# CJC: An Extensible Checker for the CleanJava Annotation Language

Cesar Yeep and Yoonsik Cheon

TR #13-29
May 2013

**Keywords:** Formal annotation language, parser, static checker, CleanJava, JastAddJ.

**1998 CR Categories:** D.3.2 [*Programming Languages*] Language classifications—Object-oriented languages; D.3.4 [*Programming Languages*] Processors—Compiler, parsing; D.2.4 [*Software Engineering*] Software/Program Verification—assertion checkers, class invariants, formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

# CJC: An Extensible Checker for the CleanJava Annotation Language

Cesar Yeep and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ceyeep@miners.utep.edu, ycheon@utep.edu

***Abstract*— CleanJava is a formal annotation language for the Java programming language to support a Cleanroom-style functional program verification technique that views programs as mathematical functions. It needs a suite of support tools including a checker that can parse annotations and check them for syntactic and static semantic correctness. The two key requirements of the checker are flexibility and extensibility. Since the language is still under development and refinement, it should be flexible to facilitate language experimentation and accommodate language changes. It should be also extensible to provide base code for developing more advanced support tools like an automated theorem prover. In addition, it should recognize Java syntax, as CleanJava is a superset of Java. In this paper we describe our experience of developing a CleanJava checker called CJC and explain how we met the above requirements by using an open-source Java compiler. We expect our techniques and the lessons that we learned be useful to others implementing Java-like languages.**

***Keywords—formal annotation language; parser; static checker; CleanJava; JastAddJ***

## I. Introduction

Formal program verification is a complementary technique to program testing. One such a technique is *functional program verification* originated from the Cleanroom Software Engineering [1] that emphasizes defect prevention rather than defect removal. In functional program verification, a program is viewed as a mathematical function from one program state to another, and the program is verified by comparing two functions, the implemented and the expected behaviors. CleanJava is a formal notation for Java to support a functional program verification technique. It allows one to annotate Java code with specifications written using mathematical functions [2]. The specifications called *intended functions* are written in an extended form of Java expression syntax (see II.A). CleanJava complements informal program comments like Javadoc comments and promotes the use of more formal comments for rigorous and formal correctness reasoning of Java programs.

Just like a programming language, a formal specification language such as CleanJava also needs a set of support tools. A wide spectrum of support tools is possiblecfrom simple parsers and static checkers to fully automated theorem provers. At minimum, one needs a tool that can parse specifications and perform static checking like syntax and type checking on the parsed specifications. However, developing such a tool for CleanJava poses several interesting challenges. Since the CleanJava language is still under development and refinement, the tool should be sufficiently flexible and extensible to facilitate experiments of new language features and to easily support future language extensions. Since we also envision the tool as a base platform for developing more advanced tools like proof assistants, extensibility is another important requirement. Another challenge is that, besides processing CleanJava-specific annotations, the tool also has to understand and process Java code because CleanJava is a superset of Java. This introduces issues and problems such as name spaces and context switching, as well as an interesting opportunity—the tool could be a drop-in replacement for a Java compiler and it will definitely help the adoption and use of CleanJava.

In this paper we describe how we addressed these challenges in developing the CJC tool, an extensible parser and static checker for CleanJava. One key to our approach is adapting an existing Java compiler that was built with an extension in mind. In particular, we used JastAddJ as our base code both to support extensibility and to avoid building yet another Java compiler. JastAddJ is an extensible Java compiler allowing one to create an extension to the Java language in a modular composition fashion [3]. JastAddJ itself was built using JastAdd, a meta-compilation system that provides support for creating modular and extensible compilers [4]. Another key to our approach is that we identified CleanJava language features that are likely to be changed or extended in the future and then provided a built-in extension mechanism for them. The current implementation of the CJC tool supports most of the CleanJava core language constructs and can be easily extended to support other CleanJava features or to be a drop-in replacement for a Java compiler such as javac.

The rest of this paper is structured as follows. Section II provides a quick overview of the CleanJava language and the JastAddJ Java compiler. Section III describes the problems and challenges of developing a CleanJava checker. Section IV explains our approach for developing a modular and extensible CleanJava checker called CJC. Section V provides a quick evaluation of the CJC tool. Section VI mentions related work, and Section VII concludes this paper.

## II. Background

### A. CleanJava

CleanJava is a formal annotation language for the Java programming language to support Cleanroom-style functional program verification [2]. In functional program verification, a program execution is modeled as a mathematical function from one program state to another. Each program state is a mapping from state variables to their values. These functions are described using a notation called a *concurrent assignment*. A concurrent assignment states changes in a program state and can express both the actual function implemented by a section of code called a *code function* and the intention of the code called an *intended function* [5]. In CleanJava an intended function is written using an extended form of Java expressions. However, the expressions should be side-effect free, and thus operators like ++ and += are not allowed. The following shows a Java code snippet annotated in CleanJava.

```
/*@ [str != null ->
  @  result := str->select(char c; c == ch)->size()] @*/
public static int numOfOccurrence(String str, char ch) {
 //@ [r, i := 0, 0]
 int r = 0;
 int i = 0;

 /*@ [str != null -> r, i :=
   @  r + str.substring(i)->select(char c; c == ch)->size(),
   @  anything] *@/
 while (i < str.length()) {
  //@ [r, i := r + (s.charAt(i) == ch ? 1 : 0), i + 1]
  if (s.charAt(i) == ch) {
   //@ [r:= r + 1]
   r++;
  }

  //@ [i := i + 1]
  i++;
 }

 //@ [result := r]
 return r;
}
```

As shown a CleanJava annotation is written as a special kind of comments either preceded by //@ or enclosed in /*@ and @*/, and every section of code including the whole method is annotated with its intended function that precedes the code. Most intended functions have the form [$x_1$, $x_2$, …, $x_n$ := $e_1$, $e_2$, .. $e_n$] stating that the new values of $x_i$'s are $e_i$'s concurrently evaluated in the initial state—the state just before executing the code. A partial function can also be specified by writing a condition followed by the -> symbol. For example, the first intended function specifies a partial function and states that the *numOfOccurrence* method is defined only when the argument *str* is not null. The method calculates how many times the given character *ch* appears in the given string *str*. The pseudo variable *result* denotes the return value of a method. An intended function is written in the Java expression syntax with a few CleanJava-specific extensions. The above example shows a few such extensions. For example, the first intended function uses two collection operations, *select* and *size*. The *select* operation is an iterator that selects all the elements of a collection—including an array and a string—that satisfies a specified condition, and the *size* operation returns the size of a collection (see Section IV.C.3). The expression, thus, denotes the number of times that the character *ch* appears in the string *str*. Another CleanJava-specific extension is the keyword *anything* appearing in the third annotation. It specifies that the final value of the corresponding state variable is not constrained; it states that we don't care about the final value of the loop variable *i*. Besides these two extensions, CleanJava also supports several other extensions such as informal descriptions, user-defined functions, model variables, and model methods, some of which will be introduced in later sections.

## *B. JastAdd and JastAddJ*

JastAdd is a meta-compilation system for generating extensible language support tools such as compilers and source code analyzers [4]. In JastAdd the data structures that support a compiler such as symbol tables and flow graphs are embedded in the abstract syntax tree (AST) in the form of attributes. An *attribute* is an AST node property to introduce functionalities and behavior to the AST [6]. AST nodes are implemented using Java classes, and their attributes provides APIs to the AST node classes. One of the features of JastAdd is its ability to define AST attributes declaratively. Attributes can be defined in any order using so-called aspects and can have different types of values, e.g., simple values like integers, composite value like sets, and references to other nodes in the AST. A reference-valued attribute allows one to explicitly define graph properties of a program; e.g., one can link an identifier like a variable name to its declaration node. Attribute values are defined by writing *equations* that may refer to other attributes. Attributes and equations are defined in an *intertype declaration*, a declaration that appears in an aspect file or a behavior specification, and are automatically added to their corresponding AST classes by JastAdd. It is also allowed to have regular Java member declarations like fields and methods in an intertype declaration. Object-orientation and intertype declarations are two key mechanisms of JastAdd to facilitate the construction of extensible language support tools.

A JastAdd application is typically consists of several extensible components (see Fig. 1). A *component* is composed of specification files, a frontend or application program, and a build file. A specification file can be a lexer specification, a context-free grammar or an abstract grammar. The lexer and context grammar files are inputs to a scanner and parser generators like JFlex [7] and Beaver [8], and the abstract grammar file defines AST nodes and behavior specifications. A build file is for specifying the elements of a component and compilation options.

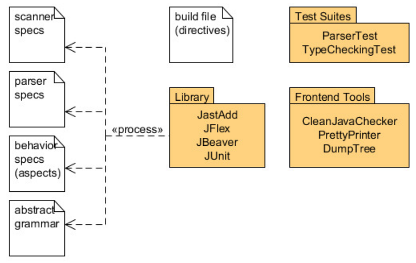

Fig. 1. Typical elements of a JastAdd component.

Rules in a specification file can be organized into *modules*. Modules are useful to organize specification rules based on their properties or classifications, e.g., similar compilation problems like as name and type analysis and grouping of language features. As mentioned before, specification files are transformed into Java classes, thus producing APIs that can be used by a frontend tool or main application such as a compiler. The CJC frontend component uses modules from several JastAddJ frontend components (see Fig. 2).

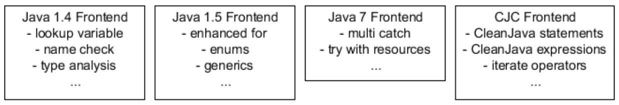

Fig. 2. JastAddJ and CJC Components.

A good example of a JastAdd application is JastAddJ, an extensible Java compiler [3]. JastAddJ facilitates the construction of static analysis tools for Java and the extension of the Java language with new language constructs. JastAddJ itself is a language extension in that the base implementation of JastAddJ supports Java 1.4 and two independent extension components add the features of Java 5 and 7. Every JastAddJ version consists of two components, a frontend and a backend. The frontend contains tools to parse Java source code, print compile-time error messages, and print the normalized version of a program (pretty printing) and its generated AST. The backend contains tools to generate Java class files. The backend tools are extensions of frontends. An extension of JastAddJ can act either as a pure checker by extending a frontend or as an extended Java compiler by extending a backend.

## III. THE CHALENGES

CleanJava is a formal specification language to support functional program verification. Its goal is to facilitate formal correctness verification and reasoning of Java programs by providing a Java-like notation for writing intended functions. Performing formal correctness proofs manually often times requires a considerable amount of time and effort and thus makes it less attractive to programmers. However, there is no support tool available for the CleanJava language. Basic support tools like a static semantic checker are important for several reasons. They can promote the adoption and use of CleanJava, facilitate the refinement and further development of CleanJava itself, and serve as a platform for constructing more advanced support tools like a correctness proof tool.

The first milestone toward the construction of CleanJava support tools is a checker that can parse Java code annotated in CleanJava and perform static checks including syntax and type checks. Ideally, the checker should be able to be used as an alternative to Java compilers such as javac. However, there are several challenges in developing a CleanJava checker. Since the CleanJava language is continuously being refined, new language features are likely to be introduced and some of current ones are likely to evolve as well. Thus, the checker should be sufficiently flexible to facilitate the experimentation of various language features and the accommodation of new language extensions. Another key requirement is extensibility. We envision a standard set of CleanJava tools to promote the use of CleanJava, especially in academia, where such a toolset will serve as a platform for teaching formal program verification. For this, we would like to use the checker as the base platform for constructing more advanced support tools like a specification analyzer and a proof assistant. Another challenge is that the checker should also understand the Java syntax and semantics because CleanJava annotations are embedded in Java source code as special kinds of comments and are written by referring to various Java elements such as variables, fields, and methods. It is simply out of question for us to build a new Java compiler, and it is also a daunting task to keep up with the Java language changes.

## IV. CLEANJAVA CHECKER

As stated in the previous section, the construction of a CleanJava checker is the first milestone toward the creation of a standard set of CleanJava tools including an automatic or semi-automatic proof tool. A key requirement for the checker is flexibility and extensibility to facilitate the addition of new language features. We decided to develop our CleanJava checker, called CJC, by extending an existing open-source Java compiler to avoid the trouble of writing a new Java compiler and to accelerate its development as well. We considered several open-source Java compilers including OpenJDK [21], ECJ (Eclipse Compiler for Java), and GCJ (GNU Compiler for Java) [20], and our decision was to extend JastAddJ. Two particularly interesting features of JastAddJ is its support for extension by using object-orientation and declarative attributes.

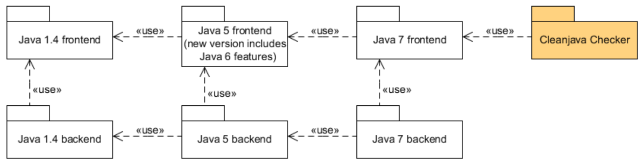

Fig. 3. CJC as an extension of JastAddJ.

### A. *Architecture*

CJC was built as an extension of the JastAddJ 7 frontend component inheriting all its features such as language constructs and static checks (see Fig. 3). CJC component is composed of a set of specification files, JUnit tests, and a build file. The lexical and abstract grammar rules of CleanJava are defined in the main specification files CleanJavaScanner.flex and CleanJava.ast. Various behavior specifications are grouped into modules based on CleanJava language features like expressions and statements, thus modularizing behavior specifications. An example of a CleanJava language feature is an intended function, which is a type of CleanJava statements. Its behavior rules are defined in a CleanJava statement module in a specification file CleanJavaStatement.jrag. Although the current implementation of CJC only modularizes behavior specifications, new features can be implemented using modules that contain different specification types such as parser and grammar specifications.

The build file is an important element of the CJC component. It specifies the specification files that are imported from other components, the order of compiling specification files, the names of output files to be generated, and their target locations. Although there is no standard way for defining the structure of a JastAdd component or the contents of the build file, it is important to use some conventions. Our structure and conventions can facilitate the creation of new CJC extensions by providing a better understanding of the architecture and allowing tools such as JASG framework [9] to perform certain operations automatically on an extension component. Future extensions of CJC can be made by importing CJC specification files into a new JastAdd component or by creating new modules in the current CJC component.

## *B. Implementation Process*

One neat feature of JastAddJ is its support for an incremental development, e.g., one language feature at a time. For an incremental development we first grouped CleanJava language features into feature groups and then implemented one feature group at a time by following the same basic development steps: (a) define parse nodes, (b) define lexical and parsing rules, (c) add attributes to the AST, and (d) create JUnit test cases. Below we illustrate this development process in detail by implementing a sample feature of CleanJava. However, before starting the implementation of the CleanJava features, we first need to create a new extension component for CleanJava. A working boilerplate component can be created from JastAddJ and can serve as the starting point for our extension. It will have all the necessary files like specification files and a build file to compile and build a Java 7 checker. The build file also defines new standard properties to facilitate a document creation in JASG [9].

The sample CleanJava feature to be implemented is the keyword *anything* denoting an arbitrary value. It is commonly used to indicate that one doesn't care about the final value of a temporary variable such as a loop variable, as shown below.

```
//@ [x, y, temp := y, x, anything]
temp = x;
x = y;
y = temp;
```

The first step for implementing the *anything* keyword is to declare a new parse node for it. Technically, *anything* is a literal because it denotes an arbitrary value. Therefore, we introduce a new parse node named AnythingLiteral as a subclass of Literal, a JastAddJ parse node class representing a Java literal. A new rule is added to the CJC abstract grammar defined in the specification file CleanJava.ast.

```
AnythingLiteral: Literal;
```

AnythingLiteral inherits all the behavior of Literal, including a type attribute that specifies the type of the expression represented by a node. AnythingLiteral can be used as a regular Java literal expression in a CleanJava annotation.

We then define a new lexical rule and a new parsing rule. Since *anything* is a keyword, we define a new terminal "anything" by introducing a new JFlex rule in the specification CleanJavaScanner.flex:

```
"anything" { return sym(Terminals.ANYTHING_Literal); }
```

The associated action routine states that the scanner will return a token named ANYTHING_LITERAL; this new token is used to define a parsing rule. We also define a new parsing rule in the CleanJavaExpression parser specification as follows.

```
Expr literal = ANYTHING_LITERAL
     {: return new AnythingLiteral(ANYTHING_LITERAL); :}
```

This statement introduces a new definition for an existing JastAddJ non-terminal *literal* of type *Expr*. Expr is an abstract AST node class defined by JavaAddJ to represent a Java expression. The definition states that a *literal* now can be an *anything*. The associated action states that when an anything literal is parsed a new object of type AnythingLiteral is created and returned. However, there is one complication here. We extended the definition of the *literal* non-terminal inherited from JastAddJ. This means that the newly introduced keyword *anything* can be also used in a Java expression; it's a Java literal too. We fix this problem by conditionally activating the newly introduced lexical rule using a JFlex feature called a lexical state [7]. A *lexical state* acts like a start condition in that if the scanner is in a specific lexical state, only expressions that are preceded by the same named start condition can be matched. We defined two lexical states for CleanJava, one for parsing single line annotations (//@) and the other for parsing multi-line annotations (/*@ … @*/). All CleanJava-specific lexical rules including that of *anything* have CleanJava lexical states as start conditions.

Once we define a parse node for the *anything* literal, we are ready to add behavior to the node by writing new behavior specification definitions. This can be done either declaratively (attributes, equations, and rewrites) or imperatively (ordinary Java field and method declarations). A *behavior* can be an AST attributes like a type attribute or an ordinary method declaration like the toString() method that define the parse node API. A subclass inherits the behavior of all its superclasses, and a superclass often forces its subclasses to implement a certain behavior, e.g., defining attribute values. For example, class AnythingLiteral extends class Literal, an indirect subclass of class Expr defined in JastAddJ 4. JastAddJ 4 specifies a *type* attribute for Expr that has to be implemented by all of its subclasses. Thus, to complete the implementation of the AnythingLiteral node we need to define the value of its inherited *type* attribute. We also need to implement a toString() method that is used by the JavaPrettyPrinter frontend program to print a normalized version of the literal. This behavior can be implemented by adding a new set of intertype declarations to a behavior specification file CleanJavaExpression.jrag that contains intertype declarations related to CleanJava expressions.

```
eq AnythingLiteral.type() = unknownType();
public void AnythingLiteral.toString(StringBuffer s){
  s.append(" anything ");
}
```

The above statements show two different types of intertype definitions. The first one is a declarative definition containing an equation for the *type* attribute of the AnythingLiteral class. The second is an imperative definition that overrides the toString() method inherited from the class ASTNode, the ultimate superclass of all AST node classes; this method is for pretty printing the parsed source code.

The type checking behavior for Java expressions is defined by JastAddJ. The only necessary step is to specify the type of the *anything* literal expression by assigning a value to the *type* attribute. Since the *anything* literal can be of any type, we use a special JastAddJ declaration type *unknownType*; this type acts as a wildcard type during type checking, making the *anything* literal assignment-compatible to any expression.

The last step is to add test cases. We add a new set of JUnit test cases for AnythingLiteral to two different test suites. One test suite includes test cases for checking syntax errors and the other for checking static semantic errors like type errors. This completes the implementation of the *anything* literal.

## C. Implementation of CleanJava Features

The current version of CJC supports all the core features of CleanJava language; core features are the basic features that provide the foundation of CleanJava, and advanced features like specification-only methods are built on top of the core features. In this section we describe the implementation of some of the supported features along with challenges that we encountered.

### 1) CleanJava Annotations

As shown in Section II.A, there are two kinds of CleanJava annotations: member-level annotations and statement-level annotations. The first annotation is for annotating class members like methods, and an example is an intended function for a method. The second is for annotating a single statement or a block of statements, and an example is an intended function for a *while* statement appearing in a method body. One interesting question is how to represent these two types of annotations in an AST. The goal is to reuse the provided framework code of JastAddJ as much as possible to reduce the development time and effort and to have a more reusable and maintainable implementation as well. Specifically we would like to delegate most of static semantic checks like type checks along with associated functions such as a symbol lookup to the framework code that performs these for Java code. In JastAddJ, an AST also plays the role of a symbol table, and we definitely don't want to duplicate it in our extension. Our solution is intuitive and straightforward and is to define a CleanJava-specific parse node as a subclass of the corresponding Java parse node (see Fig. 4).

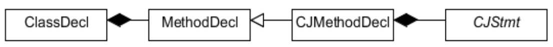

Fig. 4. AST nodes for representing member-level annotations.

We define a CleanJava-specific AST node CJMethodDecl as a subclass of a framework node MethodDecl representing a Java method declaration. The CJMethodDecl class represents a Java method declaration with a CleanJava annotation, i.e., an intended function for the method. Thanks to this subclassing, a Java class declaration (ClassDecl) can now have method declarations with (CJMethodDecl) or without a method intended function (MethodDecl). Similarly, we introduce a new AST node class, AnnotatedStmt, to represent a Java statement with an annotation (see Fig. 5). It is a subclass of Stmt representing a Java statement. Thus, a method body may consist of Java statements with or without CleanJava annotations.

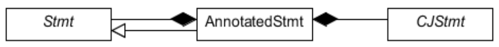

Fig. 5. AST nodes for representing statement-level annotations.

A CleanJava AST node class such as CJMethodDecl inherits features like name resolution and type checks from the corresponding Java AST node class, thus it only needs to implement CleanJava-specific features, e.g., type checking the intended function. Our approach is also extensible in that if CleanJava introduces a class-level annotation, e.g., a class invariant [16], it can be easily accommodated in a similar fashion by defining a CleanJava-specific new subclass of ClassDecl.

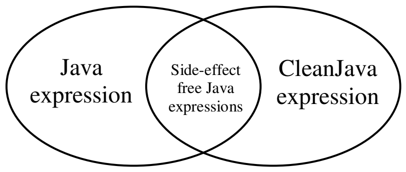

Fig. 6. CleanJava expression.

### 2) CleanJava Expressions

CleanJava expressions consist of side-effect free Java expressions and CleanJava-specific extensions (see Fig. 6). Java expressions that have side-effects are not allowed in CleanJava annotations. Examples of Java expressions that may have side-effects are those expressions written using operators with side-effects, e.g., the increment operator (++) and the decrement operator (--), and those that invoke methods that may have side-effects; only query methods are allowed in CleanJava expressions. Examples of CleanJava extensions are the *anything* literal, informal descriptions, and collection operations and iterators (see below).

An interesting question is how to disallow Java expressions with side-effects in CleanJava annotations. Our approach is to recognize them during the lexical analysis phase of the compilation. Specifically, we introduce a new scanning context to exclude Java operators that are not allowed in CleanJava. When the scanner sees one of the CleanJava annotation start markers (//@ and /*@), it changes its scanning context to this new context. Upon completion of scanning an annotation—i.e., when the scanner sees the corresponding annotation end marker (end-of-line or *@/)—it switches its scanning context to that of Java. We used JFlex's lexical state mechanism to implement our approach (see Section IV.B for lexical states). We use the same approach to support various CleanJava-specific keywords, operators, and symbols; they are recognized as tokens only within a CleanJava lexical context, i.e., only when parsing CleanJava annotations. One shortcoming of our approach, however, is that it cannot detect side-effects caused by invocations of mutation methods. This check may be done during the static semantic analysis phase assuming that the compiler can determine whether a method has a side-effect or not at compile time (see Section V).

One interesting feature of CleanJava is that it allows one to tune the level of formality in writing annotations. It is possible to mix formal and informal texts in annotations by using a facility called an *informal description*. An informal description is any text enclosed in a pair of "(*" and "*)". It can be used to escape from formality in writing an intended function, as shown below.

```
/*@ [cnt := cnt +
  @  (* occurrences of c in str starting at index i *)]
  @*/
```

One interesting fact about the informal description is that it is treated as an expression of any type. To be precise, its context determines its type. For example, the type of the above informal description is *int*, as it appears in a place where an *int* value is expected. How to check type correctness of an expression written using an informal description? If the type of an informal description can be determined or inferred based on its context of use, the expression is type correct; otherwise, it is not. To implement this, we assign a special type *unknown* to an informal description. It is a special type from JastAdd and acts as a wildcard when performing a type check of an expression involving values of unknown types.

*3) Iteration Operators*

In CleanJava, one can manipulate a collection of values by using *iteration operators*, also called *iterators* for short. An iterator can access or iterate over the elements of a collection to manipulate them. Iterators are defined for collections stored as Iterable objects, arrays, and strings, and some example iterators are *forAll*, *exists, select*, and *iterate*. The following intended function shows a typical use of an iterator:

```
[ok := accounts->forAll(Account acc; acc.balance() > 0)]
```

The *forAll* iterator returns true if each element of the collection satisfies the specified condition. Thus, the new value of *ok* is true if every account contained in *accounts* has a positive balance; otherwise, it is false. As shown, an iterator introduces a local variable (*acc*) to denote the element being iterated over. The scope of such an iterator variable is the body of the iterator (e.g., a.balance() < 0). Also note that an arrow notation (e.g., account->forAll(…)) is used to invoke an iterator.

Among the various iterators the *iterate* operation is the most powerful and general in that other iterators can be expressed in terms of it. It has the form *iterate*($T_1$ $x_1$, $T_2$ $x_2 = E_2$; $B$; $E_1$), where $T_1$ is the element type of the collection being iterated over, $T_2$ is the result type of this *iterate* operation, $B$ is an optional Boolean expression typically written in terms of $x_1$, and $E_1$ is an expression of type $T_2$ typically written in terms of $x_1$ and $x_2$; local variables $x_1$ and $x_2$ are called an *iterator* and an *accumulator* respectively. If $B$ is omitted, it defaults to true. Informally the meaning of this operation is as follows. Each value of the collection is assigned to the iterator variable $x_1$, and if the condition $B$ is true, the expression $E_1$ is evaluated and the result is stored in the accumulator $x_2$. When all the elements are iterated over, the value stored in $x_2$ is returned as the result. For example, the previous intended function can be rewritten using the *iterate* operation as follow.

```
[ok := accounts->iterate(Account acc, boolean r = true;
        true; r && acc.balance() > 0)]
```

An iterator poses several challenges in the implementation of the CJC tool. One challenge is that it is an expression but can have locally-scoped variables, i.e., an iterator and an accumulator. One complication is that the local variables have to be maintained in the symbol table during the parsing and checking of the iterator body. There is no such a language construct in Java, and thus the JastAddJ framework provides no support for an iterator-like language construct. Semantically an iterator behaves like a query method invocation in that it returns a value. However, structurally it is different from a method invocation, meaning that we can't represent it using the AST node class for a method invocation. Another challenge is the extensibility requirement of the CJC tool. Iterators are one particularly feature of the CleanJava language that is likely to be changed in the future as the language is being refined. More iterators are likely to be added, and the structure of the iterators may be changed. Ideally, the iterator design and implementation should accommodate such future changes easily.

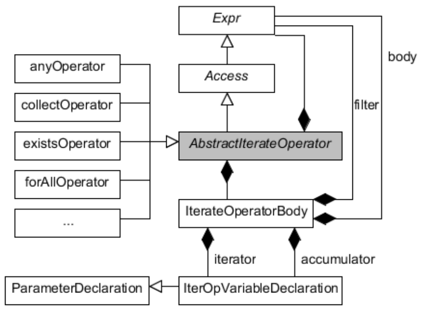

Fig. 7. AST node classes for iterators

Our solution to these challenges is to provide an extensible framework of AST node classes for iterator-like constructs by reusing JastAddJ framework classes as much as possible. Fig. 7 shows our design of iterator node classes. The common features of all iterators are factored out and implemented in the abstract class AbstractIterateOperator that extends the JastAddJ framework class Access. The Access class is an abstract node class representing a variable reference or a method invocation. Defining the AbstractIterateOperation as a subclass of Access allows us to treat an iterator invocation in a similar fashion as we do a Java method invocation. However, one difference is the existence of iterator variable declarations.

Structurally, an iterate operator is somewhat similar to a **for** loop statement in that it has a set of local variable declarations and a body expression. An iterate operator variable declaration is similar to the loop variable declaration of a **for** loop statement in that the scope of the variable is the body of the iterate operator and the name of the variable should be unique within its scope. As in Java, an iterator variable can shadow a class field that can be accessed by using the keyword **this**. The best way to handle an iterator variable declaration would be to treat it as a Java local variable declaration by representing it using the VarDeclaration node class. This would allow us to maximize code reuse, as we don't duplicate code for managing the symbol table for iterator variables. However, one problem is that the VarDeclaration class extends the node class Stmt that represents a Java statement, and thus it cannot be contained in an expression context such as an iterator invocation. As shown, our approach is to use a ParameterDeclaration node class instead. It has similar behavior to a variable declaration but is not tied to a particular type of AST nodes and thus can be used as part of any expression. However, it requires some code duplication from the VarDeclaration behavior specification, e.g., code for checking variable initialization and duplicate declarations.

The abstract class AbstractIterateOperator also implements the default behavior for type checking. For example, the receiver of an iterate operator must be of type Iterable or an array. The type of the iterator variable must be assignment-compatible to the element type of the receiver. If there is an accumulator, its type must be compatible with the type of the body expression.

A concrete iterator is implemented by creating a new AST node class as a subclass of the abstract AbstractIterateOperator class, adding parser and scanning rules, and creating specific behavior for the iterator in the iterate operator module. As an example, let us consider the *exists* iterator that tests if a collection contains at least one element for which a given condition is true. The *exists* iterator has the general form *exists*(*T x*; *B*; *E*), where *B* is an optional Boolean expression, and *E* is a Boolean-typed body expression. If there is any element that makes both *B* and *E* true, the iterator returns true; otherwise, it returns false. Its implementation requires definitions of only two new equations or behaviors: one for constraining the return type and the other for the type of the body expression (*E*).

```
eq ExistsOperator.type() = typeBoolean();

public void ExistsOperator.typeCheck() {
  super.typeCheck();
  if (!getBody().getArg().type().isBoolean())
      error("Body must be a boolean expression");
}
```

As shown in this behavior segment of code, the *exists* iterator inherits the type checking logic of its superclass; e.g., the checking of the optional Boolean expression (*B*) is performed by the superclass.

#### 4) Concurrent Assignments

In CleanJava, a concurrent assignment is used to write an intended function. Structurally it is similar to a Java assignment statement; however, semantically it denotes a mapping from one program state to another. There are several variations of concurrent assignments, e.g., simple concurrent assignments, conditional concurrent assignments, splitting definitions, and sequential composition. CJC supports all these different concurrent assignment statements by organizing their AST node classes into a class hierarchy (see Fig. 8 and Fig. 9). As shown in Fig. 8, the common superclass of all these variations is the AbstractConcurrentAssignment that extends the CJStmt class. Below we describe our design of simple concurrent assignments.

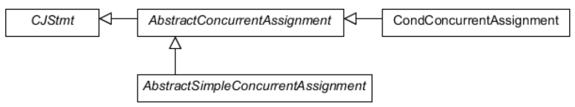
CJStmt
AbstractConcurrentAssignment
CondConcurrentAssignment
AbstractSimpleConcurrentAssignment

Fig. 8. Simple and conditional concurrent assignments.

A simple concurrent assignment is a concurrent assignment statement that doesn't have a condition. Semantically it denotes a total function; there's no constraint on the input state. To specify a partial function, one uses a conditional concurrent assignment that has conditions. The most commonly used simple concurrent assignment has the form [$L_1$, $L_2$, …, $L_n$ := $E_1$, $E_2$, …, $E_n$], where $L_i$ and $E_i$ are expressions. For the statement to be well-formed, it has to satisfy the following conditions which are checked by the CJC tool.

- $L_i$'s and $E_i$'s are well formed.
- The number of $L_i$'s is equal to the number of $E_i$'s.
- Each $L_i$ denotes a location, and the locations denoted by $L_i$*'s* must be different. This check is done by looking up each $L_i$ location in the AST and keeping track of the set of visited variable nodes.
- The type of $E_i$ is assignment compatible to that of the corresponding $L_i$.

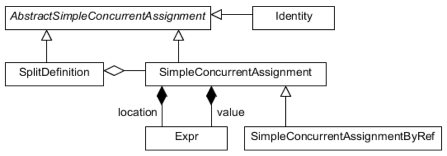
AbstractSimpleConcurrentAssignment
Identity
SplitDefinition
SimpleConcurrentAssignment
location
value
Expr
SimpleConcurrentAssignmentByRef

Fig. 9. Variations of simple concurrent assignments.

There are several variations allowed for the simple concurrent assignment (see Fig. 9). For example, there are two different semantic interpretations possible, *value semantics* and *reference semantics*. The value semantics means that the new value of $L_i$ is equivalent to $E_i$, and the reference semantics means the $L_i$ now refers to, or is the same as, $E_i$. The values semantics models the *equals* relation in Java and the reference semantics models the object equality (==). In CleanJava, the reference semantics is concretely denoted by the use of the symbol @= instead of := in the assignment statement. In AST, it is represented by the SimpleConcurrentAssignmentByRef class that does additional checking for reference types. Another variation is splitting the definition of a simple concurrent assignment. For example, instead of writing [x, y, z := 1, 2, 3], one can write [x := 1, y := 2, z := 3]. This is useful when writing an intended function with a long definition that spans multiple lines. The CJC tool combines the definitions and performs type and static checks on the combined definition. Yet another variation is a special statement called *identity* that denotes no change in the state of a program. This function is mainly used in the specification of a partial function using the conditional concurrent assignment.

### D. CJC Tools

The current CJC toolset contains several programs for the CleanJava users and developers. The programs includes in the toolset are a CleanJava checker, a pretty printer, and an AST tree viewer.

- CleanJava checker. This is the main program of the CJC toolset and performs static checks on a Java program annotated in CleanJava. It runs on a command prompt and behaves like a Java compiler except that it doesn't produce bytecode files. Internally, it extends the JastAddJ framework class *FrontEnd* that processes compilation arguments and options such as input files and classpath. It invokes the CleanJava parser that builds an AST from the input files and performs several static checks such as type checking.

- Pretty printer. This program prints parsed source code in the CleanJava presentation syntax and is mainly for the developers. The presentation syntax is mainly for human readers and the ASCII syntax is for the tools. In the ASCII syntax, for example, CleanJava-specific keywords like anything are preceded by the character '\', e.g., \anything, as they can be valid Java identifiers; note that in this paper we use the presentation syntax to show CleanJava annotations. The pretty printer is implemented by overriding the *toString*() methods of AST node classes..
- AST viewer. This is another tool for developers and displays or dumps the AST of the parsed source code.

## V. Evaluation and Discussion

We evaluated the checker using two different criteria: extensibility and language coverage.

### A. Extensibility

Extensibility is one of the key requirements of the CJC toolset. The checker should accommodate the introduction of new language features and also facilitate the creation of new support tools like a proof checker. In addition to the extensibility obtained by using the JastAddJ framework as the base code, the checker also provides a built-in extension mechanism for language features that are likely to be changed or extended. For example, one can easily add a new collection iterator, say the *any* iterator. The *any* iterator returns an arbitrary element of a collection that satisfies a condition, and it has the form any($T$ $x$; $B_1$; $B_2$), where $B_1$ is an optional Boolean expression and $B_2$ is a Boolean-typed body expression typically written in terms of the iterator variable $x$. Adding a new iterator like the *any* iterator is straightforward. We need to follow the JastAdd development cycle as follows.

- Step 1. Declare a new AST node class as a subclass of the AbstractIterateOperator class (see Section IV.C.3).

```
AnyOperator: AbstractIterateOperator
```

- Step 2. Declare a new token *any*.

```
"any" { return sym(Terminals.ANYOP); }
```

- Step 3. Define a grammar rule for the iterator.

```
Access iterate_access =
  iterate_receiver ANYOP iterate_body;
```

- Step 4. Define the behavior for the new AST node class. Most static checks are already implemented in the superclass, and we only need to define the type of the new iterator and check the type of its body expression as follows.

```
eq AnyOperator.type() =
   getBody().getIterator().getTypeAccess().type();

public void AnyOperator.typeCheck() {
  super.typeCheck();
  if (!getBody().getArg().type().isBoolean())
   error("Boolean body expected!");
}
```

It still needs to be evaluated if the checker provides a suitable base code for developing more advanced support tools like a proof checker. However, we hope the extensibility obtained by employing JastAdd as the underlying framework has positive impacts.

### B. Language coverage

The current version of the CJC tools (version 0.3.5) supports most of the core features of the CleanJava language. It supports 26 features out of 32 core features, thus giving 81% of language coverage. It should be noted that the main focus of the first version of CJC was to support the basic notation for writing intended function, while providing mechanisms and guidelines for creating future extensions. The unsupported language features include statement-level annotations, associating annotations with code sections, annotation labels (e.g., f1 in f1: [x := x + 1]), and advanced features such as class invariants, model methods, and user-defined functions .

### C. Discussion

One of the challenges in constructing the CJC tools was learning about and understanding the JastAddJ framework. The initial work was done using JastAddJ 5 that lacked API documents and support. For example, it was difficult to figure out different aspect declarations needed to inherit behaviors from various existing AST node classes. Without proper documents, it was also difficult to know the purposes and meanings of certain attributes and equations. Fortunately, the arrival of JastAddJ 7 along with a new version of JastAdd in 2012 alleviated some of these difficulties. The newer version provides improved APIs and better documents, and the JastAdd community has increased and became more active providing help to developers. Even with JastAddJ 7, however, the learning curve is steep. It is a daunting task especially for a beginning developer to figure out all the elements needed for the construction of a new feature or extension. The framework has 317 AST nodes with behaviors defined in many different aspect files, about 230 parser rules, and 600 rule definitions in four different components. This problem was partially addressed with the inclusion of RagDoll in JastAddJ 7 that creates API documents from AST node classes, listing attributes and methods along with the locations of their definitions. As our own solution to this problem we created the JastAdd Specification Generator Framework (JASG) to facilitate the creation of new component features and their documents [9]. It allows one to create a new feature or extension by completing XML templates. An included tool parses a JASG XML specification and generates all required JastADD specification files such as parser, scanner, AST, and AST behavior. We used JASG not only for implementing the iterate operators incrementally but also for generating the scanner and parser APIs containing all the parser and scanner rules from all the four different modules of the CJC extension.

The development of the CJC tools also made a contribution to refining the CleanJava language. The actual construction of the checker revealed several ambiguities and weaknesses in the language. For example, we were able to identify an ambiguity in composing different variations of the conditional concurrent assignments. We also learned that there is no simple, modular way to check the side-effect-freeness of an expression unless the language provides a notation to state the side-effect-freeness of a method; there are different approaches proposed for checking side-effect-freeness of Java methods [10]. The CJC tools development also contributed the design and refinement of the ASCII syntax of the language.

## VI. Related Work

The development of CJC tools was inspired by other language processing tools that faced similar challenges. One such a tool is JAJML, a JastAddJ-based compiler for the Java Modeling Language (JML) [11]. The main objective of JAJML was creating an extensible JML compiler. According to case studies [12] [13], the use of attribute grammars not only facilitated the implementation of language extensions based on JastAddJ but also proved to be more extensible than other approaches, such as JML 4 [17] developed based on the Eclipse Java Development Tools (JDT). However, one downside mentioned is the uncertainty of JastAddJ regarding its future support, especially new revisions of Java. At the time of the case studies, for example, JastAddJ only supported Java 4 and 5 but not Java 6; the current version of JastAddJ supports Java 7. One of the features that JAJML and CJC share is the inclusion of program specifications or annotations inside Java comments. The CJC tools adopted the approach of JAJML to create scanner lexical states and recognize CJC-specific terminals inside Java comments. The development of CJC not only leveraged the findings of JAJML, e.g., fast and easy creation of extensible compilers but also introduced language-specific mechanisms for creating future extensions, e.g., abstract iterate operators and JASG templates [9]. JAJML also showed how a JastAddJ compiler extension could be further extended. For example, SafeJML is an extension of the JML language that supports specification of safety critical Java programs, and it was implemented as an extension to JAJML by introducing new language constructs [14]

The AspectBench Compiler (ABC) is an extensible AspectJ compiler created initially using Polyglot and Soot for the frontend and backend respectively [15]. An alternate frontend was later created as an extension of JastAddJ 4. One of the main advantages of the JastAddJ frontend was the automatic scheduling of attribute computations compared with polyglot's manual scheduling [3]. Other advantages of JastAddJ over polyglot were the reduction of source code, increased speed, and less compiler errors.

There are Java programs for supporting Cleanroom-style functional program verification [18]. One program can even perform a trace on a Java program with intended functions. It looks like that a subset of Java expression syntax is used to write intended function, but due to lack of documentation, we were not able to know the exact notation for writing intended functions. It is also unclear if the programs were designed with future extensions in mind.

## VII. Conclusion

We developed a CleanJava checker called CJC to help the design and refinement of the CleanJava language and to promote Cleanroom-style formal program development as well. The CJC tool parses a Java program annotated with CleanJava specifications and performs static checks such as type and syntax checks. It's our first effort toward constructing a suite of support tools for CleanJava and hopefully provides base code for more advanced tools like an automated verification tool. Developing the CJC tool posed several interesting and engineering challenges. The tool has to be sufficiently extensible to not only facilitate the experimentation of various language features but also support future language extensions. It had to understand the Java syntax and process Java source code because CleanJava specifications are embedded in Java source code and written by referring to Java program elements such as variables, fields, and methods; the CleanJava language is an add-on to the Java language and its specifications are checked and interpreted in the context of a Java program.

We presented our solution to the above challenges, focusing on the design and implementation of the main features of the CleanJava language. The keys to our solution are to use an extensible Java compiler as a base platform for our development and to provide a built-in extension mechanism for language constructs and features that are likely to be refined and changed in the future. The particular Java compiler framework that we chose was JastAddJ and it supported extensibility and avoided building a new Java compiler. The particular features of JastAddJ that contributed to the extensibility include an object-oriented modeling of AST node classes and a declarative, aspect-oriented way of defining AST behaviors. In addition to the built-in extension mechanism for CleanJava constructs like iterators, we also developed an XML-based CJC extension framework, called the JastAdd Specification Generator Framework (JASG), to facilitate the creation of new CJC extensions and their documents.

The current version (version 0.3.5) of the CJC tools supports most of the CleanJava core language features and includes a few front-end tools. The next step is to extend the checker to a CleanJava compiler that generates Java bytecode. This can be done by extending the JastAdd backend component. A CJC backend will be useful to perform dynamic or runtime verification, e.g., by injecting executable CJC specifications into Java class files. Other future work includes creating a CJC extension that computes a code function—a function computed or implemented by a section of code—to assist formal program verification, supporting runtime analysis and verification, and integrating CJC with an IDE such as Eclipse.

## Acknowledgment

This work was supported by NSF grant DUE-0837567.

## References

[1] H. D. Mills, M. Dyer and R. Linger, "Cleanroom Software Engineering," *IEEE Software,* vol. 4, pp. 19-25, Septermber 1987.

[2] Y. Cheon, C. Yeep and M. Vela, "The CleanJava Language for Functional Program Verification," *International Journal of Software Engineering,* vol. 5, no. 1, pp. 47-68, 2012.

[3] E. Torbjorn and H. Gorel, "The JastAdd Extensible Java Compiler," *ACM SIGPLAN Notices—Proceedings of the 2007 OOPLSA,* vol. 42, no. 10, pp. 1-18, October 2007.

[4] E. Torbjorn and H. Gorel, "The JastAdd System—Modular Extensible Compiler Construction," *Science of Computer Programming,* vol. 69, no. 1-3, pp. 14-26, December 2007.

[5] A. Stavely, Toward Zero Defect Programming, Addison-Wesley, 1999.

[6] H. Gorel, "An Introductory Tutorial on JastAdd Attribute Grammars," *Generative and Transformational Techniques in Software Engineering III*, Lecture Notes in Computer Science, vol. 6491, pp. 166-200, 2011.

[7] G. Klein, "JFlex: The Fast Scanner Generator for Java." Available: http://jflex.de.

[8] A. Demenchuk, "Beaver: A LALR Parser Generator." Available: http://beaver.sourceforge.net.

[9] C. Yeep, "JASG: JastAdd Specification Generator Framework." Available: https://github.com/ceyeep/JASG.

[10] A. Rountev, "Precise Identification of Side-effect-free Methods in Java," *Proceedings of the 20th IEEE International Conference on Software Maintenance,* pp 82-91, 2004.

[11] G. Haddad , "JAJML." Available: http://sourceforge.net/apps/trac/jmlspecs/wiki/JAJML.

[12] G. Haddad and G. T. Leavens, "Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations," School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL, 2008.

[13] P. Chalin, P. R. James and G. Karabotsos, "An Integrated Verification Environment for JML: Architecture and Early Results," *Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, 2007.

[14] G. Haddad, F. Hussain and G. T. Leavens, "The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification," *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, New York, NY, 2010.

[15] P. Avgustinov, A. Christensen, L. Hendren, et al., "abc: An Extensible AspectJ Compiler," *Transactions on Aspect-Oriented Software Development*, Springer, pp. 293-334, 2006.

[16] C. Avila and Y. Cheon, “Functional Verification of Class Invariants in CleanJava,” *Innovations and Advances in Computer, Information, and Systems Sciences, and Engineering,* Lecture Notes in Electrical Engineering, vol. 152, Springer-Verlag, pp. 1067-1076, August 2012.

[17] A. Sarcar and Y. Cheon, “A New Eclipse-Based JML Compiler Built Using AST Merging,” *Second World Congress on Software Engineering,* Dec. 19-20, 2010, Wuhan, China, pp. 287-292, IEEE Computer Society.

[18] G. J. Ferrer, "Tools for Cleanroom Software Engineering," Available: http://ozark.hendrix.edu/~ferrer/software/cleanroom.

[19] T. Parr and R. W. Quong, "ANTLR: a Predicated-LL(k) Parser Generator," *Software—Practice and Experience,* vol. 25, no. 7, pp. 789-810, July 1995.

[20] Free Software Foundation, “GCJ: The GNU Compiler for the Java Programming Language”, November 2012, Available: http://gcc.gnu.org/java.

[21] Oracle Corporation, “OpenJDK,” 2013, Available: http://openjdk.java.net.