Runtime Verification for Generic Classes with ConGu2*

Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos

LaSIGE and Faculty of Sciences, University of Lisbon,
Campo Grande, 1749–016 Lisboa, Portugal,
{pedro.crispim,mal,vv}@di.fc.ul.pt

Abstract. Even though generics became quite popular in mainstream object-oriented (OO) languages, approaches for checking at runtime the conformance of such programs against formal specifications still lack appropriate support. In order to overcome this limitation within CONGU, a tool-based approach we have been developing to support runtime conformance checking of Java programs against algebraic specifications, we recently proposed a notion of refinement mapping that allows to define correspondences between parametric specifications and generic classes. Based on such mappings, we also put forward a notion of conformance between the two concepts. In this paper we present how the new notion of conformance is supported by version 2 of the CONGU tool.

1 Introduction

The formal specification of software components is an important activity in the process of software development, insofar as specifications are useful, on the one hand, to understand and reuse software and, on the other, to automatically verify the correctness of components implementations. Among the several approaches that can be adopted for automatically analysing the reliability of software components one finds *runtime verification*. This approach involves the monitoring and analysis of system executions. As the system executes, the behaviour of its components is tested for correction with respect to the specification. Runtime monitoring has the advantage that can be used to analyse properties for which static verification fails. Moreover, it does not require the user expertise and effort typically required of a static verification system.

Although generics became quite popular in mainstream OO languages, existent approaches for runtime checking the conformance of generic OO programs against formal specifications still lack appropriate support. This was also the case of CONGU, a tool-based approach to runtime verification of Java implementations against algebraic specifications [10, 17]. CONGU is intensively used by our undergraduate students in the context of a course on algorithms and data structures for checking abstract data types (ADTs) implementations.

Given that generics became extremely useful and popular in the implementation of ADTs in Java, in particular those that are traditionally covered in such courses, the lack

* This work was partially supported by FCT through the project QUEST (PTDC/EIA-EIA/103103/2008).

of support for generics became a major drawback. In order to overcome this limitation, we recently proposed a notion of refinement mapping that allows to define correspondences between parameterized specifications and generic classes [16]. Based on such mappings, we also put forward a more comprehensive notion of conformance between Java programs and algebraic specifications. This work paved the way for the extension of runtime conformance checking to a more comprehensive range of situations. In this paper, we present a new approach to runtime conformance checking of Java implementations against specifications (applicable to parameterized specifications) and discuss how this solution is realized in the new version of CONGU tool.

The solution for runtime checking that was developed in order to accommodate generics is substantially different from that used before in CONGU [17]. Therein, the strategy was to replace the original classes by proxy classes and generate further classes annotated with monitorable contracts, written in JML [14]. The main innovative aspects of the solution adopted in CONGU2 are the following:

- Introduction of new mechanisms that allow to deal with generics, namely to check whether classes used to instantiate the parameters of generic classes conform to what was specified in the parameter specifications.
- Original classes are not replaced by generated proxy classes. Instead, the solution now relies on the instrumentation of the bytecode of original classes, overcoming the difficulties on the generation of appropriate proxy classes for classes making use of, e.g., public fields or inner classes.
- JML, which does not support generics (among other features introduced in Java 1.5 [9]), is no longer used. Instead, runtime checking of the specified properties at specific execution points is now achieved directly by the generated code, relying only on Java assertions. The compilation with *jmlc* of contract annotated classes was a bottleneck in terms of performance and, with the new solution, we were able to substantially reduce the compile time.

The remainder of the paper is organised as follows. In Section 2 we provide an overview of the CONGU approach, namely we introduce specifications and refinement mappings adopted in CONGU. Then, Section 3 presents the notion of conformance between specifications and Java classes and discusses the properties induced by specifications that are monitored at runtime. The solution for the monitoring of these properties that is realized in CONGU2 is presented in Section 4. Section 5 concludes the paper.

2 Overview of the CONGU Approach

As mentioned before, CONGU supports the runtime conformance checking of Java programs against algebraic specifications. In this section we provide an overview of the CONGU approach, focusing on some of the aspects that are visible to users: the specification language and the notions of specification modules and refinement mapping (see [16] for details). This is achieved by means of an example around a simple ADT — *lists with merge*.

This ADT represents lists composed of "mergeable" elements and that have an operation — mergeInRange — that merges the elements of the list in a given range $i \ldots j$

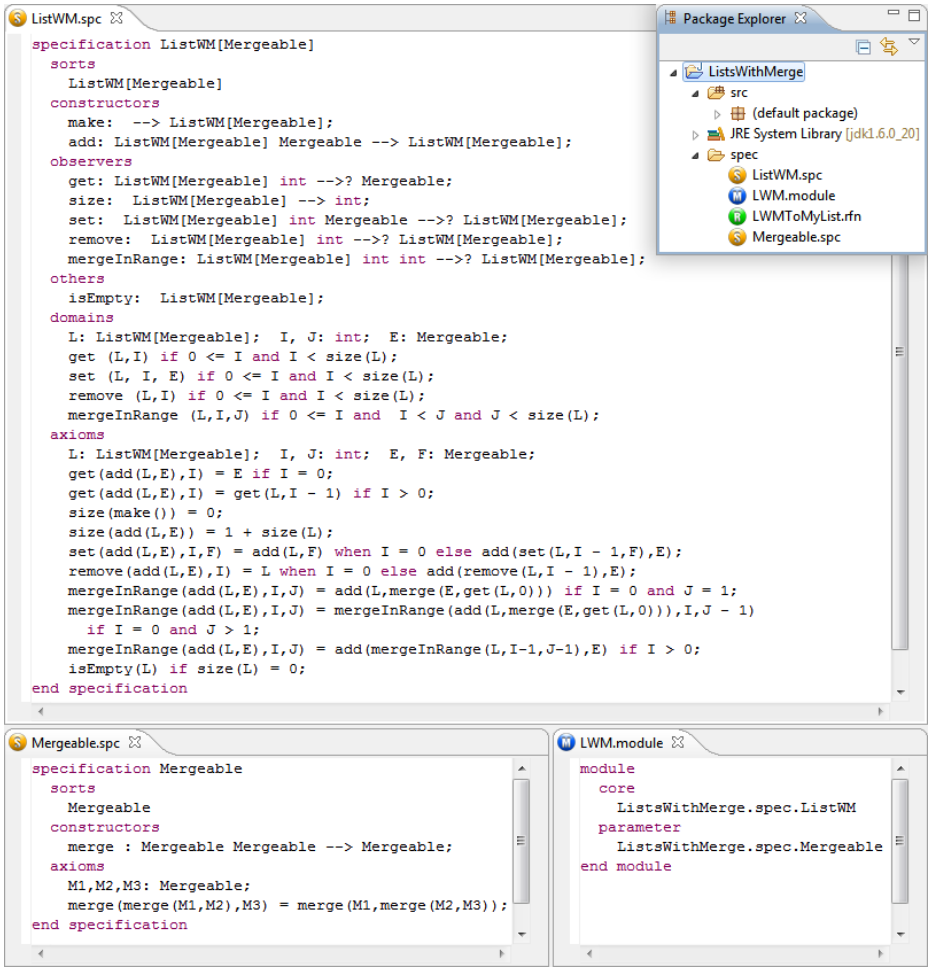

Fig. 1. The three elements involved in the specification of *lists with merge* ADT.

(the resulting element is placed in the position i of the list). Figure 1 shows the three elements involved in the specification of this ADT using CONGU's specification language. In most aspects the language closely follows CASL [4], which is considered a standard for algebraic specification.

The specification ListWM presented in Figure 1 is an example of a parameterized specification. Its parameter is the specification Mergeable, also presented in the figure. Each specification introduces a sort. In our example, Mergeable introduces a simple sort named after it while ListWM introduces the parameterized sort ListMW[Mergeable]. The sort **int**, representing the domain of integer numbers, is primitive in the language.

Then, each specification declares three sets of operations and predicates. Operations declared as constructors are those from which all values of the introduced sort can be

built. The other two sets include the operations and also predicates that provide fundamental information about the values of the sort, or are redundant, but useful, operations. The sort of the first argument of these operations is required to be the introduced sort. The difference between the two groups is only on the syntactical structure of the axioms that can be used to define their properties. Axioms for observers are required to be expressed in terms of their application to constructors as first argument and variables as the remaining arguments; axioms for others are less restrictive, allowing them to be expressed in terms of their application to constructors or variables as first argument and without restriction for the remaining arguments.

Because operations can be partial, specifications also define the *domain condition* of every partial operation, i.e, the situations in which the operation is required to be defined. For instance, in ListWM, get is declared to be partial (as indicated by the partial arrow `-->?`) and its domain condition defines that get(L, I) must be defined if I is indeed an index of list L.

As shown in Figure 1, specifications are put together using *specifications modules*. Specifications identified as *core* define the data types that need to be implemented while the role of *parameter* specifications is simply to impose constraints over their admissible instantiations. Now, suppose we have a candidate implementation for module LWM and that we would like to check its conformance against what was specified. First we need to establish a correspondence between each core sort s of the module LWM and a Java type T defined by one of our classes. Moreover, we need to establish a correspondence between the operations and predicates of the specification that introduces s and the methods and constructors of T. In CONGU, this correspondence is defined by means of a *refinement mapping*. In order to capture the role of parameter specifications, these mappings also allow to link parameter specifications with the type variables of generic classes. More concretely, a refinement mapping also defines a correspondence between each parameter sort s and a Java type variable E and also a method signature for each operation/predicate of the specification that introduces s.

Suppose that our candidate implementation for LWM consists of the generic class `MyListWMerge` and the generic interface `Mergeable` presented in Figure 2. The correspondence between LWM and this candidate implementation is defined in the refinement mapping presented in Figure 3. It maps the compound sort ListWM[Mergeable] (Figure 1) into the generic type `MyListWMerge<E extends Mergeable<E>>` (Figure 2) and the operations and predicates of the former into methods and constructors of the latter. For instance, we can see that operation add is mapped into the method `void addFirst(E e)`. The first argument of an operation always correspond to object `this` and, hence, an operation with n arguments is mapped into a method with arity $n-1$. Only operations declared constructors whose first argument is not the sort being specified can be mapped into class constructors. Predicates are necessarily mapped into boolean methods. For operations that produce elements of the sort being specified, the corresponding method can either be `void` or of the corresponding type. In this way, it is possible to deal with different implementation styles, namely immutable and mutable implementations. In our example, the class `MyListWMerge` provides a mutable implementation of lists and, hence, all operations in this situation are mapped to `void` methods.

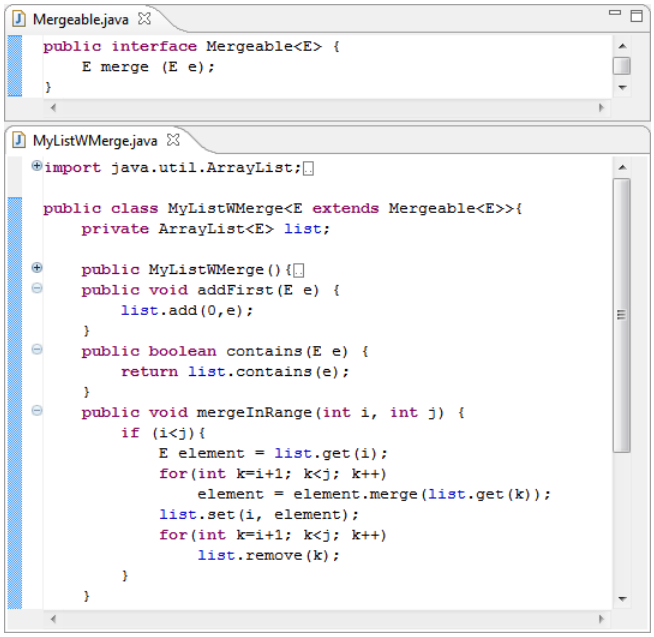

Fig. 2. The interface `Mergeable<E>` and an excerpt of the Java class `MyListWMerge<E>`.

The mapping in Figure 3 also establishes a correspondence between sort Mergeable and thetype variable `E`. It is defined that the operation merge corresponds to the method signature `E merge(E e)`. As we will explain in the next section, this refinement mapping is only correct if the instantiation of `E` in `MyListWMerge<E>` is limited to classes `C` that have a method with signature `C merge(C e)` (which is indeed the case because `E` has `Mergeable<E>` as an upper bound).

After defining the refinement mapping from module LWM to our candidate implementation, CONGU instruments `MyListWMerge.class` so that, during the execution of any program that uses `MyListWMerge`, the behaviour of this class (made precise in the next section) is checked against what was specified in ListWM[Mergeable]. Moreover, the behaviour of the classes used for instantiating `E` in the creation of objects of `MyListWMerge<E>` is also checked against what was specified in Mergeable. Suppose, for instance, that we have a program that includes a class `Color` that implements `Mergeable<Color>` and, another class, that creates and manipulates objects of type `MyListWMerge<Color>`. In this case, the behaviour of `Color` is checked against what was specified in Mergeable.

3 Runtime Conformance of Programs against Modules

In this section, we present the notion of conformance of Java programs against specification modules that is considered in CONGU2 and discuss some key aspects of CONGU approach to the runtime checking of this notion of conformance.

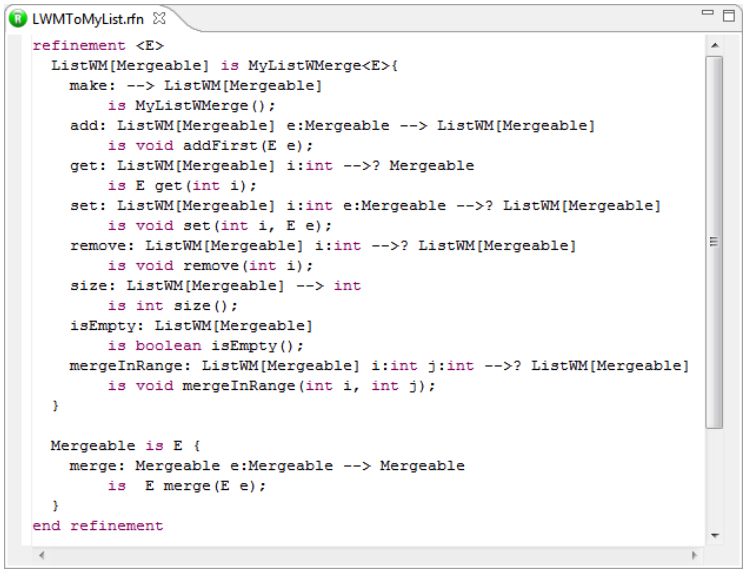

Fig. 3. A refinement mapping from LWM to {`MyListWMerge<E>`, `Mergeable<E>`}.

3.1 Object Properties Induced By Specifications

The conformance of a Java program against a specification module can only be defined if a direct connection between the specifications of the module and the classes of the Java program is provided. As discussed before, in CONGU, the correspondence between specifications and classes is established through the use of refinement mappings. In the previous section we have already mentioned some conditions required by refinement mappings, namely those concerning the matching of method signatures with operations and predicates. The complete set of conditions that a mapping has to meet in order to define a refinement mapping is defined below.

A *refinement mapping* consists of a set V (of type variables) equipped with a pre-order $<$ and a refinement function $\mathcal{R}$ that maps:

1. each core simple specification to a non-generic type defined by a Java class;
2. each core parameterized specification to a generic class, with the same arity;
3. each core specification that defines a sort $s < s'$, to a subtype of $\mathcal{R}(S')$, where S' is the specification defining s';
4. each parameter specification to a type variable in V;
5. each operation of a core specification to a method of the corresponding Java type with a matching signature;
6. each operation of a parameter specification to a matching method signature.

Additionally:

7. if a parameter specification S' defines a subsort of the sort defined in another parameter specification S, then it must be the case that $\mathcal{R}(S') < \mathcal{R}(S)$ holds;

8. if S is a parameterized specification with parameter S', it must be possible to ensure that any type C that can be used to instantiate the parameter of the generic type $\mathcal{R}(S)$ possesses all methods defined by $\mathcal{R}$ for type variable $\mathcal{R}(S')$ after replacing all instances of the type variable $\mathcal{R}(S')$ by C.

Let us consider again the refinement mapping presented in Figure 3. In this case, the set V is the singleton set $\{\texttt{E}\}$ and only the satisfaction of condition 8 requires some reasoning. According to the definition of the class `MyListWMerge`, type variable `E` must extend `Mergeable<E>`, which, in turn, declares method `E merge(E e)`. Hence, the instantiation of `E` is limited to classes `C` that implement `Mergeable<C>` and, hence, it is ensured that `C` possesses a method with signature `C merge(C e)`.

In the sequel, we assume a fixed refinement mapping $\mathcal{R}$ between a specification module $\mathcal{M}$ and a Java program $\mathcal{J}$. Intuitively, $\mathcal{J}$ is in conformity with $\mathcal{M}$ iff:

(i) the properties specified in $\mathcal{M}$ and

(ii) the algebraic properties of the notion of equality

hold in every possible execution of the program $\mathcal{J}$.

More concretely, the properties of a core specification S impose constraints on the behavior of every object of type $T_S = \mathcal{R}(S)$, whereas the properties of a parameter specification S used in a parameterized specification, say $S'[S]$, impose constraints on the behavior of every object of a type T_S in $\mathcal{J}$ that is used to instantiate the respective type variable of the generic type $\mathcal{R}(S')$. Axioms and domain conditions impose different type of constraints:

Axioms. Every axiom in a specification S defines, for the objects of type T_S, a property that must hold in all client visible-states.

Let us consider, for instance, the first axiom for get in ListWM[Mergeable]. Let `lwm` be an object of type `MyListWMerge<C>`. This axiom defines that for every non-null expression `e` of type `C`, after the execution of `lwm.addFirst(e)`, the expression `lwm.get(0).equals(e)` evaluates to true[1].

Domains. Every domain condition ϕ of an operation op of a specification S defines that, for every object of type T_S, whenever ϕ holds, the invocation of $\mathcal{R}(op)$ must return normally (i.e., does not throw an exception).

The constraints induced by axioms and domain conditions just presented define a notion of conformance. CONGU, by default, uses a stronger notion that, in addition, also imposes restrictions on the clients of the classes T_S, namely when they invoke a method $\mathcal{R}(op)$: it is required that the **`null`** value is not passed as argument and the domain condition ϕ holds at the time the method $\mathcal{R}(op)$ is invoked.

This stronger notion of conformance is useful for checking that client code does not call methods in situations where it is not possible assess the normal (non-exceptional) return of a method called outside its domain condition. This is however only appropriate in the absence of additional information about the safe calling conditions for such methods. For instance, if the documentation of class `MyListWMerge<E>` says that **`void`** `set(E e,`**`int`**` i)` has no pre-condition and simply produces no effect on the

[1] We currently assume that the interpretation of sorts does not include the **`null`** value but, we envisage that, in the future, refinement mappings may define whether this is appropriate or not.

state of the list when `i>size()`, the fact that a class in our program calls this method with an argument that violates this condition should not be identified as a problem of conformance between the program and the specification module LWM. For this reason, we found useful to support the two notions of conformance in CONGU2.

3.2 Checking Object Properties

In CONGU, the strategy for runtime checking the conformance of a program against a specification module consists in checking the properties induced by the axioms at the end of specific methods, determined by the structure of the axioms. Let us first consider the axioms with a left-hand side expressed in terms of the application of operations or predicates to constructors as first argument and variables as the remaining arguments. In this case, the property induced by the axiom is checked at the end of the method that refines the referred constructor. All method invocations that are performed in order to check a property make use of clones, whenever cloning is possible. Otherwise, the side effects of these methods would affect the monitored objects (if method `clone()` is not available, it is assumed that the class's objects are immutable). For instance, the property induced by the first axiom for get is checked at the end of **`void`** `addFirst(E e)` through the execution of the following code, where `eOld` is a copy of `e` obtained at the entry of the method.

```
if (eOld != null) {
  E e2 = this.clone().get(0);
  assert(e2 != null && e2.equals(eOld));
}
```

Similarly, the second axiom of the get operation is checked by the below code, where `rangeOfInt`, of type `Collection<Integer>`, is populated with integers that cross the boundary (either as parameters or as returned values) of some method in class `MyListWMerge`.

```
if (eOld ! = null)
  for (int i: rangeOfInt)
    if (i>0 && i<this.clone().size()) {
      E e2 = this.clone().get(i);
      assert((i-1)>=0 && (i-1)<thisOld.clone().size());
      E e3 = thisOld.clone().get(i-1);
      assert(e2!=null && e3!=null && e2.equals(e3));
    }
```

On the other hand, the properties induced by axioms that feature a variable as first argument are checked at the end of the method that refines the corresponding operation/predicate. For instance, the last axiom for isEmpty is checked at the end of method **`boolean`** `isEmpty()` by:

```
if (result) assert(thisOld.clone().size()==0);
```

where `result` is the return value of method `isEmpty()`.

Equality of integers and booleans is translated into comparisons with == whereas the equality of terms of a non-primitive sort, say s, is translated into invocation of method `equals` of the class T_S. Therefore, it is essential that all involved classes define a proper implementation of `equals`. Namely, because equality of terms is a congruence, `equals` should be defined in such a way objects are considered equal only if they are

behaviourally equivalent with respect to the methods that refine some operation of s (i.e., calling these methods over equal objects must produce equal results).

Correctness of `equals` is checked at runtime as follows. At the end of the method, if the return value is true, then it is checked that by applying the method that refines each observer to the two objects, we obtain equal results. For instance, checking the correctness of **`boolean`** `equals(Object other)` in `MyListWMerge` includes the below code for the get operation.

```
if (result)
  for (int i: rangeOfInt)
    if (i>=0 && i<thisOld.clone().size()
             && i<otherOld.clone().size()) {
      E e1 = thisOld.clone().get(i);
      E e2 = otherOld.clone().get(i);
      assert(e1!=null && e2!=null && e1.equals(e2));
    }
```

In addition, at the end of method `clone()`, it is checked that the returned object is equal to the original. Additional properties such as symmetry and transitivity of `equals()` can also be checked, in a similar way, at this point.

Finally, checking that client classes are well-behaved, i.e., do not invoke a method that refines an operation when its domain condition does not hold and also do not pass **`null`** as argument, can be easily performed at the beginning of the method. For instance, in the case of method **`void`** `set(int i,E e)`, this is checked by:

```
assert(e != null && i >= 0 && i < this.clone().size());
```

4 The New CONGU Tool

The new CONGU tool, which we named CONGU2, implements the runtime checking approach conformance of Java classes against specifications described in the previous section. As shown in Figure 4, the tool takes as input a specification module, a set of specifications, a refinement mapping and a Java program (in bytecode form). The program is then transformed so that, when executed under CONGU, the behavior of each class is checked against the corresponding specification. This is achieved by intercepting the calls to all methods that, according to the refinement mapping, refine some operation, and dispatching them to property-monitoring classes generated by the tool.

As mentioned in the introduction, in the previous version of CONGU, the interception of methods was accomplished by proxy classes that wrapped and mimicked the original class's interface as close as possible, capturing the client method calls and diverting them to classes annotated with JML contracts. These contracts were checked at runtime with resort to JML's Runtime Assertion Checker. In order to overcome the difficulties on the generation of appropriate proxy classes for classes making use of more advanced features of the Java language, such as public fields or inner classes, CONGU2 intercepts client method calls through bytecode instrumentation. On the other hand, JML is no longer used and, instead, properties to be checked are now encoded using Java assertions. In this way, CONGU was released from several limitations imposed by the use of JML, namely the lack of support for features introduced in Java 1.5 (generics included), the long compilation times imposed by *jmlc* (namely, because of the compilation of the JML models of the Java API) and the poor and unstable information about

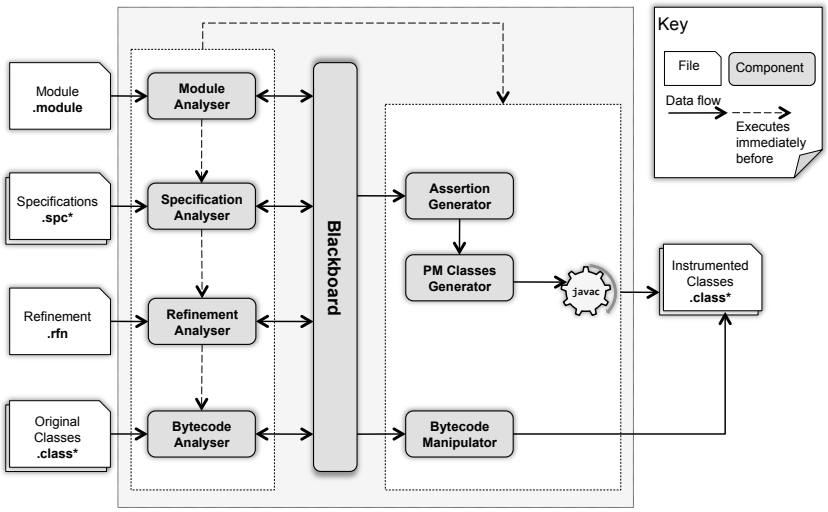

Fig. 4. Overview of CONGU2 architecture.

assertion violations that hindered the connection of errors with the axioms of specifications (the CONGU solution for this problem, developed for JML 5.4 quickly became obsolete).

As shown in Figure 4, there are two main tasks in the CONGU support of runtime checking of Java programs against specifications: the analysis of the different input sources and the synthesis of output classes. In the analysis task, the major challenges posed by the extension of the approach to parameterized specifications and generic classes arise in the analysis of refinement mappings.

4.1 Analysis of Refinement Mappings

The extension of the CONGU approach to specification modules including parameterized specifications introduces various challenges in what concerns the analysis of refinement mappings. As detailed in Section 3, refinement mappings impose restrictions on the Java types to which they refer. Enforcement of these restrictions requires querying the Java binaries of the respective classes. This achieved by taking advantage of Java's reflection facilities, provided by the `java.lang.reflection` package of the Java API.

The process of analyzing refinement mappings comprises two phases. The first phase focus on the Java classes that refine core specifications while the second addresses the verification of the conditions related with parameter specifications.

Verifying that a non-generic type has the methods mentioned in the refinement is straightforward: it involves querying for the specified method and then checking whether the return type is the expected one. Matters complicate when generic types are at play. Generics in Java are mostly a source code artefact. Simply put, the compiler

erases the generic type information, a mechanism known as *type erasure*. Thus, a once generic type becomes a simple type, i.e., a raw type, where all uses of its generic type variables are replaced by their respective upper bounds [11]. For this reason, built-in support for querying classes for methods is limited to signatures defined only in terms of raw types and, hence, a new strategy for verifying that generic classes possess the methods mentioned in the refinement is needed.

Although information about generics is not used at runtime by the JVM, this information is still retained in the bytecode, in the form of metadata and can be queried through Java's reflection API. The strategy to verify that a generic class has the methods mentioned in the refinement mapping involves retrieving its methods, getting the generic parameter types and generic return type of each one and then comparing with what was expected (a recursive process on the structure of the types).

The second phase of the refinement analysis addresses the verification of the conditions concerning the parameter specifications. Recall that, in this case, specifications are not refined into concrete Java types and what is necessary ensuring is that the classes that can be used to instantiate the corresponding type variable have the right methods. For instance, in our example, in this phase it is verified that every class `C` that can be used to instantiate `E` in `MyListWMerge<E>` possesses a method with signature `C merge(C e)`. This is achieved by going through the upper bounds of `E` in `MyListWMerge` (in our case there is a single upper bound but in general types may have more than one). Methods whose signature depends somehow on type `E` (in our case, `E merge(E e)`) need only to be searched on the upper bounds types that are themselves generic and dependent of `E` (in our case, `Mergeable<E>`) while the remainder methods are searched on all upper bounds types. Searching for these methods in generic types follows the strategy described before.

Additionally, the analysis of the refinement mapping also involves ensuring that hierarchy relationships established for specification sorts are maintained when refining to Java types. This is directly accomplished again making use of Java reflection API, which enables us to query a type for its super class and/or implemented interfaces.

4.2 Bytecode Instrumentation

For monitoring the behaviour of Java programs, CONGU relies on the interception of method calls by client classes. In CONGU2, this is achieved with method call interception through bytecode instrumentation of a copy of the original class (the original bytecode remains unchanged so that the program can also be executed normally). The objective of this instrumentation is to inject bytecode instructions in the methods to forward the call to a corresponding method in a property-monitoring class. From the start, the goal was to minimise the impact on the original bytecode, avoiding any undesirable side effects of its faulty manipulation as much as possible, preferring to generate Java code and rely on the compiler for type safety. Realising this new strategy posed interesting challenges, namely:

Inner calls. How to avoid interception of intra-class method calls? Intra-class calls can not be monitored otherwise we obtain a non-terminating program.

Calls from within superclasses. How to prevent interception of calls from within a superclass? Such calls cannot be monitored for the reason above.

Constructors. How to intercept calls to object constructors and redirect them?
Clone and equals. What to do when these methods are not overriden in the class? (Recall that CONGU relies on these methods and there are properties that have to be monitored when they are invoked.)

Answering these questions was central in overcoming the limitations of earlier version of the CONGU tool. The chosen strategy consists in renaming to *m_ Original* each method *m* that refines some operation (each method whose external calls we wish to intercept), and placing in its lieu a method *m* with the exact same interface but dispatching the call to a corresponding method in a property-monitoring class. Moreover, all calls to *m* from within the class and inner classes are replaced by calls to *m_Original* and all calls to *m* in each superclasses are replaced also to calls by calls to *m_Original* that are also added to the superclasses. Although constructors are not methods, they can be for the most part treated as such. Hence, the approach towards the interception of constructor calls is identical to that employed for methods, with the safeguard that constructors require initialisation calls, which are removed from the renamed method and inserted in the replacement method.

More concretely, if *m* is a method of a class C that refines some operation, then the bytecode instrumentation process involves the following steps:

1. Within C, rename method *m* to *m_Original*;
2. Still within C, replace invocations to *m* by invocations to *m_Original*;
3. In C's superclasses, replace invocations to *m* by invocations to *m_Original* and generate a method *m*, with the signature of the original, which just forwards the call to *m_Original*;
4. Generate a replacement for method *m*, with signature of the original, that calls the respective method in the property-monitoring class:
   ```
   congu.properties.CPMonitoring.m(this, ...);
   ```
5. Rename and generate a replacement for method `equals`; if `equals` is not overridden in C, first create a such method that delegates into the superclass;
6. Rename and generate a replacement for method `clone`; if C does not implement interface `Cloneable` or does not override method `clone` by making it public, a `clone_Original` method is generated that simply returns **`this`**. This method is for the exclusive use of the monitoring process.

Implementation of this bytecode-instrumentation approach resorts to the ASM Java bytecode engineering library [6]. ASM is a lightweight and efficient, offering a very simple, well-documented API, full support for Java 6 and an interesting open-source license which allows for convenient packaging within the CONGU2 tool itself.

4.3 Generation of Property-Monitoring Classes

The checking of object properties described in subsection 3.2 is performed in classes generated by the tool, which we call *property-monitoring* classes (or PM-classes, for short). For each Java type C under monitoring, there is a corresponding PM-class, named by appending the suffix `PMonitoring` to the name of C. In the instrumented

bytecode, the intercepted client method calls are dispatched to methods in the respective PM-class.

Each method under monitoring has a counterpart in its respective PM-class, in the form of a static method with the same name, the same return type, the same argument types. In addition it features an argument `callee` of type C (a reference to original method callee) and a boolean argument `monitoring` (signalling whether monitoring should be performed or not). When invoked from the instrumented bytecode, this flag is set to **`true`**; when the invocation is realised in the context of a property monitoring it is set to **`false`**.

These static methods are responsible for the monitoring of the relevant object properties following a general pattern:

1. At the entry of the method, store all elements that, later, are needed for checking some property (these elements are stored in variables starting with *old*).
2. Verify that the client is well-behaved, namely that the domain condition holds (only applicable in the case of strong conformance) and the values passed as arguments are non **`null`**;
3. Call the original method upon `callee` and keep its return value;
4. Check the properties defined by the axioms.
5. Return the original method return value.

The execution of steps 2 and 4 for method *m* (which are only executed if flag `monitoring` is true) rely on two separate static methods: `mPre` and `mPos`. These test *callee* for the object properties induced by the specification as described in 3.2 but, instead of calling the methods of the original class, they call the method of the corresponding PM-class with `monitoring` set to false.

More concretely, the method `mPre` receives the same arguments as the original one, plus a boolean flag signalling whether to break on a violation, and returning a boolean value, corresponding to whether or not the respective domain condition is satisfied. Method `mPos` is void and takes as arguments: (i) two references to the callee (one before application of the original method, i.e., its *old* value, and another after the original method call); (ii) the arguments of the respective method in the PM-class and (iii) a boolean flag signalling whether to break on a violation.

Figure 5 presents a sequence diagram that illustrates the flow of execution in the concrete case of a call to method `mergeInRange(`**`int,int`**`)`.

The monitoring process also heavily depends on an auxiliary method generated as part of each PM-class, named `conguAssert`. This method is responsible for issuing adequate error messages whenever a violation occurs. It takes as parameters the assertion to evaluate, an enumerate value flagging what kind of property was violated (either a domain condition or an axiom-induced property) and error description elements. The method tests the assertion and throws an exception if it is **`false`**, detailing the violation with the descriptive elements received as arguments. The error description elements passed to this method are the file of the specification to which the property belongs, the domain or axiom to which it pertains as written in the specification and its line. In this way, it is possible to pinpoint the origin of the error, in terms of the specification, which can be invaluable when developing in a specification guided manner.

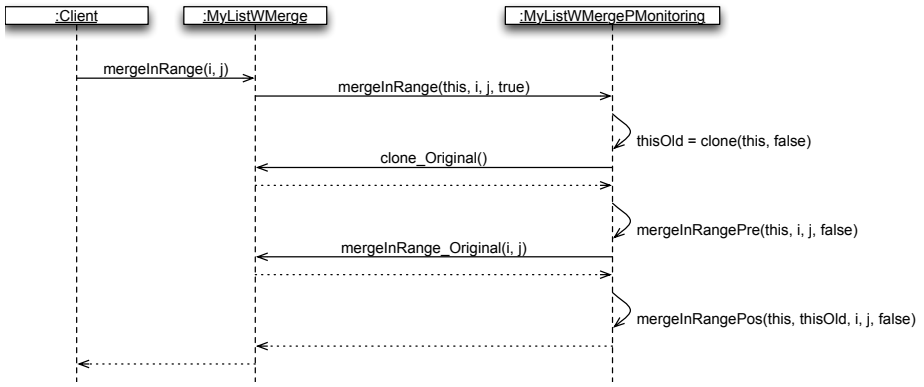

Fig. 5. The property-monitoring process.

In step 3, in addition to invocation of the original method, it is also checked that if the domain condition holds, the invocation of the method returns normally. This is achieved by surrounding the original method call with a try-catch statement. The catch clause is only reached when the original method fails to return normally, in which case a violation is issued if the domain condition was true. If the domain condition was false, no constraints apply and, hence, the caught exception is re-thrown, allowing the program to handle it as it would had it been executing normally.

It is worth noting that the properties monitored by each of PM-class do not necessarily originate from a single specification. Refinement mappings do not restrict the number of specifications that a Java type may implement, therefore the PM-class for a given type is responsible for monitoring the properties arising from all the specifications that have been refined to mentioned type.

All of the above holds true for both core and parameter specifications. However, parameter specifications require another level of indirection. Let C be a generic class with a parameter E that refines a core parameterized specification, say $S[S']$. Even though each class that is used to instantiate E in C has a corresponding PM-class, the code generated for monitoring the properties of S that involves to call a method over an object e of type E, cannot commit to a specific PM-class (the actual type of e is only known at runtime and will vary from call to call).

CONGU2's solution is to generate a dispatcher class associated to each type variable of the refinement mapping. This class has the exact same methods of a PM-class, but, instead of monitoring properties, only resolve to which PM-class should the call be forwarded to, based on the actual type of object *callee*. In the PM-class for class C, whenever the testing of a property requires to invoke a method of E, the call is placed to the respective dispatcher class.

Suppose that our program manipulates an object `lc` of type `MyListWMerge<Color>` and another `lt` of type `MyListWMerge<Text>`. Monitoring the behaviour of these objects is performed by the `MyListWMergePMonitoring` class. Whenever such operation involves a call to method `E merge(E)`, we call the respective method in the `EPMonitoring` dispatcher class. While monitoring `lc`'s behaviour, the actual type of

the callee is `Color` and, hence, the dispatcher forwards the invocation of `merge` to `ColorPMonitoring`. Similarly, while monitoring `lt`, the calls are forwarded to class `TextPMonitoring`. Notice however that when, as a result of calling `mergeInRange` over `lc`, method `E merge(E)` is called (see the body of the method in Figure 2), this call is intercepted and forwarded to `ColorPMonitoring` in order for the properties of this operation to be checked.

5 Conclusions

The importance of tools that support runtime conformance checking of implementations against formal specifications has long been recognised. In the last decade, many approaches have been developed for monitoring the correctness of OO programs w.r.t. formal specifications (e.g., [1, 3, 7, 8, 12, 15]). However, despite the actual popularity of generics in mainstream OO languages [5], current approaches still lack support such a feature. This was also a limitation CONGU that was overcome in CONGU2.

In this paper we showed how the CONGU approach and the corresponding tool were extended in order to support the specification of parametrized data types and their implementation in terms of generic classes. The extension of the specification language in order to support the description of generic data types was relatively simple. Given that CONGU relies on property-driven specifications, this mainly required the adoption of parameterized specifications available in conventional algebraic specification languages. In order to bridge the gap between parameterized specifications and generic classes we proposed a new notion of refinement mapping around which a new notion of conformance between specifications and OO programs was defined. To the best of our knowledge, this issue has not yet been addressed in other contexts. Other approaches exist that deal with the problem of the implementation of architectural specifications including parameterized specifications as, for example [2], but the target are ML programs. Relationships between algebraic specification and OO programs that we are aware of, namely those that address runtime conformance checking or testing, exclusively consider flat and non-parameterized specifications (e.g., [1, 12, 18]).

CONGU2 implements the runtime monitoring of this new notion of conformance which, in the case of generic data types, involves checking that both the class that implements the data type and the Java types used to instantiate it conform with what was specified. With CONGU2, runtime conformance checking becomes applicable to a range of situations in which automatic support for detection of errors becomes more relevant. Generics are known to be difficult to grasp and, hence, with generics in action, obtaining correct implementations becomes more challenging. For us, it is particularly important to be able to use CONGU with the generic data types that appear in the context of a typical Algorithms and Data Structures course: we believe this course constitutes an excellent opportunity for exposing undergraduate students to formal methods. As discussed by Hu [13], accurate descriptions of abstract data types, agnostic w.r.t. programming paradigms and languages, are important for teaching these concepts. From our experience in teaching this course for several years (initially without tool support and, more recently, using CONGU), we are convinced that, from an educational an motivational point of view, it is quite important that students experience, in their practice,

that they can take real advantage of formal descriptions. The use of a simple tool that allows them to gain confidence that their classes correctly implement a given data type has shown to be a good starting point. The extension of the tool to support generics will contribute to the success and effectiveness of the CONGU approach to the introduction to formal methods in the computer science curriculum.

References

1. S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.
2. D. Aspinall and D. Sannella. From specifications to code in CASL. In *Proc. Algebraic Methodology and Software Technology (AMAST) 2002*, volume 2422 of *LNCS*, pages 1–14. Springer, 2002.
3. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
4. M. Bidoit and P. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer, 2004.
5. G. Bracha. Generics in the Java programming language, 2004. Available at java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
6. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles*, 2002.
7. F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2005*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.
8. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proc. International Conference on Software Engineering Research and Practice (SERP'02)*, pages 322–328. CSREA Press, 2002.
9. D. R. Cok. Adapting JML to generic types and Java 1.6. In *Proc. Specification and Verification of Component-Based Systems Workshop*, 2008.
10. Contract Based System Development. http://gloss.di.fc.ul.pt/congu/.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005.
12. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP 2003*, volume 2743 of *LNCS*, pages 431–456. Springer, 2003.
13. C. Hu. Just say 'a class defines a data type'. *Communications of the ACM*, 51(3):19–21, 2008. See also Forum in *Communications of the ACM*, 51(5):9–10, 2008.
14. G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–208, 2005.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997.
16. I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Runtime Verification*, volume 5779 of *LNCS*, pages 115–131. Springer, 2009.
17. I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L.S. Reis. Checking the conformance of Java classes against algebraic specifications. In *Proc. International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *LNCS*, pages 494–513. Springer, 2006.
18. B. Yu, L. King, H. Zhu, and B. Zhou. Testing Java components based on algebraic specifications. In *Proc. International Conference on Software Testing, Verification and Validation*, pages 190–198. IEEE, 2008.