Bridging the Gap Between Algebraic Specification and Object-Oriented Generic Programming

Isabel Nunes, Antónia Lopes, and Vasco T. Vasconcelos

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

Abstract. Although generics became quite popular in mainstream object-oriented languages and several specification languages exist that support the description of generic components, conformance relations between object-oriented programs and formal specifications that have been established so far do not address genericity. In this paper we propose a notion of refinement mapping that allows to define correspondences between parameterized specifications and generic Java classes. Based on such mappings, we put forward a conformance notion useful for the extension of CONGU, a tool-based approach we have been developing to support runtime conformance checking of Java programs against algebraic specifications, so that it becomes applicable to a more comprehensive range of situations, namely those that appear in the context of a typical Algorithms and Data Structures course.

1 Introduction

Many approaches have been developed for runtime checking the conformance of object-oriented programs with formal specifications. A limitation of existing approaches, including our own work [20], is the lack of support to check generic components. Despite the popularity of generics in mainstream object-oriented languages, available approaches, such as [14, 17, 2, 18, 5, 22, 8], are not applicable to programs that include generic classes.

At the specification side, the description of programs that include generic elements is not a problem. In particular, in the case of algebraic specification, languages s.a. CASL [6] support the description of parameterized specifications as well as the definition of specifications through instantiation of parameterized ones. What is lacking is to bridge the gap between parameterized specifications and generic classes.The conformance relations of specifications with object-oriented programs that have been established so far (e.g. [2, 12, 1, 11, 4, 9, 22]) only consider simple, non-parameterized, specifications.

In our own previous work on runtime conformance checking between algebraic specifications and Java programs — the CONGU approach [20] — we have also considered simple specifications only. However, since generics are available in Java, the fact that generic specifications are not supported by CONGU has become a severe drawback. The CONGU tool [10] is intensively used by our undergraduated students in the context of a course on algorithms and data structures. They use the tool to analyze their implementations of ADTs with respect to specifications. Given that generics became extremely useful and popular in the implementation of ADTs in Java, in particular those that are traditionally covered in this course, it is crutial to extend CONGU in order to support parameterized specifications and generic classes.

The extension of the CONGU approach requires *(i)* finding an appropriate mechanism for expressing correspondences between parameterized specifications and generic classes, *(ii)* defining a more comprehensive notion of conformance between specifications and Java programs, and *(iii)* extending the CONGU tool, which is based on the generation of annotated classes with monitorable contracts written in JML [13], to cope with this broader notion of conformance.

This paper mainly focuses on the first two aspects, considering not only generic specifications, but also specifications that make use of subsorting. In the developed approach, subsorting revealed to be a very useful construct, fundamental to cope with the range of situations that appear in the algorithms and data structures course.

To our knowledge, our work is the first to tackle the problem of bridging the gap between algebraic specifications and generic object-oriented programming (a problem that does not arise in the context of other approaches to verification that were extended to handle generics in Java, namely in [21] that is focused on the proof of functional properties of programs). Our contributions are twofold. On the one hand, we propose a way of describing the modular structure of reusable libraries that use generics, s.a. the Java Collections Framework (generics are especially common in this context). We believe our proposal can be easily understood by Java programers in general and our students in particular. This is because, as we will show, there exists a straightforward correspondence between the key concepts at the specification and programming levels. On the other hand, we put forward a notion that allows to express correspondences between specifications and Java programs and a conformance notion that paves the way for the extension of application of runtime checking to a more comprehensive range of situations, namely to APIs that use generics and code that uses these APIs.

The remainder of the paper is organised as follows. Section 2 presents the structure of specification modules and the adopted specification language which includes subsorting and parameterization. Then, in Section 3, we propose an interpretation of those modules in terms of Java programs and, in Section 4 we put forward a notion of conformance. The solution envisaged for extending the CONGU tool is addressed in Section 5, and Section 6 concludes the paper. We illustrate our approach to runtime conformance checking of generic Java programs against parameterized data types with two typical examples: sorted sets and (closed) intervals.

2 Specifications and Modules

As specifications we take essentially a subset of the set of specifications that can be defined in CASL [6], considered a standard for algebraic specification. However, for the sake of simplicity, we adopt a different concrete syntax. This section introduces our specifications in three steps: simple, with subsorting and parameterized specifications. These are the building blocks of modules, introduced at the end of the section.

```
specification TOTAL_ORDER
  sorts
    Orderable
  observers
    geq: Orderable Orderable;
  axioms
    E, F, G: Orderable;
    E = F if geq(E, F) and geq(F ,E);
    geq(E, E);
    geq(E, F) if not geq(F, E);
    geq(E, G) if geq(E ,F) and geq(F, G);
end specification
```

Fig. 1. A specification of a total order.

2.1 Specifications

Simple specifications are those currently supported by CONGU, described in detail in [20]. A specification defines exactly one sort and introduces a set of operations and predicates. The first argument of every operation and predicate is required to be of that sort. Operations declared with $\longrightarrow?$ may be partial, i.e., can be interpreted by partial functions. Operations can be classified as **constructors** corresponding to the usual (loose) datatype constructors; in this case, they may have no arguments. Furthermore, the language imposes some restrictions on the form of axioms, namely the separation, under the keyword **domains**, of the domain conditions of operations, that is, the conditions in which operations have to be defined.[1]

Figure 1 presents an example of a simple specification — TOTAL_ORDER. This specification is self-contained, i.e., it does not include *external symbols*. However, simple specifications may refer to sorts, operations and predicates defined elsewhere. For instance, for specifying a total order that, additionally, has a correspondence with a set of *natural numbers*, we would have to refer to a sort Nat, say, defined in a different specification.

Specifications can be more complex, namely making use of *subsorting*. More precisely, a specification may introduce a new sort that is declared to be a subsort of one or more sorts introduced elsewhere. Following [6], this means that the values of the subsort have to be understood as a special case of those of the supersort. Figure 2 presents an example of a specification of a total order with a successor operation, achieved by defining elements of this data type (values of sort Successorable) as special cases of elements of a total order.

Similarly to CASL, an operation or predicate defined in the supersort $s > s'$ can receive a term of sort s' as argument wherever an element of sort s is expected. For instance, in our example, this justifies why the formula geq(suc(E),E) with E:Successorable is well-formed.

The third kind of specifications are *parameterized specifications*. They have one or more specifications as parameters, and introduce one compound sort of the form

[1] These restrictions are related with the way CONGU supports runtime conformance checking, that involves the automated generation of monitorable contracts from the specified properties.

```
specification TOTAL_ORDER_WITH_SUC
  sorts
    Successorable < Orderable
  constructors
    suc: Successorable ⟶ Successorable;
  axioms
    E, F: Successorable;
    geq(suc(E), E);
    geq(E, suc(F)) if geq(E,F) and not (E = F);
    E = F if suc(E) = E and suc(F) = F;
end specification
```

Fig. 2. A specification of a total order with a successor operation.

```
specification SORTED_SET[TOTAL_ORDER]
  sorts
    SortedSet[Orderable]
  constructors
    empty: ⟶ SortedSet[Orderable];
    insert: SortedSet[Orderable] Orderable ⟶ SortedSet[Orderable];
  observers
    isEmpty: SortedSet[Orderable];
    isIn: SortedSet[Orderable] Orderable;
    largest: SortedSet[Orderable] ⟶? Orderable;
  domains
    S: SortedSet[Orderable];
    largest(S) if not isEmpty(S);
  axioms
    E, F: Orderable;  S: SortedSet[Orderable];
    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 if not isEmpty(S) and geq(E, largest(S));
    largest(insert(S, E)) = largest(S) if not isEmpty(S) and not geq(E, 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
```

Fig. 3. A specification of a sorted set.

$name[sort_1, \ldots, sort_k]$ where $sort_i$ is the name of the sort introduced in the ith parameter of the specification. Figure 3 presents a specification of the data type *Sorted Set*. It is an example of a parameterized specification with one single parameter — the specification TOTAL_ORDER — that introduces the compound sort SortedSet[Orderable].

Parameterized specifications are usually used as a means to support reuse at the specification level, through the instantiation of their parameters with different specifications. In this work, however, we are mainly interested in parameterized specifications as a means of specifying generic data types. Specifications defined through instantiation of parameterized specifications are not first-order, in the sense that they cannot be used as elements of specification modules (introduced below). Still, these specifications are useful as they introduce sorts and operations that can be used in other specifications.

The parameterized specification INTERVAL presented in Figure 4 illustrates this situation. Intervals are defined as pairs of elements of a total order with a successor operation. The operation elements, that calculates the set of elements of an interval, returns elements of sort SortedSet[Successorable]. This sort is defined by the specifica-

```
specification INTERVAL[TOTAL_ORDER_WITH_SUC]
  sorts
    Interval[Successorable]
  constructors
    interval: Successorable Successorable ⟶? Interval[Successorable];
  observers
    max: Interval[Successorable] ⟶ Successorable;
    min: Interval[Successorable] ⟶ Successorable;
    before: Interval[Successorable] Interval[Successorable];
    elements: Interval[Successorable] ⟶ SortedSet[Successorable]
  domains
    E, F: Successorable;
    interval(E,F) if geq(F,E);
  axioms
    E,F: Successorable; I,J: Interval[Successorable];
    max(interval(E, F)) = F;
    min(interval(E, F)) = E;
    before(I, J) iff geq(min(J), max(I));
    elements(I) = insert(empty(), min(I)) if max(I) = min(I);
    elements(I) = insert(elements(interval(suc(min(I)), max(I))), min(I))
        if not (max(I) = min(I));
end specification
```

Fig. 4. A specification of an interval.

tion SORTED_SET[[TOTAL_ORDER_WITH_SUC]] — the specification that results from the instantiation of the parameter of SORTED_SET with TOTAL_ORDER_WITH_SUC. Moreover, the specification INTERVAL uses the operation and the predicate

```
insert:SortedSet[Successorable] Successorable ⟶SortedSet[Successorable]
empty: SortedSet[Successorable]
```

also defined in this specification.

Usually, specification instantiation is defined if the argument specification provides symbols corresponding to those required by the corresponding parameter, and the properties required by the parameter hold. For the sake of simplicity, we decided not to support the explicit definition of fitting morphisms and so instantiation is restricted to situations in which this correspondence can be left implicit, being established by an obvious injection. For instance, in the example of the specification SORTED_SET[[TOTAL_ORDER_WITH_SUC]], the injection is indeed obvious as sort Successorable was defined to be a subsort of Orderable.

2.2 Specification modules

In the previous subsection, in order to intuitively provide the meaning for different specification constructs, we have always considered there was a well known context for dereferencing external symbols — the set of specifications previously presented. *Specification modules* provide a means for establishing this context. The meaning of external symbols is only defined when the specification is integrated in a specification module, together with other specifications that define those external symbols.

As shown before, the specification of a generic data type may involve much more than two specifications. For instance, in the case of *Interval*, we used four: TOTAL_ORDER_WITH_SUC, TOTAL_ORDER, SORTED_SET and INTERVAL. Clearly, the role of the first two specifications is different from the last two. Specifications SORTED_SET and INTERVAL define data types that have to be implemented (they are the *core* of the module), while the role of the other two specifications is simply to impose constraints over their admissible instantiations. Specification modules, defined as pairs of sets of specifications, explicitly identify the nature of each specification.

More concretely, a *specification module* is a pair $\langle core, param \rangle$ of sets of specifications s.t.:

- the intersection of *core* and *param* is empty and all sorts are different;
- the *param* set contains every specification used as parameter;
- both *core* and *param* sets are closed under the supersort relation;
- all external symbols —parameters, sorts, operations and predicates— are resolved.

With the specifications introduced in the previous subsection, we can build different modules. Three examples are presented below, ranging from a very simple module TO with a single core specification to the module ITV that specifies two generic data types.

```
TO = <{TOTAL_ORDER}, {}>
SS = <{SORTED_SET},{TOTAL_ORDER}>
ITV = <{SORTED_SET,INTERVAL},{TOTAL_ORDER,TOTAL_ORDER_WITH_SUC}>
```

Specification modules play a role that, to some extent, is similar to that of architectural specifications [7] in CASL. In both cases, it is prescribed the intended structure of implementations, that is to say, the implementation units that have to be developed. The difference is that architectural specifications additionally describe how units, once developed, are put together to produce the overall result. However, since the operators used for combining units in architectural specifications are not available in object-oriented languages s.a. Java, this aspect of architectural specifications is not useful in this context.

3 Interpreting Modules in terms of Java Programs

The standard interpretation of algebraic specifications in terms of algebras has proved very important, namely to understand ADTs. However, as pointed out in [3], this interpretation only provides us with an indirect connection with the abstractly described programs. For a broader use of specifications, namely in the context of runtime conformance checking, a direct connection between specifications and programs is needed.

In this paper we propose an interpretation of specification modules in terms of sets of Java classes and interfaces. This is presented in two steps. In this section, we characterise the Java programs appropriate for interpreting a specification module, taking into account the structural constraints, while in the next section, we define the class of Java programs that correctly realize the specified requirements. These programs are said to be conforming with the specification module.

3.1 Constraints over the structure of programs

Let us take the perspective of a Java developer that has to implement a collection of ADTs, described in terms of a specification module. Each core specification of the module abstractly describes a Java class. This class has to be generic if the specification is parameterized. Moreover, in the presence of specifications that make use of subsorting, the induced type hierarchy must be enforced by the implementations.

More specifically, a Java program, regarded as a set $\mathcal{C}$ of classes and interfaces, is appropriate for interpreting a specification module $\langle \mathit{core}, \mathit{param} \rangle$ only if there exists a correspondence from every core specification to a Java type in $\mathcal{C}$. Additionally, the following conditions also need to be fulfilled:

- the sort introduced by each simple specification in *core* corresponds to a non-generic type T in $\mathcal{C}$;
- the generic sort introduced by each parameterized specification in *core* corresponds to a Java generic type in $\mathcal{C}$ with the same arity (i.e., the same number of parameters);
- the sort $s < s'$ introduced by each subsorting specification in *core* corresponds to a type T in $\mathcal{C}$, and T is subtype of the type that corresponds to s'.

As an example consider again the module ITV. According to the constraints just described, an implementation of this module in Java has to include two generic classes, one implementing SORTED_SET and the other implementing INTERVAL.

3.2 Constraints over the structure of classes and interfaces

Method signatures. The signature of a specification S introducing a sort s imposes constraints over the methods that need to be available in the corresponding Java type T. Every operation and predicate declared in a specification S must correspond to a public method of T with a "matching" signature in terms of arity and return and parameter types. More precisely:

- **Arity**: **(i)** every (n+1)-ary operation or predicate corresponds to an n-ary method — this is due to the fact that the object that corresponds to the first parameter of every operation is the current object (`this`); **(ii)** every zero-ary operation corresponds to a constructor of the corresponding class.
- **Return type**: **(i)** every predicate corresponds to a boolean method; **(ii)** every operation with result sort $s' \neq s$ corresponds to a method with return type T', if s' corresponds to Java type T'; **(iii)** every operation with result sort s corresponds to a method with any return type, `void` included.
- **Parameter type**: given a method m corresponding to operation/predicate op, the i-th parameter of m has the type corresponding to (i+1)-th parameter sort of op.

This is similar to what we defined in [20]. However, the underlying correspondence of types, in addition to what was defined before, has to satisfy the following condition:

- the occurrence of a sort of the form $s'[s_1, ..., s_n]$, defined by an instantiation of a parameterized specification, must correspond to 1) an instantiation $T'\langle T_1, ..., T_n\rangle$, if the specifications defining every s_i belong to *core*; 2) a generic type $T\langle E_1, ..., E_n\rangle$, if the specifications defining every s_i belong to *param*.

```
interface IOrderable⟨E⟩{
  boolean greaterEq(E e);
}

public class TreeSet⟨E extends IOrderable⟨E⟩⟩{
  public TreeSet⟨E⟩(){...}
  public void insert(E e){...}
  public boolean isEmpty(){...}
  public boolean isIn(E e){...}
  public E largest(){ ...}
  ...
}
```

Fig. 5. An excerpt of a Java implementation of a sorted set.

As an example let us consider the module SS introduced before and a Java program containing the class `TreeSet` and the interface `IOrderable` (see Figure 5). If we consider that the sort SortedSet[Orderable] corresponds to the type `TreeSet⟨E⟩`, then it is easy to see that every operation and predicate in specification SORTED_SET has a corresponding method in the class `TreeSet` with a matching signature. For example, the operation insert : SortedSet[Orderable] Orderable ⟶SortedSet[Orderable] corresponds to **`void`** `insert(E e)`, the predicate isIn : SortedSet[Orderable] Orderable corresponds to **`boolean`** `isIn(E e)` and the operation largest : SortedSet[Orderable] ⟶ Orderable corresponds to `E largest()`.

Parameters of Java generic classes. Other type of constraints imposed over the structure of classes concerns the way parameters of generic classes are bound. For every parameterized specification, the instantiation of the parameter type of the corresponding generic class has to be limited to types that have a method for every operation and predicate in the corresponding parameter specification, with a signature that matches that of the operation/predicate. More concretely, if a Java class K can be used to instantiate a given generic Java type $T\langle E\rangle$ that corresponds to a generic specification $S[S']$, then every operation and predicate of S' must correspond to a method of K with a matching signature considering that the sort s' corresponds to type K.

Going back to the example discussed before, we see that the instantiation of the type parameter in `TreeSet` is limited to classes K that implement `IOrderable⟨K⟩` and hence, it is ensured that they have the method **`boolean`** `greaterEq(K e)`. The signature of this method clearly matches with the declaration geq:Orderable Orderable in specification SORTED_SET considering that sort Orderable corresponds to type K.

3.3 Refinement mappings

The correspondence between specifications and Java types as well as between operations/predicates and methods can be described in terms of what we have called a *refinement mapping*. In order to support the analysis of a Java program with respect to a specification module, it is crucial that a correspondence between them be explicitly defined.

```
refinement <E>
  SORTED_SET[TOTAL_ORDER] is TreeSet<E> {
    empty: ⟶ SortedSet[Orderable] is TreeSet<E>();
    insert: SortedSet[Orderable] e:Orderable ⟶ SortedSet is void insert(E e);
    isEmpty: SortedSet[Orderable] is boolean isEmpty();
    isIn: SortedSet[Orderable] e:Orderable is boolean isIn(E e);
    largest: SortedSet[Orderable] ⟶? Orderable is E largest();
  }
  TOTAL_ORDER is E {
    geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
end refinement
```

Fig. 6. A refinement for a sorted set.

A *refinement mapping* from a specification module $\mathcal{M}$ to a set $\mathcal{C}$ of Java types consists of a set V of type variables equipped with a pre-order, and an injective refinement function $\mathcal{R}$ that maps:

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

Additionally:

– 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 (recall that the set V of type variables is equipped with a pre-order);
– if S is a parameterized specification with parameter S', it must be possible to ensure that any type K that can be used to instantiate the corresponding parameter of the generic type $\mathcal{R}(S)$ possesses all methods *op* defined by $\mathcal{R}$ for type variable $\mathcal{R}(S')$ after appropriate renaming — the replacement of all instances of the type variable $\mathcal{R}(S')$ by K (among these methods are the methods defined by $\mathcal{R}$ for any type variable V s.t. $\mathcal{R}(S') < V$).

In Figure 6 we present an example of a refinement mapping between the module SS and the java types `{TreeSet⟨E⟩, IOrderable⟨E⟩}`, using a concrete syntax that extends the one that is currently supported by CONGU. In order to check that the described function indeed defines a refinement mapping we have to confirm that the last condition above holds (the first one is vacuously true as TOTAL_ORDER does not define a subsort). This can be ensured by inspecting in the class `TreeSet` whether any bounds are declared for its parameter `E`, and whether those bounds are consistent with the methods that were associated to parameter type `E` by the refinement mapping — **`boolean`** `greaterEq(E e)`. This is indeed the case: the parameter `E` of

```
refinement <E, F extends E>
  SORTED_SET[TOTAL_ORDER] is TreeSet<E> {
    empty: ⟶ SortedSet[Orderable] is TreeSet<E>();
    insert: SortedSet[Orderable] e:Orderable ⟶ SortedSet is void add(E e);
    isEmpty: SortedSet[Orderable] is boolean isEmpty();
    isIn: SortedSet[Orderable] e:Orderable is boolean isIn(E e);
    largest: SortedSet[Orderable] ⟶? Orderable is E greatest();
  }
  INTERVAL[TOTAL_ORDER_WITH_SUC] is MyInterval<F> {
    interval: e1:Successorable e2:Successorable ⟶? Interval[Successorable] is
                    MyInterval<F>(F e1,F e2);
    max: Interval[Successorable] ⟶ Sucessorable is F fst();
    min: Interval[Successorable] ⟶ Sucessorable is F snd();
    before: Interval[Successorable] e: Interval[Successorable]  is
                    boolean before(MyInterval<F> e);
    elements: Interval[Successorable] ⟶ SortedSet[Successorable] is
                    TreeSet<F> elems();
  }
  TOTAL_ORDER is E  {
    geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
  TOTAL_ORDER_WITH_SUC is F  {
    suc: Successorable ⟶ Successorable is F suc();
  }
end refinement
```

Fig. 7. A refinement for an interval.

```
interface ISuccessorable⟨E⟩ extends IOrderable⟨E⟩{
  E suc();
}

public class MyInterval⟨E extends ISuccessorable⟨E⟩⟩{
  public MyInterval⟨E⟩(E e1, E e2){...}
  public E fst(){...}
  public E snd(){...}
  public before (MyInterval⟨E⟩ i) {...}
  public TreeSet⟨E⟩ elems(){...}
  ...
}
```

Fig. 8. An excerpt of a Java implementation of an interval.

`TreeSet` is bounded to extend `IOrderable⟨E⟩`, which, in turn, declares the method **`boolean`** `greaterEq(E e)`.

At a first glance, it may look strange that, the definition of refinement mapping, does not require instead that the parameter specification TOTAL_ORDER be mapped directly to `IOrderable⟨E⟩`, the bound associated to `E`. This would be simpler to write but it would be much too restrictive, without practical interest. In our example, this notion would only be applicable if **`boolean`** `greaterEq(E e)` in `IOrderable⟨E⟩` was replaced by **`boolean`** `greaterEq(IOrderable⟨E⟩e)`. This would require that any two objects of any two classes that can be used to instantiate `E` in `TreeSet` can be compared which is, clearly, much stronger than what is necessary.

A more complex and interesting example that shows the full potential of our notion of refinement mapping is presented in Figure 7. It describes a refinement mapping between the module ITV and the java types `TreeSet⟨E⟩`, `IOrderable⟨E⟩` and

`MyInterval`⟨`E`⟩, `ISuccessorable`⟨`E`⟩. Figure 8 partially shows how the last two types were defined. Notice that, in this case, because $F < E$, in order to check that the described function defines a refinement mapping we have to confirm that any `K` that can be used to instantiate `E` in `MyInterval` possesses methods **`boolean`** `greaterEq(K e)` and `K suc()`. This is indeed the case as the parameter `E` of `MyInterval` is bounded to extend `ISuccessorable`⟨`E`⟩, which, in turn, declares **`boolean`** `greaterEq(E e)` and inherits from `IOrderable`⟨`E`⟩ the method **`boolean`** `greaterEq(E e)`.

In order to check the expressive power of the proposed notion, we have additionally considered a large number of examples that appear in the context of a typical Algorithms and Data Structure Course.

4 Conformance between Modules and Java Programs

In the previous section, we characterised the class of Java programs that are appropriate for interpreting a specification module. In this section, we characterise the programs that are in conformity with the module, i.e., those that correctly realize the specified requirements. In what follows, we consider a set $\mathcal{C}$ of Java types that is appropriate for interpreting a module $\mathcal{M}$ with the refinement mapping $\mathcal{R}$. For illustration purposes we use the refinement mapping presented in Figure 6 between the module SS and $\mathcal{C}$={`TreeSet`⟨`E`⟩, `IOrderable`⟨`E`⟩}.

Specifications introduced in Section 2 define two types of properties: axioms and domain conditions. In the first paragraphs we address these two types of properties when they are defined in the context of core specifications. Finally, in the last paragraph, we address properties of parameter specifications.

4.1 Constraints imposed by Axioms of Core Specifications

The axioms included in a core specification S impose constraints over the behaviour of the corresponding class, $\mathcal{R}(S)$. More concretely, the axioms specified in a non-parameterized specification S must be fulfilled by every object of type $\mathcal{R}(S)$. Notice that in the case of specifications that make use of subsorting, the above condition implies that the axioms defined for values of the supersort are also fulfilled by the values of the subsort. This is a simple consequence of the type system: if an object has type $T < T'$, then it also has type T'. In case S is a parameterized specification, the axioms must be fulfilled by every object of any type T that can be obtained through the instantiation of $\mathcal{R}(S)$, which in this case is a Java generic type.

From the point of view of an object, the properties described by axioms are *invariants* — they should hold in all client visible-states, i.e., they must be true when control is not inside the object′s methods. In particular, they must be true at the end of each constructor′s execution, and at the beginning and end of all methods [13, 16].

It remains to define which object invariants are specified by axioms. Consider for instance the following two axioms, included in SORTED_SET:

```
E, F: Orderable;  S: SortedSet[Orderable];
largest(insert(S, E)) = E if isEmpty(S);
isIn(insert(S,E),F) iff E = F or isIn(S,F);
```

The translation of these axioms to properties of an object `ts:TreeSet`$\langle$`K`$\rangle$, for some type `K` that implements `IOrderable`$\langle$`K`$\rangle$, has to take into account that method `insert` is **`void`**. In what concerns the first axiom, `ts` has to satisfy the following property:

In all client visible-states, for all `k:K`,

- if `ts.isEmpty()` holds then, immediately after the execution of `ts.insert(k)`, the expression `ts.largest().equals(k)` evaluates to true.

The translation of the second axiom is more complex because it is an equivalence. In this case, the property that `ts` has to satisfy is the following:

In all client visible-states, for all `k,k':K`,

- if `k.equals(k')` or `ts.isIn(k')` is true then, immediately after the execution of `ts.insert(k)`, the expression `ts.isIn(k')` evaluates to true;
- if, immediately after the execution of the invocation `ts.insert(k)`, the expression `ts.isIn(k')` evaluates to true, then `k.equals(k')` or `ts.isIn(k')` is true.

Space limitation prevents us from presenting the translation function induced by a refinement mapping $\mathcal{R}$. For modules composed of simple specifications only, this translation was not only formally defined but also encoded in the form of a runtime conformance tool in the context of CONGU (see [19] for details). As the example shows, generics and subsorting do not raise any additional difficulty in the translation process.

4.2 Constraints imposed by Domain Conditions of Core Specifications

The domain conditions included in a core specification S impose constraints over the behaviour of the corresponding class, $\mathcal{R}(S)$, as well as over all classes in $\mathcal{C}$ that are clients of $\mathcal{R}(S)$.

Let ϕ be a domain condition of an operation *op* in a non-parameterized (resp. parameterized) specification S. On the one hand, the implementation of method $\mathcal{R}(op)$ must be such that, for every object o of type $\mathcal{R}(S)$ (resp. for every object o of a type that results from the instantiation of $\mathcal{R}(S)$), in all client visible-states, if the property ϕ holds, then a call of $\mathcal{R}(op)$ terminates normally (i.e., no exception is raised) [16]. Notice that operations that are not declared to be partial, implicitly have the domain condition *true* and, hence, corresponding methods must always return normally. On the other hand, ϕ defines a pre-condition for method $\mathcal{R}(op)$. That is to say, ϕ must hold at the time the method is invoked. Hence, ϕ defines a constraint over the behaviour of all classes in $\mathcal{C}$ that are clients of $\mathcal{R}(S)$ and use method $\mathcal{R}(op)$. These classes cannot call $\mathcal{R}(op)$ if ϕ does not hold.

As an example, let us consider the domain condition of largest included in specification SORTED_SET. Let `K` be a class that implements `IOrderable`$\langle$`K`$\rangle$. For any object `ts:TreeSet`$\langle$`K`$\rangle$ in a client-visible state in which `!ts.isEmpty()` holds, the call of `largest` must return normally. Because $\mathcal{C}$ does not contain any client of `TreeSet`, no more restrictions are imposed by this domain condition.

4.3 Constraints imposed by Parameter Specifications

Axioms and domain conditions described in parameter specifications of a module impose constraints similar to those defined in previous sections for core specifications. The difference lies on the target of these constraints only. The axioms included in a parameter specification S impose constraints over the behaviour of the types of $\mathcal{C}$ that are used to instantiate the corresponding generic type. More precisely, if S' is a parameterized specification with parameter S, then every object of a type K used in $\mathcal{C}$ to instantiate the Java parameter type variable $\mathcal{R}(S)$ of $\mathcal{R}(S')$ must fulfil the axioms of S. In the case of domain conditions, there are also constraints that apply to the clients of these classes.

In order to illustrate these ideas, consider a Java program $\mathcal{C}'$ that, in addition to `TreeSet⟨E⟩` and `IOrderable⟨E⟩`, includes classes `Date`, `Card` and `Main`. Suppose also the last class is a client of `TreeSet⟨Date⟩`, `TreeSet⟨Card⟩`, `Date` and `Card`. Because $\mathcal{C}'$ is a superset of $\mathcal{C}$, the mapping $\mathcal{R}$ is also a refinement mapping between SS and $\mathcal{C}'$. The program $\mathcal{C}'$ is in conformity with SS only if (1) Both `Date` and `Card` behave according to the properties described in TOTAL_ORDER; (2) `Main` respects the domain conditions defined in SORTED_SET for operation largest, when invoking the corresponding method either over an object of `TreeSet⟨Card⟩` or an object of `TreeSet⟨Date⟩`.

5 Extending the ConGu Tool

The ConGu tool [10] supports the runtime checking of Java classes against algebraic specifications. The input to the tool are specification modules and refinement mappings, and of course, Java programs, in the form of *bytecode*. Running a program under ConGu may produce exceptions due to domain condition violations or to axiom violations. The first case is a manifestation of a ill-behaved client, i.e., a client that invokes a method in a situation in which the method should not be invoked. 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 approach to runtime checking used in ConGu involves replacing the original classes and generating further classes annotated with monitorable contracts, presently written in JML [13]. The generated pre and post-conditions allow to check if the constraints imposed by axioms and domain conditions hold at specific execution points. The approach is applicable even if the specified operations are implemented as **`void`** methods or methods with side effects. Roughly speaking, this is achieved as follows.

For every specification S in the module, considering that $\mathcal{R}(S)$ is the class `MyT`:

- Rename bytecode `MyT` to `MyT_Original`;
- Generate a static class `MyT_Contract` annotated with contracts automatically generated from the axioms and domain conditions specified in S. This class is a sort of functional version of the original class `MyT`. For instance, if class `MyT_Original` has the method **`void`** `m(...)`, then class `MyT_Contract` has a static method `MyT_Original m(MyT_Original o,...)` that starts by executing `co=o.clone()` and then returns `co` after the execution of `co.m(...)`. Notice that the usage of

clones in all methods of MyT_Contract ensures that the methods are *pure* (i.e., without side effects) and, hence, can be used in contracts. This step also generates classes to describe state-value pairs and ranges to be used in *forall* contracts;
– Generate a proxy class MyT with exactly the same interface as the original one. This is a *wrapper class* in the sense that it intercepts all method invocations coming from clients of the original class and forwards these invocations to the corresponding method of MyT_Contract. In this way, contracts corresponding to axioms and domain conditions are monitored.

In the rest of this section, we briefly discuss how this approach can be extended in order to support the more comprehensive notion of conformance between specification modules and Java programs presented in the previous section.

From the point of view of the structure, there are two main extensions that CONGU must suffer in order to deal with generic classes. One implies the creation of contract classes for every Java type declared as the upper bound of parameters to generic classes. For instance, in the implementation of the SORTED_SET specification, in which we have IOrderable⟨E⟩ interface as the upper bound for the parameter of the generic class TreeSet⟨E⟩, CONGU must create a class IOrderable_Contract. In this class, the method greaterEq(IOrderable e1, IOrderable e2) uses dynamic dispatching to invoke the correct greaterEq method over a clone of the e1 argument. Every contract that needs to invoke the greaterEq method over some element of a TreeSet (for example, the post-condition for the insert method in which two elements of the TreeSet must be compared), must do so using the IOrderable_Contract class.

Central to the implementation of CONGU is the ability to clone objects for contract monitoring purposes. CONGU distinguishes Cloneable from non-Cloneable types, cloning references of the former kind, calling contracts directly on the references of latter form. Java types declared as upper bounds of parameters to generic classes must be declared Cloneable in order to make it possible to create clones of its subtypes. This is because all objects of classes that implement some upper bound T in the context of a generic class instantiation, are statically used as T objects in contract classes. For example, the IOrderable interface in Figure 5 must extend the marker interface Cloneable (and announce the signature of method clone) if any of its implementations turns out to be mutable.[2]

Presently, CONGU generates JML contracts. At the time of this writing JML does not support generic types in source code, hence CONGU generates only non-generic (Java 1.4) code. In order to continue using the existing framework, the generated JML-related classes must be non-generic. Specifically, we generate non-generic contract classes (TreeSet_Contract and IOrderable_Contract), together with the remaining auxiliary (pair and range) classes. JML then compiles these classes relying on the bytecode for the program to be monitored (whose source code may contain generics that are not reflected in the bytecode) to resolve dependencies.

The other main extension concerns classes arguments to generic classes. In order to enforce the constraints imposed by parameter specifications of a module over a Java

[2] One may wonder if forcing all subtypes of a parameter type to be of the same kind (cloneable or non-cloneable) is over-restrictive. We note however that immutable classes can implement method clone by simply returning **this**, if so desired.

program $\mathcal{C}$, ConGu must first be capable of determining, for each generic specification S and each of its parameter specifications S', the set $\mathcal{T}(S, S')$ of types that are used in the program to instantiate the corresponding parameter of $\mathcal{R}(S)$ (for instance, in the program considered in the previous section, $\mathcal{T}(\mathsf{SORTED\_SET}, \mathsf{TOTAL\_ORDER})$ consists of classes `Date` and `Card`). Then, we can apply the methodology described before, and generate a proxy for each type in this set, each referring to the same contract class capturing the requirements specified in S' (e.g., `IOrderable_Contract`).

The problem of determining the set $\mathcal{T}(S, S')$ can only be completely solved if the source code of the Java program is available, for such information is lost in the bytecode generated by the Java compiler (through a bytecode inspection one can compute the set of subtypes of the upper bound of $\mathcal{R}(S')$ found in $\mathcal{C}$, which is a superset of the required $\mathcal{T}(S, S')$). In the absence of the source code, a possible solution is to ask the user to explicitly identify the classes that ConGu should monitor for each parameter.

The sets $\mathcal{T}(S, S')$ may include classes that do not correctly implement S'. If this is the case for some class `A`, then the execution of program $\mathcal{C}$ under ConGu may behave in two different ways, depending on the nature of the violation. It may produce an exception signalling the violation of an axiom of S' in `A`, or it may as well signal the violation of an axiom of a generic specification that has S' as a parameter. This problem of imprecise blame assignment can be mitigated if, prior to executing $\mathcal{C}$, we execute some simpler programs under ConGu that do not involve clients of the generic classes but, instead, involve direct clients of the classes in $\mathcal{T}(S, S')$. In this way, only the classes that correctly implement S' would be used to test the generic implementations $\mathcal{R}(S)$ of generic specifications S that have S' as a parameter. To some extent, this problem is related with the problem of testing generic units independently of particular instantiations addressed in [15]. Herein, given that the set of all possible instantiations is almost always infinite, it is suggested that a representative finite class of possible parameter units be selected.

6 Conclusion and Further Work

The work presented in this paper contributes to bridging the gap between parameterized specifications and Java generic classes, in the context of checking the conformance of the latter against the former. By evolving the ConGu approach to cope with more sophisticated specifications — involving parameters and subsorting — it becomes applicable to a more comprehensive range of situations, namely those that appear in the context of a typical Algorithms and Data Structures course and in the context of reusable libraries and frameworks.

Algebraic specifications are known to be especially appropriate for describing software components that implement data abstractions. These components play an important role in software development and are, today, an effective and popular way of supporting reuse. Many ready-to-use components produced for libraries and frameworks that support reuse of common data abstractions are generic. Given that generics are known to be difficult to grasp, obtaining correct implementations becomes more challenging and, hence, automatic support for detection of errors becomes more relevant.

We presented a specific interpretation of specification modules in terms of sets of Java classes and interfaces. This is useful if we take the perspective of a Java developer that has to implement a collection of units, described in terms of a specification module. The interpretation rigourously defines what the module's correct implementation in Java is and paves the way to the development of tools that support the runtime conformance checking of Java programs. Using these tools increases the confidence in the source code, and facilitates component-based implementation. The CONGU tool already supports checking Java classes against simple specifications. A solution for extending it in order to cope with specifications using parameters and subsorting was also presented.

We are currently implementing the sketched extension to CONGU tool. As for future work, in order to increase the effectiveness of the monitoring process, while keeping it completely automated, we also intend to develop testing techniques based on the specifications. Our aim is the automatic generation of unit and integration testing that make use of runtime conformance checking. A black box approach to the reliability analysis of Java generic components brings the challenge of not having any a priori knowledge of which Java types are to be used to instantiate parameters. Therefore, tests are harder to generate, because test data must be generated of types that are unknown at testing time. Existing methods and techniques to automatically generate test suites from property-driven specifications cannot be directly applied, and coverage criteria and generation strategies of test cases in this new context needs also to be investigated.

References

1. S. Antoy and J. Gannon. Using term rewriting to verify software. *IEEE Transactions on Software Engineering*, 4(20):259–274, 1994.
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. 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.
4. M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Proc. Workshop on Specification and Verification of Component-Based Systems 2001*, 2001.
5. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
6. M. Bidoit and P. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer, 2004.
7. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *Algebraic Methodology and Software Technology (AMAST) 1999*, volume 1548 of *LNCS*, pages 341–357. Springer, 1999.
8. 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, 2005.
9. F. Chen, N. Tillmann, and W. Schulte. Discovering specifications. Technical Report MSR-TR-2005–146, Microsoft Research, 2005.
10. Contract based system development. http://gloss.di.fc.ul.pt/congu/.
11. S. Edwards, G. Shakir, M. Sitaraman, B. Weide, and J. Hollingsworth. A framework for detecting interface violations in component-based software. In *Proc. International Conference on Software Reuse (ICSR) 1998*, pages 46–55, 1998.

12. J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.
13. G. Leavens and Y. Cheon. Design by contract with JML, 2006. http://www.eecs.ucf.edu/ leavens/JML/jmldbc.pdf.
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. P.L. Machado and D. Sannella. Unit testing for CASL architectural specifications. In *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, volume 2420, pages 506–518. Springer, 2002.
16. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997.
17. B. Meyer. Eiffel as a framework for verification. In *Proc. Verified Software: Theories, Tools, and Experiments (VSTTE) 2005*, volume 4171 of *LNCS*, pages 546–550, 2007.
18. B. Nikolik and D. Hamlet. Practical ultra-reliability for abstract data types. *Software Testing, Verification and Reliability*, 17(3):183–203, 2007.
19. I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis. Testing implementations of algebraic specifications with design-by-contract tools. DI/FCUL TR 05–22, 2005.
20. I. Nunes, A. Lopes, V.T. Vasconcelos, J. Abreu, , and L.S. Reis. Checking the conformance of Java classes against algebraic specifications. In *Proc. of Eighth International Conference on Formal Engineering Methods (ICFEM) 2006*, volume 4260 of *LNCS*, pages 494–513. Springer, 2006.
21. K. Stenzel, H. Grandy, and W. Reif. Verification of java programs with generics. In *Proc. Algebraic Methodology and Software Technology (AMAST) 2008*, volume 5140 of *LNCS*. Springer, 208.
22. 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.