Runtime Monitoring of Java Code Using ConGu

Vasco T. Vasconcelos
Faculty of Sciences
University of Lisbon
Portugal
vv@di.fc.ul.pt

Isabel Nunes
Faculty of Sciences
University of Lisbon
Portugal
in@di.fc.ul.pt

Antónia Lopes
Faculty of Sciences
University of Lisbon
Portugal
mal@di.fc.ul.pt

Luís S. Reis
Faculty of Sciences
University of Lisbon
Portugal
lmsar@di.fc.ul.pt

ABSTRACT

The main goal of the **ConGu** project is the development of a framework to create property-driven algebraic specifications and test Java implementations against them. In this paper we present a brief overview of the framework's fundamental components – specifications, modules, refinements – and describe the **ConGu** tool both from the user's and from the architect's point of view. The tool allows users to test Java classes – no source code needed, just bytecode – against a module of specifications, and to discover runtime axiom violations. The tool generates intermediate classes equipped with contracts and wraps the original classes in classes that allow contract monitoring in a way that is transparent to the clients of the original classes. The tool generates JML assertions, but it could as well generate contracts in other assertion languages – the **ConGu** modules that generate classes for monitoring are independent from those that generate contracts. We also report on the use of the **ConGu** tool in the context of an undergraduate programming course.

Keywords

Algebraic specification, implementation checking, design by contract

1. INTRODUCTION

The formal specification of software components is a desirable activity within the task of software development, insofar as formal specifications are useful, on the one hand, to understand and reuse software and, on the other hand, to test implementations for correctness.

Design by Contract (DBC) [14] is widely used for the specification of object-oriented software. There are a number of languages and tools (e.g., [4, 5, 12, 13]) that allow equipping classes and methods with invariants, pre and post-conditions, which can be monitored for violations at runtime. In the DBC approach, specifications are class interfaces (Java interfaces, Eiffel abstract classes, etc) annotated with contracts expressed in a particular assertion language.

To build contracts one must observe the following: *(i)* contracts are built from boolean assertions on values, thus any method invoked within an assertion must return a value; *(ii)* contracts should refer only to the public features of the class because client classes must be able not only to understand contracts, but also to invoke operations that are referred to in them — e.g., clients must be able to test pre-conditions; *(iii)* to be monitorable, a contract cannot have side effects, thus it cannot invoke methods that modify the state.

These restrictions bring severe limitations to the kind of properties we can express directly through contracts. Unless we define a number of, otherwise dispensable, additional methods, we are left with very poor specifications.

As an example, let us analyze the support given by DBC to the specification of a set of persons, through the integration of assertions in the class PersonSet outlined in Figure 1. In the next paragraphs we discuss how this could be achieved following Meyer [14], and using Java and JML [13] rather than Eiffel.

We want to say that a newly created set contains no elements; we thus add the following post-condition to the PersonSet constructor.

```
ensures isEmpty();
```

On what concerns method add (e), we would like to say that, after adding element e to the set, it is not empty:

```
ensures !isEmpty();
```

We also want to say that sets contain no duplicated elements, in other words, that inserting equals e and f in a row yields the same set as when inserting e alone. The inclusion of a post-condition in method add (e) with flavor (ignoring for now the details about the origin of f)

```
equals(\old(clone().add(f))) <== e.equals(f);
```

would not work because method add is **void**. As already referred to, **void** methods cannot be used in assertions.

```
public class PersonSet implements Cloneable {
  public PersonSet () { ... }
  public void add (Person e) { ... }
  public boolean isEmpty() { ... }
  public boolean isIn (Person e) { ... }
  public Person largest () { ... }
  ...
}
```

Figure 1: A class describing a set of persons.

This example shows that, whenever a specification is implemented by a mutable type, unless we have methods that allow to inspect the whole structure of the elements in a data structure without modifying it, we are not capable of writing complete post-conditions for our methods. These inspection methods are, in general, artificial, and even against the nature of the type itself and, hence, they are not a solution to the problem.

Model-based approaches to DBC, like those proposed for Z [16], Larch [9], JML [13], and AsmL [3], overcome this limitation by specifying the behavior of a class, not via the methods available in the class, but else through very abstract implementations based on basic elements available in the adopted specification language. Rather than a *model based* approach, we instead adopted a *property based* algebraic approach to specifications, described in reference [15].

We continue the specification of other methods of class PersonSet, showing that there are also inherent difficulties and hidden pitfalls in contract definition. Two examples follow.

We want to state the meaning of the isEmpty method by saying that it is true whenever there are no persons in the set, and that it is false whenever there is at least one person in the set. We do this by adding the following post-condition:

```
ensures (\ forall Person e;someSet.has(e);! isIn (e))<==\result;
ensures (\ exists Person e;someSet.has(e); isIn (e))<==!\result);
```

requiring that we maintain someSet of persons for the sole purpose of monitoring class PersonSet (the same approach would have to be applied to the above contract for method add, in order to define values for f).

For method largest we must add a pre-condition of the form

```
requires !isEmpty();
```

We would also like to say that, when inserting an element e in a set s, the element becomes the largest in the new set, when it is greater than the largest element in s. We write this as a post-condition to method add:

```
ensures largest (). equals(e) <== \old(e.greaterThan(largest()));
```

The problem with this assertion is that \old(largest ()) may not be defined, due to the pre-condition to method largest above. We must then take into consideration the *definedness condition* [15], and revise the contract to:

```
ensures \old(!isEmpty()) ==> (
   largest (). equals(e) <== \old(e.greaterThan(largest())));
```

The difficulties and pitfalls the last two examples illustrate, motivated the development of an approach supported by a tool that generates contracts from abstract specifications.

In this paper we present **ConGu** (Contract Guided System Development), a project whose aim is the development of a framework to create property-driven algebraic specifications and fully test Java implementations against them. We find it important to equip property-driven approaches with tools similar to the ones currently available for model-driven approaches. Support for checking implementations against algebraic specifications is, as far as we know, restricted to a few approaches (cf [2, 11]), which have limitations as described in Section 6.

The key idea of the **ConGu** approach is to reduce the problem of testing implementations against algebraic specifications to the run-time monitoring of contract annotated classes, supported today by several run-time assertion- checking tools. The **ConGu** tool reads algebraic specifications and a mapping relating specification and Java entities, and generates a number of classes that are used to test the original implementation against the given specifications. All specification properties are checked against implementations — monitorable contracts are generated that cover them all. The technique used by **ConGu** surpasses the above referred limitations in what contracts are concerned. The tool has been in use since 2005, with truly positive results, in the context of an undergraduate course on algorithms and data structures.

The next section presents an overview of the **ConGu** approach, describing the framework's main components and requirements. Then, Section 3 explains how to use the tool in the context of testing Java implementations against algebraic specifications. A description of the **ConGu** architecture follows in Section 4, where the various individual components are addressed. In Section 5 we report on the use of the tool in the context of an undergraduate programming course. In Section 6 we compare **ConGu** with essentially two other tools that also allow testing implementations against algebraic specifications. Finally, in Section 7, we address future work and then conclude.

2. OVERVIEW OF THE METHODOLOGY

The **ConGu** framework aims at simplifying the task of checking whether Java implementations conform to algebraic specifications. The key idea of the approach is to reduce this problem to the runtime monitoring of contracts that are automatically generated from specifications.

The **ConGu** framework's main components are specifications, modules, and refinements. The specifications we use in this context are algebraic, property-driven insofar as they define sorts and operations on those sorts, determining classes of algebras (models) which can be regarded as possible implementations of the specified data types. In general terms, **ConGu** supports partial specifications — whose operations can be interpreted by partial functions — with conditional axioms. Operations are defined through their signature, restrictions on their domain, and axioms defining their properties. Each specification defines a single sort but it may use other sorts while defining, for example, parameters or results of operations. All entities (sorts, operations) that are referred to in a specification, but that are not defined in it, are said to be *external* references.

```
specification
  sorts
    Orderable
  observers
    greaterThan: Orderable Orderable;
end specification
```

Figure 2: Specification of an ordered element.

```
specification
  sorts
    Set
  constructors
    empty: —> Set;
    insert: Set Orderable —> Set;
  observers
    isEmpty: Set;
    isIn: Set Orderable;
    largest: Set —>? Orderable;
  domains
    S: Set;
    largest(S) if not isEmpty(S);
  axioms
    E, F: Orderable;  S, T: Set;
    isEmpty(empty());
    not isEmpty(insert(S, E));
    not isIn(empty(), E);
    isIn(insert(S,E), F) iff E = F or isIn(S, F);
    largest(insert(S, E)) = E if isEmpty(S);
    largest(insert(S, E)) = E
      when greaterThan(E, largest(S)) else largest(S);
    insert(insert(S, E), F) = insert(S, E) if E = F;
    insert(insert(S, E), F) = insert(insert(S, F), E);
end specification
```

Figure 3: Specification of an ordered set.

Figures 2 and 3 present two examples. The specification in Figure 2 has no external references, while the specification in Figure 3 has Orderable as an external reference to sort Orderable.

Specifications with external references are meaningful only when they are put together with the specifications that define all those references. We use the notion of *module* to denote the set of specifications that, together, are self-contained, in the sense that all external references are defined therein.

Specific features of the language we adopted for writing specifications, such as the classification of operations in specific categories, and strong restrictions in the form of the axioms, not only simplify the task of creating specifications, but are also effective in driving the automatic identification of contracts for implementing classes.

In order to check the behavior of Java classes against specifications — violations of an axiom or a domain restriction — the gap between specifications and Java classes must be bridged. For this purpose, refinement mappings have to be defined indicating which sort is implemented by which class, and which operation is implemented by which method. Because this activity does not require any knowledge about the concrete representation, refinement mappings are quite simple to define.

Figure 4 illustrates a refinement mapping that maps the module containing the two specifications of Figures 2 and 3

```
refinement
  Orderable is class Person {
    greaterThan: Orderable o:Orderable is
      boolean older(Person o);
  }
  Set is class PersonSet {
    empty: —> Set is PersonSet();
    insert:Set e:Orderable—>Set is void add(Person e);
    isEmpty: Set is boolean isEmpty();
    isIn: Set e:Orderable is boolean isIn(Person e);
    largest: Set —>? Orderable is Person largest();
}
end refinement
```

Figure 4: An example of a refinement mapping.

into the Java classes Person and PersonSet, respectively.

The languages used to build specifications and refinements, as well as semantic restrictions on signatures, domains and axioms of specifications, are described elsewhere [1].

Given a specification module and a refinement mapping, **ConGu** generates JML monitorable contracts for every axiom and domain restriction of every specification in the input module. These contracts rely on the existence of proper equals and clone methods in the target classes. Two observations contribute to this need. On the one hand, as mentioned in the previous section, if contracts are to be monitored, they cannot contain side-effects. In order to generate monitorable contracts for testing all specification properties, the **ConGu** approach is such that methods invoked within contracts are always invoked on clones of the original objects. On the other hand, due to the fact that axioms are defined through term equality, **ConGu** contracts need to test for Java object equality.

In specifications we adopted a semantics of strong equality for the equality symbol used in axioms, that is, either both sides are defined and are equal, or both sides are undefined. In addition to the contracts that are generated from axioms, our tool also automatically generates contracts that are consistent with the adopted notion of equality.

The implementations of the equals and clone methods should meet the following criteria: *(i)* clone method is required not to have any effect whatsoever on **this**; *(ii)* the implementation of clone is required to go deep enough in the structure of the object so that any shared reference with the cloned object cannot get modified through the invocation of any of the methods that implement the specification operations. For example, an array based implementation of some collection, in which one of its methods changes the state of any of its elements, requires the elements of the collection to be cloned as well as the array itself; *(iii)* method equals returns true whenever it compares a clone with the original object.

The kind of relationship the main components — specification modules, refinements, and Java implementations — define are such that the task of testing a same class against several different specifications is easily accomplished — it suffices to write a different refinement mapping, which is quite simple to define. Also, the possibility of having a refinement mapping from two different components into the

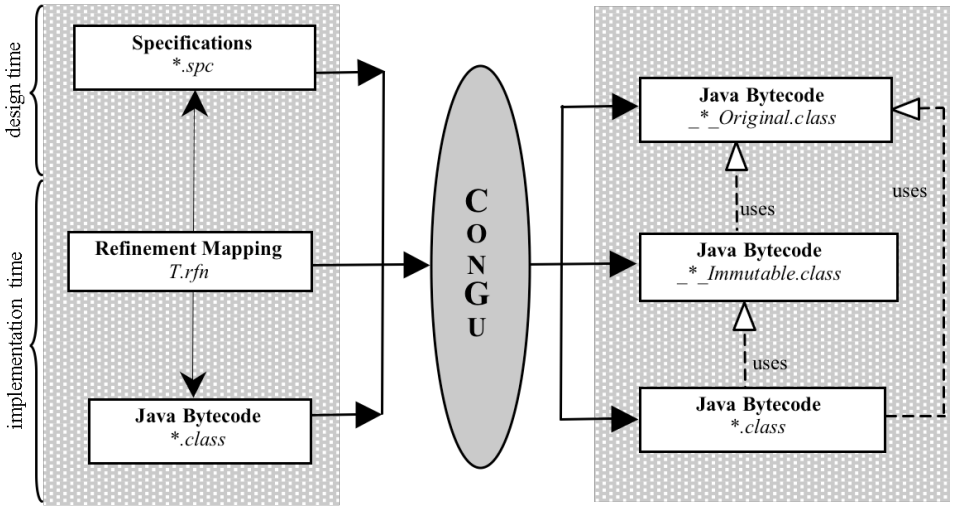

Figure 5: Overview of the ConGu tool.

same type is extremely useful since it promotes the writing of generic specifications that can be reused in different situations, as illustrated in reference [15].

3. CONGU AT WORK

The general idea underlying the tool is to automatically generate contracts from specifications in order to monitor the execution of classes that implement the specifications. Towards this end, **ConGu** replaces the original program by another program equipped with contracts that can be monitored for axiom and domain conditions violations.

Figure 5 shows that the input of **ConGu** consists of a specification module (a directory containing a self-contained collection of `.spc`, cf. Section 2), Java bytecode, and a refinement (a `.rfn` file, as in Figure 4) mapping the specification into Java. The tool renames each `C.class` bytecode file mentioned in the refinement into `_C_Original.class`, and creates an immutable class, `_C_Immutable.java`, equipped with contracts regarding specifications axioms and domain conditions. It further generates classes that wrap and substitute the original `C` classes, allowing to monitor their execution. Section 4 explains the roles of each of these classes and the relations among them.

Given a specification module (`*.spc`), a refinement mapping (`T.rfn`), and Java bytecode (`*.class`), **ConGu** generates and compiles Java classes that replace the original bytecode. The new program must be executed using the `jmlrac` command rather than `java`, in order to monitor the contracts that the tool generates. The process is described in Figure 6; the workflow is described below.

One first compiles the Java program, for **ConGu** works with bytecode, not source code. **ConGu** proper is then called with a command of the form

```
java congu.Congu <module-directory>
```

where `<module-directory>` indicates the directory where the module + refinement is. If absent, the current directory is used. The jar file for **ConGu** and the byte code are both expected to be in the class path. The tool generates a new `output` directory to hold the generated classes. We are then in a position to monitor our program, using a command of the form

```
jmlrac -Xbootclasspath/p:output/:.  Test
```

Running the original program under JML scrutiny may produce pre-condition exceptions due to domain condition violations, or to post-condition exceptions related to axiom violations. The output produced by JML then guides the developer into the the violated domain condition or axiom, from where she can start looking for the defect. For example, an output of the from

```
Exception in thread "main" java.lang.Error:
/* not isEmpty ( insert ( S , E ) ) ; */
  at PersonSet.add(PersonSet.java:100)
  at SmallBang.main(Test.java:11)
Caused by:
org.jmlspecs.jmlrac.runtime.
  JMLInternalNormalPostconditionError: by method
_PersonSet_Immutable.add regarding specifications at
File "_PersonSet_Immutable.java", line 52, character 62 when
  '\result' is {}
  ...
```

indicates an error in monitoring axiom **not** isEmpty(insert (S, E)), which in turn, may indicate a problem in method isEmpty or insert .Once the bug is spotted, the process commences once again with the compilation of the source code, if the error was in Java code, or with the run of **ConGu** if the glitch was in the specification+refinement.

4. THE TOOL

ConGu is organized into several logical components each responsible for one of the tasks that together make up the **ConGu** functionality (see Figure 7). 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 various generators and the Class Renamer, uses that internal information to produce the output of **ConGu**. The implementation of

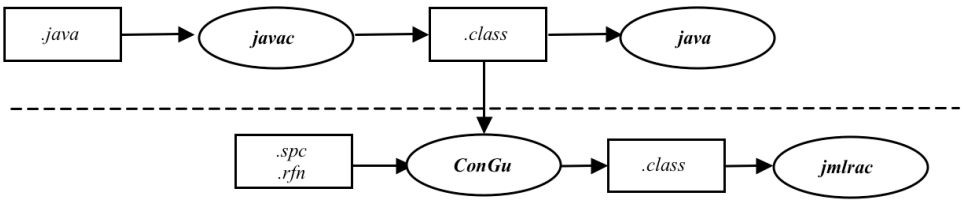

Figure 6: Traditional workflow versus working with ConGu.

ConGu maps this logical structure. Below we focus on the individual role each of these components plays, while highlighting their most interesting features.

ConGu takes as input a set of specification files and one refinement file (cf. Figure 5). In order to simplify the parsing phase (both of specifications and refinements), the **SableCC** parser generator is used [7]. **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 grammar, including the creation of an abstract syntax tree representing the syntactic structure of the contents of the file.
- Visiting the nodes of the tree, while executing certain actions. These actions are specified by extending classes generated by **SableCC**: the tree-walkers.

4.1 The Analyzers

The Specification Analyzer Module (SMA) takes as input a specification module in the form of a list of `.spc` files, 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.

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. As an example, signatures are such that the first parameter of an observer operation signature must have the sort under specification (the *main* sort); the result of any constructor operation must have the *main* sort. There are also strong restrictions on the form of the axioms that depend on the properties of the operations or predicates [1].

The Refinement Binding Analyzer (RBA) takes as input a refinement mapping in the form of a `.rfn` file, and a SpecQuerier object that represents the specification module. 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. In terms of implementation, RBA has the general structure of the Specification Module Analyzer.

While the specification module and the refinement binding are provided to **ConGu** as text files to be parsed, the Java code is presented as bytecode, whose characteristics are obtained by **ConGu** via Reflection [6]. When the refinement analysis encounters a class name it tries to find the class in the classpath. If the class is not found an error is issued; otherwise all information regarding the class (methods, for example) is collected. By using Java Reflection **ConGu** requires bytecode only as input (accessible through the classpath), rather than the original `.java` source code. This strategy has two advantages:

- It allows users to test implementations for which source code is unavailable.
- 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.

The first point above allows to check large programs incorporating both trusted parts (such as the Java API), and parts which we do not completely trust, yet would like to make sure its behavior conforms to the expected (to a given specification).

4.2 The Generators

The back-end of **ConGu** is composed of several components (see Figure 7) whose purpose is to generate Java code from the data synthesized by the front-end. This code comes in the form of bytecode adapted from the input, and of new Java classes that are then compiled within the tool itself. Next we describe the relevant features of the components in the back-end.

The File Generator component acts as the **ConGu** back-end maestro, insofar as it takes the information generated by the front-end, orchestrates the remaining components, and compiles their output. The classes generated (and compiled) by the tool fall into four categories.

Immutable static classes that contain a version of every method in the original classes, and that are equipped with contracts reflecting the axioms and the domain conditions in the specification;

Wrapper classes that contain instances of the original classes, and that force contract monitoring in every call to the methods in the original classes;

Pair classes to hold state-result pairs for non-void methods in the original classes;

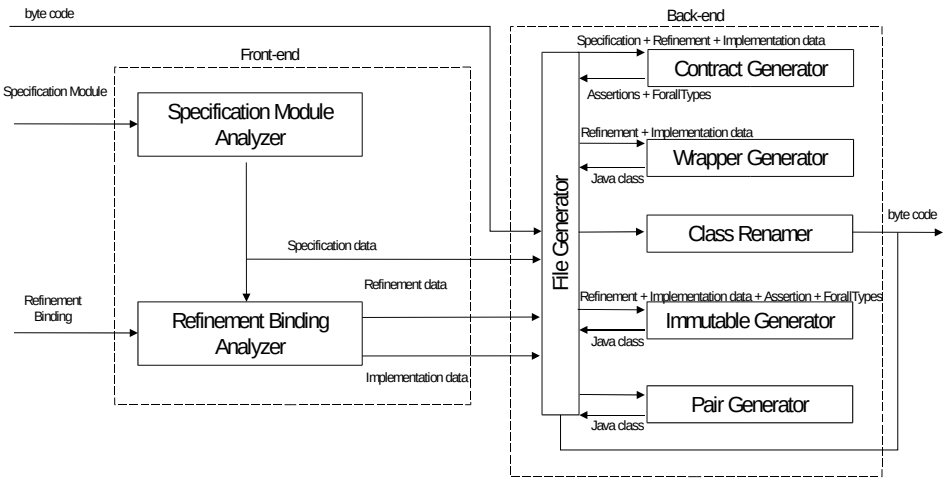

Figure 7: The architecture of ConGu. Each box represents a component.

Range class to be used in **forall** expressions in contracts.

Only the Immutable classes are compiled with `jmlc`, since all JML assertions are gathered at these classes. The remaining generated classes are compiled with `javac`.

The Class Renamer component prepares a newly assembled Java bytecode, `_C_Original.class` from an original bytecode `C.class`, for each class `C` mentioned in the refinement. Towards this end, the component loads the bytecode, updates all explicit and implicit attribute references within the bytecode (including SourceClass, ThisClass, constant-pool NameAndType, constant-pool FieldsInfo , constant-pool MethodsInfo, access to attributes in inner classes access$100) and, recursively, for nested classes. It then writes to disk the thus created bytecode under name `_C_Original.class`.

The Wrapper Generator component creates a wrapper class for each class mentioned in the refinement: the wrapper for class `C` has the same set of public methods (including constructors) and declares an instance of the original `C` class (in the meantime renamed to `_C_Original`) as its only attribute. The wrapper class is called `C`, thus effectively replacing the original class meant to be monitored. In order to force monitoring the methods in the original class, the corresponding wrapper methods make an indirection in every call, invoking instead a method in the corresponding Immutable class. Such a method, in turn, calls the corresponding method in the `_C_Original` class.

The Pair Generator component generates a series of classes defining new types used in the representation of state-value pairs. State-value pairs are required for non-**void** methods that change the state of the object; the typical example being the pop method included in some stack implementations (including the one in the Java API) that removes and returns the element at the top of the stack. The methods in the immutable class return a pair representing the effect of their execution both on the object state and on the value, as witnessed in the code snippet in Figure 8.

The Contract Generator generates JML assertions (pre and post-conditions) that translate the specification domain restrictions and axioms, respectively. Each operation domain restriction (if it exists) is translated to a pre-condition to be attached to the method that refines that operation. In what concerns axioms, the assignment of post-conditions to methods depends on the kind of axiom: *(i)* any axiom that defines values of an observer operation over a constructor operation — e.g., isIn (insert (S, E),F) **iff** E = F **or** isIn (S,F) — gives rise to a post-condition to be attached with the method that refines the constructor operation (in this case, the add method); *(ii)* any axiom that defines the behavior of some operation over a general variable of the main sort, gives rise to a post-condition to be attached to the method that refines that operation. Full details, including the rules for contract generation are described elsewhere [1, 15]. The JML assertions generated by the Contract Generator are then associated with the corresponding method in the relevant Immutable class.

Although **ConGu** generates JML contracts meant to be monitored with the JML run-time assertion checker, its architecture is general enough to encompass other assertion languages and checkers. The **ConGu** modules that generate classes are independent from the contract generation module, in the sense that the latter asks the former for contracts to add to methods in the Immutable classes.

The Immutable Generator component creates one class for each class mentioned in the refinement. Given a class `C`, a static class named `_C_Immutable` is created, containing one method for each public method in `C`. Each method of `_C_Immutable` has for parameters those of the original method, plus one extra: an instance of `_C_Original`. It then invokes the method over a clone of this object and returns

```
/*@
  @ /*largest(S) if not isEmpty(S);*/ requires ...
  @ ...
  @*/
static /*@pure@*/ public _Person_Pair_PersonSet
    largest(_PersonSet_Original s) {
  _PersonSet_Original c=(_PersonSet_Original)clone(s);
  return new _Person_Pair_PersonSet(c.largest(), c);
}

/*@
  @ /*isIn(insert(S, E),F) iff E = F or isIn(S,F);*/
  @  ensures ...
  @ ...
  @*/
static /*@pure@*/ public _PersonSet_Original add(
    _PersonSet_Original s, Person p) {
  _PersonSet_Original c=(_PersonSet_Original) clone(s);
  c.add(p);
  return c;
}
```

Figure 8: Excerpt of _PersonSet_Immutable class generated by ConGu.

the result and/or the object itself, depending on the return type of the original method.

This component reads the contracts generated by the Contract Generator component and associates them to the corresponding method in the immutable class. An excerpt of the composed result of the output produced by components Immutable Generator and Contract Generator for two methods in class `_PersonSet_Immutable` is in Figure 8.

4.3 Further Issues

Range class. The translation of certain forms of axioms require **forall** expressions in contracts. One such example is axiom

```
isIn(insert(S, E),F) iff E = F or isIn(S,F);
```

in Figure 3 that becomes a post-condition to method add in Figure 8 (cf. Section 2). In order to iterate over all Orderable F, an attribute range of class _Range equips the immutable class. _Range is a class (independent of any sort) that implements a bounded collection, allowing to generate JML **forall** code.

```
(\forall Person f; range.contains(f); ...)
```

All Person objects, parameters to the methods of the immutable class (hence to the methods of the original class PersonSet) or returned by these, are placed in the range object, via a call to method **boolean** put(Person f) (that always returns true) from within the contracts. The maximum size of this collection clearly affects the runtime of the monitoring process; see reference [15] for benchmarks.

Wrap and unwrap. Class C under test coexists with the surrogate class prepared by **ConGu**. After running the tool the former is called _C_Original, while the latter C. There are occasions when conversion is required: contracts deal with _C_Original objects; client code expects C objects. Axiom

```
largest(insert(S,E)) = E if greaterThan(E,largest(S));
```

generates a post-condition for method add that calls method older in class _Person_Immutable. Such a method accepts an object of class _Person_Original, hence the contract code must unwrap the argument.

```
ensures..._Person_Immutable.older(Person._unwrap(e),...)..
```

In the other direction, objects must be delivered to clients as they expect them, as PersonSet for example. Suppose our example includes an operation to obtain the subset with the elements smaller than a given element. In this case, the generated class PersonSet would include the following method.

```
public PersonSet lowerSet(Person p) {
  _PersonSet_Pair_PersonSet _pair =
    _PersonSet_Immutable.lowerSet(_wrappedObject, p);
  _wrappedObject = _pair.state;
  return _wrap(_pair.value);
}
```

Obtaining a _C_Original from a C object is easy since the latter holds the corresponding _C_Original as attribute. For the reverse direction, each wrapper class maintains a (static) hash table that collects mappings (_C_Original, C). Such a scheme guarantees the correct behavior of == in client code, for objects of classes under monitoring. The hash table that maps _C_Original objects into C objects can become quite large. The usage of the WeakHashMap class allows garbage collection of the held references.

Clone and equals. As discussed in Section 2, there is a close relationship between method clone and equals. **ConGu** checks that classes either both declare a clone method and implement Cloneable, or do neither. In the latter case it alerts to the fact that objects will not be cloned, which should happen only for immutable classes. **ConGu** prepares contracts for clone and equals in class _C_Immutable. For the former, the following code is generated, where we have abbreviated PersonSet to PS.

```
/*@
  @ /* Clone */ ensures equals(t, \result).value;
  @*/
static /*@pure@*/ public Object clone(_PS_Original t) {
  return t.clone();
}
```

For the latter, we take the view that any two terms that are regarded as equal must produce equal values for every observer operation and predicate. In order to check the consistency of an implementation in what respects these properties, we generate post-conditions for the equals method that test the results returned by all methods that implement observer operations and predicates [15], as illustrated below.

```
/*@
 ...
 @ /* Observer operation (isEmpty: Set) */
 @ ensures \result.value ==>
 @   o instanceof _PS_Original &&
 @   (_PS_Immutable.isEmpty(t).value <==>
 @     _PS_Immutable.isEmpty((_PS_Original) o).value);
 @*/
static /*@pure@*/ public _boolean_Pair_PS
    equals(_PS_Original t, Object o) {
  if (t == null)
    return new _boolean_Pair_PS(t == o, t);
  _PS_Original c = (_PS_Original)clone(t);
  return new _boolean_Pair_PS(c.equals(o), c);
}
```

Strong equality. Section 1 introduces the definedness condition whereby the meaning of an equality t1 = t2 in the axioms of a specification is that the two terms are either both defined and have the same value, or they are both undefined. Then, definedness condition of an operation invocation is the conjunction of the definedness conditions of its arguments and the domain condition of the operation itself [15]. An excerpt of the post-condition produced for method add (cf. Figure 8) is as follows.

```
ensures
  !_PS_Immutable.isEmpty(t).value &&
  _Person_Immutable.older(_unwrap(e),
      _unwrap(_PS_Immutable.largest(t).value)).value
  ==> ...
```

Applicability and Limitations.

- Partial class specification is supported, for the wrapper class produced contains a method for each public method in the original class, irrespective of the method being mentioned in the refinement. In the example in this paper, it is conceivable that class Person has a lot more methods than those appearing in the refinement in Figure 4.
- Constructor operations can be refined into the **null** expression. This is particularly useful for methods that return **null** on particular cases. One such example is the get method of a map that returns **null** if the key is not in the table.
- Refinement into java.lang classes is supported as long as all operations are refined into **null**, the reason being that JVM internally uses objects of these classes, making it difficult to monitor their execution. Refinement into any other class in the Java API is fully supported.
- Refinement into Java 5 generic classes is supported as long as the erased type of type parameters is Object. This allows for example to refine the specification of a map into, say HashMap<Key, Value>. Further, when all operations of a given sort are refined into **null**, bounded polymorphism is accepted. For example, a set (without the larger operation) can be refined into class **class** PersonSet <E **extends** Comparable<E>>.
- Refinement into Java interfaces is not yet supported, nor is inheritance, that is to say, we have no way to define a specification T' as an extension of another specification T, and control their refinement into classes that are related through inheritance.
- The contracts, as generated by the tool, are not meant to be read by humans. They are usually quite long and intricate, particularly because of the definedness conditions for formulæ of the specification language [15].

5. OUR EXPERIENCE WITH CONGU

We have experimented the tool with a group of 3rd and 4th grade students, which helped us finding relevant limitations of **ConGu** within a learning context. In a first phase of the experience, students were given a specification module and a refinement mapping, and were asked to implement (and manually test) two Java classes according to the material given. In a second phase, students were asked to use the tool to monitor the code they had written in the first phase, and to produce a report containing all violations detected (pre and post-condition exceptions) and how each problem was solved.

This gave us a precious input which we used to ameliorate the tool: for example, we found that the feedback given to the user whenever a specification violation occurred was truly insufficient. Although we still aim at a much better feedback — it is our intention to equip classes with human-readable contracts — we have improved the output so that exceptions thrown by pre and post-condition violations now explicitly mention the domain condition or the axiom in the original specification, as described in Section 3. Also, the tool now compiles the Java code produced, as opposed to have the user decide when to use `javac` and `jmlc` to compile the output.

ConGu is in use since 2005, in the context of a first year, second semester, undergraduate course on algorithms and data structures at the University of Lisbon. In this course students are introduced to new data types through their algebraic specifications, which they must understand and implement.

During a first semester programming course students are introduced to Design by Contract [14], where they learn how to write simple pre and post-conditions for their methods. Afterward, in the second semester, and before the advent of **ConGu**, they would manually translate algebraic specifications domain restrictions into pre-conditions, and axioms into post-conditions. During the process they would understand that there were a number of axioms that could not be translated into contracts (either because these implied operations refined into **void** methods, or because of the machinery required for **forall** contracts). These limitations were rather frustrating and, in some cases, made students disbelieve the design by contract methodology.

Since 2005, students taking this course routinely use **ConGu** to monitor their implementations against specifications provided by the teaching staff. They now have a means to fully test their classes since **ConGu** generates contracts for each and every axiom and domain restriction in the specification. The teaching staff, on the other hand, write specifications for *all* the data structures addressed in the course, and test their code with **ConGu**, before offering it to students. Two years of intense use have made the tool quite robust, and have helped to find many defects in the implementation.

6. RELATED WORK

There is a vast amount of work in the field of specification and verification of algebraic specifications and software components in general; the interested reader may refer to previous publications [8, 10] for a survey. Here we focus on attempts to check OO implementations for conformance against property-driven algebraic specifications.

Henkel and Diwan developed a tool [11] that allows to check the behavioral equivalence between a Java class and its specification, during a particular run of a client application. This is achieved through the automatic generation of a prototype

```
forall l:LinkedList forall o:Object forall i:int
  removeLast(add(l, o).state).retval == o
  if i > 0 get(addFirst(l, o).state,
    intAdd(i, l).retval).retval == get(l,i).retval
```

```
axioms
  l: List; o: Element, i: int;
  removeLast(add(l, o)) = o;
  get(addFirst(l, o), i) = get (l, i-1) if i > 0;
```

Figure 9: An example of the specification of two properties of linked lists as they are presented by Henkel and Diwan and as they would be specified in our approach.

implementation for the specification which relies on term rewriting. The specification language adopted is, as in our approach, algebraic with equational axioms. The main difference is that their language is tailored to the specification of properties of OO implementations whereas our language supports more abstract descriptions that are not specific to a particular programming paradigm. Being more abstract, we believe that our specifications are easier to write and understand.

Figure 9 presents an example. The axioms define that operation removeLast returns the last element that was added to the list and define the semantics of the get operation: get(l, i) is the i-th element in the list l. The symbols retval and state are primitive constructs of the language adopted by Henkel and Diwan [11] to talk about the return value of an operation and the state of the current object after the operation, respectively (cf. our pair classes in Section 4.2).

When compared with our approach, another difference is that their language does not support the description of properties of operations that modify other objects, reachable from instance variables. In contrast, our approach supports the monitoring of this kind of operation.

Another approach whose goal is similar to ours is Antoy and Hamlet's [2]. They propose an approach for checking the execution of an OO implementation against its algebraic specification, whose axioms are provided as executable rewrite rules. The user supplies the specification, an implementation class, and an explicit mapping from concrete data structures of the implementation to abstract values of the specification. A self-checking implementation is built that is the union of the implementation given by the implementer and an automatically generated direct implementation, together with some additional code to check their agreement. The abstraction mapping must be programmed by the user in the same language as the implementation class, and asks user knowledge about internal representation details. Here lies a difference between the two approaches: our refinement mapping needs only the interface information of implementing classes, and it is written in a very abstract language. Moreover, there are some axioms that are not accepted by their approach, due to the fact that they are used as rewrite rules; for example, equations like insert (insert (S, E), F) = insert(insert (S, F), E) (cf. Figure 3) cannot be accepted as rewrite rules because they can be applied infinitely often.

We further believe that the rich structure that our specifications present, together with the possibility to, through refinement mappings, map a same module into many different packages all implementing the same specification, is a positive point in our approach that we cannot devise in the above referred approaches.

7. CONCLUSION AND FURTHER WORK

We contextualized the **ConGu** tool within the framework we developed to test Java implementations against property-driven, algebraic specifications. The idea underlying this approach, which encompasses automatic generation of contracts, is to combine the possibility of building meaningful and complete specifications with the capacity of checking implementations — users are able to monitor *all* the axioms and domain restrictions in their specifications.

We described the **ConGu** tool both from the user's and from the architect's point of view, emphasizing particular aspects of each of the tool components, and pointing to its applicability and restrictions.

We think there are still some aspects that need to be addressed, despite the fact that the tool is now fully operational and in use.

We intend to investigate the best way to solve the problem of side-effects in contract monitoring due to changes in the state of method parameters—our approach does not cover this problem yet. Cloning all parameters in every call to a method in the generated immutable class—as we do for the target object—does not seem a plausible solution. We think a better solution would allow the user to explicitly indicate in the refinement mapping whether parameters are modified within methods (the default being that they are not modified).

The relation between domain conditions of specifications and exceptions raised by implementing methods is also a topic to investigate and develop, insofar as it would widen the universe of acceptable implementation classes.

A further topic for future work is the generation, from specifications and refinement mappings, of Java interfaces annotated with human readable contracts. Once one is convinced that given classes correctly implement a given module, it is important to make this information available in the form of human-readable contracts to programmers that want to use these classes and need to know how to use and what they can expect from them.

The refinement of sorts into primitive Java types has already been studied in the context of the **ConGu** approach but it has not yet been implemented at the level of the refinement language that the **ConGu** tool accepts. At present, the specification language supports the int primitive type that is automatically mapped into the int Java primitive type; the refinement language does not support refinement into primitive types.

Acknowledgments. Thanks are due to José Luiz Fiadeiro

for insightful discussions, to João Abreu for designing and initially implementing the tool, and to Alexandre Caldeira for fruitful input.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] João Abreu, Alexandre Caldeira, Antónia Lopes, Isabel Nunes, Luís S. Reis, and Vasco T. Vasconcelos. Congu—checking Java classes against property-driven algebraic specifications. DI/FCUL TR 07–7, Department of Informatics, Faculty of Sciences, University of Lisbon, March 2007.

[2] S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. *IEEE TOSE*, 26(1):55–69, 2000.

[3] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Proc. WSVCBS — OOPSLA 2001*, 2001.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proc. of CASSIS 2004*, number 3362 in LNCS. Springer, 2004.

[5] D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass - Java with assertions. *ENTCS*, 55(2), 2001.

[6] Mary Campione, Kathy Walrath, Alison Huml, and Tutorial Team. *The Java Tutorial*. Sun Microsystems, online edition, 2006. http://java.sun.com/docs/books/tutorial/.

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

[8] J. Gannon, J. Purtilo, and M. Zelkowitz. Software specification: A comparison of formal methods, 2001.

[9] J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[10] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proceedings of ECOOP 2003*, LNCS, 2003.

[11] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proc. ICSE 2004*, 2004.

[12] Rachel Henne-Wu, William Mitchell, and Cui Zhang. Support for design by contract in the C# programming language. *Journal of Object Technology*, 4(7):65–82, 2004.

[13] Java Modelling Language. `http://www.jmlspecs.org/`.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997.

[15] Isabel Nunes, Antónia Lopes, Vasco T. Vasconcelos, João Abreu, and Luís 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.

[16] J. Spivey. *The Z Notation: A Reference Manual*. ISCS. Prentice-Hall, 1992.