Monitoring Java Code Using **ConGu**

V.T. Vasconcelos, I. Nunes, A. Lopes, N. Ramiro, P. Crispim

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

Abstract

The **ConGu** project aims at developing a framework to create property-driven algebraic specifications and to test Java implementations against these specifications. 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 bytecode against a module of specifications, and to discover violations of specified properties. Towards this end, the tool generates intermediate classes equipped with contracts, and wraps the bytecode under test in newly generated classes that allow contract monitoring, in a way that is transparent to the clients of the original classes.

1. Introduction

The formal specification of software components is an important activity in the process 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) [13] is widely used for the specification of object-oriented software. There are a number of languages and tools (e.g., [4,5,11,12]) that allow equipping classes and methods with invariants, and 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, which is usually an extension of the language of boolean expressions of the OO language.

To build contracts using these languages one must observe the following: *(i)* contracts are built from boolean assertions, thus procedures (methods that do not return values) cannot be used; *(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.

Model-based approaches to DbC, like those proposed for Z [15], Larch [8], JML [12], 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, introduced in reference [14].

The key idea of our 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 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 JML contracts are generated that cover them all. The tool has been in use since 2005 in the context of an undergraduate course on algorithms and data structures at the University of Lisbon, where student's code for *all* data-structures covered in the course is checked against algebraic specifications.

This paper presents **ConGu** (Contract Guided System Development [6]), a framework to create property-driven algebraic specifications and to fully test Java implementations against these. Support for checking implementations against algebraic specifications is, as far as we know, restricted to a few approaches (see [7,9] for a survey), which are either tailored to the specification of properties of OO implementations [10], or requires programming an abstract mapping from the class to the specification [2].

ConGu can be used from the command line or from within Eclipse. In either case it requires JDK 1.5 or above, JML 5.4 or above, and runs on multiple platforms (Windows XP and Vista, Mac OS X, and Linux). The current version of the Eclipse plugin includes customized editors, allowing for syntax highlighting and problem marking, as well as menu options for verification and compilation.

The next section describes Congu from a user's point of view by means of a running example. Section 3 presents a brief overview of the input/output behaviour of the tool, and the following section, a description of its architecture. Section 5 discusses the features and limitations of **ConGu**, and Section 6 concludes the paper.

2. Using ConGu

The inputs to the tool are *specification modules* and *refinements*, and of course, Java code to be monitored. 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. Operations, which can be interpreted by partial functions, are defined through their signature, restrictions on their domain, and axioms defining their properties. The exact nature of the specification language is described in references [1,14]; here we introduce its main features based on an example.

The screenshot in Figure 1 shows a module composed of two specifications.

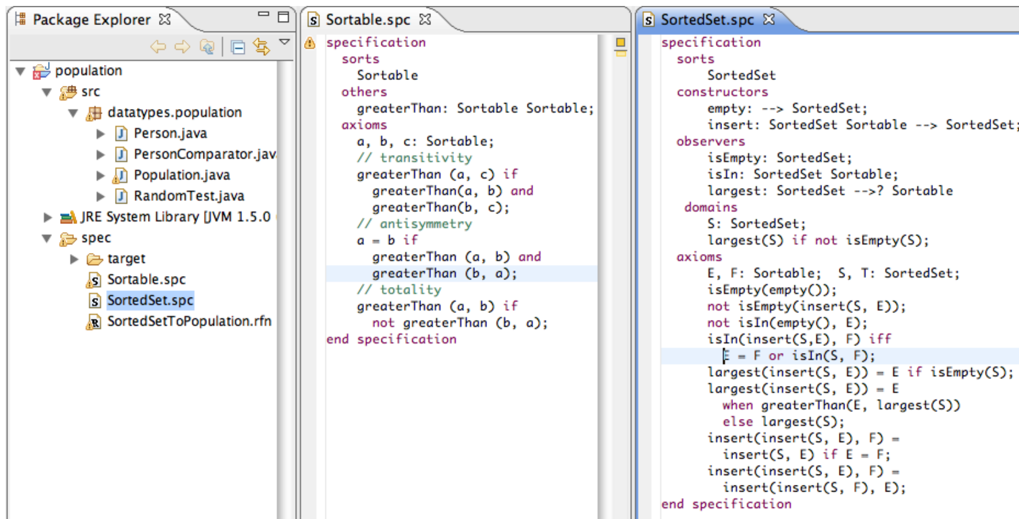

Fig. 1. A module composed of two specifications

Each specification defines exactly one sort, Sortable and SortedSet in this case. Operations are classified as **constructors**, **observers**, or **others**. These categories comprise, respectively, the operations from which all values of the type can be built, the operations that provide fundamental information about the values of the type, and the remaining operations. Predicates can only be classified as either **observers** or **others**.

In the example we find two constructors for SortedSet, one that builds an empty set; the other that builds a set from a given set and an element. As observers we have two predicates (to check whether the set is empty, and to verify whether a given element belongs to a set), and an operation (to obtain the largest element in the set).

The largest operation may not be defined for an empty set; that is the reading of the **domains** section. The last section, **axioms**, describes the relationship between the various operations. For example, the first axiom reads: an empty set (that is, the set constructed with the empty operation) is empty, and the last axiom states that insertion order is irrelevant.

The specification for Sortable elements defines a standard total order relation: transitive, antisymmetric, and total. The two specifications together are self-contained, in the sense that all external references are defined therein. We call such a collection of specifications a *module*.

Figure 2 presents an example of the two classes to be monitored against the specifications presented in Figure 1. In general an application will have many classes, of which only a subset is to be monitored. In our example, the application besides these two classes comprises two other classes: PersonComparator to be used in the search tree that implements Population, and RandomTest which simply executes a fixed number of randomly chosen operations of class Population.

Classes under test must have proper equals methods, expressing that objects are equal if it is not possible to distinguish between them using any observers of their type (also called *similarity*). Moreover, the classes under test that define mutable types, must be cloneable with clone methods that 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. In our example, class Person defines a immutable type (i.e., the state of objects of this type never change) and does not implement Cloneable while class Population defines a mutable type and does implement Cloneable.

The last ingredient of **ConGu** links the world of specifications (in Figure 1) to the world of Java implementations (in Figure 2). A *refinement mapping* indicates which sort is implemented by which class, and which operation is implemented by which method.

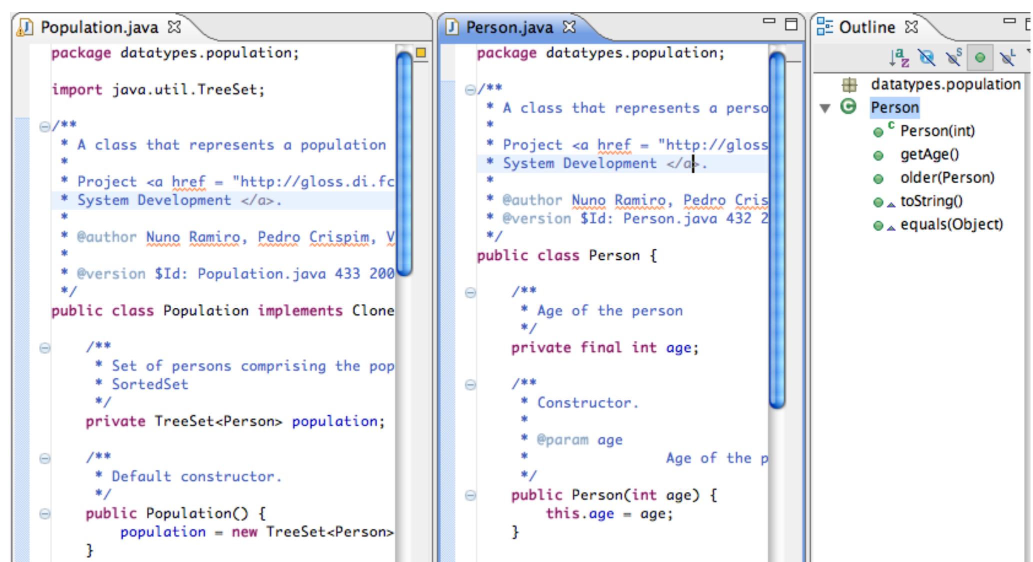

Fig. 2. The Java classes to be monitored

The mapping in Figure 3 links sort Sortable to class Person and sort SortedSet to class Population. On what concerns operations, one can see that, for example, operation greaterThan is mapped to method older in class Person. The example also shows that not all methods need to be the target of refinements: class Person is composed of five public methods of which one only (older) will be directly monitored by the tool.

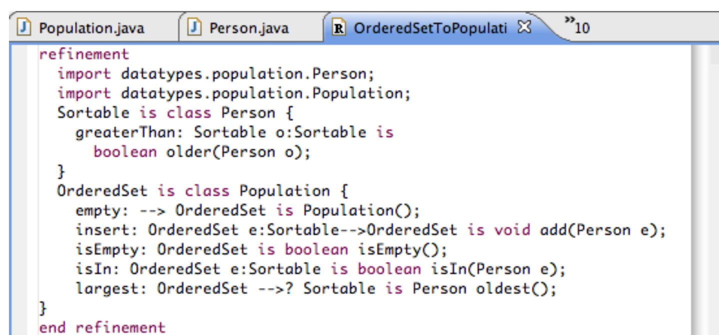

Fig. 3. A mapping from the module in Figure 1 and the classes in Figure 2

Once all problems regarding specifications, refinements and Java code have been fixed, time comes to run the application under the scrutiny of the contracts generated by **ConGu**. For this purpose, the plugin provides a new entry in the Run Dialogue facility of Eclipse, as shown in Figure 3.

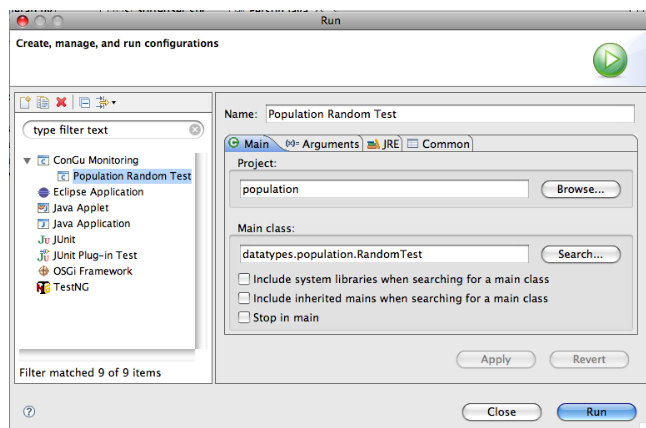

Fig. 4. Monitoring the application

Running the original application under Congu may produce pre-condition exceptions due to domain condition violations or to post-condition exceptions due to axiom violations. The first case is always a manifestation of a ill-behaved client, i.e., a client that invoked a method in a situation in which the method shouldn't be invoked. For instance, in our example, a pre-condition exception would be produced if RandomTest class calls the oldest method on an empty population. The second case is a manifestation of a faulty supplier class. One of the classes under test is failing to ensure at least one of the specified properties.

The output produced by **ConGu** then guides the developer into the violated domain condition or axiom, from where she can start looking for the defect. The screenshot in Figure 5 indicates a post-condition error, revealing a fault on the supplier class Population, while monitoring axiom largest (insert (S,E)), immediately after a particular execution of method add. This suggests to start looking for the defect in method oldest or add in class Population.

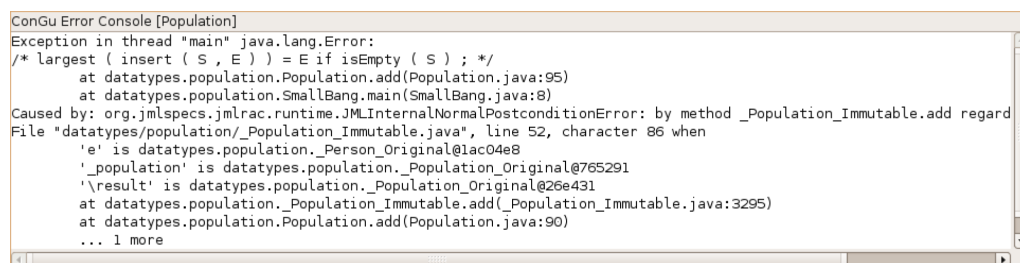

Fig. 5. The console showing a post-condition error, thus signaling a defect in the producer

3. Overview of the tool

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.

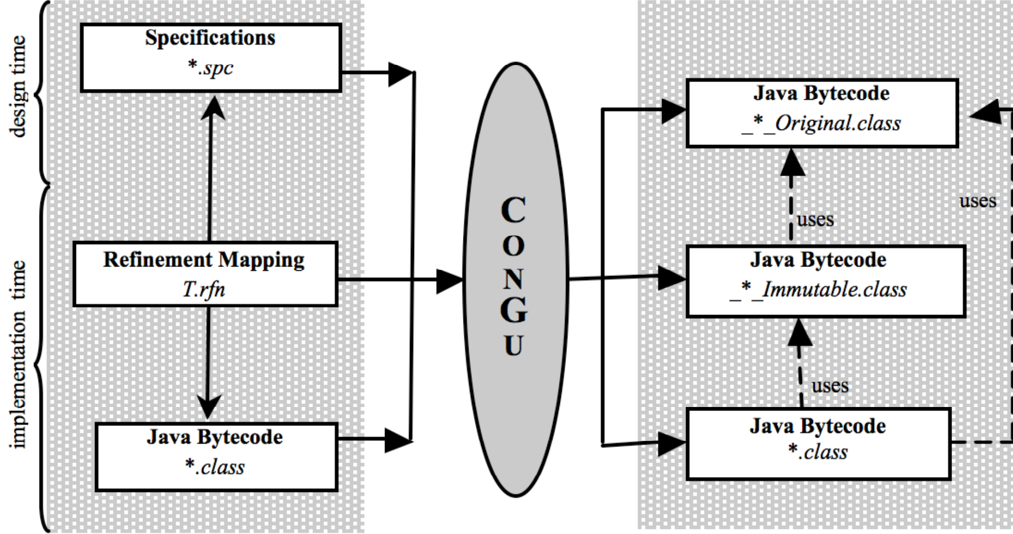

Fig. 6. The input and the output of **ConGu**

Figure 6 shows that the input of **ConGu** consists of a specification module, a refinement, and Java bytecode. For each class `C` mentioned in the refinement, the tool renames `C.class` bytecode file into `_C_Original.class`, and creates a new `C.class` that wraps and substitutes the original code. It further generates a static class, `_C_Contract.java`, equipped with contracts regarding specifications axioms and domain conditions, that is used by the wrapper class to force contract checking.

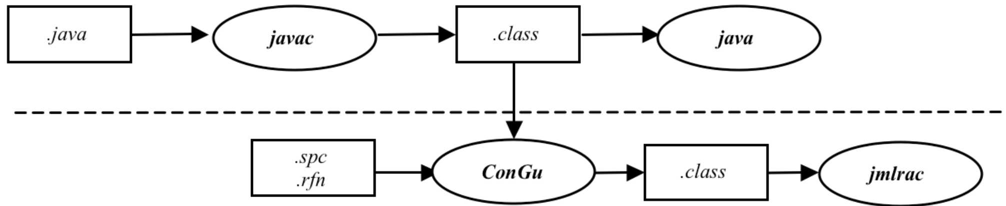

Fig. 7. Traditional versus **ConGu** workflow

Contracts are generated in the The Java Modeling Language (JML [12]). The new program must be executed using `jmlrac`[1] rather than `java`, in order to monitor the

[1] A script for running java programs compiled with the JML runtime assertion checker compiler.

contracts that the tool generates. The process is described in Figure 7, where standard Java-based workflow is depicted above the dashed line, and the **ConGu**'s workflow below the same line.

4. Architecture

ConGu is divided into two projects: the compiler and the Eclipse plugin. The compiler exposes its functionality as an API that is used both by the plugin and by a command line executor.

The plugin extends in several ways the features and facilities provided by the Eclipse IDE and its Java Development Tools, therefore requiring the latter to be installed. Extending the typical Java project, the **ConGu** project adds contract monitoring functionality, providing module directories for the contract specifications files. Furthermore, specifications development is enhanced by customized editors, allowing for syntax highlighting and problem marking, as well as menu options for verification and compilation, also available as toolbar buttons. Compiler output is integrated into the problems tab, providing easy navigation to the respective file and even line by double-clicking on the problem, as well as into the console tab, at several degrees of verbosity, allowing a more detailed description of the problems. In addition to that, execution of the application in monitoring mode is also possible from within Eclipse and its output presented in the console tab, in coherence with the IDE's environment. All these features, and more, can be configured in the Eclipse preferences menu, some being project-dependent and modified in the respective project's properties dialog.

The compiler is organized in several logical components, as described in Figure 8. Components Specification Module Analyzer, Refinement Binding Analyzer, and Bytecode Analyzer make up the front-end of **ConGu**. The back-end, formed by various generators and the Class Renamer, produces and compiles Java code. The classes generated (and compiled) by the tool fall into four categories:

Contract (static) classes that contain a version of every method in the original classes, and that are equipped with JML 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;

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

Only Contract 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.

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 and declares an instance of the original `C` class (in the meantime renamed to `_C_Original`) as its only attribute.

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

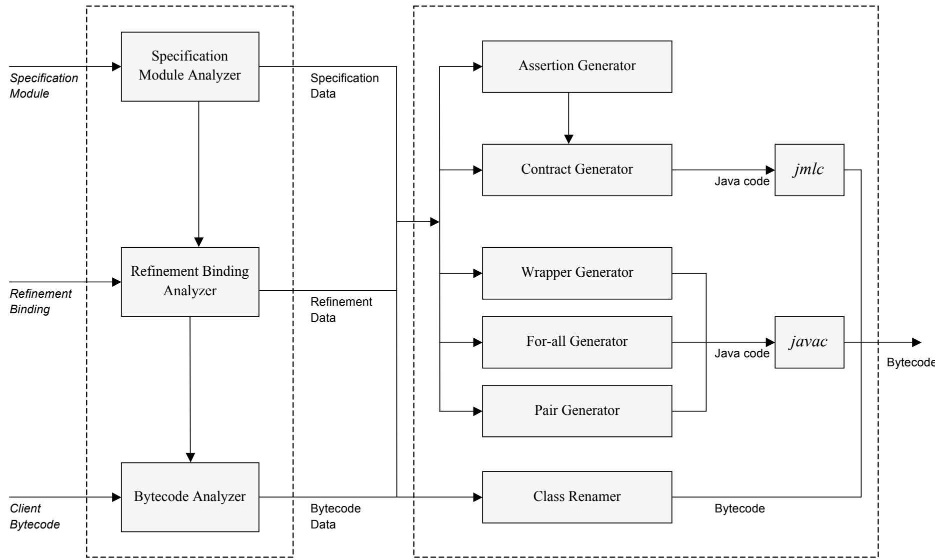

Fig. 8. The architecture of the **ConGu** compiler

in some stack implementations (including the one in the Java API) that removes and returns the element at the top of the stack.

The Assertion Generator generates JML assertions (pre and post-conditions) that translate the specification domain restrictions and axioms, respectively. These assertions are then inserted in a class prepared by the Contract Generator. The modularity thus obtained allows to quick replace the language used to express the contracts.

Full details, including the rules for contract generation are described elsewhere [1,14]. Given a class `C`, a static class named `_C_Contract` is created, containing one method for each public method in `C`. Each method of `_C_Contract` includes a parameter for each in the original method, plus one extra: an instance of `_C_Original`. It then invokes the method over a clone of this object and returns the result and/or the object itself, depending on the return type of the original method.

Figure 9 shows an excerpt of the output produced for class `_PersonSet_Contract`. These contracts are not meant to be read by humans; they are usually quite long and intricate, due to the four reasons discussed below.

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);
```

that becomes a post-condition to method _PopulationOriginal add in Figure 9. In order to iterate over all Sortable F, an attribute _Range range equips the contract class. Class _Range implements a bounded collection, allowing to generate JML code of the form

```
/*@
  @ /* largest ( S ) if not isEmpty ( S ) ; */ requires true ==> !(datatypes.population._Population_Immutable.isEmpty(_p
  @ ensures _domaindatatypes_population_Person._put(\result.value);
  @*/
static /*@pure@*/ public _Person_Pair_Population oldest(_Population_Original _population) {
    _Population_Original _clone = clone(_population);
    return new _Person_Pair_Population(_clone.oldest(), _clone);
}

static /*@pure@*/ public _boolean_Pair_Population isEmpty(_Population_Original _population) {
    _Population_Original _clone = clone(_population);
    return new _boolean_Pair_Population(_clone.isEmpty(), _clone);
}

/*@
  @ /* insert ( insert ( S , E ) , F ) = insert ( insert ( S , F ) , E ) ; */ ensures (\forall datatypes.population.Pers
  @ /* largest ( insert ( S , E ) ) = largest ( S ) if not ( greaterThan ( E , largest ( S ) ) ) ; */ ensures true && tr
  @ /* largest ( insert ( S , E ) ) = E if greaterThan ( E , largest ( S ) ) ; */ ensures true && true && (!(datatypes.p
  @ /* not isEmpty ( insert ( S , E ) ) ; */ ensures (true && true && true) ==> !(datatypes.population._Population_Immut
  @ /* largest ( insert ( S , E ) ) = E if isEmpty ( S ) ; */ ensures true && datatypes.population._Population_Immutable
  @ /* insert ( insert ( S , E ) , F ) = insert ( S , E ) if E = F ; */ ensures (\forall datatypes.population.Person F;
  @ /* isIn ( insert ( S , E ) , F ) iff E = F or isIn ( S , F ) ; */ ensures (\forall datatypes.population.Person F; _c
  @ ensures _domaindatatypes_population_Person._put(e);
  @*/
static /*@pure@*/ public _Population_Original add(_Population_Original _population, datatypes.population.Person e) {
    _Population_Original _result = clone(_population);
    _result.add(e);
    return _result;
}
```

Fig. 9. The contracts produced for class Population

(\ **forall** Person f; range.contains(f); ...). All Person objects (parameters to the methods of the contract class or returned by these) are placed in the range object. The maximum size of this collection is one of the factors that more affects the running time of the monitoring process; see reference [14] 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. Obtaining a _C_Original from a C object is easy since the latter holds the corresponding _C_Original as an 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 the == operator in client code, when used with objects of classes under test.

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 contract classes (the case of class Person in Figure 2, for example). The tool prepares contracts for clone and for equals in class _C_Contract. For the former, the following code is generated.

```
//@ ensures equals(t, \result).value;
static /*@pure@*/ public Population clone(_Population_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 [14].

Strong equality. 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 [14]. The screenshot above shows a redundant "(**true** && **true** && **true**) ==>" in the contract for method add, since, at the time of this writing, we do not simplify the generated assertions.

5. Applicability and Limitations

- Although **ConGu** generates contracts meant to be monitored with the JML runtime assertion checker, its architecture is general enough to encompass other assertion languages and checkers: there is one single class with contracts per specification in the input module and the component that generates these contracts can be easily replaced by a different one generating pre and pos-conditions in a different assertion language. Notice that although JML provides support for a variety of aspects, such as, invariants, abnormal behaviour, model variables, etc, **ConGu** relies exclusively on assertions for pre and post-conditions.
- **ConGu** requires bytecode only as input, rather than source code. This strategy has two advantages: i) it allows users to test implementations for which source code is unavailable, thus permitting to check large programs incorporating both trusted parts (such as the Java API), and parts not completely trusted but that we would like to make sure it behaves as expected, i.e., conforms to a given specification; ii) it simplifies the implementation of **ConGu** by avoiding the need to parse and analyze Java source code.
- Java code is kept separated from specifications, allowing several Java classes or packages to be tested against the same specification.
- Java code is kept separated from refinement bindings, meaning that the same Java code can be easily tested against different specifications, even if this requires a bit of an overhead with respect to, say, Java annotations.
- Partial class specification is supported; the wrapper class produced contains a method for each public method in the original class, irrespective of the methods mentioned in the refinement. In our example, it is conceivable that class Person has a lot more methods than those appearing in the refinement.
- 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 map.
- Refinement into classes of the Java API is supported, subject to the restriction that java.lang classes can only be used 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 Java 5 generic classes is supported as long as the upper bound of the type parameters is Object. This allows, for example, to refine the specification of a map into, say, HashMap<Key, Value>.
- Refinement into Java interfaces is not yet supported, nor is inheritance.
- The tool does not support changes to the state of objects passed as parameters. Although this feature is not particularly important for implementing abstract data types, this limitation does not allow to uncover changes of parameters introduced inadvertently by programmers.

6. Further Work

In this paper, we presented a tool for runtime monitoring of Java programs against property-driven, algebraic specifications. The tool was developed with the goal of helping the adoption of formal specifications in the realm of abstract data types implementations. Our experience of using it in the context of an undergraduate course on algorithms and data structures has been confirmed that these specifications are indeed accessible and the approach is effective in helping to find errors.

As future work, we intend to pursue investigation on detecting the side-effects in contract monitoring due to changes in the state of method parameters.

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.

Acknowledgments. This work was partially supported by FCT, through the Multiannual Funding Programme.

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 Transactions on Software Engineering*, 26(1):55–69, 2000.

[3] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Workshop on Specification and Verification of Component-Based Systems*, 2001. Published as Iowa State Technical Report 01-09a.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[5] D. Bartetzko, C. Fisher, M. Möller, and H. Wehrheim. Jass, Java with assertions. In *Proceedings of the First Workshop on Runtime Verification*, volume 55(2) of *ENTCS*. Elsevier, 2001.

[6] Congu: Monitoring Java code against algebraic specifications. `http://gloss.di.fc.ul.pt/congu/`.

[7] John D. Gannon, Marvin V. Zelkowitz, and James M. Purtilo. *Software Specification: A Comparison of Formal Methods.* Greenwood Publishing Group Inc., 1994.

[8] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[9] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proceedings of the 17th European Conference on Object-Oriented Programming 2003*, volume 2743 of *LNCS*, pages 431–456. Springer, 2003.

[10] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458. IEEE Computer Society, 2004.

[11] 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.

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

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

[14] 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 the International Conference Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 494–513. Springer, 2006.

[15] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall, 1992.