2013-01-01

CJC: An Extensible Checker for the CleanJava Annotation Language

Cesar Eduardo Yeep
University of Texas at El Paso, ceyeep@miners.utep.edu

CJC: AN EXTENSIBLE CHECKER FOR THE CLEANJAVA ANNOTATION LANGUAGE

CESAR EDUARDO YEEP RIVAS

Department of Computer Science

APPROVED:

Yoonsik Cheon, Ph.D., Chair

Rodrigo Romero, Ph.D.

Eric Smith, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

to my

FAMILY

with love

CJC: AN EXTENSIBLE CHECKER FOR THE CLEANJAVA ANNOTATION LANGUAGE

by

CESAR EDUARDO YEEP RIVAS

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2013

Acknowledgements

I would like to thank all of those who helped me in one way or another to complete my research work and finish my thesis. First of all, I would like to thank Dr. Yoonsik Cheon for his support during all these years. He has been a great mentor to me as a researcher and as an educator. He helped me to become a better engineer and gave me the opportunity to guide and learn from my fellow students as a Teaching Assistant. I would also like to thank the members of my committee Dr. Rodrigo Romero and Dr. Eric Smith for their review and contributions to my work.

I thank Jennifer Kuchle for her feedback and valuable contributions to my thesis. She provided me support during the toughest times and motivated me to continue pursuing my goals.

Finally, I would like to thank my family for their love and support during all these years. They are my guide and motivation to continue becoming a better person every day.

Abstract

Formal program verification can be used as a complementary technique to software testing. It allows checking the correctness of all the states of a program which may be impossible using only software testing as a verification technique. One software development process that relies on formal verification is Cleanroom Software Engineering. Cleanroom's main principles are to certify the software with respect to its specification and to produce zero-fault or near-zero-fault software. Cleanroom has being tested primarily in safety-critical systems that require a high level of correctness by development teams in places such as NASA and IBM, demonstrating lower fault rates and improved reliability. One of the techniques derived from Cleanroom is *functional program verification*. Functional program verification consists of calculating the function computed by the code (*code function*) and comparing it with its formal specification (*intended function*). A program is correct with respect to its specification if both functions are equivalent. CleanJava is a formal annotation language for the Java language that supports Cleanroom-style functional program verification. CleanJava has two main purposes: to promote the use of functional program verification (especially in the academia) and to serve as a platform for the development of techniques and tools that enable automatic or semi-automatic functional program verification.

Currently there are no support tools for the CleanJava language. The main step towards building CleanJava tools is the creation of a language checker that parses CleanJava specifications and performs static analysis such as syntax and type checking on those specifications. However, developing a checker for CleanJava poses several interesting challenges. The checker needs to be sufficiently flexible and extensible since the CleanJava language is still under development requiring constant experimentation and implementation of new language features. The checker will serve as a base platform to more advanced tools such as fully automated theorem provers, so it needs to support extension mechanisms and integration with other development tools. Because CleanJava notation is based on the Java language syntax and CleanJava annotations are embedded in Java programs, the checker needs to understand and process Java code as well. This would require building yet another Java compiler or ideally extending an existing Java compiler.

In this thesis I describe a series of solutions to address the above mentioned challenges related to developing a CleanJava checker (CJC). A key element in my solutions is to implement the CJC tool as an extension of an existing Java compiler that provides extensibility features. JastAddJ was used as the base code to support extensibility and avoid building a new Java compiler. JastAddJ is an extensible Java compiler that allows creating Java language extensions in a modular way. JastAddJ extensibility capabilities are provided by JastAdd, a meta-compilation system for creating modular and extensible compilers. Another key element in my solution approach was to build a set of tools that facilitate the creation of CleanJava language features including JastAdd specification file generators using XML templates. The current implementation of CJC supports most of the CleanJava language features and can be used as an alternative to a Java compiler such as javac.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In our modern society, software has become a fundamental aspect of our daily lives. We use software in a day to day basis to perform many different tasks from work to entertainment. Software can be found in almost any electronic device going from applications in smart phones to software in an avionics system. Software systems are also used to control the operations of all kind of organizations from small companies to multinational corporations. Software quality is then an important issue in a software dependent society. High quality software is expected especially in safety-critical systems such as financial, healthcare, aeronautics, defense systems, or any other system that involves high risk for humans or may cause huge economic loses. Most software development methodologies include a software verification or quality assurance phase to detect software defects and improve the quality of the software. One of the most popular techniques for improving the quality of software is software testing. Software testing provides reliable and low-cost solutions such as fully automated testing solutions. Software testing is especially attractive in the industry where software is constantly changing and low-cost solutions are necessary in order to maintain competitiveness. Testing depends on a finite number of test cases to catch defects in the code. For most software systems, testing all possible states of a program will require an extensive or almost infinite number of test cases. Because it is impossible to create an infinite number of test cases, in most cases testing needs to be rationalized and applied extensively only to critical components. Still, there are a finite number of test cases that can be performed during a certain amount of time. Formal program verification is a software verification technique that can be used as a complementary technique to software testing. Formal program verification has the potential of checking the correctness of all the possible states of a program. A software development process that relies on formal verification is Cleanroom Software Engineering [1]. Cleanroom's main principles are to certify a program with respect to its specification and to produce zero-fault or near-zero-fault software [2]. Cleanroom has been used primarily in safety-critical systems that require a high level of correctness by development teams in places like NASA and IBM, demonstrating lower fault rates and improved reliability [1] [3]. One of the techniques derived from Cleanroom is *functional program verification* [4].

The key steps of functional program verification consist of calculating the function computed by the code (*code function*) and comparing it with its formal specification (*intended function*). A program is correct with respect to its specification if both functions are equivalent. CleanJava is a formal annotation language for the Java language that supports Cleanroom-style functional program verification. CleanJava has two main purposes: to promote the use of functional program verification (especially in the academia) and to serve as a platform for the development of techniques and tools that enable automatic or semi-automatic functional program verification.

The CleanJava language requires the construction of an initial set of support tools that facilitate software development with CleanJava, such as a compiler and an IDE. The first step towards building CleanJava tools is the creation of a language checker that parses CleanJava specifications and performs static analysis such as syntax and type checking on those specifications. However, developing a checker for CleanJava poses several interesting challenges. The checker needs to be sufficiently flexible and extensible since the CleanJava language is still under development requiring constant experimentation and implementation of new language features. In addition, the checker needs to be integrated with other tools such as IDEs and also serve as a platform for more advanced tools such as fully automated theorem provers. Because the notation of CleanJava is based on the Java language syntax and CleanJava annotations are embedded in Java source code, the checker needs to understand and process Java code; this requires building a CleanJava checker that fully supports the Java language and performs static checking on Java programs and CleanJava annotations as well. This introduces new issues such as defining language context switching during static analysis e.g., for parsing CleanJava-specific syntax and defining CleanJava-specific namespaces.

In this thesis I describe a series of solutions and techniques to address the above mentioned challenges related to developing a CleanJava checker (CJC). A key element in my approach is to implement the CJC tool as an extension of JastAddJ, a Java compiler that provides extensibility features. JastAddJ was developed using the JastAdd framework which allows creating programming language compilers in a modular way [5]. Another key element in my solution approach is to build a set of techniques and tools that facilitate the creation of CleanJava language features including a JastAdd

specification file generator that uses XML templates. The current implementation of CJC supports most of the CleanJava language features and can be used as an alternative to a Java compiler such as javac.

1.1 Outline

The rest of this thesis is structured as follows:

Chapter 2 provides a background on the main technologies used to develop the CJC tools.

Chapter 3 describes in more detail the different problems addressed in this thesis and the requirements of a CleanJava checker.

Chapter 4 describes the architecture and implementation process of the CJC tool.

Chapter 5 describes design and implementation challenges as well as their solutions related to the development of the main CleanJava features.

Chapter 6 describes the implementation and features of the JASG framework, including the use of the framework in other projects created with JastAdd.

Chapter 7 provides an evaluation of the CJC tool in terms of extensibility and language coverage.

Finally, Chapter 8 concludes the thesis by summarizing the findings and main contributions. It also mentions related work and provides a list of future research topics.

Chapter 2: Background

This chapter provides an introduction to the main technologies and concepts necessary to better understand the development of the CJC tool. A more in-depth description of these concepts is provided as needed at the beginning of each section.

2.1.1 CleanJava

CleanJava is a formal annotation language for the Java programming language that supports Cleanroom-style functional program verification [6]. In functional program verification a program execution is modeled as a mathematical function that describes the state of program by mapping state variables to their values. These functions are described using a notation called a *concurrent assignment*. A concurrent assignment is used to describe the state of a program after executing a section of code. Concurrent assignments can express both the actual function computed by a section of code, known as *code function*, and the intention for this code, called an *intended function* [4]. Concurrent assignments 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. Another type of concurrent assignment is a *conditional concurrent assignment* used to define partial intended functions. A conditional concurrent assignment has the form [B_1 -> A_1 *else* B_2 -> A_2 *else* … *else* B_n -> A_n] where B_i's are conditions and A_i's are simple concurrent assignments.

CleanJava annotations such as concurrent assignments are written as a special kind of comments preceded by //@ for single line specifications or enclosed within /*@ and @*/ symbols for multi-line or block annotations. Intended functions are created using an extended form of Java expressions that excludes side-effect expressions such as "++" and "--" operators.

The following example shows a section of code annotated in CleanJava:

```
//@ [x, y := y, x]
x = x + y;
y = x - y;
x = x - y;
```

The intended function, preceded by //@, describes the behavior of a swap function using a concurrent assignment [x, y := y, x]. This intended function states that variable *x* gets the value of *y* and *y* gets the value of *x* concurrently (i.e., *y* gets the old value of *x).* In order to verify the correctness of a section of code using functional program verification, the code function implemented by the section of code is compared with its intended function.

For the previous example we can construct a *tracing table* to verify the correctness of the code against its intended function. A *tracing table* is used to illustrate step-by-step the state of program by showing the current value of the state variables after the execution of a statement.

Statement	**x**	**y**
x = x + y;	x + y	
y = x – y;		(x+y) – y = x
x = x – y;	(x + y) – x = y	
Final value	**y**	**x**

The function computed by this section of code is`[x, y := y, x]` which is the same as the intended function`[x, y := y, x]`, therefore we can say that the previous segment of code is correct with respect to its specification. The following section of code shows a more complex example:

```
/*@ [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 + (str.charAt(i) == ch ? 1 : 0), i + 1]
        if (str.charAt(i) == ch) {
             //@ [r:= r + 1]
            r++;
        }
    }
}
```

The first intended function in the previous example specifies a partial function and states that the *numOfOccurrence* method is defined only when the argument *str* is not null. The method determines how many times the given character appears in the given string. The previous example also shows some examples of CleanJava-specific features like *result* keyword which denotes the return value of a method and two iteration operations, *select* and *size*. The *select* operation is an iterator that selects all the elements of a collection or a String that satisfies a given condition; *size* operation returns the size of a collection. The previous intended functions denotes the number of times that the character *ch* appears in the string *str*. Another CleanJava-specific feature contained in the previous example (third annotation) is the keyword *anything*. It is used to express that the final value of a variable is not constrained to any particular value. This keyword is commonly used to describe local or incidental variables such as loop variable *i* (e.g., in the third intended function.)

The previous features along with other main features such as informal descriptions and sequential compositions are considered as the CleanJava core language. CleanJava also contains extension mechanisms used to extend the vocabulary of CleanJava by adding new symbols and expressions. Examples of extension mechanism are user-defined functions and model methods. Additional CleanJava features include model variables and specification inheritance.

2.1.2 JastAdd and JastAddJ

JastAdd is a meta-compilation system for generating extensible language support tools such as compilers and source code analyzers [5]. 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 used to provide added functionality or *behavior* to the AST. In JastAdd, a program is viewed as an object-oriented model, where AST nodes are implemented using Java classes and their attributes serve as an API to the AST classes. The main feature of JastAdd is the ability to define AST attributes declaratively, that is, they can be defined in any order using aspects. Attributes can have simple values such as integers, composite value like sets, or references to other nodes in the AST; values are stated using *equations* that may access other attributes. Reference-valued attributes allow to explicitly define graph properties in a program; one such application is the ability to

link an identifier such as a variable name to its declaration node. Attributes and equations are defined using *intertype declarations*, a declaration that appears in an aspect file or *behavior specification* and gets inserted into their corresponding AST class by JastAdd. It is also possible to insert regular Java fields or method declarations using intertype declarations. Object-orientation and intertype declarations are two key JastAdd mechanisms that facilitate the construction of extensible language support tools.

JastAdd applications are typically composed of a set of extensible components. A *component* consists of a set of specification files, a frontend or application program, a build file, and test suites (see Figure 2.1). There are four main types of specification files: scanner, parser, abstract grammar, and behavior specifications. The specification files are compiled by the JastAdd framework using tools such as JFlex [7] and Beaver [8] to generate Java-based programs of the scanner, parser, and abstract grammar. These Java programs are used by *frontend tools*, which are also Java programs, to perform different tasks such as parsing source code and building ASTs.

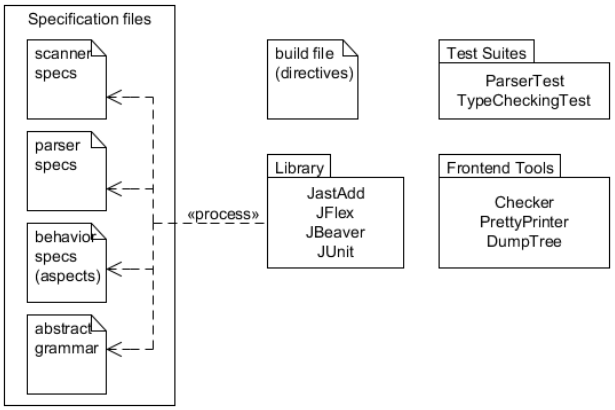


Figure 2.1: Typical elements of a JastAdd component.

A specification file is composed of a set of rules that defines a particular aspect of the component. For example, lexical rules in a scanner specification define the different tokens of the language supported by the component. Rules in a specification file can be organized into *modules*. Modules are useful to organize specification rules based on similar compilation problems such as name or type analysis, or be grouped by language features (see Figure 2.2).

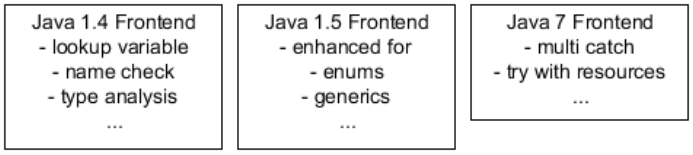


Figure 2.2: Example of JastAddJ modules (enclosed by a JastAddJ component).

An Apache Ant build file is used to define the elements of a component (including elements in other components) and configure their compilation options.

A good example of a JastAdd application is JastAddJ, an extensible Java compiler [9]. 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. Some of the applications of JastAddJ include components for non-null type checking and type inference [9]. 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.

Chapter 3: The Problem

CleanJava is a specification language that supports Cleanroom-style functional program verification. *Cleanroom Software Engineering* methodology emphasizes defect prevention instead of defect removal with the main purpose of producing near zero-defect software systems [4] [10]. *Functional program verification* is a powerful technique for supporting Cleanroom Software Engineering. This technique requires minimal mathematical background (i.e., sets and functions) and supports forward reasoning when evaluating program correctness. One of the main purposes of CleanJava is to facilitate the adoption of functional program verification both in the academia and industry, by providing a Java-based notation for creating *intended functions*. While CleanJava provides a robust notation for creating intended functions, performing formal verification proofs manually may take a considerable amount of effort and time, becoming an unfeasible task in projects that do not require a high level of correctness verification (i.e., non-safety-critical systems).

Currently there is no tool that supports the CleanJava language, making it difficult to use for research and software development purposes. Therefore, there is a need for creating tools and techniques that ultimately lead to semi-automated or automated functional program verification of programs annotated in CleanJava. The first milestone towards the construction of such automated tools is a static checker for the CleanJava language. Because CleanJava is still under development, the checker should be sufficiently extensible so researchers can implement and test new features of the language with a minimal exposure to the technical details of the checker. An example of this is the creation of new iteration operators or the implementation of CleanJava extension mechanisms such as user-defined functions and collection literals. The tool also needs to be extensible in order to facilitate the development of other CleanJava tools such as the creation of a CleanJava compiler that could be used as a drop-in replacement for a Java compiler.

As CleanJava syntax is based on the Java language syntax and CleanJava annotations are embedded in Java source code, the checker needs to understand and process Java code. The tool needs to support static checking on Java programs and CleanJava annotations as well. This introduces new issues such as defining language context switching during static analysis (e.g., parsing CleanJava-specific

constructs inside of Java comments) and extending the behavior of the Java checker to support CleanJava static checking. For example, the tool needs to check if a Java variable that it's being used in an intended function was properly declared and initialized and the function is in the scope of the variable. It also needs to check that variables that were declared inside of an intended function are not visible to other Java constructs, but are visible to other intended functions using CleanJava-specific namespaces.

In addition, it is necessary to provide a CleanJava development process that facilitates the creation of new CleanJava features and tools by minimizing the learning curve of the different technologies used to build the CleanJava checker. This includes defining CleanJava checker architecture, a development cycle, a testing framework, and providing adequate documentation

The development of a standard set of CleanJava tools will help to increase the user base of CleanJava, especially in the academia, where such tools will serve as a platform for teaching formal verification and support further development of the CleanJava language. A CleanJava checker is essential to provide feedback on the design of current and new features of the language by performing different tests or case studies using Java programs annotated with CleanJava.

Chapter 4: CleanJava Checker Architecture and Implementation

As stated in Chapter 3 (The Problem), the construction of a CleanJava checker is the first milestone towards the creation of automatic or semi-automatic verification tools for CleanJava. A key requirement for the checker is flexibility and extensibility; it must facilitate the addition of new language features. The CleanJava Checker, named CJC, also needs to be built as an extension of an existing Java compiler to reduce the burden of implementing a new Java compiler and accelerate its development.

One possible approach towards creating CJC is to extend an existing open-source Java compiler such as OpenJDK [11], the Java compiler in the Eclipse JDT project [12], or GCJ (GNU Compiler for Java) [13]. These compilers are popular for many reasons including support for the latest version of Java, a vast amount of documentation, supporting tools, and a large community of users and developers (e.g., Eclipse community) that provide continuous feedback and support for improving the quality and performance of these compilers. However, extending one of the above compilers will require a high level of expertise in compiler construction and a good knowledge of their APIs even for small extensions. Using one the mentioned open-source Java compilers as a platform base for CJC will not be suitable as the CleanJava language will be continuously evolving, requiring new language extensions.

A way to reduce the complexity of compiler construction is using a compiler-compiler system such as JastAdd. JastAdd is a Java-based system for constructing modular and extensible compilers. It provides object-oriented Abstract Syntax Trees (ASTs), typed access methods for traversing the AST, aspect-modularization for imperative (using regular Java code) and declarative code (using Reference Attributed Grammars) [5]. JastAdd Extensible Java Compiler (JastAddJ) is an open-source Java compiler implemented in JastAdd [9]. JastAddJ was designed to support extensibility by extending or creating new JastAdd components (see section 2.1.2).

CleanJava checker was constructed as an extension of JastAddJ to take advantage of its native extensible features, particularly its support for extension by using object-orientation and declarative attributes.

4.1 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 Figure 4.1).

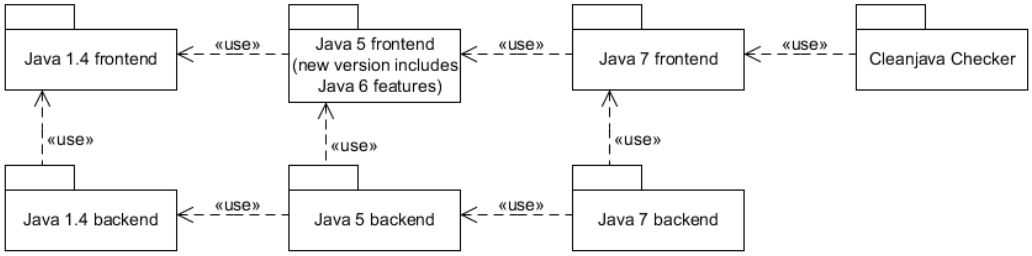


Figure 4.1: CJC as an extension of JastAddJ.

In JastAdd a component is a directory that contains all the necessary files to define, construct, and run a JastAdd application. A typical component includes specification files, compilation tools (e.g., JastAdd and JFlex), frontend tools, JUnit tests, and a build file (see Figure 4.2).

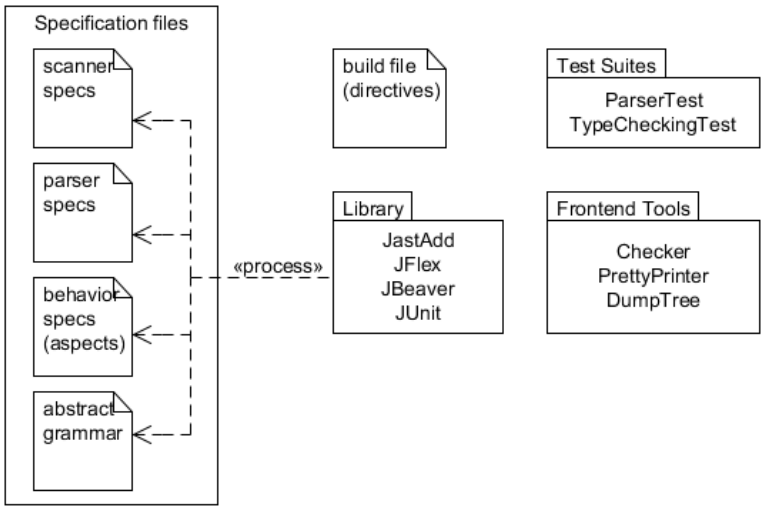


Figure 4.2: Typical elements of a JastAdd component.

In CJC, specification files are organized into modules (see Figure 4.3). Each module consists of a set of feature-related specification rules defined in different specification files. For example, the CleanJava statement module is composed of two different specification files, one for defining parser rules (CleanJavaStatement.parser) and one for defining AST behavior specifications

(CleanJavaStatement.jrag). Each specification file contains definition rules related to CleanJava statements such as simple concurrent assignments and conditional concurrent assignments.

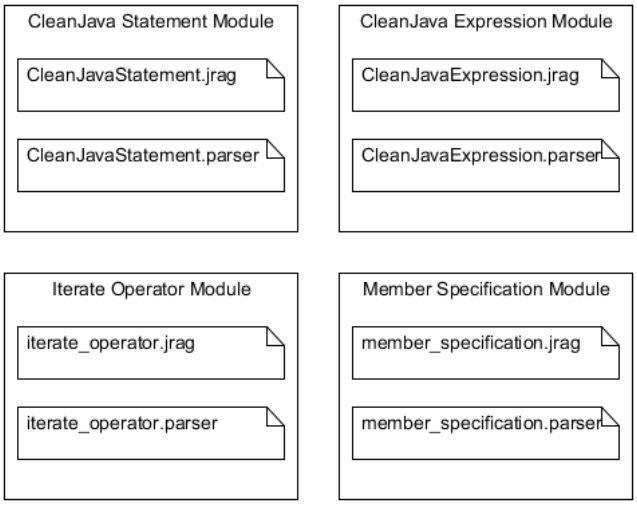


Figure 4.3: Main modules of the CJC component.

CJC modules are defined in two different ways: using a common name for different types of specification files (e.g., CleanJavaStatement.jrag and CleanJavaStaement.parser) or placing related specification files under a same directory. Although all the different types of specifications can be modularized into modules, the initial version of CJC only modularizes parser and behavior specifications. Abstract grammar and scanner rules are not part of a specific module. CleanJava.ast contains all the abstract grammar rules from the CleanJava language. This approach was inherited from the JastAddJ 1.4 abstract grammar definition (java.ast) which allows having the abstract grammar of a complete language in a single place, which is useful during the design phase of a language. The scanner specification of CJC is defined by different specification files to follow the structure of a JFlex specification file. For example, preamble specifications containing user code, scanning options, and lexical state definitions are defined in different specification files separated from the lexical rules. This allows creating lexical-only specification files that can be organized into modules and describe rule precedence by defining the compilation order of such modules in a build file (discussed below). All lexical rules in CJC are defined in a single scanner specification file CleanJavaScanner.flex following the same approach from the abstract grammar specification.

Specification modules provide a way to modularize rule definitions and facilitate locating such definitions for example by feature type. The CJC framework categorizes CleanJava features into four different modules: CleanJava statements (e.g., concurrent assignments), CleanJava expressions (e.g., informal descriptions), special types of Java class body and member declarations (e.g., method declarations with CleanJava annotations), and iteration operators. Although iteration operators are also CleanJava expressions, they have a more specific functionality and have a common design and implementation. New iterators can be easily created by reusing the code of existing iterators in the iterator module; this allows using the iterator module as a built-in extension mechanism for creating new iterators (see section 5.3). New modules can be used to introduce new types of features with common functionality, for example, a case study in section 7.1.2 introduces a new module for CleanJava extension mechanisms that adds user-defined functions and collection literals to the current CJC framework. CJC modules can be automatically generated by the JASG framework which organizes created specification files into modules (creating a new directory and using naming conventions for modules) based on a given module name.

An important element of the CJC component is the build file. The build file specifies the different elements that are imported from other components such as specification files and libraries, the compilation order of the specification files, the names and target paths of the generated output files. It also specifies directives to perform different tasks such as running tests or running a frontend program with parameters, similar to the functionality of a javac and java executable programs. The CJC build file contains two properties, scannerName and parserName, that are used by JASG to perform automatic operations such as generating a web-based API for the scanner and parser specifications of the CJC component.

The CJC component also includes a series of frontend tools. Frontend tools are a set of Java programs that facilitate the development of programs annotated with CleanJava. CleanJava tools are discussed in more detail in section 5.5.

Future extensions of CJC can be made by importing CJC specification files into a new JastAdd component or by creating new modules inside of the existing CJC component. Section 7.1 describes the design and implementation of three different types of CJC extensions that follows the architecture design discussed in this section. Chapter 6 describes how JASG framework can be used to facilitate the implementation of new CJC features using templates and documentation generation techniques.

4.2 IMPLEMENTATION PROCESS

Although there is no standard process to develop a JastAdd component, the developers of JastAdd provide a set of guidelines and examples to create new components such as JastAddJ extensions [14]. This section describes the general implementation process of CJC using JastAddJ implementation guidelines and additional strategies to facilitate the implementation of new CJC features.

CJC was implemented as an extension of JastAddJ using an incremental development process. The process consists of a cycle of steps that produces a new version of CJC at the end of each cycle. A new version may add new features to the tool or extend the functionality of existing features. CleanJava features to be implemented were divided into feature groups in order to define the outcome of each development cycle, implementing one feature group at a time during each cycle. A cycle consists of four main development steps:

1. Define parser nodes (AST classes)
2. Define lexical and parsing rules
3. Add behavior to the AST class
4. Create JUnit test cases

Before starting the implementation process, it was necessary to create a new extension component for CleanJava. The current distribution of JastAddJ 7 does not come with an extensible or boilerplate component. A JastAddJ boilerplate component was created to serve as a starting point for CJC. The boilerplate provides a compilable JastAddJ component that contains a build file configured to build a JastAddJ 7 checker, a set of specification templates, an initial set of generic frontend tools, and a test framework template. The build file introduces new standard properties to facilitate documentation generation with JASG.

To illustrate the implementation process of CJC, the implementation of a sample CleanJava feature will be described next. The selected sample CleanJava feature is the keyword *anything* which denotes an arbitrary value. This keyword is used as wildcard indicating that the final value of a variable is irrelevant, as shown in the following example:

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

The first implementation step is to define a new parser node. Because the *anything* keyword denotes an arbitrary value, it behaves like a Java literal. For that reason, a new parser node AnythingLiteral is introduced as a subclass of Literal, a parser node defined in JastAddJ that represents a Java literal. The AnythingLiteral node is defined as a new rule in the CJC abstract grammar specification file CleanJava.ast.

```
AnythingLiteral: Literal;
```

In JastAdd, non-terminals and productions of a parser are written as classes. This allows representing a program as an object-oriented model. An AST tree consists of a tree of objects of such classes called AST classes. AnythingLiteral class inherits all the behavior of Literal (see Figure 4.4), including a *type* attribute which holds the type value of an expression. Attributes behave as fields in a class. Attributes will be further discussed when attributing the AST. Because AnythingLiteral is a subclass of Literal, it can be used as a regular Java literal expression inside of a CleanJava annotation.

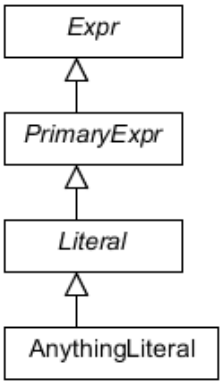


Figure 4.4: AnythingLiteral inherits the behavior of all its super classes.

```
class AnythingLiteral extends Literal {...}
```

Figure 4.5: AnythingLiteral class is generated by JastAdd from the abstract grammar definition.

The next step is to define new lexical and parsing rules. In this case, a new keyword *anything* is defined as a new terminal “anything” in the JFlex specification CleanJavaScanner.flex.

```
<CLEAN_SINGLE_LINE,CLEAN_MULTI_LINE > {
    "\anything" { return sym(Terminals.ANYTHING_LITERAL); }
}
```

The new terminal is defined using CleanJava *lexical states* CLEAN_SINGLE_LINE and CLEAN_MULTI_LINE which make the scanner recognize the token “anything” only within a CleanJava expression. A *lexical state* is a JFlex feature that acts as a start condition for matching tokens that are preceded by the same start condition [7]. The return statement indicates that the scanner will perform an action routine that returns a token named ANYTHING_LITERAL; this token can be used to define a new parsing rule.

CLEAN_SINGLE	Single-line annotations (//@)
CLEAN_MULTI_LINE	Multi-line annotations (/*@ … @*/)
CLEAN_STRING	Expression string (e.g., “CleanJava\n”)
CLEAN_INFORMAL	Informal description expressions ((*…*))

Table 4.1: In CJC there are four main lexical states.

The *anything* literal parsing rule is defined as follows.

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

A new rule is added to the existing JastAddJ *literal* non-terminal definition. The definition includes a return type *Expr*, which is an abstract AST node class defined by JastAddJ to represent a Java expression. The return statement indicates that the parser will create a new object of type AnythingLiteral when constructing the AST. Although the keyword *anything* could be used in any parsing rule that included a non-terminal *literal* such as a Java expression, CJC prevents this by using *lexical states*; the *anything* keyword can be used only within a CleanJava expression, otherwise the checker will return a syntax error.

After defining a parser node for the *anything* literal, the next step is to define its behavior by writing new behavior specification definitions. This can be done declaratively (attributes, equations, and rewrites) or imperatively (ordinary Java field and method declarations). A *behavior* provides additional functionality to the AST by defining the API of the AST classes. For example, in JastAddJ attributes are used during name analysis to associate an identifier with its appropriate declaration considering scope rules. It is also possible to override attributes inherited from super classes using *equations* or *rewrites*, or using ordinary method declarations imperatively. Continuing with our example, the class AnythingLiteral is a subclass of class Literal, which is in turn a subclass of class Expr, introduced in JastAddJ 4. JastAddJ 4 defined a *type* attribute for Expr that must be implemented by its subclasses.

Therefore, AnythingLiteral needs to implement or provide a value to its *type* attribute. It also needs to implement a toString() method used by the JavaPrettyPrinter frontend program to print a normalized version of the AnythingLiteral object. The behavior can be defined by adding a pair of *intertype declarations* to a behavior specification file CleanJavaExpression.jrag as follows.

```
eq AnythingLiteral.type() = unknownType();

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

An *intertype declaration* is a declaration that appears in a behavior specification that actually belongs to an AST class. The JastAdd framework compiles the behavior specification, which is an aspect, and weaves the intertype declarations into their appropriate AST classes (in this case the AnythingLiteral class). The first intertype declaration is a declarative definition containing an equation for the *type* attribute of AnythingLiteral class. In this case the new value for the *type* of AnythingLiteral is of *unknownType*, which is a special JastAddJ declaration type that acts as a wildcard type. This makes AnythingLiteral type-compatible with any expression during type checking. The second intertype declaration is an imperative definition that overrides the method toString() inherited from the ASTNode class, which is the superclass of all AST node classes. The overridden method adds the string “anything” to an output string buffer during pretty printing. Declared attributes and methods can be accessed from other AST classes as a regular Java method, such as AnythingLiteral.type().

The final step in the development cycle is to add test cases. The CJC testing framework contains two main test suites for JUnit. One suite includes test cases for checking syntax errors and the other for checking static semantic errors, such as type and name checks. The tests can be run using ANT targets “test-syntax”, “test-semantic”, or “test” to run all test suits.

4.3 SUMMARY

This chapter discusses the core architecture of the CJC component. It describes the main elements of the CJC component and the use of modules for organizing specifications and facilitating the

development of new CleanJava features and built-in extension mechanisms. It also discusses the general implementation process of the CJC tool by providing a step-by-step example of the development of a CleanJava feature. The next chapter includes a detailed description of the design and implementation of the main CleanJava features.

Chapter 5: CleanJava Checker Feature Implementation

The previous chapter introduced the general process of implementing a CleanJava feature. This chapter discusses in more detail the design and implementation of the major CleanJava features supported by CJC. The current version of CJC (v. 0.3.11) supports most of the core features of the CleanJava language. CleanJava core features provide the base of the CleanJava language. More advanced features such as user-defined functions and model methods (i.e., extension mechanisms) are built on top of the core. The features that are not supported by the current version of CJC are field declaration annotations and statement-level annotations.

5.1 CLEANJAVA ANNOTATIONS

Similar to Java annotations, CleanJava annotations provide additional information about a program without interacting with the program itself. There are two kinds of CleanJava annotations: member-level annotations and statement-level annotations. Member-level annotations are used to annotate class members such as an intended function for a method declaration; statement-level annotations are used to annotate a single statement or block of statements. A CleanJava annotation is written using a special kind of comment using the symbol `//@` for writing a single-line annotation and a pair of `/*@` and `@*/` symbols to enclose a multi-line annotation as shown below.

```
//@ [x := 2 ]

/*@ [x < 0 -> return := -1 |
 *   x == 0 -> return := 0  |
 *   x > 0 -> return := 1
 @*/
```

5.1.1 Challenges

The main challenge for implementing CleanJava annotations is to represent the different kinds of annotations in an AST; JastAddJ does not define a parser node for Java comments. The tool must identify CleanJava annotations inside Java comments and be able to process them. It also needs to associate a CleanJava annotation with a particular segment of code or class member. Additionally, one of the main requirements of CJC is to reduce the development time and effort of CleanJava features by

reusing the code provided by the JastAddJ framework as much as possible. Another requirement is extensibility; future extensions of CJC may need to support new types of annotations such as user-defined functions.

5.1.2 Implementation

The development of CleanJava annotations follow the development process described in section 4.2). The first step is to define a parser node for a CleanJava annotation. Because an annotation must be associated with a Java parser node, the most intuitive solution is to define a CleanJava-specific parse node as a subclass of the corresponding Java parse node with a composite annotation. For example, a CleanJava-specific AST node CJMethodDecl extends the node MehodDecl from JastAddJ which represents a Java method declaration (see Figure 5.1). The CJMethodDecl class represents a Java method declaration with a CleanJava annotation, such as an intended function for the method. Because CJMethodDecl is a subclass of MethodDecl a ClassDecl, which represents a Java class, can now have method declarations with or without an annotation.

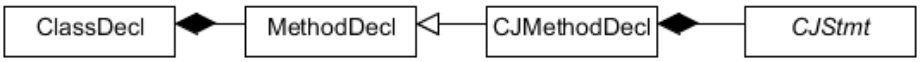


Figure 5.1: AST nodes for representing a member-level annotation.

The same approach was taken for other Java class members including static initializer, instance initializer, and constructor declaration (see Figure 5.2).

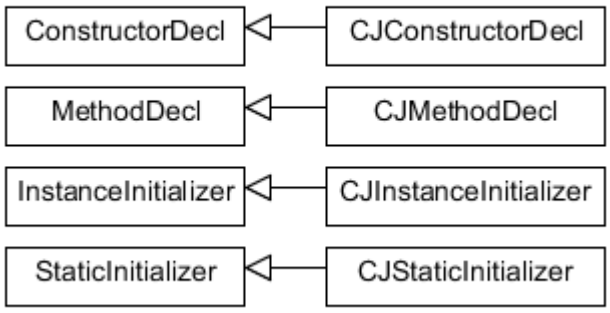


Figure 5.2: CleanJava-specific class member AST classes.

Statement-level annotations follow a similar design to member-level annotations but instead of extending a concrete AST node, the annotated statement (AnnotatedStmt) extends an abstract parser node Stmt that represents a Java statement. Stmt can contain other Stmt nodes; thus an AnnotatedStmt can contain regular Java statements and/or other annotated statements (see Figure 5.3).

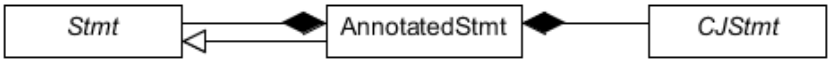


Figure 5.3: AST nodes for representing statement-level annotations.

In addition to creating AST nodes for CleanJava annotations, defining scanner and parser rules also introduced several challenges. Because CJC can be used as an alternate Java compiler, CleanJava annotations need to be treated as Java comments for traditional compilation purposes (i.e., generating Java bytecode) with an extended behavior for performing CleanJava code checking. This was possible by specifying CleanJava lexical states, as mentioned in the previous chapter. A start maker (//@ or /*@) triggers a CleanJava lexical state CLEAN_SINGLE and CLEAN_MULTI_LINE respectively and an end marker (end-of-line or @*) switches the scanning context back to Java. A CleanJava annotation is still viewed as a Java comment, but CleanJava lexical states allow further processing of the annotation, providing CleanJava-specific tokens to the parser. CJC also supports single-line comments (//) inside of a CleanJava annotation by defining a comment rule within a CleanJava lexical state context (see Figure 5.4).

```
/*@
    //a comment inside of a comment
    [x := 3]
    //another comment
 @*/
```

Figure 5.4: A Java comment inside of a CleanJava annotation.

5.1.3 Discussion

By extending JastAddJ AST classes the CleanJava-specific AST classes inherit features such as name and type checking from the corresponding superclass. This allows implementing only CleanJava-specific features such as type checking for intended functions, saving both implementation effort and time. This approach is also extensible in the sense that it can easily accommodate a new class-level annotation like class invariants [15] by defining a new CleanJava-specific subclass of ClassDecl. In addition, annotation nodes can be used to identify CleanJava subtrees in an AST as the annotation node serves as an entry point or root node of a CleanJava AST; this can be useful when performing analysis with CleanJava-specific language constructs.

The current version of CJC only supports member-annotations, as the primary goal of the initial version of CJC was to provide a tool to experiment with the notation of the specification language. The design and implementation of member-annotations will serve as a base for implementing statement-annotations in a future CJC iteration.

5.2 CLEANJAVA EXPRESSIONS

CleanJava expressions are written using an extended syntax of Java expressions. All Java expressions and operators are allowed in CleanJava annotations except for side-effect expressions, expressions that change the value of variables when evaluating the expression, such as the increment operator (++), decrement operator (--), and assignment operator (=). Other valid expression are *query methods* (method that cause no side-effects), and literals such as *null* (see Figure 5.5).

```
//@ [ sum := x + y ]
//@ [ max := Math.max(x, y) ]
```

Figure 5.5: Example of intended functions with side-effect free Java expressions.

In addition to Java expressions, CleanJava introduces its own set of expressions such as *anything* literal, informal description, and collection operations and iterators (see Figure 5.6).

```
//@ [x := \anything ]

//@ [x := (* maximum of x and y *) ]

//@ [ x := myArray=>\any(int a; a == 0) ]
```

Figure 5.6: Example of intended functions with CleanJava-specific expressions.

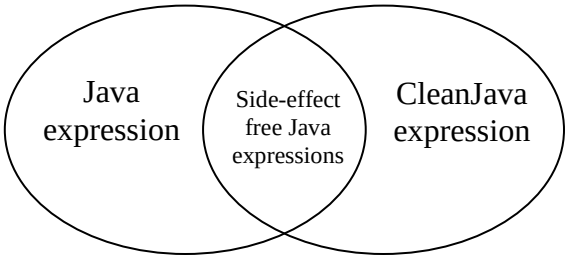


Figure 5.7: CleanJava expression.

5.2.2 Challenges

The main challenges towards implementing CleanJava expressions is to disallow Java expressions with side-effects in CleanJava annotations and implement static semantic checks for CleanJava expressions such as type and name checking. Implementing CleanJava-specific expressions introduces several additional challenges such as extending proper JastAddJ expression nodes and defining extended AST behavior. The design and implementation of iteration operators will be discussed in a separate section, as they possess several interesting challenges by their own.

5.2.3 Implementation

The main method for filtering out side-effect Java expressions in a CleanJava annotation is to recognize side-effect Java operators during lexical analysis. A way to do this is to exclude those operators from the CleanJava scanner specification. Only side-effect free operators are added to the scanner specification using CleanJava expression lexical states. The same approach was used to include CleanJava-specific keywords, operators, and symbols using CleanJava expression lexical states. Only CleanJava-specific tokens (those recognized within a CleanJava lexical context) can be used when parsing CleanJava annotations.

The following sections discuss the implementation of some of the most important CleanJava-specific expressions. Iteration operators are discussed in more detail in Section 5.3.

Anything literal

The *anything* literal is used to assign an arbitrary value to a variable. The step-by-step implementation of this feature is discussed in Section 4.2.

```
//@ [x, y := 2, \anything]
```

Figure 5.8: Example of an intended function with an *anything* literal.

Note: One thing to notice is the use of the symbol "\" in the keyword *anything*. In order to distinguish variable names (e.g., anything) and other Java-specific symbols from CleanJava-specific keywords, a symbol "\" was prepended to most of the CleanJava keywords. This approach creates differences between the syntax of the CleanJava language and the CJC syntax. A list of these differences is provided in Appendix A.

Informal description

An *informal description* is a mechanism that allows one to tune the level of formality of an annotation; it allows escaping from formality when writing annotations. It can be used to simplify the definition of an intended function, or to partially define a function before creating a fully formal definition. It also allows mixing formal and informal descriptions. An informal description is written between a pair of "(*" and "*)" symbols as shown below.

```
//@ [minPlusOne := (* minimum value of array A *)+ 1 ]
//@ [min := (* minimum value of array A *) ]
```

An informal description is treated as an expression of any type as its type is truly defined by its context of use. In the previous example, the intended function assigns to variable *min* the minimum value of array A. Variable *min* is of type integer, therefore the type of the informal definition expression should be type compatible with an integer type (e.g., integer). This requires that the type of the informal

definition expression gets determined or inferred based on its context. To implement this feature, the informal description parser node was defined as subclass of Expr of type *unknown*. Similar to the *anything* literal, this makes an informal description type-compatible with any expression during type check.

Note: Although this is a mechanism supported by CleanJava, it is recommended to minimize the use of informal notations to fulfill the real purpose of functional verification [6].

Result operator

The *result* operator denotes a return value in a Java method declaration. It is used to specify the resulting value of a non-void method declaration (see Figure 5.9).

```
//@ [ \result := x + y ]
public int sumXY(){
    return x + y;
}
```

Figure 5.9: Example of an intended function with a *result* operator.

The *CJResult* node was defined as a Java expression extending the Expr AST class. One interesting aspect of this feature is assigning the correct type to *CJResult*. The type of the expression is determined by the type access of the annotation's parent node (e.g., CJMethodDecl). CJC introduces a new ASTNode attribute *containingDeclaration* of type *MethodDecl* that returns the contained CJMethodDecl of the parent node of an annotation. If the *result* operator is used in an annotation that is not associated with a method declaration, CJC will return a semantic error.

5.2.4 Discussion

Although using lexical states is useful for identifying side-effect expressions, it cannot be used for identifying non-query methods. Currently JastAddJ compiler cannot determine if a method has side-effects at compile time. Another side-back is keeping track of the changes made to the scanner specification in multiple JastAddJ modules. The list of Java operators supported by CJC was copied directly from the JastAddJ lexical specification. Still, this approach provides a flexible mechanism to

add CleanJava-specific operators and easily remove those that are not supported by the CleanJava language.

5.3 ITERATION OPERATORS

CleanJava provides a mechanism called *iteration operator* (also called *iterators*) used to manipulate a collection of objects. There are different types of operations that can be performed on a collection iteratively using different iteration operators. Some example iterators include *forAll*, *exists*, *select*, and *iterate*. Iterators are defined for arrays, strings, standard CJ collections, and Iterable objects.

```
//@ [ acc := accounts=>\any(Account a; a.getBalance() > 10000) ]
```

Figure 5.10: Example of an *any* iterator in an intended function.

The previous intended function describes an *any* iteration operation in which an arbitrary account with balance greater than 10000 is selected from a collection of accounts.

All iteration operators follow the general form:

$$receiver =>iterateOperator(T_1\ x_1,\ T_2\ x_2 = E_1;\ B;\ E_2) \qquad \text{Form 1}$$

where T_1 is the element type of a supported collection (e.g., Iterable or CJ collection), B is an optional Boolean expression called a *filter*, that may be written in terms of x_1, and E_2 or *body* is an expression of type T_2 that may be written in terms of x_1 and x_2; local variables x_1 and x_2 are *called* an *iterator* and an *accumulator* respectively. Different iteration operators have different type and number of attributes and return type. Most iterators follow one of the next compact general forms:

$$receiver=>iterateOperator(T\ x;\ B;\ E) \qquad \text{Form 2}$$

$$receiver=>iterateOperator(T\ x;\ B_1;\ B_2) \qquad \text{Form 3}$$

where x is an iterator of type T and each expression is bound to x. B and B_1 are optional Boolean expressions with default value *true*. In form 2, if B is true E is evaluated returning the specified value by the operator. In form 3, depending on the type of the operation if B_1 and B_2 hold then **true** is return, otherwise **false** is returned.

The intended function in Figure 5.10 presents an *any* iterator which contains a receiver with name *accounts*, an *iterator* variable *a* of type Account, and a Boolean expression (or *filter*) a.getBalance() > 10000. The any iterator has a general form of type 2, in which the return type of the iterator expression is the same type of the iterator variable (see Table 5.1), in this case, an Account type.

Table 5.1: List of iteration operators supported by CJC.

Operator	**Description**	**Return type**	**General form**
any(E)	any element for which *E* is true	*T* type	form 3
collect(E)	a collection that results from evaluating *E* for each element	CJCollection<S>, where S is *T* type	form 2
exists(E)	has at least one element for which *E* is true?	Boolean	form 3
forAll(E)	is *E* true for all elements?	Boolean	form 3
isUnique(E)	does *E* have unique values for all elements?	Boolean	form 2
iterate(T_1 x_1,T_2 x_2; E)	iterates over all elements accumulating the result to x_2	T_2 type	form 1
one(E)	has only one element for which *E* is true?	Boolean	form 3
reject(E)	a collection containing all elements for which *E* is false	CJCollection<S>, where S is *T* type	form 3
select(E)	a collection containing all elements for which *E* is true	CJCollection<S>, where S is *T* type	form 3

5.3.2 Challenges

Implementing an iterator has several challenges. One challenge is that the iterator is an expression that can have locally-scoped variables, such as local variables *iterator* and *accumulator*. These variables have to be maintained in the symbol table during parsing and semantic checking of the

iterator body. Additionally, an iterator body may contain nested iterators creating nested variable scope (see Figure 5.11).

```
//@ [ x := intArray=>\any(int a; a > intArray=>\any(int b; b > 0)) ]
```

Figure 5.11: Example of a nested iteration operator.

Semantically an iterator is similar to a query method, in the sense that it returns a value, but structurally is different. An iterator looks more like a for-loop statement containing variable declarations, which are also statements, and a set of expressions that are evaluated iteratively. JastAddJ does not have an AST node that supports such behavior, especially having a statement node as a child of an expression node. Another challenge is feature extensibility; as the CleanJava language keeps evolving, new iteration operators may be introduced or current ones may be changed. CJC must have a mechanism that facilitates the creation of new iterators and/or modify the behavior of current ones.

5.3.3 Implementation

In order to address the mentioned challenges a framework that facilitates the creation of iterator-like constructs and supports extensibility was created. The main element of this framework is the AST class AbstractIterator (see Figure 5.12). This abstract class contains the common features of all iterators represented in the iterator general form known as form 1. Iteration operators are implemented by extending the AbstractIterator class and specifying specific behavior for each iterator.

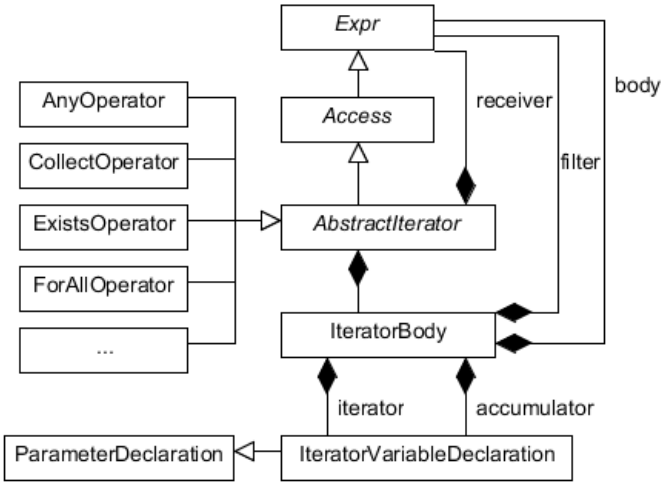


Figure 5.12: AST node classes for iteration operators.

AbstractIterator extends the *Access* class from JastAddJ which represents a variable reference or a method invocation. By extending the Access abstract class, an iterator can be accessed by other expressions such as variable access, method access, or another iterator. This functionality allows treating an iterator similar to a method invocation. One main difference is that a receiver of an iterator invocation can only be of type Iterable, an array, or a string. This constraint is checked during type checking by evaluating the type declaration of the associated *receiver* expression.

Functionally, an iterator variable declaration is similar to a variable declaration in a Java for-loop or a parameter declaration in a try-with-resources statement. However, the JastAddJ node VarDeclaration which represents a variable declaration extends Stmt which represents a statement. JastAddJ prevents having a statement as part of an expression. ParameterDeclaration is an AST node that represents a parameter declaration in a try-with-resources statement. Differently to a variable declaration, ParameterDeclaration does not extend any particular AST class. This feature allows having a parameter declaration inside of an expression. An AST node IteratorVariableDeclaration that extends the class ParameterDeclaration was introduced to represent an iterator variable declaration. The downside of this approach is a limited functionality from ParameterDeclaration compared to VarDeclaration, e.g., managing the symbol table for local variables. This limitation required duplicating code from the VarDeclaration class for implementing the desired behavior.

An IteratorBody AST node represents the body of an iterator. The body contains an iterator variable declaration, an optional accumulator variable declaration, an optional Boolean expression (also known as *filter*), and the *body* expression. The accumulator variable is optional as iterators that follow the general forms 2 and 3 do not have accumulators. In case that a new form needs to be introduced (e.g., different number or type of arguments) it is possible to create a customized definition for the body either by extending the IteratorBody class or creating a different body node.

Implementing iterator variable scope was challenging as depending on the type of the variable (*accumulator* or *iterator*) the scope variable varies. The scope of an accumulator variable is only the body expression. The scope of the iterator variable is the optional Boolean (or filter), and the body expression. The solution was to create a set of equations that assign a value to the *lookupVariable* attribute of the iterator variables and body expressions depending on the associated target scope. For example, during name checking analysis, CJC will look up a variable contained in the *body* expression in both the *accumulator* and *iterator* variable declaration AST nodes. If CJC does not find the name of the variable in any of those nodes, it will recursively search for the variable in their parent nodes. In the case of a *filter* expression, CJC will only start looking up the variable in the *iterator* variable AST node. This feature allows to have nested iterators and nested variable scopes as well. The Java keyword *this* can be used to access a shadowed iteration operator variable.

```
//@ [ x := intArray=>\any(int x; this.x > 0) ]
```

Three main types of type checking are performed. The first one is checking that the *receiver* is type-compatible with Iterable class, Array, or String. The second is to the check if the iterator variable is type-compatible with the *receiver*. Finally, if the iterator has an *accumulator* variable its corresponding assignment expression (E_1) must be type-compatible with the type of the accumulator variable.

The type check and name check method implementations are defined in the AbstractIterator class. They support the three different types of iterator general forms, with the exception of Form 3, which requires checking that the body expression is of type Boolean. Additional behavior can be added by overriding the current implementation (e.g., adding an extra type check rule for Form 3).

New iteration operators can be easily introduced by extending IteratorBody. As an example, lets us describe the implementation of *any* iteration operator.

First, a new AnyOperator node that extends AbstractIterator is defined as follow:

```
AnyOperator:AbstractIterator;
```

Then, new terminal and parser rules are defined as follow:

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

iterate_operator_caller.c ANYOP  iterate_operator_body.b
{: return new AnyOperator(c, ANYOP, b); :}
```

Finally, the behavior of the new AST node is defined:

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

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

The first AST behavior equation defines the return type of the iterator as different iterators have different return types, which in this case is an Iterator type. The second definition extends the implementation of the type check method verifying that the *body* expression is of type Boolean, as the *any* iterator follows the general form 3. As shown above, in just a couple of lines a complete iteration operator was defined addressing one of the main requirements of this feature.

5.3.4 Discussion

An additional implementation approach that required defining iterators in an external file was considered before the final implementation of the iteration operators. This approach consisted of a single IterateOperator AST node that behaves as a Java method access. Every operator needs to be declared in a similar fashion to a method declaration in Java. The iteration operator “signature” is defined in an external file (iteration operator library) adding behavior to the IterateOperator node to check valid iteration operator calls.

The advantage of this approach is that only a single iteration operator node needs to be implemented and new iteration operators can be declared in an external file without having to recompile the checker. The main disadvantages are the need of reading an external file every time that an iteration operator is processed and the limitation of having all the iteration behavior implementation in a single node, decreasing node cohesiveness. Even though the final approach required a complete implementation cycle for each new iteration operator introduced in CJC, it has several advantages over the first approach. The implemented approach provides a mechanism to easily implement new iterators by extending the AbstractIterator class, which contains most of the implementation for type and name checking. It was possible to reuse the behavior of some of the JastAddJ classes such as Access, ParameterDeclaration, and VarDeclaration. Although this approach required some code duplication (e.g., VarDeclaration name checks) the amount of duplicated code was minimal and it provided a powerful framework for constructing highly customizable iteration operators.

5.4 CONCURRENT ASSIGNMENTS

In CleanJava, a *concurrent assignment* is used to write *intended functions*. Structurally, a concurrent assignment is similar to a Java assignment statement in the sense that it is composed of expressions, and contrary to expressions they are not evaluated to a single value and therefore do not have a data type. Semantically, a concurrent assignment denotes the mapping of one program state to another. It represents a function that maps a location (e.g., variable) to a value.

```
//@ [ x, y, z := 1, 2, 3 ]
```

Figure 5.13: Example of simple concurrent assignment.

There are different types of concurrent assignments including simple concurrent assignments, conditional concurrent assignments, splitting definitions, and sequential compositions.

Simple Concurrent Assignment

A *simple concurrent assignment* is a concurrent assignment that does not contain a condition or constraint (see Figure 5.13). The function denoted by a simple concurrent assignment is a *total function*. A simple concurrent assignment has the form [L_1, L_2, ...,L_n := E_1, E_2, ..., E_n], where L_i is a location expression (e.g., variable) and E_i is a value expression. A well-formed simple concurrent assignment must satisfy the following conditions:

- L_i and E_i expressions must be well formed
- Each location entry must be unique (no location duplicates)
- The concurrent assignment must be balanced (same number of locations and values)
- A value expression must be assignment-compatible with its location expression (their types must be compatible following the rules of assignment compatibility in Java) [16].

A simple concurrent assignment can have different semantic interpretations: *value semantics* and *reference semantics*. Value semantics express that the new value of L_i is equivalent to E_i and reference semantics express that L_i refers to, or is the same object as E_i. Value semantics are analogous to the *equals* relation in Java and reference semantics to the object equality (==). Value semantics are represented by the AST node SimpleConcurrentAssignment and reference semantics by the AST node SimpleConcurrentAssignmentByRef. A simple concurrent assignment by reference has the general form [L_1, L_2, ...,L_n @= E_1, E_2, ..., E_n], where $L_i$is a location expression (e.g., variable) and E_i is a value expression. When using reference semantics all L_i's must be of reference types.

It is possible to split the definition of an intended function using *split definitions*. Split definitions are mostly used to improve the presentation and readability of an intended function by splitting the concurrent assignment into a list of location-value tuples (see Figure 5.14). The same rules of well-formness for simple concurrent assignments also apply to split concurrent assignments (e.g., unique location expressions).A split definition has the form [A_1, A_2, …, A_n], where A_i is a simple concurrent assignment.

```
/*@ [ x > 0 -> x := 1 \,
              y := 2 \,
              z := 3 \else
      \I]
@*/
```

Figure 5.14: Example of an intended function with a split concurrent assignment and an identity statement.

The *identity* statement (I) is a special kind of simple concurrent assignment that denotes no change in the state of a program. An identity statement is typically used in a conditional concurrent assignment (see next section) as the default option if no condition is met (see Figure 5.14).

Conditional Concurrent Assignment

Another major type of concurrent assignments is a *conditional concurrent assignment*, which represents a *partial function* defined upon a subset of initial states (see Figure 5.15). Each condition of the conditional concurrent assignment is evaluated sequentially from first to last until a condition is met providing a complete definition of the function.

```
/*@
*  [ x > 0 -> y := 1 \else
*  x< 0 -> y := -1 \else
*  \I ]
@*/
```

Figure 5.15: Example of a conditional concurrent assignment.

A conditional concurrent assignment has the general form [B_1 ->A_1 | B_2 ->A_2 | ... | B_n->A_n], where B_i known as *condition* is a Boolean expression and A_i known as *then* expression could be a simple concurrent assignment or an *Identity* statement. The last B_n condition can be replaced with the keyword *otherwise*, similar to an *else* keyword in a Java if-else statement. A conditional concurrent assignment is then composed of pairs of *condition* and *then* expressions, and an optional *otherwise* expression. There are two main types of conditional concurrent assignments: *deterministic* conditional concurrent assignment, which is known just as conditional concurrent assignment, and *non-deterministic*

conditional concurrent assignment. The main differences are that non-deterministic conditional concurrent assignments use a "," symbol instead of a "|" to separate expression pairs, and conditions are evaluated differently; if conditions are not mutually exclusive and there is more than one condition that holds, one condition is chosen non-deterministically among the conditions that hold. In deterministic conditional concurrent assignments conditions are chosen deterministically, that is, the first condition that holds.

Sequential Composition

CleanJava also provides a mechanism to define an intended function by composing other intended functions known as *sequential composition* (see Figure 5.16). It has the form $[A_1; A_2; \ldots; A_n]$, where A_i denotes a concurrent assignment.

```
//@ [ x, y := 3, 10 ]
//@ [ x := 2 ]
//@ [ x := 1]
```

```
/*@
 *  [ x, y := 3, 10;
 *    x := 2;
 *    x := 1; ]
@*/
```

Figure 5.16: Example of a sequential composition (right). Code examples are analogous.

5.4.2 Challenges

There are two main problems to solve for implementing concurrent assignments. The first one is to create a design that favors reusability of code, as there are several variations of concurrent assignments. The second problem is checking well-formedness of concurrent assignments. This process involves different types of checks including type compatibility, name checking, and equation balance checking. Also, depending on the type of concurrent assignment different additional checks may be required, such as checking reference types on a simple concurrent assignment by reference.

5.4.3 Implementation

In CJC, concurrent assignments are organized into class hierarchy of AST nodes. The parent node of a concurrent assignment is the abstract class *CJStmt* which represents a CleanJava annotation. CJStmt is the root node of a CleanJava specification. As discussed in Section 5.1, CJC introduced a set

of specialized class members and statements such as CJMethodDecl and AnnotatedStmt that contain an annotation node of type CJStmt. An IntendedFunction AST node is introduced to represent an intended function. IntendedFunction extends CJStmt allowing an intended function to be directly associated with a class member or statement node (see Figure 5.17). IntendedFunction contains a set of AbstractConcurrentAssignment nodes; this allows combining several concurrent assignment statements to create the definition of an intended function.

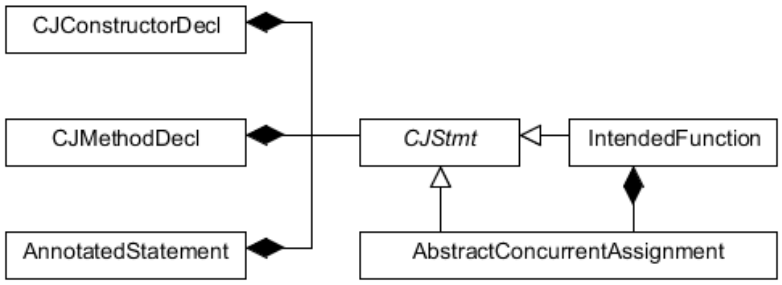


Figure 5.17: An intended function is composed of a set of concurrent assignments.

The two main types of concurrent assignments are represented by two abstract classes AbstractSimpleConcurrentAssignment and AbstractCondConcurrentAssignment (see Figure 5.18). The concrete implementations of these classes represent the different variations of concurrent assignments.

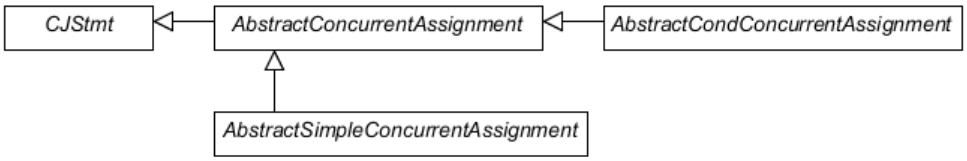


Figure 5.18: Abstract AST classes representing the main types of concurrent assignments.

In order to simplify parsing of concurrent assignments, it was necessary to introduce CJC-specific syntax definitions (see Table 5.2). For example, to split definitions in CJC a symbol "\," is used instead of the "," from CleanJava. The following table contains a list of CJC-specific symbols.

Table 5.2: CJC-specific symbols used in CleanJava.

CJC Syntax	CleanJava Syntax	Feature
\I	I	Identity statement
\else	|	Else expression in deterministic conditional concurrent assignments
\nelse	,	Else expression in non-deterministic conditional concurrent assignments
\,	,	Splitting definitions
\else x > 0 -> x := 1 \else x := -1	| otherwise x > 0 -> x := 1 | otherwise x := -1	Last omitted condition in a conditional concurrent assignment
@:	&:	Reference semantics

Simple Concurrent Assignments

In CJC there are three main types of simple concurrent assignments: a simple concurrent assignment, an identity statement, and a split definition. These types are represented by extending the abstract class AbstractSimpleConcurrentAssignment class (see Figure 5.19). The abstract simple concurrent assignment class is the only class that can be used in a *then* expression of a conditional concurrent assignment; this prevents having nested conditional concurrent assignments, thus complying with the CleanJava language design.

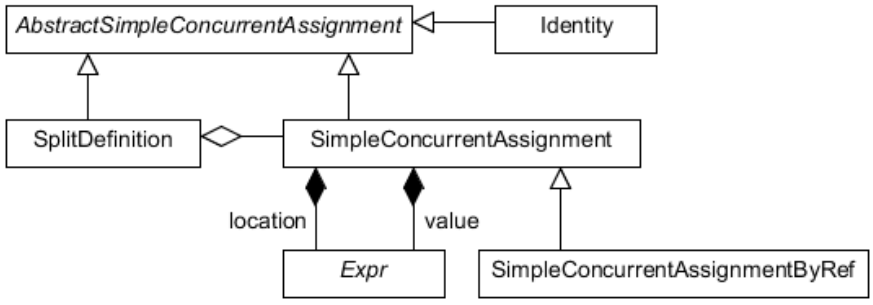


Figure 5.19: AST classes representing the types of simple concurrent assignments.

Although structurally a simple concurrent assignment looks similar to an assignment expression, semantically they are different. However, it is possible to reuse some of the static checks from the assignment expression defined in the Type Check module in JastAddJ 4. Differently from an assignment expression, a concurrent assignment can have a list of expressions in both sides of the assignment operators. CJC models a simple concurrent assignment as tuple of *location* and *value* Expr lists (see

Figure 5.19). CJC checks that both lists have the same number of expression nodes; if not, it throws an error. Name checking is handled by JastAddJ using the default behavior of Expr. Type checking is handled by CJC, as it needs to check that every pair of *location-value* expressions is type-compatible. This was done by accessing the *type* attribute of the expressions.

In order to implement simple concurrent assignments by reference, a new AST class SimpleConcurrentAssignmentByRefis introduced as a subclass of SimpleConcurrentAssignment. SimpleConcurrentAssignmentByRef adds an additional constraint to the type checker to ensure that all *location* expressions have a reference data type. This is done by checking that the type of the expression is not a primitive type. The following statement is part of the type check method for simple concurrent assignments located in the CleanJavaStatement behavior module. If a location expression (*loc*) is primitive, CJC returns an error.

```
if(loc.type().isPrimitive())
      error("cannot use " + loc + " of primitive "+
            loc.type()+" type using referential semantics");
```

SplitDefinition represents a split definition of a simple concurrent assignment. It contains a set of SimpleConcurrentAssignment nodes. SplitDefinition combines the *location* and *value* expressions from the partial definition nodes to create a single SimpleConcurrentAssignment object. SplitDefinition checks that all the partial definitions use the same kind of semantics; if not, it throws an error. It also verifies that the created object is well-formed by calling the respective SimpleConcurrentAssignment checks. Because a simple concurrent assignment can be defined by a single partial definition, all simple concurrent assignments except for identity statements (see below) are really split definitions in CJC.

The identity AST class represents an identity statement, which is a special kind of simple concurrent assignment that denotes no change in the state of a program. This class does not perform any type of checking. It is usually used by conditional concurrent assignments as the last *then* expression.

Conditional Concurrent Assignments

The design of conditional concurrent assignment is similar to that of the single concurrent assignment. An abstract node AbstractCondConcurrentAssignment is used to represent a conditional

concurrent assignment; a pair of concrete classes represent its variations (see Figure 5.20).AbstractCondConcurrentAssignment contains a set of ConditionThenExprPair nodes. This node represents a pair of *condition-then* expressions; *condition* expression is of type Expr and *then* expression is of type AbstractSimpleConcurrentAssignment. In this way, all variations of conditional concurrent assignments contain a list of *condition-then* expression pairs.

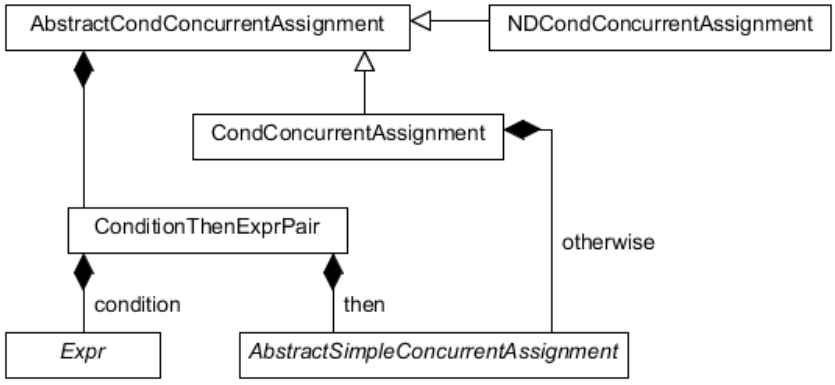


Figure 5.20: AST classes representing the types of conditional concurrent assignments.

The class CondConcurrentAssignment represents a deterministic conditional concurrent assignment and NDCondConcurrentAssignment represents a non-deterministic conditional concurrent assignment. The main difference between these two variations of conditional concurrent assignments is that a non-deterministic concurrent assignment does not support an *otherwise* expression, as conditions are not evaluated in any particular order. As shown in Figure 5.20, CondConcurrentAssignmentcan have an optional *otherwise* expression of type AbstractSimpleConcurrentAssignment. Optional nodes such as *otherwise* expressions are defined in the abstract grammar specification as follows.

```
CondConcurrentAssignment:AbstractCondConcurrentAssignment ::=
    [Else:AbstractSimpleConcurrentAssignment];
```

This abstract grammar rule defines a CondConcurrentAssignment AST class as a subclass of an AbstractCondConcurrentAssignment class with an optional child enclosed within “[“ and “]”symbols of type AbstractSimpleConcurrentAssignment named “Else”.

5.4.4 Discussion

Defining a variation of a concurrent assignment simply consists of implementing the concrete class of the respective abstract concurrent assignment. Splitting and combining definitions can be easily done by defining lists of concurrent assignment objects and performing simple list operations, such as combining partial definitions into a single simple concurrent assignment. It is possible to implement most of the static checks such as type and name checking by reusing behavior implementation from the JastAddJ framework. Using abstract classes also reduced the amount of implemented code by factoring out common features.

5.5 CLEANJAVA TOOLS

CleanJava tools are a set of programs that facilitate the development of Java programs annotated with CleanJava. CleanJava tools include a CleanJava checker, a CleanJava compiler, and two visualization programs, PrettyPrinter and AST Viewer (see Figure 5.21). CleanJava tools are Java programs that can be invoked either as regular Java program with parameters or using a batch file. The current implementation of CJC only contains batch files for Microsoft Windows using a .bat file extension.

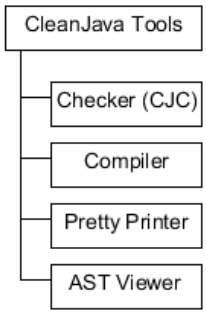


Figure 5.21: CleanJava tools.

5.5.1 CleanJava Checker and CleanJava Compiler

CleanJava checker is the main program of the CJC framework. CleanJava checker extends the Frontend class from JastAddJ. The checker works similar to the javac compiler invoked from a command prompt except that it does not produce Java bytecode. The CleanJava checker has similar options to javac such as version, classpath, help, etc. It receives a set of Java files as parameters, invokes

the CleanJava parser that builds an AST, and performs several static checks such as type checking. If any, it returns a list of compilation errors (syntax or semantic errors).

The CleanJava compiler produces bytecode from Java programs annotated with CleanJava. It extends the frontend or compiler program from the JastAddJ 7 backend component. A new component CJC backend was created to import the specification files from the JastAddJ 7 backend (see Figure 5.22). This component does not add any new feature to CJC, it was created only to support the frontend-backend architecture of JastAddJ.

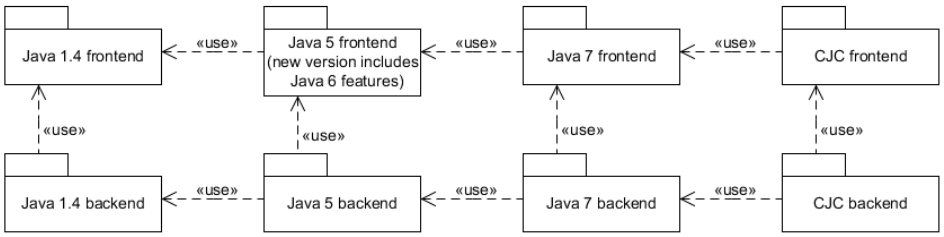


Figure 5.22: CJC frontend contains a CleanJava checker; CJC backend contains a CleanJava compiler.

5.5.2 Pretty printer (PrettyPrinter)

Pretty printer returns a standardized view of Java programs annotated with CleanJava. It prints CleanJava annotations using the CleanJava presentation syntax instead of the CJC syntax (see Figure 5.23). It also performs static checking by calling the CleanJava checker.

```
/*@ [ x> 0 -> x := 3
    \else x > 10 -> x := 20
    \else \I] @*/
```

```
/*@ [ x> 0 -> x := 3 |
    x> 10 -> x := 20 |
    I]
@*/
```

Figure 5.23: Example of an annotation using CJC syntax (left) and CleanJava presentation syntax (right).

5.5.3 AST viewer (DumpTree)

The AST viewer displays the AST of the parsed source code. This tool is especially useful to developers for analyzing the structure of CleanJava annotations.

Chapter 6: The JastAdd Specification Generator Framework

In addition to the CJC tool described in the previous chapters, this research work includes the development of JastAdd Specification Generator Framework (JASG), a framework that facilitates the construction of JastAdd components. JASG allows creating JastAdd specification files from XML-based templates and generating a parser API from an existing JastAdd component.

6.1 MOTIVATION

Although JastAddJ significantly eases and accelerates the implementation process of a Java compiler extension such as CJC, one of its main weaknesses is a limited amount of API documentation. Building a construct in JastAddJ requires knowing the location and purpose of many different types of specification rules. Specification rules can be distributed among different JastAdd components and modules; this can make it difficult to locate specific rules, especially if their names are not self-descriptive. It could be particularly difficult to find behavior specifications such as inherited AST attributes, as they can be defined declaratively in several specification files. For example, creating a new AST node that extends Expr requires implementing behavior defined in JastAddJ 4 by several different modules such as Name Analysis, Type Analysis, Prettyprinter, etc. Previous versions of JastAddJ 7 did not have a formal AST class API making it difficult to find related AST class information such as inherited attributes and methods and the location of their specification definitions. The only way to find such information in a single place was by accessing the generated AST Java class file. JastAddJ 7 introduced a tool called RagDoll that generates a XML-based API for the AST classes using JavaDoc annotations. The generated API contains a list of all the attributes and methods of each AST node in a component including inherited attributes and methods. It also includes a description of the AST class members and the location of the member declaration (e.g., behavior specification). RagDoll facilitated to improve the development process of CJC by reducing the learning curve of the JastAddJ AST class API. It also facilitated further extension of CJC by automatically generating API documentation for CJC. However, it is still difficult to locate specific parser and scanner rules declared in different JastAddJ components and modules.

Another difficulty when creating or extending a JastAdd component is getting familiar with the notation and structure of the different specification file types. Usually a JastAdd component is composed of at least four different types of specification files. Implementing a new feature in a JastAdd component requires background knowledge on the different types of specification notations, thus increasingly affecting the learning curve of the JastAdd framework.

In order to tackle the above mentioned problems, the JastAdd Specification Generator (JASG) framework was implemented as a multi-purpose solution. The first purpose is to compile documentation from all the JastAdd components needed to build or extend a component and make this documentation available in a single place. The second solution is to facilitate the creation of new specification files by providing specification templates and specification construction wizards. The JASG framework will consist of a set of tools that facilitate project documentation generation and creation of new language features.

The following sections explain the features supported by JASG along with their implementation challenges and solutions. The last section illustrates how JASG can be used to implement a JastAdd component extension using CJC as an example.

6.1 JastAdd Documentation Generator (JADG)

As part of the documentation generation solution, JASG provides a tool (JADG) for generating parser and scanner rule directories for JastAdd components. This tool allows centralizing parser and scanner information from a *workspace* or a set of JastAdd components that are used to build a new JastAdd component.

6.1.1 Challenges

Implementing a tool that generates a directory of the scanner and parser rules contained in different JastAdd components and modules has two main challenges: obtaining the data from different sources such as components or modules and presenting the data. Presenting the data is more interesting, as the purpose of this tool is to facilitate locating parsing rules during a feature implementation not just presenting a merged list of scanner and parser rules from different sources.

6.1.2 Implementation

Our approach is to develop a Java program that compiles and organizes the terminal and non-terminal parser rules of a workspace into a XML file. JADG provides a CSS and XSL files to allow viewing the XML file as a styled web page. The documentation generator program requires pre-compiling the target JastAdd component. JADG uses as input files a set of intermediate scanner and parser specification files generated by the JastAdd framework during compilation (see Figure 6.1). These intermediate specification files are composed of fragments of specification files defined in different components and/or modules.

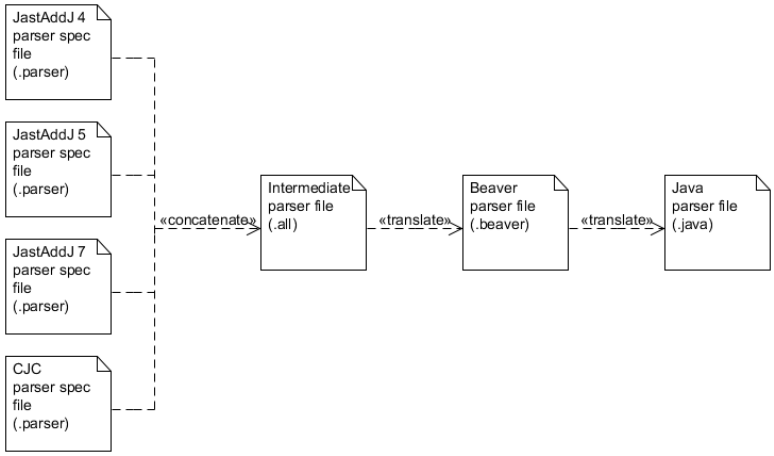


Figure 6.1: JADG uses as input an intermediate specification file generated by JastAdd.

The documentation generator collects the intermediate files from the component and parses them to create a *model* of the specification files. The model consists of a set of collections that represent the elements of the specification files such as terminals, rule names, and rule definitions. The implementation consists of several classes such as a JastAddParserReader class that uses JastAddParser from the JastAdd framework to parse the intermediate parser specification file and create a parser model object (see Figure 6.2). JFlexSpecificationParser is used to parse the intermediate scanner specification file and create a scanner model object containing a list of tokens.

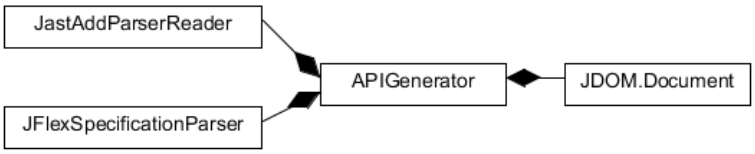


Figure 6.2: Main components of the parser generator tool.

Model objects are used to build a JDOM Document object. JDOM is a Java API that allows accessing, manipulating, and outputting XML data from a Java program [17]. JDOM is a light-weight alternative to other standards such as Simple API for XML (SAX) and Document Object Model (DOM) which can also be integrated with DOM and SAX Java implementations. JDOM allows random access without allocating the complete XML document in memory. The elements of an XML tree node are represented by different JDOM Java classes such as Document, Element, and Attribute. JADG creates a representation or model of the XML tree node using the JDOM API. The XML document model is populated with the contents of the specification model objects. The XML document model contains two main types of elements: terminals and non-terminals. Non-terminals are composed of a set of rule definitions; this allows locating rules defined in several components or modules in a single XML node. Once the XML document model is constructed, JDOM generates an XML output file in a target directory (see Figure 6.3).

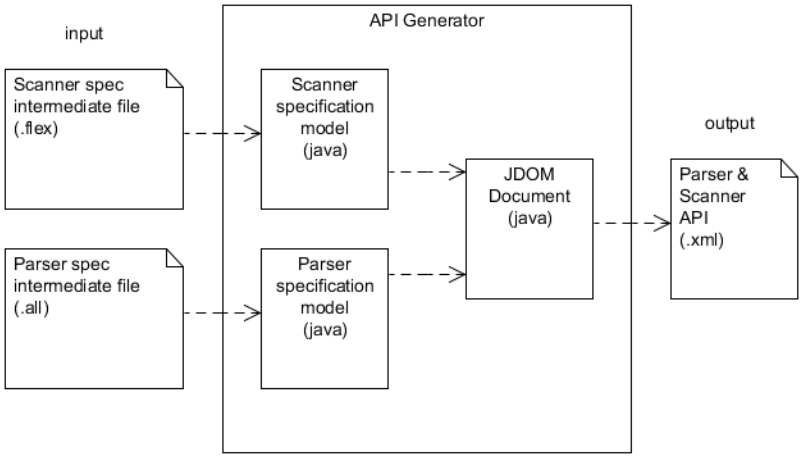
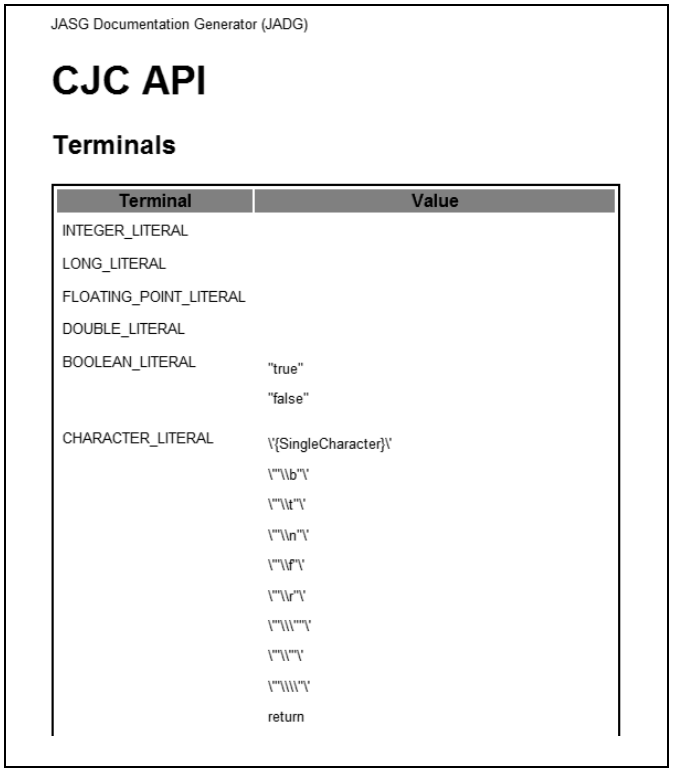


Figure 6.3: Data transformation in JADG.

The final output of JADG is a XML file containing the scanner and parser rules from a JastAdd component and a set of CSS and XSL files that define the structure and presentation style of the output data (see Figure 6.4 and Figure 6.5).



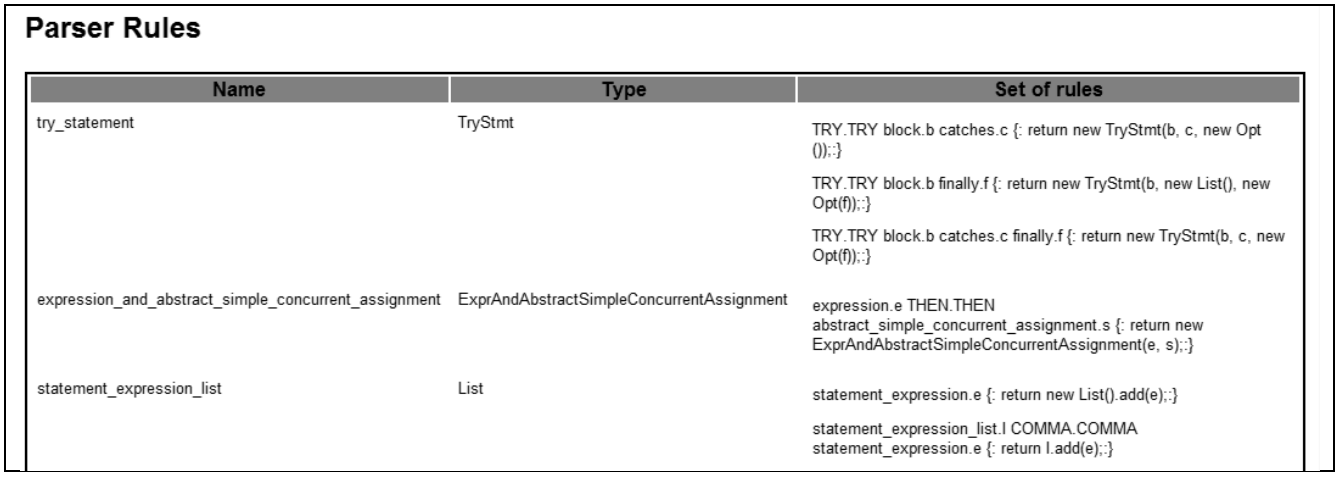
JASG Documentation Generator (JADG)

CJC API

Terminals

Terminal	Value
INTEGER_LITERAL	
LONG_LITERAL	
FLOATING_POINT_LITERAL	
DOUBLE_LITERAL	
BOOLEAN_LITERAL	"true"
	"false"
CHARACTER_LITERAL	\'{SingleCharacter}\'
	\'"\\b"\'
	\'"\\t"\'
	\'"\\n"\'
	\'"\\f"\'
	\'"\\r"\'
	\'"\\\""\'
	\'"\\'"\'
	\'"\\\\"\'
	return

Figure 6.4: Snapshot of a parser and scanner rule directory generated by JADG showing lexical rules.



Parser Rules

Name	Type	Set of rules
try_statement	TryStmt	TRY.TRY block.b catches.c {: return new TryStmt(b, c, new Opt ());:}
		TRY.TRY block.b finally.f {: return new TryStmt(b, new List(), new Opt(f));:}
		TRY.TRY block.b catches.c finally.f {: return new TryStmt(b, c, new Opt(f));:}
expression_and_abstract_simple_concurrent_assignment	ExprAndAbstractSimpleConcurrentAssignment	expression.e THEN.THEN abstract_simple_concurrent_assignment.s {: return new ExprAndAbstractSimpleConcurrentAssignment(e, s);:}
statement_expression_list	List	statement_expression.e {: return new List().add(e);:}
		statement_expression_list.l COMMA.COMMA statement_expression.e {: return l.add(e);:}

Figure 6.5: Snapshot of a parser rule directory generated by JADG.

6.2 JastAdd Specification Generator (JASG)

The JastAdd Specification Generator tool facilitates the creation of JastAdd component specifications by generating them from XML-based specification template files. The main purpose of this tool is to reduce the learning curve of the different specification notations used by the JastAdd framework. JASG allows defining a JastAdd construct using XML as an intermediate specification notation known as *JASG specification language* (see Figure 6.6). This notation supports writing specification rules for four of the most common types of specification files in JastAdd: JFlex, Beaver, AST grammar, and JastAdd aspects (behavior specifications). The JASG specification allows defining different types of specification rules in a single JASG specification file.

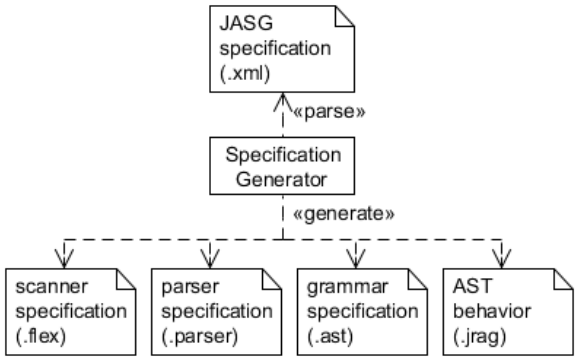


Figure 6.6: Creation of JastAdd specification files using JASG.

6.2.1 Challenges

The implementation of the JASG specification generator has two main challenges: (a) defining a notation for the JASG specification that represents the four main types of specifications used in JastAdd and (b) generating corresponding specification files from a JASG specification file. Creating JastAdd specifications using a single JASG specification notation should facilitate the creation of JastAdd constructs, especially for inexperienced JastAdd developers.

6.2.2 JASG Specification Notation

JASG specifications are used to aid the creation of specification rules by providing specification templates and semi-automatic creation of specification rules using XML. A JASG specification consists of a XML file that contains the different specification rules to build a new JastAdd construct. A XML

DTD defines the structure of the JASG specification document. The DTD definition includes the grammar of the different specification notations. For example, the following DTD element definition represents a parser rule.

```
<!ELEMENT parserRule (pe_idUse?, pe_idDecl, pe_definition*)>
```

The parser rule element contains an optional return type (pe_idUse), rule name (pe_idDecl), and a set of rule definitions (pe_definition).

```
<parserRule>
    <pe_idUse>Collection</pe_idUse>
    <pe_idDecl>collect</pe_idDecl>
    <pe_definition>
        <pe_element>
            <pe_idUse>expression</pe_idUse>
            <pe_name>a</pe_name>
        </pe_element>
        <pe_element>
            <pe_idUse>expression</pe_idUse>
            <pe_name>b</pe_name>
        </pe_element>
        <pe_code>return new ArrayList();</pe_code>
    </pe_definition>
</parserRule>
```

Figure 6.7: Example of a parser rule definition using JASG specification.

In the example shown in Figure 6.7 a parser rule *collection* of type *Collection* is defined. The definition contains two elements and a segment of code associated with this rule.

A JASG specification can contain zero or more specification file declarations and must have a *namespace* declaration; JASG generated specification files are placed in a new directory with the namespace name in the target directory. Namespaces are used to group a set of related specifications files. Namespaces are analogous to modules in CJC.

```
<!ELEMENT jasg (nameSpace,parser?,scanner?,AST?,ASTBehavior?)>
```

A specification file (e.g., parser, scanner, AST) may contain a name, a template type, and a set of specification rules as follows as shown in Figure 6.8.

```
<!ELEMENT parser (fileName?,template?,parserRules?)>
<!ELEMENT scanner (fileName?,template?,scannerRuleSets?)>
<!ELEMENT AST (fileName?,ast_definitions?)>
<!ELEMENT ASTBehavior (fileName?,aspects?)>
```

Figure 6.8: DTD definition for JastAdd specification files.

If a file name is omitted, JASG provides a default name to the specification file based on the name space value of the *jasg* element. The *template* element allows defining different types of parser and scanner specification files; JastAdd allows using different parser and scanner generators such as beaver, JavaCC and CUP, although the current implementation of JASG only supports beaver and JFflex for the parser and scanner respectively. If *fileName* and *template* elements are not defined, JASG assigns a default value; *fileName* gets the value of the *namespace* and beaver and JFlex are used as the default parser and scanner generators. If no rules are defined in the JASG specification, JASG creates a skeleton specification file. For example, for AST behavior specifications, JASG creates a file containing an empty aspect declaration with the name of the name space. JASG specifications are validated by JASG against a jasg.dtd file to check well-formedness. JASG contains a set of JASG specification templates that can be used to create new specification rules.

6.2.3 Specification Generation

The specification generation process consists of parsing a JASG XML specification file and generating a set of JastAdd specification files (e.g., parser and scanner). The specification generator tool consists of set of Java classes that includes a XML parser (XMLParser), a specification generator (SpecificationGenerator), and document factories (e.g., BeaverDocumentFactory). The XMLParser class parses the JASG specification and creates a JDOM Document object that represents the specification document. Next, XMLParser passes each *Element* child of the JDOM Document root that represents a specification document (e.g., parser specification) to the SpecificationGenerator class. The SpecificationGenerator class creates a document factory for each particular type of specification

document (e.g., beaver specification) generating a *model* of each specification document. Finally, SpecificationGenerator uses a FileFactory to generate an output specification file from the specification document model (see Figure 6.9). Figure 6.10 shows a parser specification file created by JASG after processing a JASG specification. The parser specification example contains a rule specified in the fragment of code presented in Figure 6.7.

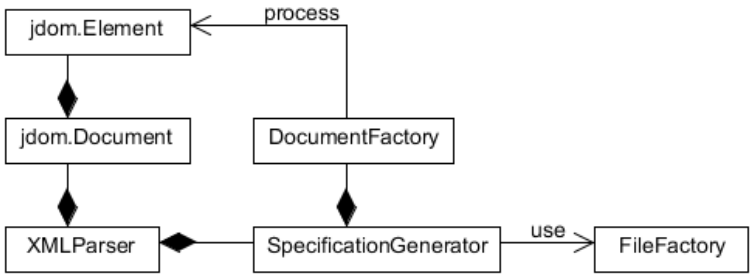


Figure 6.9: Generating JastAdd specification files from a JASG specification.

```
1  //parser file generated by JASG using beaver template
2  Collection any = expression.a expression.b  {: return new ArrayList(); :}
3    | expression.a  {: return new ArrayList(); :};
4
5  Collection collect = expression.a expression.b  {: return new ArrayList(); :};
6
7
```

Figure 6.10: Example of a parser specification file created by JASG.

The most interesting component of the specification generator is the document factory. There is a different document factory for each specification type (e.g., parser and AST) and specification template (e.g., beaver and CUP). The parser and scanner factories are represented by a set of Java interfaces ParserDocumentFactory and ScannerDocumentFactory respectively; a concrete set of Java classes represent the *template* or specific type of parser and scanner specification (see Figure 6.11). For example, the class BeaverFactory implements the ParserDocumentFactory acting as a template for the construction of beaver parser specifications. This design allows creating new specification templates such as CupFactory. The document factory builds a complete specification document *model* using a StringBuffer which includes required specification headers, footers, and specification rules. The

document factory creates complete specification rules that include symbols such as (=) and (;) based on the type of template.

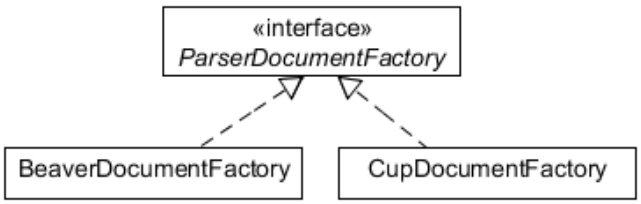


Figure 6.11: Classes representing a parser document factory and its different variations (templates).

6.3 DISCUSSION

JASG framework provides a set of tools to facilitate the construction of JastAdd components. It allows generating parser and scanner APIs providing a mechanism to easily find scanner and parser rules in a JastAdd project, especially those with several JastAdd components and/or modules. Although a JASG specification can be more verbose than a traditional specification document because it uses markup, it allows creating specification documents for different specification types using the same notation. JASG specifications are not intended to substitute traditional JastAdd specifications in the development process of JastAdd projects; instead they are intended to aid inexperienced JastAdd developers to create basic specification files using JASG specification templates. They can also be used to quickly create a JastAdd component or module boilerplate. The current version of JASG includes a JastAdd component boilerplate and a JASG specification template for creating general-purpose modules.

Future work may include a GUI-based JASG specification generator. The GUI-based specification generator can use the parser and scanner XML models produced by the API generator to aid in the construction of JASG specification definitions. For example, when creating a new parser rule, the GUI-based tool can provide a list of available rules to select from the parser model.

Chapter 7: Evaluation

This chapter describes the evaluation that was performed to verify that the main requirements of CJC are satisfied and the problems stated in Chapter 3 are successfully addressed. The checker was evaluated using two main criteria: extensibility and language coverage. The first part of this chapter contains case studies used to evaluate extensibility. The second part of the chapter analyzes the checker in terms of language coverage. The goal of case studies is used to analyze the implementation of new CJC extensions using the solutions described in the methodology section (see Chapter 4 and Chapter 5). The case studies cover different types of CJC extensions such as new CJC checker features and framework extensions. The findings of the studies identify a series of advantages as well as possible areas of improvements in the CJC framework. The studies also provided a way for analyzing the implementation of different CJC extensions quantitatively, e.g., number of lines of codes (LOC), implementation time, and percentage of implementation completeness. The language coverage analysis includes a list of implemented features, missing features, and related implementation complications.

7.1 Case Studies

As stated in Chapter 3, the main requirement of CJC is extensibility. The checker should facilitate the introduction of new CleanJava language features and the creation of support tools such as proof checkers. In addition to the extensibility features provided by JastAddJ, CJC provides a built-in mechanism that facilitates modifying language features that are likely to change or be extended during the development of the CleanJava language (see section 5.3). In order to evaluate the extensibility features of the CJC framework, a series of case studies was performed. The case studies consist of three different scenarios that cover three of the most common types of CJC extensions. The first scenario consists of implementing a CJC feature using a built-in extension mechanism. The second scenario consists of implementing a new CJC feature without using a built-in extension mechanism. The third scenario consists of extending the CJC framework to create a new CJC component. The first scenario is used to evaluate extensibility for features that are most likely to be changed such as iterators. The other two scenarios are used to evaluate the construction of more complex features by creating new CJC modules and/or components. Each case scenario contains a description of the problem (including design

and implementation issues), a description of the solution, and a set of observations related to the development process. The results and observations of the case studies are used to evaluate the tools and techniques proposed in this thesis in terms of extensibility. The case studies also helped to identify the types of extensions that are best supported by the framework as well as identifying opportunities for future improvement.

7.1.1 Implementing a new iteration operator

The first case scenario is for implementing a new *count* iteration operator using a built-in extension mechanism for iteration operators. The purpose of this case scenario is to evaluate the creation or modification of features of which specifications are most likely to be changed but share common functionality with an existing feature. The *count* iterator is not included in the current specification of CleanJava but share common functionality with existing iterators; it counts all the elements in a collection that satisfy a given condition. The iterator has the form *count*(*T x*; B_1; B_2), where B_1 is an optional Boolean expression with default value *true* and B_2 is a Boolean expression written in terms of the iterator variable *x*. The iterator returns an integer value, the number of elements that satisfy both conditions.

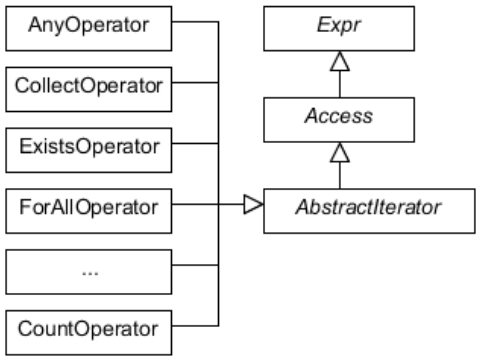


Figure 7.1: Class diagram representation of the iterator built-in extension mechanism.

Implementation

Section 5.3 describes a built-in extension mechanism for creating new iteration operators extending the abstract AST class AbstractIterateOperator (see Figure 7.1). The implementation process is straightforward as shown in the following implementation steps:

a) Define a new CountOperator AST node class as a subclass of the AbstractIterateOperator class in CLEANJAVA.ast

```
CountOperator:AbstractIterateOperator;
```

b) Define a new *count* lexical rule in CLEANJAVAscanner.flex

```
"\\count"{ return sym(Terminals.COUNTOP); }
```

c) Define a new *count* parser rule in iterator_operator.parser in the iterator module

```
Access iterator_access =
iterator_receiver COUNTOP iterator_body;
```

d) Define AST class behavior in iterator_operator.jrag included in the iterator module

Most of the static checks such as name and type checks are defined by the AbstractIterateOpearator class. Additional implementation for *count* operators includes defining the type of the iterator expression and checking that the iterator body is of type Boolean:

```
eq CountOperator.type() = typeInt();

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

e) The last step is to create a set of JUnit test cases for syntax checking (tests/ParserTest.java) and type checking (tests/TypeCheckingTest.java).

Observations

Most of the *count* iterator implementation is provided by the abstract iterator class reducing the implementation effort of the operator. The complete implementation of the *count* iterator required approximately 17 lines of code, including test cases, with an estimated development time of 1.5 hours.

This approach requires minimal understanding of the CJC framework and the different JastAdd specification notations. The *count* operator is defined in a module dedicated to iterator operators which facilitates code reusability from other existing iterators. However, using built-in extension mechanisms still require some basic knowledge of JastAdd concepts such as abstract AST nodes and attributes, as well as knowing the different types of specification notations used in CJC.

Built-in extension mechanisms are adequate for developing features that have related functionality e.g., same AST parent node. More complex features such as iterators with different base forms may require a deeper understanding of the different specification notations used by JastAdd, especially parser specifications. Still, complex behavior definitions such as behavior specifications can be implemented using traditional Java code.

7.1.2 Creating a new CleanJava extension module

The second case scenario is for implementing a new CJC module that contains two CleanJava extension mechanisms: user-defined function declarations and collection literals. ClenJava extension mechanisms allow programmers to extend the vocabulary of CleanJava by introducing custom symbols and expressions. The purpose of this case scenario is to evaluate the implementation of features that do not share common functionality with existing CJC features or cannot be implemented using the CJC built-in extension mechanism.

User-defined Functions

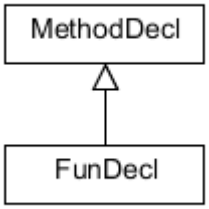
A user-defined function allows a programmer to introduce new mathematical functions that can be used for writing intended functions. A user-defined function declaration is preceded by the keyword *fun* and consists of a function signature and a body (see example in Figure 7.2). The function signature is similar to a method signature but with optional argument and return types; types are inferred statically at compile time. However, in this implementation they are allowed to be explicitly specified. The function body is composed of a Java expression, including CleanJava expressions.

```
//@ fun int abs(int x) = x >= 0 ? x : -x
```

Figure 7.2: Example of a user-defined function

User-defined functions follow Java scoping rules. Structurally, a user-defined function declaration is similar to a Java method declaration; the main difference is that user-defined functions can be declared in any section of a Java program including method bodies. This introduces some implementation challenges such as user-defined function scope definitions. Differently to iterator operators, CJC does not provide a built-in extension mechanism for implementing user-defined functions. In this case, the implementation approach is to create a *prototype* feature from an existing Java or CleanJava feature. A prototype feature is a partially-implemented feature that is based on an existing feature with similar characteristics such as structure or functionality. In this case a Java method declaration can be used to create a user-defined function declaration prototype providing an implementation start point. The prototype feature provides an initial design and code base that can be further refined to comply with the original feature specification. Creating prototype features are especially useful for implementing language features that have not been completely defined yet or whose specifications are constantly changing. A prototype feature allows estimating development time and effort of a new feature.

A user-defined function declaration (FunDecl) extends the method declaration (MethodDecl) inheriting functionality such as type and name checking as shown below.

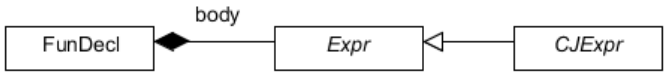


Custom structure and functionality are implemented by creating new parser and behavior specifications. A new user_defined_functions module is created to separate the implementation of the new feature. A new CJC module is defined by naming its contained specification files, e.g., <module name>.<extension> (i.e., user_defined_functions.jrag and user_defined_functions.parser) and/or by

storing these files in a dedicated module directory. In CJC most modules consist of a parser and behavior specifications; scanner and abstract syntax specifications are defined in common specification files CLEANJAVA.ast and CLEANJAVAscanner.flex to maintain the definition of the abstract grammar in a single place and facilitate searching abstract grammar rules. In addition, JASG can be used to define new modules and generate an initial set of specifications files from specification templates. JASG assigns appropriate names to the specification files (i.e., <module name>.<extension>) and places them into a new module directory. Modules that are created inside of the CJC component directory do not require to be identified in the build file unless a specific build order is required. Also, it is possible to omit specific modules from the build file in order to turn off a feature or a set of features (i.e., user-defined functions) which can be useful for testing purposes during the implementation of a new module.

```
FunDecl:MethodDecl ::= Body:Expr;
```

FunDecl has a definition of function body which is a single expression (see above). A user-defined function body can be a Java or a CleanJava expression as CJExpr is a subclass of Expr as shown below.



The first version (prototype) of the user-defined function provides similar functionality to a Java method declaration, including namespace and type check. FunDecl checks that a function name is unique and the type of its body expression is compatible with the function type. Function and argument types are not optional in this user-defined function prototype. FunDecl introduces variable scope checks for the function arguments inside a function body. Function declarations inside of blocks are not allowed at this time as FunDecl are defined as class member declarations by MethodDecl. Additional behavior specifications can be used to allow missing characteristics of a user-defined function.

Collection Literals

CleanJava provides a language construct to express a collection literal directly, that is, without having to define a specification-only method or using a CleanJava standard library method [18]. The following segment of code shows two user-defined functions with equivalent collection literals.

```
//@fun CJSet<Integer> integerCollection() = new CJSet<Integer>{10,20,30,40,50}
//@fun CJSet<Integer> integerCollection() = {10,20..50}
```

Collection literals have a similar structure to array declarations, although they differ in functionality. One advantage of the JastAdd framework is that it allows reusing only certain specifications from other features such as parser or behavior definitions. Collection literals (ColExpr) reuse the parser and scanner definitions from ArrayDeclaration but introduce new AST and behavior definitions. ColExpr is a CJExpr that has a type access (Access) and a body (ColBody) as shown below.

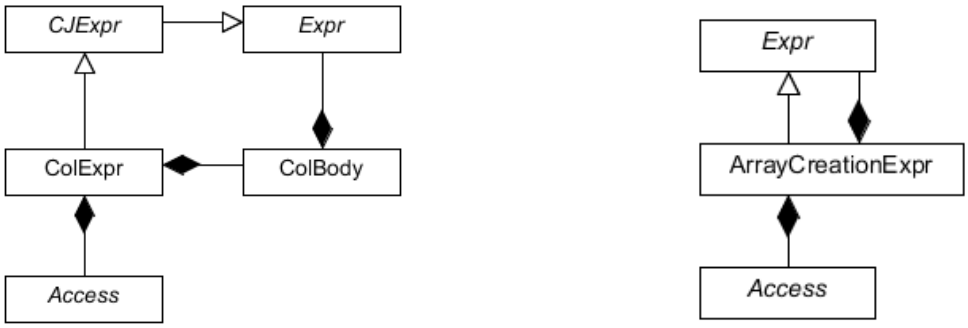


Figure 7.3: Classes for collection literals and array creation expressions.

A Collection expression (ColExpr) is similar in structure to an array creation initializer (ArrayCreationExpr) but introduces new behavior definitions that include:

- Type definition, which is a collection type defined in the CJCollection library
- Type check for evaluating that ColExpr and ColBody are type compatible
- Pretty printing for Collection Literals

The implementation of ColExpr consisted of reusing most of the parser and scanner specifications from the array creation expression and introducing new behavior specifications that included three new attributes and three method definitions with approximately 60 lines of code. The initial implementation of collection expressions do not supports optional types.

Observations

The development of the two CleanJava extension mechanisms required approximately 110 lines of code with an implementation time of around 11 hours for both implementations.

Compared to the previous case study, the implementation of user-defined functions required further knowledge of the JastAdd extension features and JastAddJ components. Still, it was possible to create a feature prototype from existing JastAddJ features, allowing to quickly identify possible design and implementation solutions, as well as estimate implementation time and effort. As shown in the implementation of these two CleanJava extension mechanisms, it is possible to create new features by extending existing solutions (feature prototypes) or by choosing parts of a solution e.g., parser to create a completely new solution with a different design and functionality. Creating a CJC module for the newly implemented feature allows organizing features by similar functionality which can facilitate the creation of related features such as new user-defined functions. JASG can be used to facilitate the construction of new CJC modules by providing specification templates and following CJC naming conventions.

However, one of the main obstacles for creating more complex features in CJC is learning the AST API. The CJC framework consists of around 335 AST classes and the behavior specifications of these classes are defined within seven different components. The inclusion of RagDoll [19] in JastAddJ 7 greatly improved the development of new features in JastAdd. It provides a web-based API that facilitates searching for specific AST classes and methods as well as their structure and relationship with other AST classes. RagDoll also provides the location of the AST method definitions which facilitates modifying such definitions or reusing the code for new feature implementations.

7.1.3 Creating a new CleanJava extension component

The last scenario is for implementing a CJC backend component that includes a CleanJava compiler to support Java bytecode generation. The component includes a CJC compiler tool that performs static checks on programs annotated in CleanJava and generates Java bytecode as a regular Java compiler. The purpose of this case study is to show how the CJC framework can be extended to create new tools from existing modules. In addition to support extensibility of language features, the CJC framework must facilitate the creation of new CJC tools and also the integration with other software development tools such as IDEs.

Implementation

Creating a new JastAdd component that extends a set of existing JastAdd components requires adding references to the target components from a boilerplate component. In this case, the target components are the JastAddJ components that provide the Java compiler functionality and the CJC frontend component that provides the CleanJava checker functionality (see Figure 7.4). JASG provides a CJC boilerplate that contains references to JastAddJ and CJC frontend components providing the functionality of a CJC static checker.

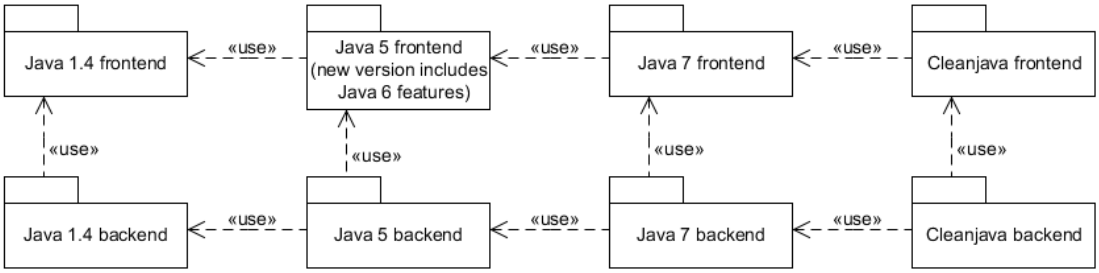


Figure 7.4: CleanJava backend and target components.

Developing a Java compiler that supports CleanJava annotations using JastAdd consists of the following steps:

a) Copying CJC boilerplate component from JASG and adding references to the JastAddJ backend components in the build file.

b) Creating a CleanJava compiler program (CleanJavaCompiler.java) using the JastAddJ7Backend compiler tool (JavaCompiler.java) as a base program and adding CJC specific functionality from the CJC frontend tool (CleanJavaChecker.java) such as customized command prompt help dialogs and support for the CJC testing framework.

c) Compiling the CJC backend component using JastAdd

The CJC backend component provides a complete Java compiler that supports static checking of programs annotated in CleanJava. Additionally, based on a JastAddJ performance analysis [9], the JastAddJ framework provides an out-of-the-box compiler that is 3 times slower than javac compiler. However, considering that performance is not a main requirement of the initial version of CJC, this can be considered a nice-to-have feature of JastAddJ.

Observations

Developing a CleanJava backend component from a CJC boilerplate required adding about 25 lines of code to the build file with a total implementation time of 2.5 hours. The development of the backend component required minimal understanding of the JastAdd specification notations, although it required moderate understanding of the CJC and JastAddJ platform, in particular, the frontend tools and build specifications.

The initial version of the CJC backend component can be further extended by introducing new modules to provide additional functionality such as embedding CJC annotations in the Java bytecode. Because the CleanJava compiler is a Java program, it is relatively easy to integrate them with other Java-based software development tools.

7.1.4 Analysis

The table below summarizes the results of the case studies presented in this chapter:

Table 7.1: Case study results

	Case 1: Count iterator	**Case 2: User-defined function declarations**	**Case 3: Collection literals**	**Case 4: CleanJava compiler**
Type of extension	CJC feature using a built-in extension mechanism	CJC module/CJC feature without using a built-in mechanism	CJC module/CJC feature without using built-in mechanism	CJC component

Total implementation time	1 hr.	5 hrs.	6 hrs.	2 hrs.
Implemented lines of code	8	50	60	25
Feature completion	100%	70%	80%	100%
Strengths of the CJC framework over different types of extensions	- Straightforward implementation of similar features - Minimal knowledge required of the JastAdd specification notation	- A huge library of existing features from the JastAddJ and CJC components providing design and code base examples - Extending a feature to provide more specific functionality	- A huge library of existing features from the JastAddJ and CJC components providing design and code base examples - Extending only parts of a solution (e.g, parser) to create custom functionality	- Combination of JastAdd components to create new component solutions (not only CJC components) - Frontend tools can be easily integrated with other Java software development tools
	Case 1: Count iterator	**Case 2: User-defined function declarations**	**Case 3: Collection literal**	**Case 4: CleanJava compiler**
Development Strategy	- Use a built-in extension mechanism for Iterators - Extend AbstractIterator class	- Create a feature prototype using a similar feature from the CJC framework to reuse its design and code base - Extend MethodDecl class from JastAddJ	- Create a feature prototype using a similar feature from the CJC framework to reuse its design and code base - Create a new AST node based on the structure of ArrayInit, which represents an array initialization expression with the form: “new” <type> “{“ <variable_initializers> “}”	- Create a CJC boiler plate component that supports the functionality of the CJC checker (boilerplate provided by JASG) - Import specification files from JastAddJ 7 backend component to support generation of Java bytecode - Create a custom compiler program using the frontend from the JastAddJ backend component
CJC framework features that address identified problems and requirements from Chapter 3	- Built-in extension mechanisms for constantly changing features - Implementation of complete features with minimal effort and development time - Creation of new features with	- Code reusability - Estimation of implementation efforts by building prototype features - Functional prototypes with minimal implementation effort - Most of the implementation effort	- Code reusability - Estimation of implementation efforts by building prototype features - Functional prototypes with minimal implementation effort - Most of the implementation effort	- Extensibility - Combining the functionality of different components can be easily performed by adding references to a build file - Frontend programs such as checkers and

	minimal understanding of the CJC framework	requires creating behavior specifications using Java thus reducing the learning curve of Java developers	requires creating behavior specifications using Java thus reducing the learning curve of Java developers	compilers are created with Java which provides easy integration with Java-based software development tools
JASG features that address identified problems and requirements from Chapter 3	Not necessary for built-in extension mechanisms	Useful for creating initial module structure and skeleton specification files	Useful for creating initial module structure and skeleton specification files	Provides a CleanJava boilerplate
Weaknesses	Not suitable solution for features that do not share similar functionality	- Need medium to high knowledge of JastAdd specification notations, especially behavior specifications - Prototype features usually don not provide complete functionality. Further refinement is necessary.	- Need medium to high knowledge of JastAdd specification notations, especially Beaver parser specifications	- Need moderate experience with ANT build files - Need medium understanding of the different component architectures, especially high understanding of needed parser and scanner specifications and their order of compilation
Future work	Adding more built-in extension mechanisms for other features with common functionality	Improve JASG to facilitate the creation of new modules (i.e., GUI that allows creating specification rules from templates)	Improve JASG to facilitate the creation of new modules (i.e., GUI that allows creating specification rules from templates)	- Create component architecture models to facilitate the integration of different components - Create documentation and standards for the build process such as naming conventions for directories and description of the build order of specifications

In addition to the features mentioned in the table, CJC provides forward compatibility with future versions of JastAddJ, thus facilitating the integration of CJC with newer versions of Java (e.g., Java 8). Also, on-going research efforts and current involvement of the industry with the JastAdd project provides a continuous improvement of the JastAdd framework including advances in performance,

quality assurance, documentation, and a growing community of JastAdd developers. Latest improvements in the JastAdd framework such as improved build performance and the creation of the RagDoll tool substantially improved the development of more complex CleanJava features reducing the implementation time and effort significantly.

7.2 CJC LANGUAGE COVERAGE

Another evaluation criterion is the number of CleanJava features covered by the CJC checker. According to the CleanJava core language specification there are 32 different features that were classified into four main groups: annotations, Java expressions, CleanJava-specific operators, and intended functions. The current version of CJC (version 0.3.5) supports 26 of the core features (81.25% of all core language features) which were tested with a set of 79 unit tests including 39 for syntax checking, 36 for semantic checking, and the rest for testing the frontend tool.

The main focus of the first version of CJC was to support the basic notation for writing intended functions, while providing a set of mechanisms and guidelines to create future extensions.

The following CleanJava features are not supported by the current version of CJC:

- Java Statement-level annotations (e.g., for loop and if-statements annotations)
- Refining intended functions using space separation and indentation notation (see Figure 7.5)
- Class field annotations
- Identifying query methods (methods with no side-effects)
- Intended function identifiers such as f_1 in

```
//@ f1:[ x := 1 ]
```

- Combining definitions using []; []; …; [] notation
- Advanced features such as class invariants, model methods, and user-defined functions (partially supported by the case study implementation).

.

```
//@ f1:[ x, y := x + 2, y + 1 ]
    //@ f2:[ x := x + 2 ]
    x = x + 2;

    //@ f3:[ y := y + 1 ]
    y++;
```

Figure 7.5: Refining intended functions with indentation notation.

The previous example contains two inner block statement annotations using indentation and blank space notation, preceded by function identifiers. In this example function *f1* is refined by using two secondary intended functions *f2* and *f3* contained in an indentation block, similar to Python notation. The scope of secondary functions is delimited by an ending blank space. Several implementation problems arise with this notation. Because Java does not allow this kind of indentation notation, it is difficult to identify an indentation block and associate this block with an intended function. In order to associate *f1* with the appropriate AST node, CJC would need to differentiate between different levels of indentation in a Java program and keep track of the number of blank spaces between statements; this is not in the scope of the current project.

Another feature that was not implemented is *query method* identification. There are different approaches towards evaluating side-effect operations in Java [20]; however implementing a feature suitable for this checker could be part of a new research topic.

Chapter 8: Conclusion

The main deliverable of this thesis work is a CleanJava checker (CJC) that parses Java programs annotated with CleanJava specifications and performs static checks such as type and syntax checks. CJC is used to help the initial design and subsequent refinement of the CleanJava language and to promote the use of Cleanroom-style formal program verification. The creation of a CleanJava checker is the first milestone towards developing a set of CleanJava support tools. The checker will be used as a base for more advanced tools such as fully automated theorem provers.

Developing the CJC tool introduced several interesting challenges. As CleanJava is still under development, the CJC tool has to be sufficiently extensible to facilitate the experimentation of new language features and support future language extensions. CJC had to understand and process Java source code as CleanJava specifications are embedded in Java source code and they can be written using a subset of the Java notation. In other words, CleanJava language is an extension of the Java language and its specifications need to be checked and interpreted in the context of a Java program.

This thesis presented a set of solutions that addressed the above challenges, focusing primarily on the design and implementation of the CleanJava core features. Two of the key solutions are to implement CJC using an extensible Java compiler as its base platform and provide built-in extension mechanisms for language features that are likely to be changed or refined during the development of the CleanJava language. JastAddJ was used as the base platform to develop CJC because of its extensibility features and to avoid implementing a new Java compiler. The main features of JastAddJ are its object-oriented modeling of AST nodes and its aspect-oriented mechanism to define AST behavior declaratively. In addition to the extensibility features provided by JastAddJ and the CJC built-in extension mechanisms, a XML-based CJC extension framework, known as JastAdd Specification Generator (JASG), was created to facilitate the creation of new CJC modules and components. Although JASG was initially created to facilitate the development of CJC tools, the framework can be also used for developing other JastAdd tools

The current version of the CJC framework (version 0.3.5) supports most of the CleanJava core features and provides a set of frontend programs to facilitate software development using CleanJava. CJC is available as open source at https://github.com/ceyeep/CJC. GitHub [21] enhances collaboration between developers by providing a source code repository, a bug tracking system, and a wiki site.

The two main frontend programs include a CleanJava checker and a compiler. The CleanJava compiler supports Java 7 and can be used as a regular Java compiler, parsing CleanJava annotations and generating Java bytecode. The frontend programs can be easily integrated in other Java software development tools as they were developed in Java.

The CJC framework was evaluated in terms of extensibility and language coverage using a case study and a language coverage analysis. The evaluation identified a series of advantages and improvement opportunities for each of the analyzed types of extensions.

The major challenges towards building CJC tools using JastAddJ was learning the different types of JastAdd specification notations and searching for construct definitions through the different specifications located in different components without a standard AST API or a more descriptive documentation. Fortunately, newer versions of JastAdd provide tools for creating AST APIs, improved documentation and available tutorials, and an increasing community of active JastAdd developers that provide constructive feedback.

8.1 RELATED WORK

The development of CJC was inspired by other language processing tools that faced similar challenges during their implementation. The most related work is JAJML, an extensible runtime assertion checker for the Java Modeling Language (JML) [22]. The main objective of JAJML was to create an extensible JML compiler. By performing a case study [23] that compared the implementation of new JML features by extending a JDT compiler (JML4) and an extensible Java compiler (JastAddJ), they showed that attribute grammars facilitated the implementation of language extensions. It also proved to be more extensible than other approaches, such as JML 4 [24].One of the disadvantages that the study found was that JastAddJ had a low guarantee for support of future Java versions. At the time of the study JastAddJ only supported Java 4 and 5 passing several years before a new JastAddJ version was

created. Currently JastAddJ supports Java 7 and the JastAddJ community is currently active issuing new compiler revisions.

JAJML served as an inspiration to CJC by using an extensible Java compiler that facilitated the creation of new language features. An example of this is SafeJML, an extension of the JML language that supports specification of safety critical Java programs. SafeJML was implemented as an extension to JAJML by introducing new language constructs [25]. However, a main difference in analysis between JAJML and CJC is that JAJML focused in the describing the differences between extending a traditional compiler versus an extensible compiler, and CJC focused in analyzing the distinct types of extensions that are most common in CJC. In addition, CJC discusses techniques to create built-in extension mechanisms to facilitate the creation of features that are constantly changing but have similar functionality. It also provides a framework to facilitate the creation of JastAdd specifications.

Before the creation of CJC, the closest work related to open-source tools that supported Cleanroom-style functional program verification was a set of tools created by Dr. Gabriel J. Ferrer called CASE tools [26]. CASE tools are used for educational purposes and include a set of programs to create and edit Cleanroom-inspired black box specifications and trace Java programs annotated with intended functions. CASE tools include a set of examples of Java programs annotated with Cleanroom specifications. Although the examples illustrate different forms of specifications using a Cleanroom-like notation, it does not define specific annotation syntax or guidelines for creating and reading specifications. CASE tools only support Java 5 and it does not mention the possibility of extension mechanisms or upcoming updates to support new Java versions.

8.2 FUTURE WORK

Possible future extensions of this work include:

- Extending the CleanJava backend to perform dynamic or runtime verification, for example by embedding executable CJC specifications into Java class files.
- Creating a CJC extension that computes a code function (the function computed or implemented by a section of code) to assist formal program verification. This extension will

also set a path towards the creation of automated verification of Java programs annotated with CleanJava.

- Supporting runtime analysis and verification.
- Integrating CJC frontend tools with an IDE such as Eclipse.
- Support for specification inheritance and importing libraries of CleanJava specifications.
- Complete implementation of CleanJava Collection library classes.
- Complete implementation of CleanJava extension mechanisms such as those partially implemented in the case study discussed in section 7.1.2.
- Support for future versions of Java (i.e., Java 8) including native Java lambda expressions.
- JASG GUI to facilitate the creation of specification rules using features such as rule suggestion and auto completion, importing and exporting JASG XML specification files.

References

[1] R. C. Linger, "Cleanroom software engineering for zero-defect sotware," in *Software Engineering, 1993. Proceedings., 15th International Conference*, 1993.

[2] S. L. Pfleeger and J. M. Atlee, Software Engineering: Theory and Practice, Upper Saddle River, NJ: Prentice Hall, 2006.

[3] V. Basili, "Software Process Improvment in the NASA Software Engineering Laboratory," 1994.

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

[5] E. Torbjorn and H. Gorel, "The JastAdd system --- modular extensible compiler construction," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 14-26, December 2007.

[6] 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.

[7] G. Klein, "JFlex - The Fast Scanner Generator for Java," 31 January 2009. [Online]. Available: http://jflex.de/. [Accessed 2 December 2013].

[8] "Beaver - a LALR Parser Generator," [Online]. Available: http://beaver.sourceforge.net/. [Accessed 2 December 2013].

[9] E. Torbjorn and H. Gorel, "The JastAdd extensible Java compiler," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, Montreal, Quebec, Canada, 2007.

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

[11] D. Megert, "Java Development Tools," 2013. [Online]. Available: http://projects.eclipse.org/projects/eclipse.jdt. [Accessed 2 December 2013].

[12] "OpenJDK," 16 November 2013. [Online]. Available: http://openjdk.java.net/. [Accessed 5 December 2013].

[13] "The GNU Compiler for the Java Programming Language," 25 November 2013. [Online]. Available: http://gcc.gnu.org/java/. [Accessed 2 December 2013].

[14] H. Gorel, "An Introductory Tutorial on JastAdd Attribute Grammars," in *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, Berlin, 2009.

[15] C. Avila and Y. Cheon, "Functional Verification of Class Invariants in CleanJava," *Innovations and Advances in Computer, Information, System Sciences, and Engineering*, vol. 152, pp. 1067-1076, 2012.

[16] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, "The Java Language Specification - Java SE 7 Edition," 28 February 2013. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf. [Accessed 5 December 2013].

[17] J. Hunter, "JDOM," [Online]. Available: http://www.jdom.org/. [Accessed 2 December 2013].

[18] M. Vela and Y. Cheon, "Enhancing the Expressiveness of the CleanJava Language," in *Technical Report 13-33, Department of Computer Science, The Univeristy of Texas at El Paso*, El Paso, TX, 2013.

[19] G. Hedin, "JastAdd - Downloads," 29 November 2013. [Online]. Available: http://jastadd.org/web/download.php. [Accessed 5 December 2013].

[20] A. Rountev, "Precise identification of side-effect-free methods in Java," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference.*, 2004.

[21] "GitHub - Build software better, together," GitHub, Inc., 2013. [Online]. Available: www.github.com. [Accessed 5 December 2013].

[22] G. Haddad, "JAJML Wiki," November 2010. [Online]. Available: http://sourceforge.net/apps/trac/jmlspecs/wiki/JAJML. [Accessed 5 December 2013].

[23] G. Haddad and G. T. Leavens, "Extensible dynamic analysis for jml: A case study with loop annotations," School of Electrical Engineering and Computer Science, Orlando, FL, 2008.

[24] A. Sarcar and Y. Cheon, "A New Eclipse-Based JML Compiler Built Using AST Merging," in *Second World Congress on Software Engineering*, Wuhan, China, 2010.

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

[26] G. J. Ferrer, "Tools for Cleanroom Software Engineering," [Online]. Available: http://ozark.hendrix.edu/~ferrer/software/cleanroom/. [Accessed 18 February 2013].

Appendix A: Syntax differences between CleanJava and CJC syntax

CJC Syntax	CleanJava Syntax
\anything	anything
\result	result
\I	I
\else	|
\nelse	,
\,	,
last omitted condition in conditional concurrent assignment x > 0 -> x := 1 \else x := -1	otherwise x > 0 -> x := 1 | otherwise x := -1
=>	->
@:	&:
\any	any
\collect	collect
\exists	exists
\forAll	forAll
\isUnique	isUnique
\iterate	iterate
\one	one
\reject	reject
\select	select

Vita

Cesar Yeep was born on December 29th, 1984 in Ciudad Juarez, Chihuahua, Mexico. He graduated from Instituto La Salle de Chihuahua High School in 2003. He attended Universidad La Salle de Chihuahua from 2003 to 2005 as an undergraduate student in Mechatronics Engineering. He continued his studies at the University of Texas at El Paso earning a Bachelor's degree in Computer Science in spring 2009. During the summer of 2007, he was awarded a research assistantship at the Data Sciences Summer Institute at the University of Illinois at Urbana-Champaign where he designed and tested a model for the Virtual Graph using Java and WEKA software. After completing his Bachelor's degree, he continued his education pursuing a Master's degree in Computer Science. During his graduate studies, he worked as a Research Assistant for the Software Specification and Verification Research Lab under the supervision of Dr. Yoonsik Cheon. He also worked as a Teaching Assistant for the Computer Science department. During the last year of his graduate studies he joined HP Enterprise Services as a Software Developer in the Healthcare division.

Permanent address: 7135 Cumbre Monserrat
Chihuahua, Chihuahua, Mexico, 31217

This thesis/dissertation was typed by Cesar Yeep.