Specification-driven Unit Test Generation for Java Generic Classes

Francisco R. de Andrade[2], João P. Faria[2,3], Antónia Lopes[1], and Ana C.R. Paiva[2]

[1] Faculdade de Ciências da Universidade de Lisboa
[2] Faculdade de Engenharia, Universidade do Porto
[3] INESC Porto
{francisco.andrade,jpf,apaiva}@fe.up.pt, mal@di.fc.ul.pt

Abstract. Several approaches exist to automatically derive test cases that check the conformance of the implementation of abstract data types (ADTs) with respect to their specification. However, they lack support for the testing of implementations of ADTs defined by generic classes. In this paper, we present a novel technique to automatically derive, from specifications, unit test cases for Java generic classes that, in addition to the usual testing data, encompass implementations for the type parameters. The proposed technique relies on the use of Alloy Analyzer to find model instances for each test goal. JUnit test cases and Java implementations of the parameters are extracted from these model instances.

1 Introduction

Algebraic specifications have been successfully used for the formal specification of abstract data types (ADTs) and several approaches exist to automatically derive test cases that check the conformance of the implementation of ADTs with respect to their algebraic specifications (e.g., [3,5,8,11,17]). In these approaches, because ADTs are described in an axiomatic way, the derivation of tests involves choosing some instantiations of the axioms or their consequences. Then, concrete tests are generated to check if these properties hold in the context of the implementation under test (IUT).

Many data types admit different versions in different applications—e.g., *sets of strings, dates, messages*. Nowadays, the implementation of these data types in mainstream object-oriented languages, such as Java and C#, strongly relies on generic classes. However, existing methods and techniques to automatically generate test suites from specifications cannot be directly applicable in these cases.

Genericity poses new difficulties for testing. To write tests for a generic class one has to commit to a set of types for its parameters and this raises several problems. First, in the case of non trivial parameters, types for instantiating them may not be available at test time (e.g., trees of intervals as defined by edu.stanford.nlp.util.IntervalTree have intervals as parameters and no implementation for this type is available there [16]). Second, the types available for instantiating the parameters may not cover all the possibilities allowed by the parameters. For instance, partially ordered sets that have a type parameter that corresponds to partial orders can be tested with strings or integers, however, the properties of these sets that hold vacuously in total orders will not be properly tested. Third, in order to isolate the source of possible failures, one may not want to depend on the implementation of other types besides the one under test (this is a unit

testing best practice). A technique that is often used to overcome these difficulties in manual test generation is the use of mock objects [14]. One of the challenges in automatic test generation for generic classes is the automatic generation of mock objects for their parameters, removing from the user the burden of providing the types for instantiating the type parameters.

In this paper we address the generation of unit test cases for Java implementations of ADTs defined by generic classes, that comprise automatically generated mock classes and mock objects that can be used to instantiate their type parameters. As illustrated in Fig. 1, we consider that ADTs are described by parameterized specifications and that the abstraction gap between the specifications and the implementations is bridged through refinement mappings. Parameterized specifications are supported by several specification languages. In contrast, refinement mappings were defined in [15] for CONGU specifications [4]. Herein, we revisit this notion and reformulate it in a more general setting.

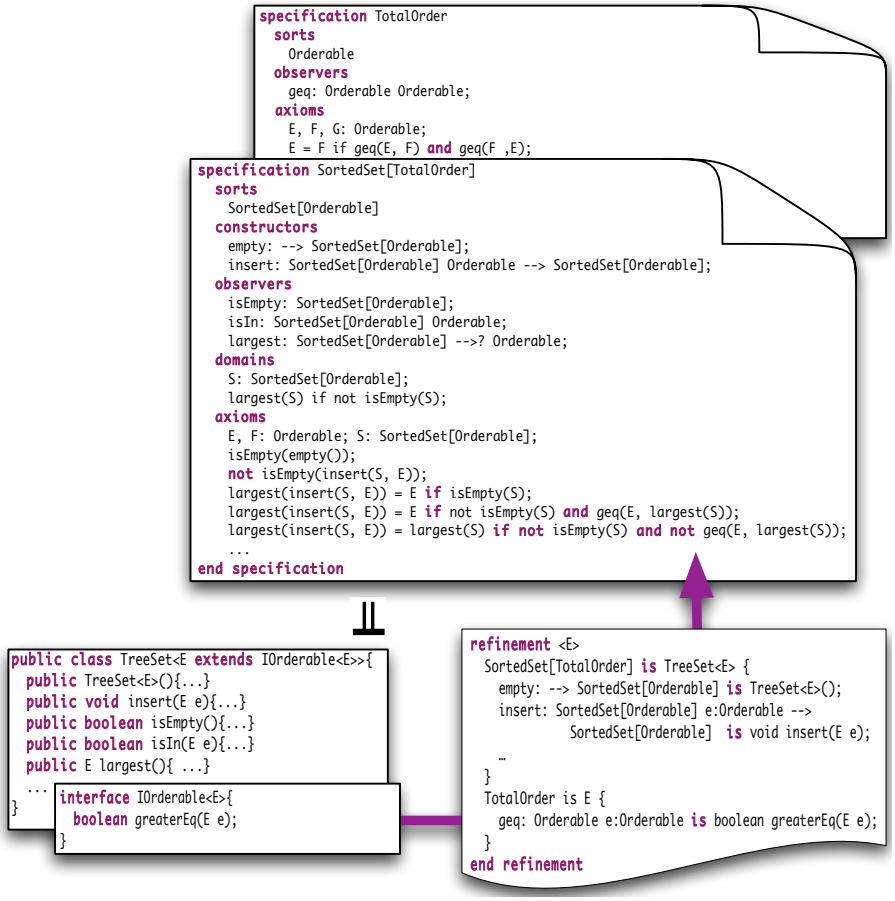

Fig. 1. The aim is to generate tests for checking if a set of classes correctly implements an ADT.

Following the tradition of specification-based testing, the developed technique involves considering some abstract tests obtained through the instantiation of the axioms.

The difference is that, in our case, this instantiation is not exclusively achieved at the syntactical level by substitution of axiom variables by ground terms; it also involves assigning a value to some variables according to a specific model of the parameter specification. For the generation of abstract tests, the proposed technique relies on Alloy Analyzer [13], a tool that finds finite models of relational structures. Abstract tests are then translated into JUnit tests for a given implementation of the specification. This translation takes into account the correspondence between specifications and Java types defined by the given refinement mapping. Fig. 2 presents an overview of this process.

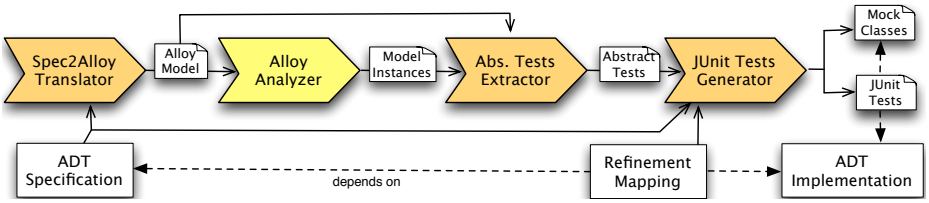

Fig. 2. Overview of the test generation process.

The organisation of the paper is as follows. Sec. 2 presents the specifications we have considered in our approach and their semantics. In Sec. 3, we introduce a notion of abstract test appropriate for parameterized specifications and present a technique for the generation of these tests that relies on the encoding of specifications into Alloy and on Alloy Analyzer for finding model instances. In Sec. 4, we show how to automatically translate abstract tests to JUnit tests for a concrete implementation. Sec. 5 presents some evaluation experiments and Sec. 6 concludes the paper and discusses future work.

2 Specifications of Generic Data Types

In algebraic specification, the description of ADTs that admit different versions is supported by parameterized specifications [6]. The description of algebraic specifications in general, and parameterized specifications in particular, is supported by different languages (e.g., [2,6,7]) with significant variations in terms of syntax and semantics. In this section, we present the specifications considered in our approach and their semantics.

Preliminaries. We use Σ to represent a many-sorted signature $\langle S, F, P\rangle$, where S is the set of sorts and F and P are the sets of, respectively, operation and predicate symbols. Moreover, we use $Spec$ to represent a specification $\langle \Sigma, Ax\rangle$, where Ax is the set of axioms described by formulas in first-order logic (with equality). An example of a specification is $TotalOrder$, partially described in Fig. 1 using the language of CONGU. It has the sort $Orderable$, a single predicate symbol geq and no operation symbols. Its set of axioms includes $\forall e{:}\,Orderable\ \forall f{:}\,Orderable\ .\ geq(e, f) \wedge geq(f, e) \Rightarrow e = f$.

We use $PSpec$ to represent a parameterized specification, i.e., a pair $\langle Param, Body\rangle$ of specifications with $Param$ (the formal parameter) included in $Body$ (the body). An example of a parameterized specification is $SortedSet[TotalOrder]$, also partially described in Fig. 1. The formal parameter is $TotalOrder$ and the body is $SortedSet$, a specification that contains what is in $TotalOrder$ and the sort $SortedSet[Orderable]$, the operation symbols $empty$, $insert$ and $largest$, the predicate symbols $isEmpty$ and $isIn$, and several axioms that express the properties of these operations and predicates.

2.1 Specifications

In this work, we restrict our attention to a set of parameterized specifications that can be described in CONGU. More concretely, we consider specifications in which operation symbols are classified as *constructors* or *observers* and that are obtained through the extension of a given specification $Spec$ with *an increment*, i.e.,: (i) a single sort s, (ii) constructors that produce elements of sort s, (iii) observers and predicate symbols that take an element of sort s as first argument, (iv) axioms that express properties of the new operation and predicate symbols only. An increment can define a specification by itself or rely on sorts, operations and predicates available in the base specification $Spec$. In any case, we use $Spec{+}Spec_s$ to represent the extension of $Spec$ with an increment centred on sort s and $Spec_{s_1}{+}...{+}Spec_{s_n}$ to represent a sequence of increments. For the body of $PSpec$, we also require that all increments different from $Param$ include at least one *creator*, i.e., a constructor that does not have elements of the introduced sort among its arguments. In the sequel, we use $Body{-}Param$ to refer to sorts, operations, predicates and axioms in $Body$ but not in $Param$. It is easy to see that $TotalOrder$ and $SortedSet$ fulfill these requirements: $TotalOrder$ is an extension of the empty specification while $SortedSet$ is an extension of $TotalOrder$ that indeed introduces one creator ($empty$) of the introduced sort ($SortedSet[Orderable]$).

In the sequel we will use $Term_\Sigma$ and $CTerm_\Sigma$ to denote, respectively, the set of ground terms and the set of canonical ground terms (i.e., terms defined exclusively in terms of constructors). For terms, canonical terms and formulas built over a set X of variables typed by sorts in Σ, we use $Term_\Sigma(X)$, $CTerm_\Sigma(X)$ and $Form_\Sigma(X)$.

In what concerns the axioms, we assume they have one of the following forms:

$\forall x_1 : s_1 ... \forall x_n : s_n \;.\; \phi \qquad\qquad \forall x_1{:}s_1 ... \forall x_n : s_n.\; \psi_{op} \Rightarrow \mathit{defined}(op(x_1, ..., x_n))$

where op is an operation symbol and ϕ, ψ_{op} are quantifier-free first-order logic formulas built over Σ. The first type of axioms is used for expressing usual properties of operations and predicates. The other type of axioms supports the definition of a domain condition of an operation, i.e., the condition under which the operation must be defined (these are needed because operations can be interpreted as partial functions). In $SortedSet[TotalOrder]$, all operations but $largest$ must be interpreted by total functions and, hence, their domain conditions are $true$ while $largest$ has to be defined for non empty sets, i.e., $\forall s : SortedSet[Orderable].\; \neg isEmpty(s) \Rightarrow \mathit{defined}(largest(s))$.

We further assume there is exactly one domain condition for each operation, which allows us to define the formula $\mathit{defined}^*(t)$ that, as we will see later on, defines sufficient conditions for the term t to be defined.

Definition 1. *$\mathit{defined}^*(t)$ is the formula defined inductively in the structure of term t as follows: (1) $\mathit{defined}^*(x) = true$ if x is a variable, (2) $\mathit{defined}^*(op(t_1, ..., t_n)) = \mathit{defined}^*(t_1) \wedge ... \wedge \mathit{defined}^*(t_n) \wedge \psi_{op}[t_1/x_1, ...t_n/x_n]$ if $op : s_1, ..., s_n \rightarrow s$ is an operation with domain condition $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\; \psi_{op} \Rightarrow \mathit{defined}(op(x_1, ..., x_n))$.*

2.2 Semantics

Specifications are interpreted in terms of Σ-algebras. More concretely, we take Σ-algebras as triples $A = \langle \{A_s\}_{s \in S}, \mathcal{F}, \mathcal{P} \rangle$, where A_s is the carrier set of sort s, $\mathcal{F}$ defines the interpretation of operation symbols as partial functions and $\mathcal{P}$ defines the

interpretation of predicate symbols as relations. We use $[\![t]\!]^{A,\rho}$ to denote the interpretation of a term t in $Term_\Sigma(X)$ with an assignment ρ of X into A, i.e., a function that assigns a value in A_s to each variable $x{:}s$ in X. Given that operation symbols can be interpreted by partial functions, $[\![t]\!]^{A,\rho}$ might not be defined. In fact, $[\![t]\!]^{A,\rho}$ is defined if and only if $A, \rho \vDash \mathit{defined}(t)$. The interpretation of equality also has to take into account the possibility of terms not being defined. Equality is interpreted as being strong, i.e., $t_1 = t_2$ holds in a Σ-algebra A when the values of both terms are defined and equal or both are undefined. In what concerns predicates, when they are applied to undefined terms they are always false (see [2] for the rational of this choice).

There are various forms of semantic construction in the algebraic approach to specification of ADTs. For the purpose at hand, the appropriate construction is loose semantics. It associates to $\mathit{Spec}{=}\langle \Sigma, Ax\rangle$ the class of all Σ-algebras which satisfy its axioms Ax; these are called Spec-algebras. According to this semantics, an implementation of the ADT in which all specified properties hold is considered to be correct.

For parameterized specifications, loose semantics associates to $\langle Param, Body\rangle$ the class of functions $\mathcal{T}_{Body}$ that assign to each $Param$-algebra A, a $Body$-algebra $\mathcal{T}_{Body}(A)$ that coincides with A when restricted to $Param$. This means that an implementation of a parameterized specification is correct if it has all the specified properties, when instantiated with any correct implementation of its parameter.

3 Generation of Abstract Tests

The envisaged strategy for deriving test cases for implementations of generic data types encompasses the generation of tests for their parameterized specifications. We call them abstract tests because their target are abstract models (algebras). For testing Java implementations, we need to convert them into object-oriented tests (JUnit tests, in our case).

3.1 Tests for Parameterized Specifications

A test for an algebraic Spec is usually defined as a ground and quantifier-free formula that is a semantic consequence of Spec and, hence, valid in every Spec-algebra [8]. This can be generalised to parameterized specifications but the result is not interesting as specifications used as parameters are not expected to have creators and so, the corresponding set of ground terms is empty. In fact, specifications used as parameters are not expected to have constructors as they often correspond to a required "ability". For instance, in our example, $TotalOrder$ corresponds to a requirement for the actual parameter of a sorted set to have a comparison operation that defines a total order.

The notion of test that we found useful for parameterized specifications is one in which we fix a specific $Param$-algebra.

Definition 2. *A* closed test *for a parameterized specification $PSpec = \langle Param, Body\rangle$ is a tuple $\langle A, X, \phi, \rho_P, \rho_B\rangle$ where A is a $Param$-algebra; X is a finite set of variables typed by sorts in $Body$; ϕ is a quantifier-free logic formula in $Form_\Sigma(X)$; ρ_P is an assignment of X_P into A, where X_P is the set of variables in X typed by sorts in $Param$; ρ_B is a function that assigns a term $\rho_B(x)$ in $Term^s_\Sigma(X_P)$ to each $x{:}s$ in $X_B{=}X\backslash X_P$; such that $\mathcal{T}_{Body}(A), \rho_P \vDash \rho^*_B(\phi)$, for every $\mathcal{T}_{Body}$ in the semantics of PSpec, where $\rho^*_B(\phi)$ is the translation of formulas induced by ρ_B.*

Notice that, in these tests, the instantiation of the variables in the formula is achieved through the combination of (i) a syntactic replacement of variables in X_B by terms and (ii) an assignment of variables in X_P into the fixed $Param$-algebra. In this way, we can exercise the test in any Σ_{Body}-algebra that extends A.

We are interested in tests that result from the instantiation of axioms of the form $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$. Closed tests may involve the replacement of variables by terms and their interpretation in a specific Σ_{Body}-algebra might be undefined and, hence, this instantiation needs to be conditioned by the definedeness of these terms. Because the formula $\mathit{defined}^*(t)$ provides a sufficient condition for the term t to be defined in any $Spec$-algebra (for details, see [1]), we can use the formula $\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x)) \Rightarrow \phi$.

Proposition 1. *Let $PSpec = \langle Param, Body \rangle$ be a parameterized specification and $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$ an axiom in $Body$. If A is a $Param$-algebra, X is a set of variables including $\{x_1{:}s_1, ..., x_n{:}s_n\}$, ρ_P is an assignment of X_P into A and ρ_B is a function that assigns a term $\rho_B(x)$ in $Term^s_\Sigma(X_P)$ to each $x{:}s$ in X_B, then*

$$\langle A, X, (\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x))) \Rightarrow \phi, \rho_P, \rho_B \rangle$$

is a closed test for $PSpec$.(See [1] for the proof.)

As an example, let us consider the axiom $\forall s{:}SortedSet[Orderable].\ \forall e{:}Orderable.\ \neg isEmpty(insert(s, e))$ of $SortedSet[TotalOrder]$. As a result of Prop. 1:

- the $TotalOrder$-algebra TO^2 with two elements, say, $Ord0$ and $Ord1$ and geq interpreted as the relation $\{(Ord1, Ord0), (Ord1, Ord1), (Ord0, Ord0)\}$
- the set of variables $\{s{:}SortedSet[Orderable], e{:}Orderable\}$
- the formula $true \Rightarrow \neg isEmpty(insert(s, e))$
- ρ_P: $\{e{:}Orderable \mapsto Ord0\}$ and ρ_B: $\{s{:}SortedSet[Orderable] \mapsto empty()\}$

defines a closed test for $SortedSet[TotalOrder]$.

3.2 Generation Technique

When tests are obtained through ground instantiation of axioms, performing a test experiment just requires evaluating a ground formula in the IUT. The generation of closed tests for parameterized specifications also involves the instantiation of axioms, but this instantiation is only partial — the instantiation of an axiom involving a set of variables X is limited to the variables in X_B. Hence, the generation of closed tests involves the generation of models for the parameter specification and evaluations in these models for the variables in X_P. In this subsection, we describe a technique for the generation of abstract test suites for parameterized specifications that can be subsequently translated into JUnit test suites for testing Java implementations.

As pointed out in [8], test thoroughness is increased by generating multiple test cases for each axiom, through the partitioning of each axiom into a finite set of "cases", either by successively unfolding the premises of equational axioms or by considering the conjunctive terms in the Disjunctive Normal Form (DNF) of the axiom expression. In our case, since axioms are not restricted to equational ones, DNF partitioning is more directly applicable, with the advantage of not mixing together different axioms. To further assure that the different cases are disjoint, and hence avoid generating redundant tests, we take a special DNF form — the Full Disjunctive Normal Form (FDNF). The FDNF of a logical formula that consists of Boolean variables connected by logical

operators is a canonical DNF in which each Boolean variable appears exactly once (possibly negated) in every conjunctive term (called a minterm) [9].

The technique consists in considering each of the axioms $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$ in $Body{-}Param$ that do not express a domain condition and start by converting it to FDNF. Assuming that the result is $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi_1 \vee ... \vee \phi_k$ then, for every $1 \leq i \leq k$, the technique involves using Alloy Analyzer to find a $Body$-algebra M such that: (1) M is finite; (2) M satisfies sort generation constraints for sorts in $Body{-}Param$ (each of these sorts is constrained to be generated by the declared constructors); (3) M satisfies a stronger version of the domain condition of every operation op in $Body{-}Param$: $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \psi_{op} \Leftrightarrow \mathit{defined}(op(x_1, ..., x_n))$; (4) M satisfies $\exists x_1{:}s_1 ... \exists x_n{:}s_n.\ \phi_i$.

Only axioms in $Body{-}Param$ are considered because these are the axioms that express the properties of the generic data type that we are interested to check in the IUT (the other axioms concern properties that are expected to hold in actual parameters). In what concerns the constraints imposed on the finding of the $Body$-algebra (the model instances of the Alloy specification): condition 1 is a requirement imposed by the model finder tool, which limits search to finite models; condition 2 excludes models that have junk in the carrier sets as we will need to subsequently convert the elements of this model to arbitrary Σ_{Body}-algebras that extend $M|_{Param}$ (the restriction of M to $Param$); condition 3 avoids the generation of some models that define an evaluation for terms that are undefined in other $Body$-algebras; condition 4 ensures we get from the model finder tool an assignment ρ of the variables in ϕ_i into M satisfying it. Since the formula is in FDNF, all variables of the axiom occur in ϕ_i and, hence, ρ is an assignment of $X{=}\{x_1{:}s_1, ..., x_n{:}s_n\}$ into M.

Because M restricted to sorts in $Body{-}Param$ is a generated model, for each x in X_B, there exists (i) a canonical term $t_x \in CTerm_{\Sigma}(Y_x)$ for some set Y_x of variables typed by sorts in $Param$ and disjoint from X, and (ii) an assignment ρ_x of Y_x into M such that $[\![t_x]\!]^{M,\rho_x} = \rho(x)$. This family of terms and assignments can be used to define a closed test for *PSpec* as follows:

$$\langle M|_{Param}, X', \phi', \rho_P, \rho_B \rangle$$

- $M|_{Param}$ is the restriction of M to $Param$
- $X' = \cup_{x \in X_B} Y_x \cup X$
- ϕ' is the formula $(\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x))) \Rightarrow \phi$
- ρ_P coincides with ρ for X_P and with ρ_x for Y_x, for every $x \in X_B$
- ρ_B is the function that maps each x in X_B into t_x

The correctness of the proposed technique is an immediate consequence of Prop. 1.

Proposition 2. *The tuple $\langle M|_{Param}, X', \phi', \rho_P, \rho_B \rangle$ is a closed test for PSpec.*

Consider, for instance, the axiom of $SortedSet[TotalOrder]$:

$$\forall s{:}SortedSet[Orderable].\forall e{:}Orderable.$$
$$\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \Rightarrow largest(insert(s, e)) = largest(s)$$

One minterm of the corresponding FDNF is $\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \wedge largest(insert(s, e)) = largest(s)$. The application of the technique just described involves using Alloy Analyzer to obtain a $SortedSet$-algebra that satisfies this minterm (and fulfills the other three requirements described before). Fig. 3 presents an example

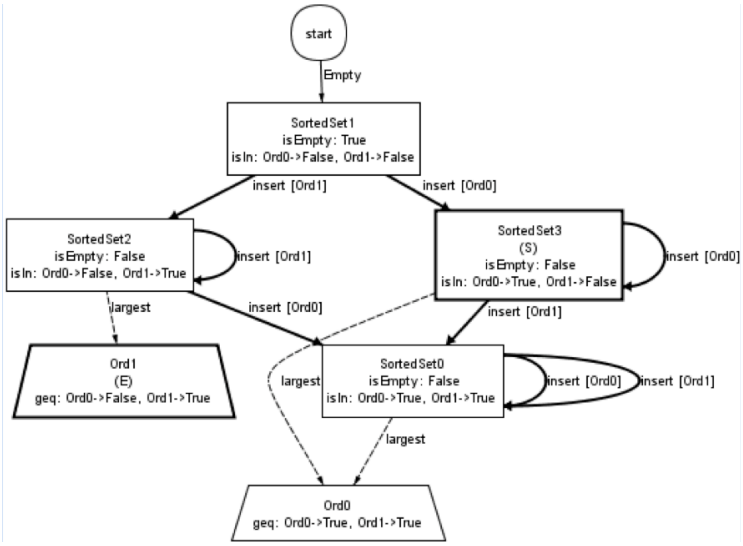

Fig. 3. A model instance defining a $SortedSet$-algebra and an assignment to variables e and s.

of one of these algebras (referred as SS^2 in the sequel), represented as an Alloy model instance. In fact, it also defines ρ:$\{e$:$Orderable \mapsto Ord1$, s:$SortedSet[Orderable] \mapsto SortedSet3\}$. The last step is to find a pair $\langle t_s, \rho_s \rangle$ that represents $SortedSet3$. Through the analysis of SS^2, we find the term $insert(empty(), f)$ and the assignment ρ_s: $\{f$:$Orderable \mapsto Ord0\}$. As a result, we obtain the closed test $\langle TO^2, X', \phi', \rho_P, \rho_B \rangle$ for $SortedSet[TotalOrder]$, where

- X'=$\{s$:$SortedSet[Orderable], e$:$Orderable\} \cup Y_s$ with $Y_s = \{f$:$Orderable\}$
- ϕ' is $defined^*(insert(empty(), f)) \Rightarrow$ $(\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \Rightarrow largest(insert(s, e))$=$largest(s))$, with $defined^*(insert(empty(), f)) = true \wedge true \wedge true$
- ρ_P is the assignment $\{e \mapsto Ord1, f \mapsto Ord0\}$
- ρ_B is the function $\{s \mapsto insert(empty(), f)\}$.

The number of tests generated for each axiom is in general smaller than the number of minterms in the corresponding FDNF since a minterm may not be satisfiable by $Body$-algebras. In particular, this happens in minterms that require the satisfaction of the negation of the definedeness condition of some term. This is the case of the minterm $isEmpty(s) \wedge \neg geq(e, largest(s)) \wedge largest(insert(s, e))$=$largest(s)$.

3.3 From Algebraic Specifications to Alloy and Back

The technique just described requires the ability to generate Alloy models from a parameterized specification $PSpec$=$\langle Param, Body \rangle$ and model finding commands for Alloy Analyzer (Alloy "run" commands) and, in the end, to extract abstract closed tests from the model instances found by Alloy Analyzer.

Encoding of algebraic specifications in Alloy. The encoding of $PSpec$ in Alloy takes into account the sorts, operations, predicates and axioms in $Body$ and, at the same time,

```
sig Orderable  {
    geq: Orderable -> one BOOLEAN/Bool
}
sig SortedSet {
    insert:Orderable -> one SortedSet,
    isEmpty:one BOOLEAN/Bool,
    isIn:Orderable ->one BOOLEAN/Bool,
    largest: lone Orderable
}
one sig start{
    empty: one SortedSet
}
fact SortedSetConstruction{
    SortedSet in (start.empty).*{x: SortedSet, y: x.insert[Orderable]}
}
fact domainSortedSet0{
  all S:SortedSet |
    S.isEmpty != BOOLEAN/True implies one S.largest else no S.largest
}
fact axiomSortedSet4{
    all E:Orderable, S:SortedSet |
       (S.isEmpty = BOOLEAN/False and E.geq[S.largest] = BOOLEAN/False)
          implies (S.insert[E].largest = S.largest)
}
// ... other axioms of Orderable and SortedSet
run run_axiomSortedSet4_0{
    some E:Orderable, S:SortedSet |
       S.isEmpty = BOOLEAN/False and E.geq[S.largest] = BOOLEAN/False
       and S.insert[E].largest = S.largest
} for 6 but exactly 2 Orderable
// ... other run commands for other minterms and axioms
```

Fig. 4. Excerpt of the Alloy model and run commands for $SortedSet[TotalOrder]$.

has to ensure conditions 2 and 3 of Sec. 3.2: there is no junk in the parameterized sorts and partial operations are defined if and only if their domain condition holds.

Due to space limitation we explain the translation rules (presented in full detail in [1]) using our running example. Fig. 4 shows an excerpt of the Alloy model produced for $SortedSet[TotalOrder]$. Sorts are translated into Alloy signatures. A special signature $start$ with a single instance is defined to represent the root of the graph view of each model instance found by Alloy Analyzer (see Fig. 3), holding fields corresponding to the creators of all sorts (e.g., $empty$). Other operations and predicates are encoded as fields of the signature corresponding to their first argument. To allow this encoding for predicates without further arguments (e.g., $isEmpty$), predicates are handled as operations of return type $Boolean$. Partial operations (e.g., $largest$) originate fields with $lone$ multiplicity (0 or 1) and a fact encoding their (strong) domain condition. To exclude junk for a sort s, a fact is introduced (e.g., $SortedSetConstruction$ fact in Fig. 4) imposing that all its instances are generated by applying constructors (a creator followed by other constructors). When constructors have extra arguments that also have to be constructed, it is necessary to ensure that all instances can be constructed in an acyclic way, e.g., by imposing in the construction fact that, in each step, it is possible to construct an instance y by using only instances x_1, ..., x_n that precede y in a partial ordering (to be found by Alloy Analyzer) of all instances. Axioms are straightforwardly encoded as Alloy facts (e.g., $axiomSortedSet4$ in Fig. 4).

Generation of model finding commands. In order to find a model instance that satisfies each minterm of the FDNF of each axiom in $Body{-}Param$, a "run" command that encodes condition 4 of Sec. 3.2 is generated. This is illustrated in the bottom of Fig. 4 for the same axiom and minterm used in the example of Sec. 3.2. The exploration

bounds can be configured by the user. In the example of Fig. 4, we are searching for models with at most 6 instances of each signature and exactly 2 instances of $Orderable$.

Extraction of abstract tests from the model instances found. When a "run" command is executed, each model instance found by Alloy Analyzer can be visualized as a graph as illustrated in Fig. 3. From this instance an abstract test can be extracted as partially explained in Sec. 3.2. The canonical term to be assigned to each variable x in X_B (e.g., S in Fig. 3) is obtained by following a path from the $start$ node to the node assigned to that variable (e.g., $SortedSet3$). When constructors have extra parameters that have also to be constructed, only paths obeying the partial ordering of all instances imposed by the construction fact are considered. In the example, the extracted Alloy expression is $start.empty.insert[Ord0]$. Since a canonical term cannot contain elements of carrier sets, values of parameter sorts ($Ord0$ in this case) are replaced by variables in the expression and their values are recorded in an assignment (ρ_P).

Prop. 1 ensures the correctness of the test generation technique in abstract terms. Obviously, the preservation of this correctness result depends on how specifications are encoded into Alloy. Concretely, it is necessary to ensure that all model instances of the generated model, restricted to the elements of $Param$, define a $Param$-algebra. For the encoding technique presented in this section, all model instances of the generated Alloy model define a $Body$-algebra.

4 From Abstract Tests to JUnit Tests

In this work, we focus on Java implementations of ADTs. Hence, we consider implementations of parameterized specifications to be sets of Java classes and interfaces, some of them defining generic types. The challenge we address in this section is the translation of the abstract tests generated for the parameterized specification with the help of Alloy Analyzer to concrete JUnit tests. The goal of these tests is to exercise the IUT, instantiating its parameters with mock classes and mock objects derived from the abstract tests.

The translation of abstract into concrete tests requires that a correspondence between what is specified algebraically and what is programmed is defined. We assume this correspondence is defined in terms of a refinement mapping. This notion, defined in the context of CONGU specifications in [15], is formulated in a more general setting.

4.1 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*. We will restrict our attention to the set of specifications described in Sec. 2.1. Hence, in the rest of this section we will consider a parameterized specification $PSpec$ with $Body$ defined by $\mathcal{B} = Spec_{s_1}+...+Spec_{s_n}+Spec_p+Spec_{t_1}+...+Spec_{t_k}$ in which $Spec_p$ corresponds to the parameter specification. Moreover, to ease the presentation, we will consider that $Spec_p$ has a single sort and all increments $Spec_{t_i}$ depend on $Spec_p$ (i.e., they cannot be moved to a position on the left of $Spec_p$). For the same reason, we also consider only Java generic types with a single parameter.

Definition 3. *A* refinement mapping *from $\mathcal{B}$ to a set $\mathcal{C}$ of Java types consists of a type variable V and an injective refinement function $\mathcal{R}$ that maps:*

- *each s_i to a non-generic type defined by a Java class in $\mathcal{C}$;*
- *each t_i to a generic type with a single parameter, defined by a Java class in $\mathcal{C}$;*
- *p to the type variable V;*
- *each operation/predicate of $Spec_s$, with $s \in \{s_1,...,s_n,t_1,...,t_k\}$, to a method of the corresponding Java type $\mathcal{R}(s)$ with a matching signature:* ***(i)*** *every n-ary creator corresponds to an n-ary constructor;* ***(ii)*** *every other (n+1)-ary operation/predicate symbol corresponds to an n-ary method (object* **this** *corresponds to the first parameter of the operation/predicate);* ***(iii)*** *every predicate symbol corresponds to a boolean method;* ***(iv)*** *every operation with result sort s corresponds to a method with any return type, void included, and every operation with a result sort different from s corresponds to a method with the corresponding return type;* ***(v)*** *the i-th parameter of the method that corresponds to an operation/predicate symbol has the type corresponding to its (i+1)-th parameter sort;*
- *each operation/predicate of $Spec_p$ to a matching method signature such that, for $1 \leq i \leq k$, we can ensure that any type K that can be used to instantiate the parameter of the generic type $\mathcal{R}(t_i)$ possesses all methods defined by $\mathcal{R}$ for type variable V after appropriate renaming — the replacement of all instances of V by K.*

Fig. 1 partially shows a refinement mapping from $SortedSet[TotalOrder]$ to the Java types {TreeSet<E>,IOrderable<E>}, using CONGU refinement language. We can check if the last condition above holds by inspecting whether any bounds are declared in the class TreeSet 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 TreeSet is bounded to extend IOrderable<E>, which, in turn, declares the method **boolean** greaterEq(E e).

4.2 Mock Classes and JUnit Tests

In order to test generic classes against their specifications, finite mock implementations of their parameters are automatically generated, comprising *mock classes*, that are independent of the generated abstract tests, and *mock objects*, instances of mock classes that are created and set up in each test method according to a specific abstract test.

```
public class OrderableMock implements IOrderable<OrderableMock> {
      private HashMap<OrderableMock, Boolean> greaterEqMap =
            new HashMap<OrderableMock, Boolean>();
      public boolean greaterEq(OrderableMock o) {return greaterEqMap.get(o);}
      public void add_greaterEq(OrderableMock o, boolean result) {
            greaterEqMap.put(o, result);
      }
}
```

Fig. 5. Mock class generated from the refinement mapping in Fig.1.

Mock classes. For the parameter sort p, a *mock class* named pMock is generated. This class will be used to instantiate the parameter of all generic types $\mathcal{R}(t_i)$ and, hence, has to implement all the interfaces that bound these parameters. For instance, in our example, the class OrderableMock was generated (see Fig. 5) implementing IOrderable<OrderableMock> because the parameter E of TreeSet is bounded to extend IOrderable<E>. The mock class defines extensional implementations of all interface

methods that correspond to operations or predicates of the parameter specification (in our example, just the method greaterEq). More concretely, for each interface method m, the mock class provides: a hash map mMap, to store the method return values for allowed actual parameters; an add_m method, to be used by the test setup code to define the above return values; and an implementation of m itself, that simply retrieves the value previously stored in the hash map.

JUnit tests: axiom tester method. Each axiom $\forall x_1{:}s_1 ... \forall x_n{:}s_n \ . \ \phi$ in $Body-Param$ not defining a domain condition is encoded as a method (reused by all test methods generated for that axiom) with the axiom variables as parameters and a body that evaluates and checks the value of ϕ for the given parameter values (see axiomSortedSet4Tester in Fig. 6). In the case of a variable x_k of a parameterized sort s_k, since operations of s_k may be mapped to methods with side effects (see the case of insert in Figs. 1 and 6), a factory object (of type Factory$<s_k>$) is expected as parameter instead of an object of type s_k, to allow the creation of as many copies as needed of x_k (a copy for each occurrence of x_k in ϕ) without depending on the implementation of clone. This way, methods with side effects can be invoked on one copy without affecting the other copies. Sub-expressions involving operations mapped to **void** methods are evaluated in separate instructions (see insert in Fig. 6). Equality is evaluated with the equals method.

```
private interface Factory<T> {T create();}
private void axiomSortedSet4Tester(Factory<TreeSet<OrderableMock>> sFact,OrderableMock e) {
      TreeSet<OrderableMock> s_0 = sFact.create();
      TreeSet<OrderableMock> s_1 = sFact.create();
      if(!s_0.isEmpty() && !e.greaterEq(s_0.largest())) {
            s_1.insert(e);
            assertTrue(s_1.largest().equals(s_0.largest()));
      }
}
@Test public void test0_axiomSortedSet4_0(){
      // mock objects for the parameter
      final OrderableMock ord0 = new OrderableMock();
      final OrderableMock ord1 = new OrderableMock();
      ord0.add_greaterEq(ord0, true);
      ord0.add_greaterEq(ord1, false);
      ord1.add_greaterEq(ord0, true);
      ord1.add_greaterEq(ord1, true);
      // factory objects for the axiom var's of parameterized type
      Factory<TreeSet<OrderableMock>> sFact =
         new Factory<TreeSet<OrderableMock>>() {
            public TreeSet<OrderableMock> create() {
                  TreeSet<OrderableMock> s = new TreeSet<OrderableMock>();
                  s.insert(ord0);
                  return s; }
      };
      // checking the axiom
      axiomSortedSet4Tester(sFac, ord1);
}//… other axioms and test cases
```

Fig. 6. Excerpt of JUnit test code generated corresponding to the model instance shown in Fig. 3.

JUnit tests: test methods encoding abstract tests. For each abstract test

$$\langle A, X, \wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x)) \Rightarrow \phi, \rho_P, \rho_B \rangle$$

generated according to the technique described in Sec. 3.2, a concrete JUnit test method is generated comprising three parts (for an example, see Fig. 6):

- **Mock objects:** Creation of mock objects (instances of mock classes) for the values in the carrier set of A, and addition of tuples for the functions and relations in A.
- **Factory objects:** Creation of a factory object of type Factory$<s>$ for each variable $x : s$ in X_B, that constructs an object of type s upon request according to the term

$\rho_B(x)$ and the mapping ρ_P. The verification of the condition $\mathit{defined}^*(\rho_B(x))$ is performed incrementally in each step of the construction sequence, by checking the domain condition before applying any operation with a defined domain condition and issuing a warning in case it does not hold (not needed in the example).
– **Axiom verification**: Invocation of the method that checks ϕ, passing as actual parameters the factory objects prepared in the previous step (for the variables in X_B) and the values defined in ρ_P (for the remaining variables).

5 Evaluation

To assess the efficacy (defect detection capability of the generated test cases) and efficiency (time spent) of the proposed technique, an experiment was conducted using different specifications. Herein, we report on the results of the experiment with our running example. We started by generating abstract test cases for the specification $SortedSet$. We measured the time spent by Alloy Analyzer on finding model instances for the several run commands (axiom cases) and the number of run commands for which instances were found. For the ones that Alloy Analyzer could not find instances, a manual analysis was conducted to determine whether they could be satisfied with other search settings (exploration bounds). After that, JUnit test cases generated from abstract tests and the refinement mapping to TreeSet and IOrderable were executed to check the correctness of the implementation and of the test suite. Subsequently, a mutation analysis was performed to assess the quality of the test suite. Mutants not killed by the test suite were manually inspected to determine if they were equivalent to the original code, and additional test cases were added to kill the non-equivalent ones. A test coverage analyses was also performed as a complementary test quality assessment technique. The experiment was conducted on a portable computer with a 32 bits Intel Core 2 Duo T6600 @ 2.20 GHz processor with 2.97 GB of RAM, running Microsoft's Windows 7. The results are summarized in Table 1.

In terms of efficiency, we concluded that the time spent in finding model instances ($\sim$ 2 minutes) is not a barrier for the adoption of the proposed approach. The percentage of axiom cases for which a model instance was not found was significant (44%). A manual analysis showed that these cases were not satisfiable. Mutation analysis revealed some parts of the implementation of *equals* and *largest* that were not adequately exercised, due to the fact that conditions for inequality are not explicitly specified and consequently not tested in this example, and due to the fact that the behaviour of operations outside their domain (in the example, the behaviour of *largest* over an empty set) is not specified and consequently not tested.

6 Conclusions and Future Work

Although test generation from algebraic specifications has been thoroughly investigated, existing approaches are based on flat specifications. In this paper, we have discussed testing from parameterized specifications and put forward a notion of a closed test appropriate for these specifications, which generalises the standard notion of test as a quantifier-free ground formula. Then, based on closed tests, we presented an approach

Table 1. Experimental results for the SortedSet example

Item	Sorted Set
Size of algebraic specification ($Body - Param$)	25 lines [(1)]
Total number of axioms	9
With instances found in all axiom cases	5
With instances found in some axiom cases	4
Total number of axiom cases (minterms)	36
Number of cases for which instances were found [(2)]	20 (56%)
Number of cases for which no instances were found [(2)(3)]	16 (44%)
Time spent by Alloy analyzer finding instances [(2)]	129 sec
Number of JUnit test cases generated [(4)]	20
Size of Java implementation under test	77 lines [(1)]
Number of failed test cases	0
Total number of mutants generated (with Jumble [12])	41
Killed by the original test suite	35 (85%)
Not killed by the original test suite	6 (15%)
Equivalent to original implementation	0
Not equivalent to original implementation [(5)]	6
Coverage of Java implementation under test (measured with Eclemma [10])	96,9%
Number of added test cases to kill all mutants (and achieve 100% code coverage)	3

[(1)] Ignoring comments and blank lines.[(2)] In this experiment, the exploration was limited to at most 12 instances per sort, but exactly 3 $Orderable$. [(3)] Manual analysis showed that these cases were not satisfiable. [(4)] Only one test case was generated for each satisfiable axiom case (corresponding to the first instance retrieved by Alloy Analyzer). [(5)] Related to method invocation outside the domain and to insufficient testing of $equals$ (lack of inequality cases).

for the generation of unit tests for Java implementations of generic ADTs from specifications in which the generated test code includes finite implementations (mocks) of the parameters. This paper addresses the foundational aspects of the approach. A tool that fully automates the test generation from specifications is currently under development. The tool supports the translation of CONGU specifications to Alloy, the translation of model instances found by Alloy Analyzer to JUnit as well as the tuning of exploitation bounds. We envisage the tool can also provide support for other related problems, like the automatic generation of actual parameters for methods when the type parameters are interfaces (e.g., comparator in TreeSet<E>(Comparator<E> comparator)).

The proposed approach relies on a translation of specifications into Alloy and on the capability of Alloy Analyzer to find model instances that satisfy given properties—in our case, the minterms of the FDNF representation of each axiom. In the conducted experiments, Alloy Analyzer was able to find model instances for all theoretically satisfiable axiom cases in a moderate time. Mutation testing and code coverage analyses showed that the generated test cases were of high quality, because they were able to kill all the mutants and cover all the code apart from behaviours that were not explicitly specified (behaviour outside operation domains and conditions for inequality).

Although Alloy Analyzer has scalability limitations due to the time required to find instances of complex models, we did not find that to be an issue for unit testing ADTs. The fact that Alloy Analyzer only performs model-finding over restricted scopes consisting of a user-defined finite number of objects is what imposes a limitation of the approach presented: the inability to generate tests for ADTs that do not admit finite

models, such as unbounded stacks (since the domain of push is true, it is always possible to create a bigger stack). To overcome that problem, we are currently working on an extension of the approach to automatically handle that kind of specifications, that encompasses transforming constructors into partial functions in the Alloy model and inserting definedeness guard conditions in the axioms that use those constructors.

As future work, we also intend to extend the approach to rule out automatically by static analysis unsatisfiable axiom cases and generate tests outside operations' domains and for properties not explicitly included in specifications such as those related with the fact that equality is a congruence. This will reduce the dependence of the approach on the correct implementation of equals.

Acknowledgement

This work was partially supported by FCT under contract PTDC/EIA-EIA/103103/2008.

References

1. Andrade, F., Faria, J.P., Lopes, A., Paiva, A.: Specification-driven unit test generation for Java generic classes (2011), http://paginas.fe.up.pt/∼jpf/research/TR-QUEST-2011-01.pdf
2. Bidoit, M., Mosses, P.: CASL User Manual, LNCS, vol. 2900. Springer (2004)
3. Chen, H.Y., Tse, T.H., Chen, T.Y.: TACCLE: a methodology for object-oriented software testing at the class and cluster levels. ACM Trans. Softw. Eng. Methodol. 10, 56–109 (2001)
4. Crispim, P., Lopes, A., Vasconcelos, V.T.: Runtime verification for generic classes with ConGu2. In: Proceedings of SBMF'10: foundations and applications. LNCS, vol. 6527, pp. 33–48. Springer-Verlag (2011)
5. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. 3, 101–130 (1994)
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations und Initial Semantics, Monographs in Theoretical Computer Science (EATCS), vol. 6. Springer (1985)
7. Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. In: Proceedings of the 12th POPL. pp. 52–66. ACM, New York, NY, USA (1985)
8. Gaudel, M.C., Le Gall, P.: Testing data types implementations from algebraic specifications. In: Formal methods and testing. LNCS, vol. 4949, pp. 209–239. Springer-Verlag (2008)
9. Hein, J.L.: Discrete Structures, Logic, and Computability. Jones & Bartlett Publishers (2009)
10. Hoffmann, M.R.: Ecclema: Java code coverage tool for Eclipse, http://www.eclemma.org/
11. Huges, M., Stotts, D.: Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In: Proc. ISSTV. pp. 53–61. ACM (1996)
12. Irvine, S.A., Pavlinic, T., Trigg, L., Cleary, J.G., Inglis, S., Utting, M.: Jumble Java byte code to measure the effectiveness of unit tests, http://jumble.sourceforge.net/ (2007)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
14. Mackinnon, T., Freeman, S., Craig, P.: Endotesting: Unit testing with mock objects. In: eXtreme Programming and Flexible Processes in Software Engineering – XP2000 (2000)
15. Nunes, I., Lopes, A., Vasconcelos, V.T.: Bridging the gap between algebraic specification and object-oriented generic programming. In: Runtime Verification. LNCS, vol. 5779, pp. 115–131. Springer-Verlag (2009)
16. The Stanford Natural Language Processing Group: http://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/util/package-tree.html
17. Yu, B., King, L., Zhu, H., Zhou, B.: Testing Java components based on algebraic specifications. In: Proc. International Conference on Software Testing, Verification and Validation. pp. 190–198. IEEE (2008)