structure. JForge is able to verify correctness of its code against the specifications in a few seconds for the bitwidth of 3 and 3 loop unrollings. This example demonstrate the richness of JFSL expressions:

Listing 6.5: AddressBook specification

```
@SpecField("data : String -> String from this.entries"+
        "| this.data = {x in String, y : String " +
        "| some e : this.entries[int] | e.entryName = x && e.entryEmail
           = y}")
public class AddressBook implements IAddressBook {

        @Invariant("this.entryName != null && this.entryEmail != null")
        public static class Entry {
                String entryName = "";
                String entryEmail = "";
        }

        @Invariant("null !in this.entries[int]")
        public Entry[] entries;

        @Override
        public void setEmailAddress(String name, String email) {...}
        @Override
        public String getEmailAddress(String name) {...}
        @Override
        public boolean contains(String name) {...}
}
```

Listing 6.6: GCD specification

```
@Requires("m > 0 && n > 0")
@Ensures({"some x: int, y : int | x * m + y * n = return",
        "return > 0",
        "m % return = 0",
        "n % return = 0"})
int gcd(int m, int n) {
        if (m < n) {
                int t = m;
                m = n;
                n = t;
        }
        int r = m % n;
        if (r == 0)
                return n;
        else
                return gcd(n, r);
}
```

1. calling a method inside specifications: **this** . size () = 0;
2. fully-qualifying a field and a left join: lone CyclicList @ nodes . **this**;
3. computing a sum over a bag of integers: sum n in **this** .nodes | n. data;
4. disambiguation of the plus operators: **this** .nodes.data @+ k = @old(**this**.nodes.data).

Listing 6.7: CyclicList specification

```java
package edu.mit.csail.sdg.annotations.test;
import edu.mit.csail.sdg.annotations.*;

@SpecField("nodes : set Node from this.root | this.nodes = this.root.*next - null")
public class CyclicList {
        @Invariant({"~prev = next", "lone CyclicList @ nodes . this"})
        private static class Node {Node next; Node prev; int data;}

        @Invariant("this.root in this.root.*next + null")
        private Node root;

        @Ensures("no this.nodes")@Modifies("this.nodes")
        public void clear() {root = null;}

        @Returns("#this.nodes")
        public @Pure int size() {
                if (root == null) return 0;
                int count = 1;
                Node current = root.next;
                while (current != root) {count++; current = current.next;}
                return count;
        }

        @Returns("this.size() = 0")
        public @Pure boolean isEmpty() {return size() == 0;}

        // insert at the end
        @Ensures({"this.nodes.data = @old(this.nodes).data @+ k",
                "#this.nodes = @old(#this.nodes) + 1"})
        @Modifies("this.nodes, this.nodes.next, this.nodes.prev")
        public void insert(int k) {
                if (root == null) {
                        Node n = new Node();
                        n.data = k;
                        n.next = n.prev = n;
                        this.root = n;
                        return;
                }
                Node n = new Node();
                n.data = k;
                n.next = this.root;
                n.prev = this.root.prev;
                this.root.prev.next = n;
                this.root.prev = n;
                return;
        }

        // delete the first one if found
        @Requires("k in this.nodes.data")
        @Ensures({"return.data = k",
                "#this.nodes = @old(#this.nodes) - 1",
                "this.nodes.data @+ k = @old(this.nodes.data)"
        })
        @Specification(@Case(requires = "k !in this.nodes.data", ensures = "return = null"
            ))
        @Modifies("this.nodes, this.nodes.prev, this.nodes.next")
        public Node delete(int k) {
                ... // code omitted
        }

        @Returns("sum n in this.nodes | n.data")
        public @Pure int sum() {
                ... // code ommitted
        }
}
```

Chapter 7

Future Work

I have designed JFSL language and JForge tool to support automatic bounded verification of Java code. The tool has the potential to introduce light-weight formalism into development practice with little cost of annotating the programs. This work may be extended in various directions. Here are the few ideas that seem to be the most fruitful extensions of the system:

Other Languages Java was chosen for the source code language because of its popularity and better support for abstract data types. Java is a mainstream high-level object-oriented language, and therefore, our tool supports common features of these languages such as type hierarchy, dynamic dispatch and method overriding, interfaces, aliasing, casting, and exceptions.

Still, other popular languages, such as C, bring additional challenges that we did not face in developing specifications for Java. For example, Java does not have pointers and pointer arithmetic. To represent pointers, one would need to encode an extra layer of indirection into the relational representation. This would include a relation to map the pointer to its address location and the address location to the value stored at that address. Function pointers are another complication, but they could be handled by treating writing a separate specification for the function being point to and presuming the behavior of that function conforms to its specification.

Full Java Support Our tool does not support static initialization, real arithmetic, recursion, generics, and auto-boxing. This might limit the scope applicability of the tool to the real-world code. Embracing generics in translation to FIR may offer performance improvement by reducing the scope of the relations using parametric type information.

Larger Case Study A substantial case study is needed for evaluating the tool in practice. Annotating existing code body is challenging since it requires specifying all dependency classes from Java API such as collections framework as well as full understanding what the code actually does. Whether specifying the entire system is necessary for confidence in software, or only its most complex part, is also an interesting research question. It is also interesting to understand how helpful the tool is in the early stages of designing and prototyping software.

Counter Example Validation The traces produced by JForge may not always be real traces of bugs in the program due to over-approximation of program semantics. For example, integers typically have smaller bit width in JForge. However, these traces usually indicate real bugs in the program. One interesting direction is automatic generation of JUnit tests from counter example traces that would use reflection to build the initial heap and then dynamically execute method in it.

Squander Squander is an ongoing research project of using JFSL as the basis for agile specifications [17]. Squander offers a way to execute declarative JFSL specifications. There are many challenges in this direction, such as run-time evaluation of specifications fields, and generation of fresh instances.

Assertion Checking Java's assertions can be extended to support entire JFSL. JForge can then be used to automatically check assertions either dynamically during runtime, or statically in bounded verification.

Coverage Analysis Forge provides extensive coverage information when used with appropriate SAT engine. This information may be presented to the user using standard color-scheme for branch or statement coverage in code. The main challenge here is to construct a source-to-source translator that would map control flow graph statements in FIR to Java byte code statements and then to Java source code lines.

Bibliography

[1] Code Contracts. Accessible at: http://research.microsoft.com/en-us/projects/contracts/.

[2] Java 1.5 grammar for ANTLR v3. Accessible at: http://antlr.org/grammar/1152141644268/Java.g.

[3] JForge Eclipse Plug-in. Accessible at: http://sdg.csail.mit.edu/forge/plugin.html.

[4] Nearly All Binary Searches and Mergesorts Are Broken. Accessible at: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.

[5] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the ”small scope hypothesis”. Technical report, IN POPL ’02: PROCEEDINGS OF THE 29TH ACM SYMPOSIUM ON THE PRINCIPLES OF PROGRAMMING LANGUAGES, 2002.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, 2004.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS ’03), ser. Electronic Notes in Theoretical Computer Science, T. Arts and W. Fokkink, Eds.*, 80, 2003.

[8] A. Darvas and P. Muller. Reasoning about method calls in JML specifications. In *Formal Techniques for Java-like Programs*, 2005.

[9] G. Dennis, K. Yessenov, and D. Jackson. Bounded Verification of Voting Software. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145. Springer, 2008.

[10] Greg Dennis. A relational framework for bounded program verification. PhD thesis in preparation., 2009.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[12] CAR Hoare. Proof of correctness of data representations. *Acta informatica*, 1(4):271–281, 1972.

[13] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000.

[15] K.R.M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM New York, NY, USA, 1998.

[16] C.E. Leiserson, R.L. Rivest, T.H. Cormen, and C. Stein. *Introduction to algorithms*. MIT press, 2001.

[17] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. Submitted for publication., 2009.

[18] Robby and Patrice Chalin. Preliminary design of a unified JML representation and software infrastructure. Technical Report SAnToS-TR2009-04-01, 2009.

[19] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Lecture Notes in Computer Science*, 4424:632, 2007.

[20] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999.