Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations

Ghaith Haddad and Gary T. Leavens

CS-TR-08-05
April 2008

Keywords: Formal method tools, runtime assertion checking, dynamic analysis, specification languages, JML, JastAdd, attribute grammar tool, extensible compiler.

2008 CR Categories:
D.2.2 [*Software Engineering*] Design Tools and Techniques — computer aided software engineering; D.2.4 [*Software Engineering*] Software/Program Verification — formal methods, programming by contract; D.3.4 [*Programming Languages*] Processors — compilers, parsing, translator writing systems and compiler generators.

Submitted for publication.

School of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations

Ghaith Haddad and Gary T. Leavens
College of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida, 32816
haddad@ieee.org,leavens@eecs.ucf.edu

ABSTRACT

Modern programming languages, such as Java, are large and complex, as are practically useful behavioral interface specification languages that extend them, such as JML. Their size and complexity make it difficult for researchers to build dynamic analysis tools, such as runtime assertion checkers. Researchers wishing to experiment with a small change to a specification language or a dynamic analysis face a prohibitive amount of work before they can run experiments. Even if a research prototype is built, it is difficult to keep it current with rapidly evolving languages, since the changes to the underlying compiler infrastructure are not easily separated from that infrastructure. However, extensible dynamic analysis tools can be written more easily with modern attribute grammar tools, in particular with JastAdd. We describe a small case study that shows how using JastAdd eases development of a runtime assertion checker for the specification language JML.

1. INTRODUCTION

Runtime assertion checking is an important and practical technique for improving the quality of programs. However, building a runtime assertion checker for a large practical language, such as Java [15] is a significant effort. Writing a runtime assertion checker for Java involves building or understanding and reusing a Java compiler, and then extending that compiler with features of the formal specification language used to specify what to check dynamically. Even using Java 5 annotations does not relieve much of this burden [18], as one must still compile the strings that must be used in Java 5 annotations to write assertions. Such problems afflict many efforts in formal methods, such as those envisioned for the verified software initiative [16].

The Java Modeling Language (JML) project is an example of such a formal methods project; it has experienced all these difficulties in the past several years. Although we believe the lessons we draw are general, we focus on JML because it is well-known and we are intimately familiar with it.

One of the most important of the many tools that are available for JML is its runtime assertion checker (jmlc) [6]. This tool was built as an extension of the MultiJava compiler [7]. The MultiJava compiler is itself an extension of the Kopi Java compiler [1]. This architecture allowed the JML project to reuse the Kopi compiler's support for the features of Java 1.4, including inner classes, which a previous home-grown checker did not support adequately.

However, since the Kopi and MultiJava compilers are not being maintained, the JML project has been struggling to support the features of Java 5, especially generics. Even support for Java 5's enhanced **`for`** loop is not currently available in the current release of the "Common JML Tools," also known as JML2. A major difficultly is that the size and complexity of the JML2 project, together with an architecture that is not designed for easy extension, makes it difficult for researchers to understand the JML2 software well enough to easily extend it.

One direction for overcoming these problems has been launched by Patrice Chalin and his group at Concordia University [4]. This effort, called JML4, is based on the Eclipse Java Development Tools (JDT) [13]. It builds support for JML, including runtime assertion checking, on the Eclipse JDT, using the JDT in much the same way that JML2 used the MultiJava compiler. There are two significant advantages to this architecture: (1) the Eclipse JDT is actively maintained, (2) JDT's existing integration with Eclipse eases integration of the JML runtime assertion checker into Eclipse [4].

On the other hand, our experience with the writing small parts of code for the JML4 effort has also shown that this architecture has some significant drawbacks for researchers. A major problem is that the code base of the JDT is quite large, which still leaves a significant hurdle for researchers who want to build and experiment with small extensions to JML and its runtime assertion checker. Furthermore, the JDT is not designed to be extensible, so integration of JML4 requires a fairly large amount of boilerplate code, especially to extend the parser. The JDT also aims for execution speed, not only for the compiled code, but also for parsing and static analysis (such as type checking). While compilation speed is important in an IDE such as Eclipse, it is less important for a research prototype. Thus extra coding overheads that attempt to squeeze out a bit more speed tend to complicate life for researchers hoping to make small extensions. Finally, once a researcher adds a new feature to JML4, it is not necessarily easy to combine that with other new features, as the JML4 architecture does not support extensions of extensions or combinations of extensions.

In this paper we describe a different approach to building an extensible runtime assertion checker for JML, based on the JastAdd attribute grammar tool [12, 9], and hence called JAJML. The developers of JastAdd have provided an extensible Java compiler, upon which we are building JAJML. As we will describe below, using JastAdd to build the runtime assertion checker solves the main problems described above. In particular the JastAdd extensible Java compiler is about $1/4$ the size of the Eclipse JDT: the JastAdd compiler is only 21K lines of code, while the Eclipse JDT compiler is 83K lines of code [11]. Moreover, extensions built using JastAdd are themselves extensible, due to JastAdd's declarative and modular architecture. The disadvantages of using JastAdd are that the resulting checker is harder to integrate into Eclipse, and support for future enhancements to Java is less sure. On the other hand, there are existing tools for building IDEs for Eclipse that can aleviate the first disadvantage (such as Eclipse IMP [14]). Also, JastAdd's ease of extensibility makes the second disadvantage less of a problem than with a more monolithic compiler. For example, the JastAdd developers developed their Java 5 compiler as an extension to their Java 1.4 compiler, in only 6K lines of code [11].

In the remainder of this paper we describe a case study of adding loop annotations (loop invariants and variant functions) to our JAJML prototype (Section 2). To allow a comparison with JML4, we also explain how JML4 supports these same features and give a detailed comparison (Section 3).

2. LOOP ANNOTATION CASE STUDY

In this section, we begin our case study where we add loop annotations to JAJML using JastAdd. We first give a brief overview of JastAdd and then discuss the various tasks needed to make an extension: scanning, abstract syntax tree declaration, parsing, static analysis, and code generation. Code for our case study is available from `sourceforge.net`.

The JastAdd Compiler Construction System [10, 11, 12] extends Java with rewritable circular reference attribute grammars. In addition, JastAdd has other mechanisms like static aspect-oriented programming, declarative attributes, and context-dependent rewrites, that support the tool's modularity and extensibility. These techniques make writing extensible compilers more efficient. In this section, we describe the features of JastAdd and its extensible Java compiler that we use in the remainder of this paper.

In the following we say "JAJC" for the JastAdd extensible Java compiler and "JastAdd" for the JastAdd compiler construction tool.

2.1 Scanning

JAJC uses Flex for lexical analysis. Thus, in order to extend the JAJC, we also found it necessary to use Flex.

The scanner for JAJC is split into several files. Each file contains part of the overall Java lexical grammar, organized to ease extension. For example, the lexical grammar for comments is found in a file `Comments.flex`.

Flex has a feature that is very valuable for JML, namely the ability to use states to control the lexical analysis. JML specifications are contained within special annotation comments of the form `/*@ ... @*/` or from `//@` to the end of a line. Thus the lexical grammar should use states to make sure it only recognizes JML keywords within such assertions [17, Section 4]. (Our prototype lexical analysis needs more work to accomodate all of JML's lexical details.)

2.2 The Abstract Grammar

JastAdd supports a concise "abstract grammar" for declaring abstract syntax trees (ASTs). All classes that compose the AST are defined in abstract grammar files. JastAdd automatically generates Java classes to represent these ASTs.

JAJC uses several abstract grammar files to define ASTs for Java. (There is one file for the Java 1.4 features and a file for each new feature added in Java 5.) The code below is an excerpt from the JAJC abstract grammar files that defines the ASTs for loop statements.

```
WhileStmt : BranchTargetStmt ::=
   Condition:Expr Stmt;
DoStmt : BranchTargetStmt ::=
   Stmt Condition:Expr;
ForStmt : BranchTargetStmt ::=
   InitStmt:Stmt* [Condition:Expr]
   UpdateStmt:Stmt* Stmt;
```

JastAdd makes extending the hierarchy with new AST classes easy. This is done by adding new rules to the abstract grammar.

Our added abstract grammar file describes the ASTs for JML's loop annotation statements. The JAJC already has several AST classes. We start by declaring two AST classes that will serve as superclasses for other ASTs and will help organize them. The following is the entire contents of the file `JAJML.ast`.

```
abstract JmlAnnotation: Stmt;
abstract JmlHelper;
```

The abstract AST class `JmlAnnotation` inherits from `Stmt`, which is a part of the original JAJC AST hierarchy. `JmlHelper` is just a class used to implement static helper methods used in implementation. For each feature we add a file that declares the ASTs for that feature. Loop annotation ASTs are declared in the file `possibly_annotated_loop.ast`, which contains the following declarations.

```
abstract JmlLoopStmt: Stmt ::=
    JmlLoopInvariant
    JmlLoopVariant
    LoopStmt: BranchTargetStmt;
JmlWhileStmt: JmlLoopStmt;
JmlForStmt: JmlLoopStmt;
JmlDoStmt: JmlLoopStmt;
abstract JmlLoopAnnotation:
    JmlAnnotation ::= Expr*;
JmlLoopInvariant: JmlLoopAnnotation;
JmlLoopVariant: JmlLoopAnnotation;
```

The AST class `JmlLoopStmt` is also a subclass of `Stmt`. The abstract grammar above says that a `JmlLoopStmt` contains three nodes: a `JmlLoopInvariant`, a `JmlLoopVariant`, and a third, named `LoopStmt` that is of type `BranchTargetStmt`.

The AST type `BranchTargetStmt` is an AST node type from JAJC. This AST node type is the superclass of all JAJC loop statements. Similarly, we use the abstract `JmlLoopStmt` as a superclass of the AST node types for while loops, for loops and do-while loops. We did not implement JML annotations for labeled statements and the enhanced-for statement in this prototype.

We also use an abstract node class `JmlLoopAnnotation`, which is itself a subclass of the `JmlAnnotation` class, as the subclass of loop invariant and variant function nodes. Each node of type `JmlLoopAnnotation` holds a list of expressions, which is inherited by both loop invariant ASTs and loop variant ASTs.

2.3 Parsing

Once we have added all types needed for parsing, we are ready to write the parsing rules.

Our runtime checker uses the Beaver LALR(1) parser generator [8] because it is used by JAJC for parsing. Beaver accepts grammars expressed in EBNF and is well-integrated into JastAdd. This integration makes it easy to use Beaver's AST nodes, which are represented as instances of generated Java classes when manipulating attributes from within JastAdd.

The parser for JAJC is not split over several files, but Beaver allows new context-free rules to be added to the grammar. This makes it easy to extend the grammar simply by adding new productions in separate files. For example, we can add productions for annotated loop statements by adding productions for the statement nonterminal as shown in Figure 1. This grammar follows the one in the *JML Reference Manual* [17, Section 12.2]. We put the grammar for each such feature, in a separate file.

In Beaver semantic actions are enclosed in `{:` and `:}` brackets. Each rule is responsible for creating and returning the corresponding AST node. Each AST node type is defined in the abstract grammar files as described above. The AST for a nonterminal N is referred to by the notation $N.g$ in the grammar. For example, in Figure 1 the AST for the `while_statement` nonterminal is named `w` in the first production of Figure 1. These names are thus made accessible in the semantic actions, and this allows child AST nodes to be used in constructing larger ASTs from these children.

2.4 The Attribute Grammar

After parsing, and before applying rewrites and tree transformations, JastAdd compilers perform type checking and other static analysis. This allows messages about any problems in the user's program to be generated from unmodified ASTs that directly reflect the user's input and are thus easy for users to understand.

Type checking and other kinds of static analysis are performed using JastAdd's attribute grammar facilities.

Attribute grammars attach attributes to ASTs. Attributes can be defined in either declarative or imperative style. While imperative style allows for writing regular Java code to manipulate the nodes, declarative style simply specifies the relations that compute each attribute. Attributes are evaluated in an order determined automatically by JastAdd. Imperative style attribute definitions are executed when their values are needed by the declarative style attributed definitions, which control the overall order of execution. Imperative style attribute definitions are written in Java code, and are useful when making complex decisions, for example in transfomation code.

Attributes are either inherited or synthesized. Synthesized attributes propagate information upwards in the AST (towards the root), while inherited attributes propagate information downwards.[1] Inherited

[1]Inheritance in attributed grammars is different than inheritance in object-oriented programming.

```
JMLWhileStmt jml_while_statement =
   jml_loop_statement_invariant_header.j
   while_statement.w
   {: return new JMLWhileStmt
      (new JMLLoopInvariant(j),
       new JMLLoopVariant(), w
      );
   :}
 | jml_loop_statement_variant_header.j
   while_statement.w
   {: return new JMLWhileStmt
     (new JMLLoopInvariant(),
      new JMLLoopVariant(j),w
     );
   :}
 | jml_loop_statement_invariant_header.i
   jml_loop_statement_variant_header.j
   while_statement.w
   {: return new JMLWhileStmt(
      new JMLLoopInvariant(i),
      new JMLLoopVariant(j),w
      );
   :}
        ;

Stmt statement =
  jml_while_statement.w {: return w; :}
  | jml_for_statement.w {: return w; :}
  ;

List jml_loop_statement_invariant_header :=
   JMLMAINTAINING expression.e SEMICOLON
      {: return new List().add(e); :}
 | jml_loop_statement_invariant_header.l
   JMLMAINTAINING expression.e SEMICOLON
      {: return l.add(e); :}
  ;

List jml_loop_statement_variant_header :=
   JMLDECREASING expression.e SEMICOLON
      {: return new List().add(e); :}
 | jml_loop_statement_variant_header.l
   JMLDECREASING expression.e SEMICOLON
      {: return l.add(e); :}
  ;
```

Figure 1: Beaver grammar for possibly annotated loop statements, from file `possibly_annotated_loop.parser`.

attributes are mainly used to make child nodes aware of information from their parent nodes.

Attributes are implemented inside JastAdd aspects. Inside an aspect, one can define new attributes and equations that are added to the different ASTs. This approach supports modularity as each feature is implemented in a single aspect. Each aspect can affect many different types of ASTs, which supports separation of concerns.

2.4.1 Standard Java Analyses

There are several Java analyses that must be implemented in a compiler based on the JAJC, in particular Java's definite assignment and definite unassignment analyses [2, Chapter 16]. These are implemented in JAJC with synthesized attributes that are defined in abstract classes. Thus they must be implemented by AST classes such as `JmlLoopStmt`. In JAJML this is done as follows in the file `possibly_annotated_loop.jrag`.

```
eq JmlLoopStmt.isDAafter(Variable v) =
   getLoopStmt().isDAafter(v);
```

This code simply tells the attribute evaluator that the definite assignment status for a variable `v` is determined by the `LoopStmt` node of the `JMLLoopStmt`. Several other attributes needed to be defined in this manner due to pre-existing analyses from the JAJC. Examples are shown in the following table:

analysis	attribute
assignment checking	`isDAbefore(Variable)`
definite unassignment	`isDUafter(Variable)`
unreachable statements	`canCompleteNormally()`
variable lookup	`lookupVariable(String)`
name classification	`nameType()`
type checking	`typeCheck()`).

For example, all the code for type checking in our case study is contained in Figure 2.

```
public void JmlLoopInvariant.typeCheck() {
   for (Expr exp:getExprs()) {
     TypeDecl cond = exp.type();
     if (!cond.isBoolean()) {
        error("the type of \"" + exp + "\" is "
              + cond.name()
              + ", but should be boolean");
     }
   }
}

public void JmlLoopVariant.typeCheck() {
  for (Expr exp:getExprs()) {
    TypeDecl cond = exp.type();
    if (!cond.isLong() && !cond.isInt()) {
      error("the type of \"" + exp + "\" is "
            + cond.name()
            + ", but should be int or long");
    }
  }
}
```

Figure 2: Code from `possibly_annotated_loop.jrag`, for type checking.

We also added several attributes to extract parts of the JML loop statement ASTs, but all these are trivial.

2.5 Compiling Runtime Assertion Checks

In this section we describe how our prototype uses the facilities of JastAdd to compile runtime assertion checking code for JML loop annotations.[2] Annotations in JAJML are implemented by transforming the AST to include the original code weaved with assertion checking code. Assertion checking code throws JML-specific error objects when an assertion fails.

2.5.1 Plan for Checking

JAJML checks assertions derived from both loop invariants and variant functions. A loop invariant must hold before and after each iteration of the loop. A variant function gives a loop progress metric that must decrease after each iteration of the loop. The variant

[2]Similar considerations would apply for such tasks as verification condition generation.

```
//@ maintaining Inv_1; ... maintaining Inv_n;
//@ decreasing E_1; ... decreasing E_n;
while (B) { S }
```

Figure 3: A general form of the annotated while loop

```
{  boolean fir$tTime = true;
   long variant$Var1 = 0, ..., variant$Varn = 0;
   while(true) {
      if (!(Inv_1) || ... || !(Inv_n)) {
         throw new JMLLoopInvariantError(); }
      if (fir$tTime) fir$tTime = false;
      else {
         if (!(variant$Var1 >= 0) || ...
            || !(variant$Varn >= 0)
            || !(variant$Var1 < E_1) || ...
            || !(variant$Varn < E_n)) {
           throw new JMLLoopVariantError();
         }
      }
      variant$Var1 = E_1; ... variant$Varn = E_n;
      if (!(B)) break;
      S
   }
}
```

Figure 4: A transformed form of the annotated while loop

function is an expression of type long (or int); it must also be no less than 0 when the loop is executing.

We consider a loop iteration to include the loop test and any increment statements that are implicitly executed at the end of a for loop. Exactly what constitutes a loop iteration is important for verification of Java or similar languages (such as C, C++, and C#). To explain this precisely, we specify how the annotated JML source code is transformed into Java with runtime assertion checks. We start by writing a general form of an annotated while loop. Code of the form shown in Figure 3, can be transformed to the form shown in Figure 4, where the names `fir$tTime`, and `variant$Var1`, ..., `variant$Var`n are all fresh.

Moving the loop condition, B, into the body of the loop effectively treats any side-effects in B as part of the loop iteration.[3] These may even be useful in making the variant function decrease, hence the loop variant is tested at the top of the translated loop, effectively at the end of each loop body. Note that a **`continue`** statement in the body S will thus immediately check the invariant and the variant function assertions, since the loop iteration has just ended. However a **`break`** in S will simply exit the loop, without performing any additional checks. In particular the invariant is not checked when a **`break`** is executed.

Our prototype expresses this kind of transformation directly at the level of abstract syntax trees in JastAdd. The prototype also transforms **`for`** and **`do`** loops in a similar way.

2.5.2 Transformation Using JastAdd

For tree transformation we used JastAdd's `transformation()` function. We preferred this to JastAdd's `Rewrite` construct, because transformations are done after the static analysis steps de-

[3]However, there can be no side effects in the invariant or variant function expressions in JML, since they must be pure.

```
public void JmlLoopStmt.transformation() {
   super.transformation(); // (1)
   List<Stmt> blklist = new List<Stmt>();
   Block replaced = new Block(blklist);
   LinkedHashMap<Integer, Expr>  variantExprs =
            getNumberedVariantExpressions();
   LinkedHashMap<Integer, Stmt> variantDecls =
            addVariableDeclarationsForVariants(
               variantExprs,blklist);
   VariableDeclaration vdFirstTime =  // (2)
            JmlHelper.createBooleanVarDecl(
                  "fir$tTime","true");
   blklist.add(vdFirstTime);
   List<Stmt> LoopStmtblklist =  // (3)
      new List<Stmt>();
   addIfNotFirstTimeStmt(          // (4a)
      variantDecls,variantExprs,
      vdFirstTime,LoopStmtblklist);
   addLoopInvariantChecks(LoopStmtblklist);// (4b)
   addLoopVariantUpdates(variantDecls, // (4c)
      variantExprs,LoopStmtblklist);
   // (4d) follows
   addConditionCheckAndUpdate(LoopStmtblklist);
   // (4 & 4e) follows
   blklist.add(getNewLoopStmt(LoopStmtblklist));
   replace(this).with(new Block(blklist)); // (5)
}
```

Figure 5: Transformation for runtime checking of loop assertions.

scribed above, and thus as we described above, generally leads to better error message generation. This transformation function is implemented for `JmlLoopStmt`, which allows its functionality to be inherited by all three kinds of loops.

The transformation function is shown in Figure 5. It has the following steps also indicated in the code.

1. Apply transformations bottom by first visiting the children.
2. Add a declaration for a **`long`** variable for each variant function introduced in the loop annotation.
3. Define the `fir$tTime` boolean variable to check if loop is executed for the first time.
4. Add a loop statement with condition **`true`**, the type of the loop, whether its a `while`, `for`, or `do` loop, is similar to the type of the original loop statement. For simplicity, we convert do statements into while statements in our implementation. The body of the loop statement includes:
   (a) An if statement that flips the FirstTime condition if true, and checks all variant functions otherwise,
   (b) Assert statements for all invariants in the loop,
   (c) Statements to update all variant variables with their current value calculated from the variant functions,
   (d) An if statement to break the loop if the break condition is satisfied,
   (e) The body of the original loop, and
   (f) If this is a do-while loop, then the body and break in the last two steps are switched.
5. Replace the generated block with the original JMLLoopStmt

3. COMPARISON WITH ALTERNATIVES

The JML community has been working for some time to revise their tools and provide support for both Java 5's features and better integrated development (IDE) support. As with the current generation of JML tools [3], it is very helpful to build on an existing Java compiler. The main alternatives to using JAJC are to build the JML tools on the Open JDK or Eclipse compiler.

David Cok has done some preliminary work on the Open JDK compiler,[4] which has advantages because the compiler is kept up to date by Sun and is relatively small. However, it is not designed to be extensible and there is little direct support for IDE integration.

The alternative currently favored for JML's further development is JML4. The JML4 effort was pioneered by Patrice Chalin and his group [5, 4]. It is based on the Eclipse Java development tools (JDT) compiler, maintained by the Eclipse foundation. The tight integration of the JDT compiler with Eclipse is a major advantage of the approach, compared with our approach.

However, the main disadvantage of the JML4 approach is that the JDT compiler is not designed to be extensible. In the remainder of this section we compare the extensibility of the JML4 approach with our approach using JastAdd. For this comparison we use the case study of loop annotations described above. This case study was developed as an exercise for the JML Winter School by Patrice Chalin and Perry James.[5] Besides being a tutorial example, it has been fully implemented in JML4.

3.1 JML4 Overview

At the package level, JML4 is a customization of some Eclipse packages that add JML support to the scanner, parser, code generator, and UI of the Eclipse JDT compiler. At this level, our implementation in JastAdd is very similar, because both approaches extend an existing (Java 5) compiler. However, at this level of abstraction, one should note that the implementation of JML features is introduced in distinct places in those two approaches. In JML4, features are implemented at the bytecode generation step, while in JAJML, features are implemented right before the bytecode generation step by AST transformation. This generally results in less and simpler (and thus perhaps more reliable) code in our approach.

3.2 Implementing Loop Annotations in JML4

We now explain how the loop annotations case study works in more detail in JML4, relying on the work of Chalin and James.

3.2.1 Scanning and Parsing

Scanning involves adding new keywords to the scanner's Java file, `Scanner.java` in the `internal.compiler.parser` package of `org.eclipse.jdt`. This hand-written scanner has over 4000 lines of code. One adds keywords by adding lines of the form

```
m.put("maintaining",
      new Integer(TokenNameloop_invariant));
      //$NON-NLS-1$
```

to the static initializer for `ML_KEYWORD_TO_TOKEN_ID_MAP`, which is a Java `Map`.

[4]Email message on the jmlspecs-reloaded mailing list from December 11th, 2007, see `http://tinyurl.com/3r5qbo`.
[5]See `http://tinyurl.com/3ogtxo`.

```
private void consumeLoopInvariant() {
  String lexeme
    = new String(this.identifierStack[
                      this.identifierPtr]);
  long pos
    = this.identifierPositionStack[
                      this.identifierPtr--];
  this.identifierLengthPtr--;
  Expression exp
    = this.expressionStack[this.expressionPtr--];
  this.expressionLengthPtr--;
  JmlLoopInvariant invariant
    = new JmlLoopInvariant(lexeme,
          this.jmlKeywordHasRedundantSuffix, exp);
  invariant.sourceStart = (int)(pos >>> 32);
  invariant.sourceEnd = this.endStatementPosition;
    this.pushOnAstStack(invariant);
}
```

Figure 6: The semantic action `consumeLoopInvariant`.

The parser is generated from a grammar file named `java.g` in `org.eclipse.jdt.core.grammar`. It is written in the language of the Jikes Parser Generator (JikesPG). This file is more than 3000 lines long. Productions are added by editing this file (using tags to make finding the edits easier) along with the suitable function call to perform the needed semantic action. For example, the following is a small example of a loop invariant clause written for the JikesPG (omitting comments and tags).

```
LoopInvariant ::= MaintainingKeyword Predicate
                    ExitJmlClause ';'
/.$putCase consumeLoopInvariant() ; $break ./
/:$readableName LoopInvariant:/

ExitJmlClause ::= $empty
/.$putCase consumeExitJmlClause(); $break ./
/:$readableName ExitJmlClause:/

MaintainingKeyword -> 'loop_invariant'
/:$readableName MaintainingKeyword:/
```

The reader will notice a fair bit of "boilerplate" in the grammar file. The nonterminal `ExitJmlClause` is used to have a semantic action at the end of each clause. The lines starting `/.$putCase` and ending with `; $break ./` enclose semantic actions, which are calls to methods in the file `Parser.java`, found in the package `org.eclipse.jdt.internal.compiler.parser`. The lines `/:$readableName ... :/` give a readable name to the production, used in error messages and in debugging the grammar. There is also a need to add separate entries for terminal symbols in an early section of `java.g`.

All semantic actions must be hand written in methods in the file `Parser.java`. This file is currently close to 15000 lines of code. Semantic actions are accomplished by writing code to manipulate the seven stacks that the parser maintains. For example, Figure 6 shows the semantic action `consumeLoopInvariant`, without a 15 line comment that illustrates the action of the code on the stacks. It seems fair to say that writing such semantic actions is somewhat tedious and error prone.

3.2.2 Abstract Syntax Trees and Analysis

ASTs for new productions are built manually by adding new classes to the `org.jmlspecs.jml4.ast` package. Constructors and relationships between ASTs must be hand-coded.

```
public void resolve(BlockScope scope) {
   if (hasPred()) {
      TypeBinding type =
         pred.resolveTypeExpecting(
            scope, TypeBinding.BOOLEAN);
         pred.computeConversion(
            scope, type, type);
   }
}
```

Figure 7: The JML4 type-checking action `resolve` in the superclass `JmlClause`.

```
public void resolve(BlockScope scope) {
   TypeBinding type =
      this.expr.resolveTypeExpecting(
         scope, TypeBinding.LONG);
   this.expr.computeConversion(
         scope, type, type);
   createLocalForStore(scope);
   scope.addLocalVariable(
         this.variantLocal.binding);
   this.variantLocal.binding
      .recordInitializationStartPC(0);
}
```

Figure 8: The JML4 type-checking action `resolve` in the subclass `JmlLoopVariant`.

Other methods for type-checking and static analysis are handled either automatically, by subclassing existing AST classes that implement static analysis methods, or manually, by overriding these methods in the added AST subclasses. For example, the expression `JmlLoopInvariant` must be of type **`boolean`**, so it is subclassed from `JmlClause`. The type-checking is implemented in the superclass `JmlClause` using the function `resolve` and is shown in Figure 7. However, in `JmlLoopVariant`, the expression must be of type **`long`**, so the function `resolve` is overridden as shown in Figure 8.[6]

3.2.3 Code generation

Code generation is implemented by directly generating bytecode in a method of each AST node class. For our running example, this involves modifying the method `generateCode` in the AST class `WhileStatement`. These modifications are needed to add hooks where bytecode needs to be inserted to do runtime checking. These hooks emit bytecode that creates auxiliary variables and evaluates expressions in way that is very similar to Figure 4.

3.3 Comparing JAJML and JML4

Overall there are several advantages that emerge from the extensibility features of JastAdd and the JAJC. In terms of scanning, parsing, and AST definition, JastAdd makes the work in JAJML significantly less tedious and less error prone than that in JML4's modification of the JDT. In particular automatically generating AST node classes is significantly less work than writing them by hand. The JikesPG that is used in the JDT compiler is a significant headache, both in terms of the amount of tedious boilerplate it requires and

[6]We modified this code slightly to use **`long`** instead of **`int`** to reflect recent changes in the *JML Reference Manual*.

in terms of the tedious and error prone manipulation of the seven stacks in semantic actions. By contrast, Beaver seems less tedious and has semantic actions that are shorter and more automatic.

Even more automation is provided by JastAdd's support for attribute grammars, which can automate many of the tasks that would otherwise be hand coded as visitors in JML4. The use of attribute grammars also allows the JAJC to be more easily extensible, and the JAJC is designed as an extensible compiler framework.

Another advantage is the use of transformations instead of direct generation of bytecode for runtime assertion checking. This is more automatic and less error prone than direct bytecode generation.

A crude measure of the automation given by JastAdd in obtained by counting noncommentary lines of code. For our loop annotation case study, implementing the same feature and the same functionality in JML4 and JAJML required less than 100 lines of code. This includes defining the AST nodes, writing the rules for the parser generator, and implementing the attributes. We ignored counting the lines of code that are used from the JmlHelper as these functions are used in many features. However, the total implementation for JmlHelper class is about 55 lines of code. For the same case study, JML4 require about 600 lines of code.

4. CONCLUSION

In this paper we used JastAdd to build an extensible runtime assertion checker for JML. The JastAdd extensible Java compiler provided an extensible framework for our case study, which was how to add runtime assertion checks for loop annotations.

We also compared JAJML and JML4 on our case study. We concluded that there are both advantages and disadvantages to building a runtime assertion checker with JastAdd. The main disadvantages of JAJML are the lack of tight IDE integration with Eclipse and relatively lower guarantee of support for future versions of Java. However, the simplicity and extensibility of building JAJML with JastAdd says a lot for our approach. In summary, JAJML needs less code to be implemented, and is easier to extend.

Our case study also shows that the use of modern attribute grammar tools may be advantageous for quickly building dynamic analysis tools that are easily extensible. For example, we were able to easily extend our case study by adding support for pre- and postconditions, as well as JML's **`ghost`** fields and **`set`** statements.

Future work is implementing the rest of JML, and checking how easy it is for researchers to experiment with new features.

Acknowledgments

The work of both authors was supported in part by the National Science Foundation under grant CNS 08-08913. Thanks to Perry James and Patrice Chalin for comments on an earlier draft. Thanks to Torbjörn Ekman for help with JastAdd.

5. REFERENCES

[1] Kopi project home page. `http://www.dms.at/kopi`, 2004.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, 2000.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[4] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: Architecture and early results. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47–53. ACM, Sept. 2007.

[5] P. Chalin, P. R. James, and G. Karabotsos. The Architecture of JML4, a Proposed Integrated Verification Environment for JML. Technical Report 297, Concordia University, May 2007.

[6] Y. Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, Apr. 2003. The author's Ph.D. dissertation.

[7] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. The author's masters thesis.

[8] A. Demenchuk. Beaver - a lalr parser generator. http://beaver.sourceforge.net/index.html, 2006.

[9] T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 2006. Dissertation 25.

[10] T. Ekman. *Extensible Compiler Construction*. PhD in Computer Sience, Lund University, Department of Computer Science, Lund University, Box 118, SE-221 00 Lund, Sweden, 2006. ISBN 91-628-6839-X.

[11] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[12] T. Ekman and G. Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Programming*, 69(1-3):14–26, 2007.

[13] E. Foundation. The Eclipse Project. `http://www.eclipse.org/`.

[14] R. M. Fuhrer et al. Eclipse IDE Meta-tooling Platform. `http://eclipse-imp.sourceforge.net/`.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[16] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*, page 78. Springer-Verlag, Jan. 2005.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Available from `http://www.jmlspecs.org`, Oct. 2007.

[18] K. B. Taylor. A Specification Language Design for the Java Modeling Language (JML) Using Java 5 Annotations. Master's thesis, Iowa State University, 2008. To appear.