ConGu
Checking Java Classes Against Property-Driven Algebraic Specifications

João Abreu Alexandre Caldeira
Antónia Lopes Isabel Nunes Luís S. Reis
Vasco T. Vasconcelos

DI–FCUL TR–07–7

March 28, 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

ConGu

Checking Java Classes Against Property-Driven Algebraic Specifications

João Abreu Alexandre Caldeira Antónia Lopes
Isabel Nunes Luís S. Reis Vasco T. Vasconcelos

March 28, 2007

Abstract

ConGu is a tool that supports the checking of Java classes against property-driven algebraic specifications. This document presents the specification, languages, the tool usage, and its implementation, version 1.32.

Chapter 1 describes the two specification languages: the language of abstract data types specifications, and that for defining refinement mappings between these specifications and Java classes.

Chapter 2 explains how to install and use **ConGu** tool.

And finally, Chapter 3 presents the implementation details of the tool.[1]

[1]This work was partially supported through the POSI/CHS/48015/ 2002 Project Contract Guided System Development project. Thanks are due to José Luiz Fiadeiro for many fruitful discussions that have helped putting the project together.

Chapter 1

The Specification and the Refinement Languages

1.1 Introduction

The Contract Guided System Development project aims at developing a methodology to test implementations of abstract data types against their specifications. The key idea is to reduce this problem to the run-time monitoring of classes annotated with contracts that represent the specification. In order to make this approach systematic and to allow for the development of tools, a language for specification and a language for refining these specifications into Java classes were developed; they are described in Sections 1.2 and 1.4, respectively. Refinement into Java works at the level of modules; Section 1.3 describes this notion.

1.2 The specification language

This section describes the specification language. Subsection 1.2.1 presents the general structure of specifications; Subsection 1.2.2 presents the conformance rules to which specifications must obey and that are not covered by the grammar. The grammar itself is presented in Appendix A.1.

1.2.1 General structure

According to the grammar presented in Appendix A.1 and the extra requirements on operation signatures, domains and axioms, any particular specification must obey the general structure in Figure 1.1.

The contents of each clause are described below and, in general, start with a keyword, followed by a sequence of signatures, domain conditions or axiom declarations. The **sorts** clause is the only mandatory one; the others must only be declared if they are not empty. We use the specification of a Stack in Figure 1.2 as a running example.

```
specification
    sorts field
    constructors field?
    observers field?
    derived field?
    domains field?
    axioms field?
end specification
```

Figure 1.1: The outline of a specification

```
specification
    sorts
        Stack
    constructors
        make:    —> Stack;
        push:    Stack Element —> Stack;
    observers
        peek:    Stack —>? Element;
        pop:     Stack —>? Stack;
        size:    Stack —> int;
    derived
        isEmpty: Stack;
    domains
        S: Stack;
        peek(S) if not isEmpty(S);
        pop(S)  if not isEmpty(S);
    axioms
        S: Stack;  E: Element;
        peek(push(S, E)) = E;
        pop(push(S, E))  = S;
        size(make()) = 0;
        size(push(S, E)) = 1 + size(S);
        isEmpty(S) iff size(S) = 0;
end specification
```

Figure 1.2: Specification of a stack

1.2.1.1 Sorts

This clause is used to declare the name of the sort under specification. The clause starts with keyword **sorts**, followed by the name of the sort. In the stack example these declarations are in lines 2 and 3 of Figure 1.2.

If you are specifying sort MySort this clause is declared as:

```
sorts MySort
```

1.2.1.2 Constructors

Use this clause to declare the signatures of constructor operations. Constructors form a minimal set of operations needed to build any conceivable value of the sort. The clause starts with keyword **constructors** followed by a sequence of signatures. The signature for a constructor of sort S is of a particular form: both its first parameter (if present) and its result must be of sort S. Lines 4–6 in Figure 1.2 reflect this structure.

1.2.1.3 Observers

This clause starts with keyword **observers**, and is followed by a list of operation or/and predicate signatures. Observers are the operations used to analyze (to dissect, to disassemble, to deconstruct) a given value. Observers must have at least one parameter. Also, the first parameter must be the sort under specification. Lines 7–10 of our example illustrate this structure.

1.2.1.4 Derived

Derived operations are definable directly in terms of the other operations, and make programming easier and more readable. But they are not absolutely necessary. This clause is defined exactly as the **observers** clause in Section 1.2.1.3. The only difference regards the axiom construction rules, as described in Section 1.2.2. In the given example, lines 11–12 illustrate this structure.

1.2.1.5 Domains

This section describes the conditions under which operations are required to be defined. The clause starts with keyword **domains** and is followed by a list of sort variables declaration and a list of domain conditions. Variables are used in operations and formulæ, and their sorts must be declared beforehand.

In the stack specification example lines 13–16 illustrate this clause. Both operations, pop and peek, must respect the same condition, namely, that the stack should not be empty.

1.2.1.6 Axioms

This clause is dedicated to axiom declaration. It starts with the keyword **axioms**, followed by a list of variable declarations and a list of axioms. For further details on axioms see Section 1.2.2.4. In our example this clause corresponds to lines 17–23.

1.2.2 Operation and Predicate signatures

An operation or predicate signature is an expression with one of the following basic forms, where $n \geq 0$.

```
f: MySort Sort1 ... Sortn --> Sort';
f: MySort Sort1 ... Sortn -->? Sort';
f: Sort1 ... Sortn --> Sort';
f: Sort1 ... Sortn -->? Sort';
f: MySort Sort1 ... Sortn;
f: Sort1 ... Sortn;
```

The first four forms are for operations; the later two are for predicates. A signature is built with the components:

- f, the operation or predicate name;
- MySort, the first parameter, the sort under specification;
- Sort1 ... Sortn, the remaining parameters, a list of sorts that must all belong to the module (see Section 1.3) or be the language primitive **int**;

- The identification of a total (−−>) or a partial operation (−−>?);
- Sort', the return sort, must belong to the module (see Section 1.3) or be the language primitive **int** (operations only).

Operations must return a sort belonging to the module or the primitive **int**; predicates have no return sort whatsoever.

1.2.2.1 Rules

A specification must conform to rules that do not derive solely from the grammar in Appendix A.1. The following rules apply to the signatures, domains and axioms of the specification operations and predicates.

1.2.2.2 Constructors

Constructors are characterized by returning the sort under specification. They agree with the first four forms above, where Sort' is the sort under specification (MySort). As a consequence, predicates cannot be constructors because predicates have no returning sort.

1.2.2.3 Domains

A domain condition is an expression of the form

```
op (x1, ..., xn) if <formula>
```

where op is an operation (op must be declared as an operation, not a predicate), and the arguments x1, ..., xn must be variables.

Some operations may not be defined for some specific instances (e.g, we cannot pop from an empty stack); those instances must be prohibited as arguments to the operation. The <formula> is a definition domain on which op must be defined.

When writing domain conditions, one must take in consideration:

- Any total operation, marked with −−> in the operation signature, which no domain condition is required, is provided a **true** default condition;
- Any partial operation, marked with −−>? in the operation signature, for which no domain condition is specified, is provided a **false** default condition;
- Arguments x1, ..., xn have sorts Sort1, ..., Sortn as described in the signature of operation op.
- <formula> respects the rules in this Section.

1.2.2.4 Axioms

An axiom must have one of the following basic structures, being that the part **if** <formula> is optional.

```
op (t1, ..., tn) = t if <formula>;
pred (t1, ..., tn) if <formula>;
x1 = x2 if <formula>;

op (t1, ..., tn) != t if <formula>;
not pred (t1, ..., tn) if <formula>;
x1 != x2 if <formula>;

op (t1, ..., tn) = t when <formula> else u;

op (t1, ..., tn) = t iff <formula>;
pred (t1, ..., tn) iff <formula>;
x1 = x2 iff <formula>;
```

where op is an operation, pred is a predicate, t, t1, ..., tn are terms, x1, x2 are identifiers. The following restrictions apply to these axiom patterns.

- If op is a constructor, then t1 must either be a variable or a constructor applied to variables (this last case only applies if the first parameter of op belongs to MySort); the remaining arguments t2, ..., tn must all be variables;
- If op or pred are observers, then t1 must either be a variable or a constructor applied to variables;
- If op or pred are derived, then all arguments t1, ..., tn must be variables;
- x1, x2 must be variables of the sort under specification (MySort);
- <formula> must agree with the rules in Section 1.2.2.5.

The axiom

```
op (t1, ..., tn) = t when <formula> else u;
```

is short for the two axioms below.

```
op (t1, ..., tn) = t if <formula>;
op (t1, ..., tn) = u if not <formula>;
```

Similarly, the three axioms

```
op (t1, ..., tn) = t iff <formula>;
pred (t1, ..., tn) iff <formula>;
x1 = x2 iff <formula>;
```

are abbreviations. The first for

```
op (t1, ..., tn) = t if <formula>;
op (t1, ..., tn) != t if not <formula>;
```

the second for

```
pred (t1, ..., tn) if <formula>;
not pred (t1, ..., tn) if not <formula>;
```

and the third for

```
x1 = x2 if <formula>;
x1 != x2 if not <formula>;
```

```
specification
    sorts
        Element
end specification
```

Figure 1.3: Specification of a generic element

1.2.2.5 Formulæ

Formulæ are built from variables and terms, using the disjunction operator **or**, conjunction **and**, negation **not**, equality = or !=, and the primitive integer operators.

- A variable identifier is a formula;
- if f1, ..., fn are formulæ and op is an operation (or predicate), then op(f1, ..., fn) is a formula.
- If f1, f2 are formulæ, then f1 = f2, f1 != f2, f1 **and** f2, f1 **or** f2, and **not** f1 are formulæ;
- If f1, f2 are formulæ, then so is f1 # f2, where # is one of the following $+ - * / \% > < >= <=$, and so is max(f1,f2) and min(f1,f2).
- Nothing else is a formula.

1.3 Modules

The meaning of symbols external to a specification (sort Element in the stack specification of Figure 1.2, for example) is only fixed when the specification is embedded, as a component, into a module. A module is a surjective function from a set of names N into specifications, such that all symbols (sorts, operations and predicates) are provided by some specification in the module.

For example, module Stack can be given by the mapping

$$\text{stack.spc} \mapsto \text{Figure 1.2}$$
$$\text{element.spc} \mapsto \text{Figure 1.3}$$

A file directory (implicitly) defines a module, where N is the set of .spc filenames in the directory, and the associated specifications are the contents of the files.

The module of our running example is a directory containing two files, named stack.spc and element.spc, where contents are described in figure 1.2 and 1.3, respectively.

1.4 The refinement language

This section describes the language of refinement mappings. Subsection 1.4.1 presents the general structure of refinements, and subsection 1.4.2 describes the refinement into Java classes. The grammar itself is presented in Appendix A.2.

```
refinement
    Sort_1 is class Class_1 {
        <opRepresentation_11> is <method_11>;
        ...
        <opRepresentation_1k> is <method_1k>;
    }
    ...
    Sort_n is class Class_n {
        <opRepresentation_n1> is <constructor_n1>;
        <opRepresentation_n2> is <constructor_n2>;
        ...
        <opRepresentation_nk-1> is return <method_nk-1>;
        <opRepresentation_nk> is <method_nk>;
    }
end refinement
```

Figure 1.4: The outline of a refinement

```
refinement
    Element is class java.lang.Object
    Stack is class datatypes.stack.Stack {
        make(): Stack                       is Stack();
        push(s: Stack, e: Element): Stack   is void push(java.lang.Object e);
        pop(s: Stack): Stack                is java.lang.Object pop();
        peek(s: Stack): Element             is java.lang.Object top();
        size(s: Stack): int                 is int size();
        isEmpty(s: Stack)                   is boolean isEmpty();
    }
end refinement
```

Figure 1.5: Stack Refinement

1.4.1 General structure

A refinement maps a module (see Section 1.3) into a series of Java classes. Each specification in the module is mapped into a Java class; its operations, if any, are mapped into methods of that class. In Figure 1.4, we present the general structure of a refinement; the grammar is presented in Appendix A.2.

A refinement mapping starts with keyword **refinement** and ends with keywords **end refinement**. Between these keywords we declare a non-empty set of mappings, one for each specification in the module. The refinement for each specification in the module is accomplished by a set, possibly empty, of mappings from operations and predicates to methods. Lines 2–6 represent a mapping from Sort1 in Java Class1, and, enclosed in braces (lines 3–5), the mapping from the sort operations into the class methods.

A refinement mapping for our running example (the Stack module in Section 1.3), is given in Figure 1.5.

In the particular case where the sort for the specification has no operations or predicates, the refinement mapping consists of a simple declaration of the form:

```
Sortname is class Classname
```

In our example, sort Element has no operations nor predicates; we map it into class java.lang.Object as in Figure 1.5, line 2.

In the general case the specification contains operations and/or predicates to refine. In this case the refinement consists of a block containing several mappings, one for each operation of the sort under refinement. Figure 1.5, lines 3–10, show the refinement mapping for the Stack sort.

1.4.2 Refining specifications into Java types

In this section we present the restrictions imposed at the specification language that cannot be captured by the supporting grammar (Appendix A.2).

A refinement mapping for a particular sort includes mappings for operations and for predicates.

1.4.2.1 Mapping a constructor operation into a Java constructor

Each operation in a specification may be refined into a Java constructor of the class under consideration. In this case the valid patterns for mappings are:

```
<constructorOperation> is <constructorSignature>;
```

where <constructorOperation> is the representation for any specified operation constructor, like cop with signature

```
cop: Sort1, ..., Sortn --> MySort;
```

the representations in the refinement language are, respectively:

```
cop (x1: Sort1, ..., xn: Sortn): MySort
```

where

- Each Sort1 ,..., Sortn belongs to the module;
- Sorts Sort1 ,..., Sortn in signatures and in representations agree in number and order.
- The sort under specification is MySort.

<constructorSignature> is the constructor's signature as defined in the Java class except for the visibility and the **throws** clause.

Representation

```
cop (x1: Sort1, ..., xn: Sortn): MySort
```

corresponds to a constructor signature of the form

```
constructorName(Class1 y1, ..., Classn yn)
```

where constructorName is the abbreviated constructor name (no package resolution scope), and y1 ,.., yn are identifiers in the list x1 ,..., xn (but not necessarily in that order).

A key characteristic of this mapping is that an operation with $n \geq 0$ parameters is mapped into a method with n parameters.

In the above figure this is exemplified in line 4.

1.4.2.2 Mapping an operation or a predicate into an instance method

Each operation and predicate in a specification may be refined into a Java method of the class that refines MySort. In this case the valid patterns for mappings are:

```
<operationOrPredicate> is <methodSignature>;
<operationOrPredicate> is return <methodSignature>;
```

where <operationOrPredicate> is the representation for operations and predicates. For operation op and predicate pred with signatures

```
op: MySort, Sort1, ..., Sortn --> Sort;
pred: MySort, Sort1, ..., Sortn;
```

the representations in the refinement language are, respectively:

```
op (x0: MySort, x1: Sort1, ..., xn: Sortn): Sort
pred (x0: MySort, x1: Sort1, ..., xn: Sortn)
```

where

- Each Sort1 ,..., Sortn, Sort belongs to the module;
- Sorts Sort1 ,..., Sortn, Sort in signatures and in representations agree in number and order.
- The sort under specification is MySort.

<methodSignature> is the method's signature as defined in the Java class except for the visibility and the **throws** clause. Representation

```
op (x0: MySort, x1: Sort1, ..., xn: Sortn): Sort
```

corresponds to a method signature of the form

```
Type methodName(Class1 y1, ..., Classn yn)
```

where Type is **void** or **int** or a class name corresponding to some sort in the module, and y1 ,.., yn are identifiers in the list x1 ,..., xn (but not necessarily in that order).

Representation

```
pred (x0: MySort, x1: Sort1, ..., xn: Sortn)
```

corresponds to a method signature of the form

```
boolean methodName(Class1 y1, ..., Classn yn)
```

A key characteristic of this mapping is that an operation with $n \geq 1$ parameters is mapped into a method with $n-1$ parameters, since the first parameter of the operation or predicate corresponds to the current instance (**this**) — the same Sort being specified.

Operation mappings can sometimes be ambiguous. Consider a variant of Figure 1.5 where the pop operation is associated to method

```
datatypes.stack.Stack pop();
```

The question arises as to whether the sort returned by the operation, (denoted by Stack after the colon, Figure 5, line 6), corresponds to the return type datatypes.stack.Stack of the Java method, or to the instance (**this**) upon which the method is called. In the former case use:

```
pop(s: Stack): Stack is datatypes.stack.Stack pop();
```

In the latter use:

```
pop(s: Stack): Stack is return datatypes.stack.Stack pop();
```

1.4.2.3 Mapping an operation or a predicate into a static method

Similarly to 1.4.2.2 we have the following valid patterns for mappings:

```
<operationOrPredicate> is <staticMethodSignature>;
<operationOrPredicate> is return <staticMethodSignature>;
```

<staticMethodSignature> is the only visible difference, where the static method's signature is defined as in the Java class except for the visibility and the **throws** clause, which are here omitted. Representation

```
op (x1: Sort1, ..., xn: Sortn): Sort
```

corresponds to a static method signature of the form

```
Type staticMethodName(Class1 y1, ..., Classn yn)
```

where Type is **void** or **int** or a class name corresponding to some sort in the module, and y1 ,.., yn are identifiers in the list x1 ,..., xn (but not necessarily in that order).

Representation

```
pred (x1: Sort1, ..., xn: Sortn)
```

corresponds to a static method signature of the form

```
boolean staticMethodName(Class1 y1, ..., Classn yn)
```

A key characteristic of this mapping is that an operation with $n \geq 0$ parameters is mapped into a method with n parameters, since there is not the concept of current instance (**this**).

Chapter 2

User's Guide

2.1 Introduction

ConGu (Contract Guided System Development) is a tool that supports testing Java implementations of algebraic specifications using JML. It picks a specification module—a set of .spc files containing specifications—, a package of Java classes that supposedly implements the specification module, and a refinement mapping that maps the sorts and operations of the specification module into the corresponding classes and methods of the given package—, and generates the necessary entities that will allow to monitor the execution of the given original classes against the given specifications.

This document gives a frequently asked questions overview to, and explains how to interact with, the **ConGu** tool. Chapter 1 describes the specification and the refinement languages.

2.2 Usage

First things first...

2.2.1 System requirements

What are the **ConGu** *tool system requirements?*

You need at least Java 1.4.2 and JML 5.3. Notice that in order to use Java 1.5.0 you need JML 5.4.

2.2.2 How to use

How do I use the **ConGu** *tool?*

The **ConGu** tool requires that the user indicates the path where to find the implementation class files (.class or .jar) and packages, the directory where the specification (.spc) and the refinement (.rfn) files are to be found, as suggested by the following example:

java −cp congu.jar:< client_classpath > congu.Congu [<directory>]

Note: if directory is omitted, then your current local directory is implicitly assumed. Take into consideration that the explicit client_classpath is mandatory

and should reference all implementation files class path. If everything goes smooth then an output directory is created with name output in your current local directory. This directory contains all generated files needed in the monitoring phase.

2.2.3 Generated files

What is the meaning of the generated files?

The several output files can be grouped into the following categories which can be noticed by **ConGu** tool verbose messages:

Wrapper classes in Java source format (.java) and Java byte code format (.class). These classes replace the original user's classes, monitoring the execution. They have the exact same file name as the original user's classes, and have the same API.

Renamed classes in Java byte code format (.class) only. These classes are the original user's classes, but with a distinct name so that they can be distinguished from the wrapper classes. Their filename pattern matches the _*_Original.class.

Immutable classes in Java source format (.java) and Java byte code format (.class). These are the classes that are equipped with JML contracts. They match pattern with _*_Immutable.java file names.

Pair classes in Java source format (.java) and Java byte code format (.class). These auxiliary classes allow object exchanges between the wrapper and the immutable classes. They match pattern with _*_Pair_*.java file names.

Range class in Java source format (.java) and Java byte code format (.class). This optional auxiliary datatype allows domain range contract predicates (forall) assistance. It is named as _Range.java.

2.2.4 Compiling

How do I compile the output files?

As previously stated inside the generated output directory there will be immutable, wrapper, renamed and pair classes. Those classes were already compiled automatically by congu.jar, but if needed, they can be manually compiled by first compiling the immutable classes with jmlc, and finally compiling the wrapper and the pair classes with javac.

2.2.5 Monitoring

How do I monitor contracts?

In this phase you have to take in consideration that the classes you want to test are going to be replaced by the generated classes in the output directory. Also, JML requires some special packages in order to fully monitor contracts. Using jmlrac simplifies the process. We suggest using the following generic command:

jmlrac −Xbootclasspath/p:output/:<client_classpath> <client_application>

where client_application is the user test class (where the main entry point exists), and client_classpath is the user environment class path.

The above command line tells Java Virtual Machine that the output directory is the first place where all input classes are to be found and, in case they are missing, they will be resolved inside client_classpath path.

2.3 Examples

Can you give me a practical example?

Sure, right away. The following examples are available from `http://labmol.di.fc.ul.pt/congu/examples/` website.

2.3.1 Easy

Let us start with a simple example like Stack. Checkout this example on `http://labmol.di.fc.ul.pt/congu/examples/stack.html` and download Stack.spc, Element.spc, Stack.java (the original class that implements Stack.spc), Stack.rfn and CongUltStack.java to the same folder, let us say easy/. Do not forget to also download the **ConGu** tool (congu.jar) latest version available from **ConGu** official website to the same easy/ folder.

At this point your easy/ folder will have the following file repository:

```
easy/
    CongUltStack.java
    congu.jar
    Element.spc
    Stack.java
    Stack.rfn
    Stack.spc
```

First we have to compile both implementing Stack.java and testing CongUltStack.java classes with javac (our current directory is easy/):

javac −d . Stack.java

javac −d . CongUltStack.java

```
easy/
    CongUltStack.java
    congu.jar
    datatypes/
        stack/
            CongUltStack.class
            Stack.class
    Element.spc
    Stack.java
    Stack.rfn
    Stack.spc
```

At this moment we have the required input files available. So, in a second step, we have to execute the **ConGu** tool with existing java (our current directory is still easy/):

java −cp congu.jar:. congu.Congu .

```
easy/
    CongUltStack.java
    congu.jar
    datatypes/
        stack/
```

```
            CongUltStack.class
            Stack.class
    Element.spc
    output/
        datatypes/
            stack/
                _boolean_Pair_Stack.class
                _boolean_Pair_Stack.java
                _Object_Pair_Stack.class
                _Object_Pair_Stack.java
                Stack.class
                _Stack_Immutable.class
                _Stack_Immutable.java
                Stack.java
                _Stack_Original.class
                _Stack_Pair_Stack.class
                _Stack_Pair_Stack.java
        java/
            lang/
                _boolean_Pair_Object.class
                _boolean_Pair_Object.java
                _Object_Immutable.class
                _Object_Immutable.java
    Stack.java
    Stack.rfn
    Stack.spc
```

And finally, with the third and final step, you are able to monitor contracts with jmlrac (keeping easy/ as current directory):
jmlrac −Xbootclasspath/p:output/:. datatypes.stack.CongUltStack

2.3.2 Not so easy

In the above example we mixed the specification files with the source code and byte code into the same folder. Let us now split these three categories into three distinct subdirectories inside an arbitrary notsoeasy/ folder. Check the TicTacToe example available on `http://labmol.di.fc.ul.pt/congu/examples/tictactoe.html` and download the specifications TicTacToe.spc, Board.spc, Tile.spc and the refinement mapping TicTacToe.rfn to the same specs/ subdirectory; the original classes that implement the datatype TicTacToe.java, Board.java, Tile.java and the test class RunTicTacToe.java to the same src/ subdirectory. Again, do not forget to also download the **ConGu** tool (congu.jar) latest version available from **ConGu** official website to the same notsoeasy/ folder, making sure that src−bin/ subdirectory is also present.

At this point your notsoeasy/ folder will have the following file repository:

```
notsoeasy/
    congu.jar
    specs/
        Board.spc
        TicTacToe.rfn
        TicTacToe.spc
        Tile.spc
    src/
        Board.java
        RunTicTacToe.java
        TicTacToe.java
        Tile.java
    src−bin/
```

First we have to compile all implementing TicTacToe.java, Board.java, Tile.java, and testing RunTicTacToe.java classes with javac (our current directory is notsoeasy/):

javac −d src−bin/ src/∗.java

```
notsoeasy/
    congu.jar
    specs/
        Board.spc
        TicTacToe.rfn
        TicTacToe.spc
        Tile.spc
    src/
        Board.java
        RunTicTacToe.java
        TicTacToe.java
        Tile.java
    src−bin/
        datatypes/
            tictactoe/
                Board.class
                RunTicTacToe.class
                TicTacToe.class
                Tile.class
```

At this moment we have the required input files available. So, in a second step, we have to execute the **ConGu** tool with existing java (our current directory is still notsoeasy/):
java −cp congu.jar:src−bin/ congu.Congu specs/

```
notsoeasy/
    congu.jar
    output/
        datatypes/
            tictactoe/
                Board.class
                _Board_Immutable.class
                _Board_Immutable.java
                Board.java
                _Board_Original.class
                _Board_Pair_Board.class
                _Board_Pair_Board.java
                _Board_Pair_TicTacToe.class
                _Board_Pair_TicTacToe.java
                _boolean_Pair_Board.class
                _boolean_Pair_Board.java
                _boolean_Pair_TicTacToe.class
                _boolean_Pair_TicTacToe.java
                _boolean_Pair_Tile.class
                _boolean_Pair_Tile.java
                _int_Pair_Board.class
                _int_Pair_Board.java
                TicTacToe.class
                _TicTacToe_Immutable.class
                _TicTacToe_Immutable.java
                TicTacToe.java
                _TicTacToe_Original.class
                _TicTacToe_Pair_TicTacToe.class
                _TicTacToe_Pair_TicTacToe.java
                Tile.class
                _Tile_Immutable.class
                _Tile_Immutable.java
                Tile.java
                _Tile_Original.class
                _Tile_Pair_Board.class
                _Tile_Pair_Board.java
                _Tile_Pair_TicTacToe.class
                _Tile_Pair_TicTacToe.java
                _Tile_Pair_Tile.class
                _Tile_Pair_Tile.java
        _forall/
            _Range.class
```

```
        _Range.java
specs/
    Board.spc
    TicTacToe.rfn
    TicTacToe.spc
    Tile.spc
src/
    Board.java
    RunTicTacToe.java
    TicTacToe.java
    Tile.java
src-bin/
    datatypes/
        tictactoe/
            Board.class
            RunTicTacToe.class
            TicTacToe.class
            Tile.class
```

As you can see there are many generated output files. One in particular is _Range.java from _forall package. As said before, this is an auxiliary datatype to assist domain range contract predicates (forall).

Like the precedent example, third and final step, you are able to monitor contracts with jmlrac (keeping notsoeasy/ as current directory):

```
jmlrac -Xbootclasspath/p:output/:src-bin/ datatypes.tictactoe.RunTicTacToe
```

2.4 Troubleshooting

Q: Why does **ConGu** keeps telling me that several implemented java methods are not defined in the refinement?

A: Well, that might be explained by the fact your client_classpath is incomplete or miss referenced. If this is not the case than you might have an incorrect refinement, or the implemented class does not export those methods after all.

Q: When running **ConGu**, it reports that some class does not implement the Cloneable interface. Do I have to implement the so called interface Cloneable?

A: If you know for sure that your "default" clone is sufficient since you are dealing with *immutable* classes then go ahead. Be aware that implementing the clone method implies redefining the equals method.

Q: I got a compilation error when compiling the immutable classes saying that a given class is "synchronized", why?

A: First of all, JML does not support "synchronized" input classes at the time of this writing. If your input original class is not "synchronized" then you might be using gij compiler which is known to reflect every class as synchronized. Use a Sun Java compiler instead.

Q: How to identify the violated contract source during the monitoring phase?

A: During the monitoring phase, and for each violated contract detected, a JML exception is reported. This exception, regarding the Immutable file contract specification, will identify if it is a PreconditionError or a PostconditionError and the line number where it occurs. For example let us assume in the provided

Easy example 2.3.1 we attempt to access an empty Stack top. In this case, the following exception will occur :

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError: by method _Stack_Immutable.top regarding specifications at
File "datatypes/stack/_Stack_Immutable.java", line 63, character 70 when
        '_one' is datatypes.stack._Stack_Original@7a78d3
        at datatypes.stack._Stack_Immutable.checkPre$top$_Stack_Immutable(_Stack_Immutable.java:1972)
        at datatypes.stack._Stack_Immutable.top(_Stack_Immutable.java:2060)
        at datatypes.stack.Stack.top(Stack.java:62)
        at datatypes.stack.CongUltStack.top(CongUltStack.java:119)
        at datatypes.stack.CongUltStack.showMenu(CongUltStack.java:79)
        at datatypes.stack.CongUltStack.main(CongUltStack.java:34)
```

And the line 63 of output/datatypes/stack/_Stack_Immutable.java specifies:

```
          @ /* peek ( S ) if not isEmpty ( S ) ; */ requires true ==> !(datatypes.stack._Stack_Immutable.isEmpty(_one).value);
```

As you can see the domain condition */* *peek (S) if not isEmpty (S) ;* */* triggered the JML notification. This is the same domain condition as specified in Stack.spc.

2.5 Limitations

- The input .**class** to be monitored cannot have public attributes.
- **ConGu** does not cope with packages of classes that are hierarchically related in any way.
- An operation cannot have a domain condition that can be reduced to the operation itself, since that would cause infinite recursion.
- Refinement into Java interfaces is not supported at the time of this writing.
- Client **final** methods can not be overridden from the wrapper class; if this is the case, avoid calling them.
- Be aware that in our implementation the wrapper class calls the client super-constructor twice.

Chapter 3

Implementation Guide

3.1 Introduction

ConGu is a tool that supports testing Java implementations of algebraic specifications using JML. It was designed to support the methodology described in references [4, 5]. This document gives an overview of the implementation of **ConGu**. After reading it you should have acquired a notion of how **ConGu** is implemented and how the source code is organized, and should be able to move faster into extending the **ConGu** functionalities if that is your objective. In order to read this document you should already have a good grasp on how **ConGu** and the methodology it supports work.

The input to **ConGu** is a series of source files written in the *specification language* and one source file written in the *refinement language*. The definition of these languages can be found in Chapter 1. The **ConGu** User's manual can be found in Chapter 2.

This chapter is organized as follows. Section 3.2 describes the overall architecture of the tool. Section 3.3 describes the parser. Sections 3.4 and 3.5 describe the analyzers for the specification module and for the refinement bindings. Section 3.6 presents the class renamer. Sections 3.7 to 3.11 describe the generators for the various classes produced, namely, wrapper, immutable, contracts for the immutable, pairs, and the writing to files. Section 3.12 puts it all together.

3.2 Architecture

ConGu is organized into several logical components each responsible for one of the tasks that together make up the **ConGu** functionality (see Figure 3.1). Components Specification Module Analyzer and Refinement Binding Analyzer make up the front-end of **ConGu**. Together, these two components are responsible for dealing with the input files and translating the information they contain into an internal representation. The back-end, formed by the generators and the Class Renamer, uses that internal information to produce the output of **ConGu**. The implementation of **ConGu** maps this logical structure. Each of the components is implemented, insofar as possible, by a distinct Java class

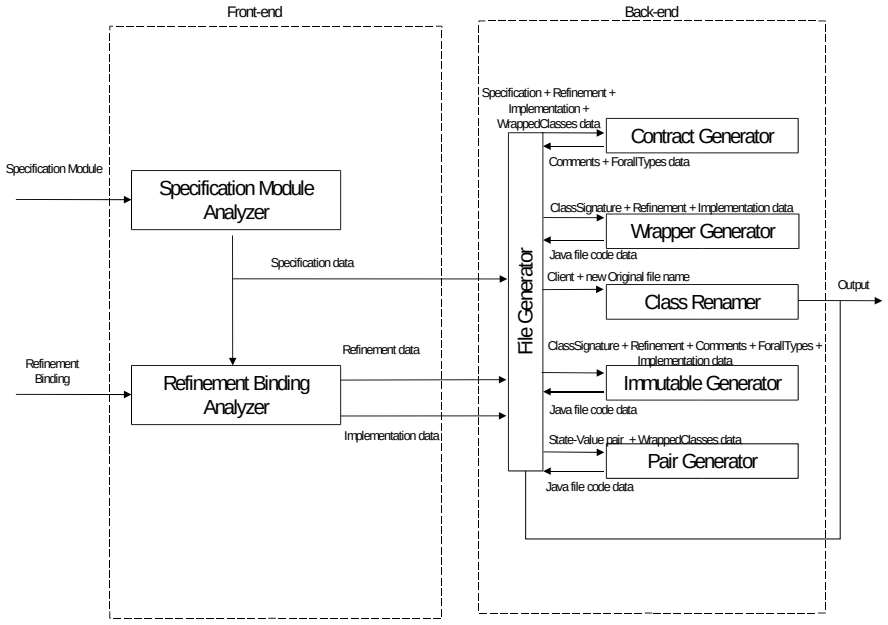

Figure 3.1: The architecture of **ConGu**. Each box represents a component.

or set of classes. In the following sections the implementation of each of these components is addressed.

3.3 Parsing with SableCC

ConGu takes as input a set of specification files and one refinement binding file. In order for **ConGu** to validate and make sense of the content of those files, the files need to be parsed and analyzed against either the specification grammar or the refinement grammar. In order to simplify the implementation of this task, the **SableCC** parser generator is used. **SableCC** takes as input a `.grammar` file that specifies the lexicon and the production rules of a language and outputs a set of Java classes that allow:

- The parsing of a text file against that language. This consists on creating a tree that represents the syntactic structure of the contents of the file.
- Walking through the nodes of that tree and executing certain actions. These actions are specified by extending some of the classes that are generated by **SableCC**: the tree-walkers.

By using **SableCC** the effort of implementing the analysis of the input files is reduced to the implementation of its semantic analysis. After defining the grammar for the language of specifications (`grammars/specification.grammar`) and the language of refinement bindings (`grammars/refinement.grammar`)

SableCC is executed on each of this `.grammar` files therefore creating two parsers, one for each language, and also the tree-walker classes. **SableCC** creates four packages for each language. For the specification language the following packages are created by **SableCC**: spec. analysis , spec. lexer , spec.node and spec. parser. Four equivalent packages were created for the refinement language: refine . analysis , refine . lexer , refine .node and refine . parser.

Each of the tree-walker classes is then extended to allow the semantic analysis of the trees that are generated by the parser. In the following sections we describe in detail this process. For more information about **SableCC** please refer to reference [3].

3.4 The Specification Module Analyzer

The Specification Analyzer Module (SMA) takes as input a list of `.spc` files with specifications, parses each file, checks the static semantics, reports errors if they exist and outputs a spec.semant.SpecQuerier object through which all the other modules of **ConGu** can obtain information about the specification. SMA is implemented mainly by the classes in packages spec.∗, together with the classes in package symbol. Class spec.semant.SpecModuleAnalyser is the main class of SMA.

SMA is the only module of **ConGu** that can be used as a stand-alone tool. In order to support the use of SMA as such, class congu.specAnalyzer.SpecAnalyzer which has a main method is created. This class provides a user interface for SMA. This class is ignored when SMA is used as part of the **ConGu** tool.

3.4.1 Semantic analysis

If every file is parsed without errors and the corresponding syntactic tree is created (spec.semant.SpecModuleAnalyser) then the specifications are syntactically correct i.e., they obey the rules of the specification language as defined in `grammars/specification.grammar`. Yet, specifications must conform to some other properties, of semantic nature, which are not captured by the grammar.

As in a standard compiler, identifier declarations and their uses must be checked for consistency by the SMA [1]. For instance, when analyzing the operation call push(R, E), SMA issues an error if the signature of operation push is push: Stack Element −−> Stack and R has the sort Rational instead of Stack as expected for the first argument of this operation. Class spec.semant.Semantics has methods responsible for this part of the semantic analysis: variable and signature declaration analysis and checking that variables and operations and predicates are used accordingly to those declarations. This corresponds to the semantic analysis phase of a standard compiler [1].

In addition to the standard semantic analysis, the underlying methodology of **ConGu** imposes restrictions on the specification language which must also be ensured by the SMA. There are strong restrictions on the form of the axioms that depend on the properties of the operations or predicates.Class spec.semant. Restrictions , which extends spec.semant.Semantics, is responsible for this part of the semantic analysis.

3.4.2 Three stage analysis

A specification is divided into three main sections: sort declaration, operation signature declaration and the axioms (in this group we include the domain definition of the operations). There are semantic dependencies among these sections, i.e., some of these sections have elements which can only be interpreted after analyzing some other section: the signatures refer to the sorts that are declared in the sorts section and the axioms have references to both the sorts and the operations (or predicates) that are declared in the signature section. These dependencies also exist among the various specifications in the module. Each specification may have references to sorts or operations (or predicates) that are declared in some other specification. In terms of semantic analysis this means the three sections must be analyzed in the correct order: first SMA must analyze all sorts in all specifications, then all signatures and, at last, all axioms.

In order to implement this three stage analysis three classes are created that extend the tree walker spec. analyses .DepthFirstAdapter generated by **SableCC**. They are:

1. spec.semant.SortAnalyser
2. spec.semant.SignatureAnalyser
3. spec.semant.DomainAxiomAnalyser

Each of these tree walkers is applied to all specifications, in this order, by the spec.semant.SpecModuleAnalyser, i.e., first the sort analyzer is applied to all the specifications, then the same thing with signature analyzer and then the axiom analyzer. These tree walkers extend spec. analyses .DepthFirstAdapter by adding calls to the appropriate methods of spec.semant.Semantics and spec.semant. Restrictions on the tree nodes they visit. Each tree walker triggers the semantic analysis of the nodes that form the section of the specification it is responsible for, while ignoring the remaining nodes.

3.4.2.1 Sort checking

Axioms are first-order logic formulæ. Each sub-formula belongs to a sort. Variables evaluate to the sort they have been declared with, while operations evaluate to their return sort. For instance, the variable E evaluates to the sort Element, if it has been declared as E: Element. The operation call push(S, E) evaluates to the sort Stack, if push: Stack Element −−> Stack is the signature of operation push. Notice that the first argument of the operation call push doesn't have to be a variable, it can be a complex expression as long as it evaluates to the sort Stack. SMA is responsible for checking that each expression evaluates to the correct sort. If the first argument of the operation call push were to evaluate to a sort other than Stack, SMA would issue an error. In order to validate an expression, SMA must determine the sort of each of the expression's immediate components. For instance an operation or a predicate call is valid in what concerns sort consistency, if all of its arguments evaluate to the expected sort. On the other hand, an equality is valid if both sides evaluate to the same sort.

Class spec.semant.DomainAxiomAnalyser, which is the tree-walker responsible for the axioms part of the specifications, plays an important role in the mechanism that evaluates the sort of each expression.

Class spec.semant.DomainAxiomAnalyser has a stack as an attribute, where the sort of each expression is pushed when the tree-walker leaves the tree node that corresponds to that expression. Since the tree-walker visits the tree in a depth-first manner, when it enters a new expression the sorts of the immediate components of that expression are already in the stack. At that point all the tree-walker has to do is: pop the sorts out of the stack, validate the expression against those sorts and then push the sort of that expression into the stack. In order to validate the expression, spec.semant.DomainAxiomAnalyser calls the appropriate method of spec.semant.Semantics which takes as arguments the sorts collected from the stack and returns the sort the expression evaluates to.

3.4.3 Storing and retrieving information

While performing the semantic analysis, class spec.semant.Semantics and class spec.semant. Restrictions store much of the information regarding the semantic properties of the several elements of the specification. This is done for two reasons: first, as mentioned in Section 3.4.1, the analysis of many parts of the specification requires knowledge about other elements of the specification, so this information must be readily available, and second the other components of **ConGu** also require knowledge about the specification module, which must be provided by SMA. Classes spec.semant.Semantics and spec.semant. Restrictions have several data structures as attribute that store all the relevant information about the specification module.

3.4.3.1 Semantic bindings

A specification module contains several elements with relevant semantic information. For instance, the identifier Stack declared in "**sorts** Stack" is a sort, in which case we need to know its name and the file where it is declared. In order to store this information, the identifier Stack becomes bound to an object of type spec.binds.Sort which stores all the relevant information about a sort. On the other hand, the identifier S, declared in "S: Stack;", is a variable, in which case we need to register the association between the name and the sort. We do this by binding the identifier S to an object of type spec.binds.Variable that stores the relevant information. Package spec.binds contains the classes that are used for storing semantic information. The elements of the specification with relevant semantic information are bound to an instance of one of this classes. This is achieved via map like structures, fields of classes spec.semant.Semantics and spec.semant. Restrictions . The elements of the specification whose semantics we wish to store are used as keys in the map; the associated values are spec.binds objects. Notice that, in addition to identifiers, some of the tree nodes are also used as keys.

3.4.3.2 Symbol table

The part of the semantic analysis that concerns identifier analysis use is usually implemented with the help of a symbol table. This symbol table maps each identifier to its semantics. As SMA reaches each identifier declaration (sort, signatures and variables) it binds the identifier (after converting it into a symbol) to its semantic properties in the table. When SMA encounters an identifier usage

it looks up the identifier in the table and checks that it is used in accordance to its semantics.

The symbol table used in SMA was originally created for the **TyCO** compiler [6], as an extension of that in book [1]. The main class that implements the table (symbol.Table) together with its auxiliary classes form the package symbol. Among other things, this symbol table is different from a simple java.util.Map in that has a scoping mechanism. The scoping mechanism is useful for SMA since the variables in the specifications must be interpreted according to the place where they are used. For instance, assuming that there are two different variable declarations with name A, if variable A is being used in a domain definition than its sort is the one declared after the **domains** token and not the one declared after the **axioms** token.

3.4.3.3 Specification querier

Class spec.semant.Restrictions, which extends spec.semant.Semantics, implements interface spec.semant.SpecQuerier. The interface spec.semant.SpecQuerier provides a set of methods that return information about the specification module. It is through this interface that the other components of **ConGu** get all the information they require about the specification module. Class spec.semant.Restrictions implements the methods from the interface spec.semant.SpecQuerier by accessing directly to its private data structure attributes but returning only the required data. The instance of spec.semant.Restrictions that is used during the analysis of the specification module is returned by the main class of SMA, spec.semant.SpecModuleAnalyser, and then passed on to the modules of **ConGu** that need to make queries about the specification module.

3.5 Refinement Binding Analyzer

The Refinement Binding Analyzer (RBA) takes as input a `.rfn` file and a SpecQuerier object that represents the specification module (see Section 3.4.3.3). RBA parses the `.rfn` file, verifies its semantics against the specification and the class system, reports errors if they exist and outputs a refine.semant.RefinementQuerier object through which the other modules can obtain information about the refinement. RBA is implemented mainly by the classes from the refine.* packages. Class spec.semant.RefinementAnalyser is the main class of RBA.

In terms of implementation, RBA has the general structure of the Specification Module Analyzer. For that reason, the structure of this section follows that of Section 3.4, albeit more succinct. We advise reading that section before proceeding.

3.5.1 Semantic analysis

The semantic analysis of a refinement consists of two main tasks:

- Making sure the refinement is compatible with the specification module and the class hierarchy. This means checking that all sorts and signatures are refined into a single type or method, checking that all sorts and signatures that are referenced in the refinement are indeed part of the

specification module, and also that all types and methods do exist in the class hierarchy.

- Checking that the refinement is consistent and that it complies to the **ConGu** methodology.This part of the analysis implies checking that operation and predicate refinements are consistent with sort refinements. For instance, the operation pop(s: Stack): Element cannot be bound to method String pop() of class Stack, unless class String implements the sort Element and class Stack implements the sort Stack.

The semantic analysis of the refinement is conducted by the methods of class refine .semant.Semantics which are invoked while tree-walking the refinement.

3.5.2 Two stage analysis

A refinement contains two types of bindings: those that specify which Java type implements which sort and those that specify which methods implement which operations or predicates. Since operation and predicate signatures mention sorts, in order to validate a binding between an operation or predicate and a Java method, RBA must know which types implement each of the sorts that compose the signature. For instance, in order to validate the binding between operation push: Stack Element −−> Stack and method **void** push(String s) of class Stack, RBA must first know that class String implements sort Element and class Stack implements sort Stack. This means that the analysis of the refinement must be done in two stages: first sort bindings; then operation bindings. In order to implement the two stage analysis two classes were created that extend the tree walker generated by **SableCC** refine . analyses . DepthFirstAdapter. These are:

1. refine .semant.SortRefinementAnalyser
2. refine .semant.OpRefinementAnalyser

These two classes call methods of class refine .semant.Semantics on the appropriate tree nodes.

3.5.3 Using Java reflection

While the specification module and the refinement binding are provided to **ConGu** as text files that need to be parsed, the Java classes that form the implementation are obtained by **ConGu** by using Java Reflection [2]. When the refinement analysis reaches a class name it tries to find the class in the classpath using Java Reflection. If the class is not found an error is issued; otherwise all information regarding the class (methods, variables, etc) is collected. By using Java Reflection **ConGu** gives the user the possibility to provide only `.class` files as input (by making them accessible through the classpath) and not the original `.java` source code. This strategy has two advantages:

- It allows the user to test implementations for which he does not have access to the source code, and

- it simplifies the implementation of **ConGu** by avoiding the need to parse and analyze Java source code. This effort is put upon the Java compiler and the Java Reflection mechanism.

3.5.3.1 Getting class information

The refinement binding defines the set of classes upon which the implementation of the specification module relies. The properties of each of these classes must be checked by the RBA in order to validate the refinement. Also, those properties must be made available for consultation by the other components of **ConGu**. Whenever RBA reaches a sort that is implemented by a class, it finds the class through reflection and collects all relevant information regarding the class. A implement.ClassSignature object is used to store this information. The constructor of implement.ClassSignature takes the full qualified name of the class as argument and uses the methods of java.lang.Class to obtain all the information about that class. It then stores that information in the fields of class implement.ClassSignature. Among other things, this constructor is responsible for creating a set of implement.MethodSignature objects that represent the methods of the given class and a set of implement.ConstructorSignature that represent the constructors of the class.

3.5.4 Storing and retrieving information

3.5.4.1 Semantic bindings

Class refine.semant.Semantics contains two java.util.Map fields that are used for storing the information concerning the refinement of sorts and the refinement of operations and predicates. The first maps sort names into objects of class refine.binds.SortRefinement that store all the relevant information regarding the sort and the type that implements it. The second maps operation or predicate signatures into refine.binds.OpRefinement objects that store all the relevant information regarding the operation or predicate and the method or expression that implements it. Class refine.binds.OpRefinement is abstract. There are four (non abstract) classes that extend it: refine.binds.OpToConstructorRef, refine.binds.OpToMethodRef, refine.binds.OpToNullRef and refine.binds.OpToExpressionRef. These classes are used according to whatever implements the operation: a Java constructor, a method, the null value or an expression.

3.5.4.2 Parameter binding

Whenever an operation or predicate is implemented by a method or a Java constructor, the mapping between the operation or predicate parameters and the method parameters must be taken into consideration, since the order of the parameters in the operation or predicate need not be matched in the method. The first parameter of an operation may, for instance, correspond to the second parameter of the method. This correspondence between parameters is specified by the user through the parameter names. When analyzing the refinement of an operation into a method, RBA must not only take this into consideration, but also store this information, so that when an operation or predicate call is

translated into a method call by the other components of **ConGu**, the correct parameter order is used.

The correspondence between the parameters of an operation or predicate and those of the method is kept in an integer array either in class refine.binds.OpToMethodRef or in class refine.binds.OpToConstructorRef. The array index gives the operation parameter and the integer value contained on that index gives the corresponding method parameter. For example, if the first integer in the array is 0, then the first parameter of the operation corresponds to the first parameter of the method. This array is a private field. The other components of **ConGu** access this information through the use of methods getMethodParameterName, getMethodParameterIndex, getConstructorParameterName and getConstructorParameterIndex that take as input the operation parameter index and give as output either the name or the index of the corresponding method or Java constructor parameter.

3.5.4.3 Refinement querier and implementation querier

Class refine.semant.RefineQuerier is the interface that defines the set of methods through which the other components of **ConGu** can obtain information about the refinement binding. Class refine.semant.Semantics implements this interface. An object of this type is returned by the main method of the RBA. Class refine.semant.RefineQuerier contains the method getImplementation which returns all the information regarding the implementation, through an object of class implement.ImplementQuerier. This interface provides methods that return information about the implementation in itself, independently of its relation to the specification module.

3.5.4.4 Implementation package

The refinement binding is what makes the connection between the specification module and its Java implementation. In order to store and process the information regarding the elements that compose the implementation (classes, expressions, etc), several classes are available. These classes form the implement package. In this package we have class implement.ClassSignature which stores the information regarding a class, class implement.MethodSignature which stores the information regarding a method and implement.Constructor which stores the information that concerns a constructor. Class implement.Null is used to represent the null value and class implement.Expression represents a generic Java expression. implement.Implementation stores the totality of the information that concerns the implementation of a specification module. This class also implements interface implement.ImplementQuerier and as so, provides the set of methods that supply all the required information about the implementation.

3.6 Class Renamer

Class congu.ClassRenamer (CR) takes as input the original classpath (it can be either a regular directory or a .jar file name), the original class filename, the new classpath directory, the new class filename and outputs the new assembled Java byte-code.

In terms of implementation, CR inner class Class File (CF) is the Java byte-code main structure used to manipulate the renaming process. CF constructor fills this structure and the dump() method outputs it to the file system. In order to successfully rename a class-file, CR:

- Initially loads CF as the original Java byte-code representation;
- Updates all explicit and implicit internal attribute references inside CF such as: SourceClass, ThisClass, constant-pool NameAndType, constant-pool FieldsInfo , constant−pool MethodsInfo and, recursively, the for inner classes. Notice that renaming one class file may not be enough if the class file has, for instance, inner classes. Although renaming can be automatically performed we still need access to bytecode files involved.
- Renames the class-file.
- Finally writes CF as the new Java bytecode representation with the new filename.

3.7 Wrapper Generator

Class congu.WrapperGenerator (WG) takes as input the implement.ClassSignature (CS), the refinement.semant.RefinementQuerier (RQ), the implement.ImplementQuerier (IQ) provided by **ConGu**'s earlier stages and outputs a wrapper Java file, acting like a String factory. This wrapper Java file intends to replace the original class.

In what concerns clients of the original class, the wrapper object is similar to the original object: its constructors and methods have the same signature and they apparently behave in the same way; however they allow monitoring the original constructors and methods execution by redirecting client calls to the corresponding constructor or method in the Immutable class.

In terms of implementation, CS gives all necessary Java class signatures and RQ the contextualized signature from the specification.

The output file is generated by instantiating WG and calling toJavaCode() method. This method generates the new Java file code by declaring the same package class, class name, public attributes, constructors and methods (excluding the final ones and omitting **native** keyword) as the original class. It also includes auxiliary methods for the purpose of wrapping and unwrapping objects when necessary.

3.8 Immutable Generator

Class congu.ImmutableGenerator (IG) takes as input the implement.ClassSignature (CS), the refinement.semant.RefinementQuerier (RQ) provided by **ConGu**'s earlier stages, the Hashtable provided by Contract Generator (CG) getComments() method, the Collection provided by CG getForallTypes() method, the implement.ImplementQuerier (IQ) provided by **ConGu**'s earlier stages, and outputs an immutable Java file containing JML contracts within Java comments,

acting like a String factory. This immutable file is responsible for contract monitoring during execution time and is generated by instantiating IG and calling toJavaCode() method.

In terms of implementation, IG generates a Java file belonging to the same package as CS, all "constructors" and methods specified in RQ, the equals() and the clone() methods (the latter only when CS implements interface Cloneable). The generated "constructors" are actually static methods that return the same datatype CS being specified. Both, "constructors" and methods are equipped with JML contracts as Java comments. In addition, the generated Java file might also includes a static _forall ._Range field (FR) for each input Collection element (when asserting over free variables). Those FR's are Object sets populated with objects that are passed as argument, and returned as results, in all method calls that happen in the context of the given Immutable class during contract monitoring execution time. This populating process is achieved by a new post-condition with JML syntax through FR updateCache() method comment. Later on, when monitoring a JML **forall** assertion, the domain range scans the appropriate FR.[1]

The final stage consists in combining each "constructor" and method with the associated JML assertion predicates. By iterating over the input Hashtable (an Hashtable of Hashtable's) one can identify the sub-Hashtable for each RQ specified "constructor" or method. The latter hashtable collects the StringBuffer containing JML formated pre- and post-conditions grouped by Operation criteria. These Operation collections are then merged with JML **also** keyword to ensure correct design-by-contract behavior. The contract for the clone() method is built without input hashtable query since it is not generated by a specified axiom but rather stating that a cloned object is always equals to the object itself.

3.9 Contract Generator

Class congu.ContractGenerator (CG) takes as input, through its constructor, the spec.semant.SpecQuerier (SQ), the refinement.semant.RefinementQuerier (RQ), the implement.ImplementQuerier (IQ), the Hashtable identifying the renamed original class name for each original client class name provided by FileGenerator (FG), all prepared by **ConGu**'s earlier stages. CG outputs two data structures: the Hashtable accessible via getComments() method and the Hashtable accessible from getForallTypes() method.

3.9.1 Output

The former output data structure (accessible via getComments() method) is a hashtable of hashtables that groups contracts either by ConstructorSignature or by MethodSignature (MS) and then by Operation keys; think that as a double key hashtable. The MS is unique since it also identifies the belonging class (also known as CS identifier) through its classFile attribute. Each contract is a StringBuffer representing a JML syntax assertion characters sequence. Later

[1] This strategy might need revision starting with JML 5.3, for updateCache() is not a **pure** method.

on, the IG will perform the task of placing each contract inside the expected immutable file and associated java method.

The latter output data structure (accessible from getForallTypes() method) is a String collection hashtable with CS as key. It contains all domain types used in **forall** assertions within contracts grouped by client class criteria. Later on, the IG when declaring the immutable class attributes will declare as many static FR's as these String collection elements. IG will also perform the task of declaring extra post conditions for each method whose signature (return and arguments type) contains any collected domain type used in **forall** assertions. These extra post conditions are nothing more than ensuring updateCache() on each static FR previously declared, giving the opportunity to dynamically grab objects during monitoring phase and reuse them when executing domain range assertions.

3.9.2 Implementation

This contract factory uses two depth first walkers named DefWalker (DW) and TranslateWalker (TW) to travel the specifications tree, although CG itself is already a depth first adapter too, on a specific simple case when collecting variable identifiers from an operation or from an equality node. After DW and/or TW walks we assemble the contract output according to our methodology.

3.9.2.1 The translation tree-walker

This class translates the formulæ of our specification language, cooperating with DW for the equality axiom. It is implemented with a translation stack that, in the end, is supposed to have one and only one translated element. During this process the operationResult node indicates when to replace the actual node with result, mainOpVar has the main operation variables substitution and mainOpNode binds the variable identifiers with associated nodes. It is also a CG client by means of reading or updating static attributes.

3.9.2.2 The definedness tree-walker

This class translates the definedness conditions for both terms and formulæ of our specification language, cooperating with TW. It is implemented with a translation stack that, in the end, is supposed to have one and only one translated element. During this process the operationResult node indicates when to replace the actual node with result, mainOpVar has the main operation variables translation ready to pass to TW, and mainOpNode reflects the specific case of getting the definedness condition from a second level depth translated domain condition. It is also a CG client by means of reading their static attributes.

3.9.3 Virtual equals node operation

Due to the fact that translating an observer operation automatically results, according to our methodology, in an equality axiom, there is the need to translate nodes that do not belong to the specification. Such node, for the purpose of code reuse, was virtually created and then translated with minor impacts to the implemented solution. One of these is the protected **static** Operation observerOp

attribute existence that allows DW and TW bind each **SableCC** Node with the main operation since (SQ) getOperation() will return **null** with equality axiom as the argument.

3.10 Pair Generator

Class congu.PairGenerator (PG) takes as input the refinement.semant.StateValuePair (SVP) provided by **ConGu** earlier stage, the Hashtable provided by FileGenerator (FG) and outputs, through toJavaCode() method, a pair of Java files, acting like a String factory. This Java file is, in turn, used by the immutable and wrapper classes to import/export objects.

The input SVP consists of the return type of a (non-void) method that exists in the refinement (value), and the class the method belongs to (state). Two methods of the same class that have the same return type are associated to the same state-value pair.

The input hashtable indicates whenever the state class type is to be subject of testing Java implementations with the **ConGu** tool framework. Basically this hashtable contains all generated wrapper classes names and their respective original name translation.

3.11 File Generator

Class congu.FileGenerator (FG) takes as input the spec.semant.SpecQuerier (SQ), the refinement.semant.RefinementQuerier (RQ), the implement.ImplementQuerier (IQ) all instantiated by **ConGu**'s earlier stages, outputting each generator output (WG, IG, CG, PG) to a file and also renaming the original client class to be confronted against specification during execution time.

For each IQ class an immutable class file is generated through IG, equipped with the contracts previously stated. During this iteration, if one class has refined methods in RQ, then both the wrapper class file, through WG, and the renamed original client class file, through CR, are dumped into the file system while the old and new names are stored in a String hashtable that will be redirected to PG.

The next step is to write into the file system each pair class, through PG. Finally if CG getForallTypes() method has any stored type, then a auxiliary FR class is also outputted to the file system. All output files are written to the "output" directory and still preserve the client package organization, which can be noticed from **ConGu** tool messages.

Final step is compiling all "output" source classes. Only the Immutable classes are compiled with jmlc, the remaining classes are all compiled with javac.

3.12 Putting it all together

The behavior of **ConGu** is achieved by synchronizing all its components. Class congu.Congu does this task. The main method of this class is responsible for providing a user's interface and initializing each of the **ConGu** components at the right time. First the SMA is executed on the specifications. If no error is produced, then the output of SMA is given as input to the RBA. If the refinement

binding is valid, the front-end of **ConGu** has finished its task and it is time for the back-end to start. The back-end, which generates the output files, is managed by class FG.

Appendix A

Grammars

A.1 Grammar of the specification language

The grammar that defines the syntax of the specification language was built under the framework of the compiler generator **SableCC** `http://sablecc.org/`. Below we list the production rules and the tokens of the grammar. The nonterminal symbols from the tokens section are left unspecified in this paper; the full grammar is available online `http://labmol.di.fc.ul.pt/congu/`.

```
/***********************************************************
 * Tokens                                                  *
 ***********************************************************/
Tokens

    white_space = (sp | ht | ff | line_terminator)*;

    traditional_comment = '/*' not_star+ '*'
        + (not_star_not_slash not_star* '*'+)* '/';
    documentation_comment = '/**' '*'* (not_star_not_slash not_star* '*'+)* '/';

    end_of_line_comment = '//' input_character* line_terminator?;

    specification_token = 'specification';
    end = 'end';
    sorts_token = 'sorts';
    constructors_token = 'constructors';
    observers_token = 'observers';
    derived_token = 'derived';
    axioms_token = 'axioms';
    domains_token = 'domains';
    if = 'if';
    iff = 'iff';
    when = 'when';
    else = 'else';

    l_parenthese = '(';
    r_parenthese = ')';
    l_bracket = '[';
    r_bracket = ']';
    colon = ':';
    semicolon = ';';
    comma = ',';
    arrow_simple = '-->';
    arrow_question = '-->?';

    plus = '+';
    minus = '-';
    mult = '*';
    div = '/';
    mod = '%';
```

```
    eq = '=';
    neq = '!=';
    lt = '<';
    gt = '>';
    le = '<=';
    ge = '>=';

    not = 'not';
    or = 'or';
    and = 'and';

    decimal_integer_literal = decimal_numeral;
    boolean_literal = 'true' | 'false';

    identifier = java_letter java_letter_or_digit*;

/*****************************************************************
 * Ignored Tokens                                                *
 *****************************************************************/
Ignored Tokens

  white_space,
  traditional_comment,
  documentation_comment,
  end_of_line_comment;


/*****************************************************************
 * Productions                                                   *
 *****************************************************************/
Productions

specification =
    header
    sorts
    constructors?
    observers?
    derived?
    domains?
    axioms?
    end specification_token;

header =
    specification_token;

sorts =
    sorts_token identifier;

constructors =
    constructors_token signature+;

observers =
    observers_token signature+;

derived =
    derived_token signature+;

domains =
    domains_token var_decl* domain+;

axioms =
    axioms_token var_decl* axiom+;

signature =
    {operation} [name]:identifier colon [parameters]:identifier* arrow
        [return_sort]:identifier semicolon |
    {predicate} [name]:identifier colon [parameters]:identifier* semicolon;

arrow =
    {total} arrow_simple |
    {partial} arrow_question;

domain =
```

```
    term if formula semicolon;

var_decl =
    identifier_list colon identifier semicolon;

axiom =
    {simple} formula semicolon |
    {conditional} formula if [condition]:formula semicolon |
    {equivalence} formula iff [condition]:formula semicolon |
    {double_conditional} relational eq [left]:relational
        when [condition]:formula else [right]:relational semicolon;

// FORMULA

formula =
    {conjunction} conjunction |
    {disjunction} formula or conjunction;

conjunction =
    {equality} equality |
    {conjunction} conjunction and equality;

equality =
    {relational} relational |
    {equality} [left]:relational eq [right]:relational |
    {inequality} [left]:relational neq [right]:relational;

relational =
    {term} additive |
    {lower_than} [left]:additive lt [right]:additive |
    {lower_equal} [left]:additive le [right]:additive |
    {greater_than} [left]:additive gt [right]:additive |
    {greater_equal} [left]:additive ge [right]:additive;

additive =
    {mult} multiplicative |
    {plus} additive plus multiplicative |
    {minus} additive minus multiplicative;

multiplicative =
    {unary} unary |
    {mult} multiplicative mult unary |
    {div} multiplicative div unary |
    {mod} multiplicative mod unary;

unary =
    {basic} basic |
    {negation} not basic |
    {minus} minus basic;

basic =
    {boolean} boolean_literal |
    {integer}  decimal_integer_literal |
    {variable} identifier |
    {operation} term |
    {paren} l_parenthese formula r_parenthese;

term =
    [name]:identifier l_parenthese [args]:additive_list? r_parenthese;

// LISTS

// A list of one or more identifiers separated by commas
identifier_list =
    identifier comma_identifier*;

comma_identifier =
    comma identifier;

// A list of one or more signatures separated by commas
signature_list =
    signature comma_signature*;

comma_signature =
```

```
    comma signature;

// A list of one or more additives separated by commas
additive_list =
    additive comma_additive* ;

comma_additive =
    comma additive;
```

A.2 Grammar of the refinement language

The grammar that defines the syntax of the language of refinements was built under the framework of compiler generator **SableCC** `http://sablecc.org`. We do not present here a list of non-terminal tokens. The full grammar is available online `http://labmol.di.fc.ul.pt/congu/`.

```
/***********************************************************
 * Tokens                                                  *
 ***********************************************************/
Tokens

    white_space = (sp | ht | ff | line_terminator)*;

    traditional_comment = '/*' not_star+ '*'
        + (not_star_not_slash not_star* '*'+)* '/';
    documentation_comment = '/**' '*'* (not_star_not_slash not_star* '*'+)* '/';

    end_of_line_comment = '//' input_character* line_terminator?;

    refinement_token = 'refinement';
    is = 'is';
    class_token = 'class';
    end = 'end';
    null = 'null';
    return_token = 'return';

    dot = '.';
    colon = ':';
    semicolon = ';';
    comma = ',';
    l_parenthese = '(';
    r_parenthese = ')';
    l_brace = '{';
    r_brace = '}';
    identifier = java_letter java_letter_or_digit*;

/***********************************************************
 * Ignored Tokens                                          *
 ***********************************************************/
Ignored Tokens

  white_space,
  traditional_comment,
  documentation_comment,
  end_of_line_comment;


/***********************************************************
 * Productions                                             *
 ***********************************************************/
Productions

refinement =
    header
    refine+
    end refinement_token;

header =
    refinement_token;
```

```
refine =
    {non_primitive} sort_to_non_primitive non_primitive_bind_block?;

sort_to_non_primitive =
    [sort]:identifier is class_token type;

non_primitive_bind_block =
    l_brace non_primitive_bind* r_brace;

non_primitive_bind =
    operation is non_primitive_value semicolon;

non_primitive_value =
    {constructor} constructor |
    {method} method |
    {null} null;

operation =
    {operation} [name]:identifier l_parenthese op_parameter_list?
        r_parenthese colon [codomain]:identifier |
    {predicate} [name]:identifier l_parenthese op_parameter_list?
        r_parenthese;

constructor =
    [name]:identifier l_parenthese method_parameter_list? r_parenthese;

method =
    return_token? [return_type]:type [name]:identifier l_parenthese
        method_parameter_list? r_parenthese;

op_parameter_list =
    op_parameter comma_op_parameter*;

comma_op_parameter =
    comma op_parameter;

op_parameter =
    [name]:identifier colon [sort]:identifier;

method_parameter_list =
    method_parameter comma_method_parameter*;

comma_method_parameter =
    comma method_parameter;

method_parameter =
    type [name]:identifier;

type =
    {simple_name} identifier |
    {qualified_name} type dot identifier;
```

Bibliography

[1] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[2] M. Campione, K. Walrath, A. Huml, and Tutorial Team. *The Java Tutorial*. Sun Microsystems, online edition, 2006. http://java.sun.com/docs/books/tutorial/.

[3] E. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, Mar. 1998.

[4] I. Nunes, A. Lopes, V. T. Vasconcelos, J. Abreu, and L. Reis. Testing implementations of algebraic specifications with design-by-contract tools. TR 05–22, Department of Informatics, Faculty of Sciences, University of Lisbon, Dec. 2005. Available at `http://www.di.fc.ul.pt/tech-reports/`.

[5] I. Nunes, A. Lopes, V. T. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In *Proceedings of ICFEM'06*, volume 4260 of *LNCS*, pages 494–513. Springer-Verlag, 2006.

[6] TYped Concurrent Objects. `http://www.ncc.up.pt/tyco/`.