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] Departamento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto
[3] INESC Porto
mal@di.fc.ul.pt, {francisco.andrade,jpf,apaiva}@fe.up.pt

Technical Report TR-QUEST-2011-01

Work supported by FCT under contract PTDC/EIA/103103/2008

Abstract. Algebraic specifications have been successfully used for the formal specification of ADTs, and several approaches exist to automatically derive test cases that check the conformance of the implementation of ADTs with respect to their specification. However, existing approaches lack support for the testing of implementations of ADTs defined by generic classes. In this technical report, we present a novel technique to automatically derive unit test cases for Java generic classes that, in addition to the usual testing data, encompass implementations for the type parameters of the classes. The proposed technique relies on the use of a model finder tool — Alloy Analyzer, which is used 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 (eg. [4,2,17,11,8]). 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.

Many data types (e.g., *set*) admit slightly different versions in different applications (e.g., *sets of strings, dates, numbers*). Nowadays, the implementation of these data types in mainstream object-oriented languages, s.a. Java and C#, strongly relies on generic classes (e.g., ArraySet<E>). Existent 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 but this raises the following problems: (i) in the case of non trivial parameters, types for instantiating the parameters may not be available at test time; (ii) the types available for instantiating the parameters may not

cover all the possibilities allowed by the parameters' specifications (e.g., using an existing type with a total ordering to test a PartiallyOrderedSet<E>); (iii) 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 or fake objects for their parameters. This also enables a higher degree of test automation, because the user needs not to indicate the types to use for instantiating the parameters of the generic type.

In this technical report we address the generation of test cases for implementations of ADTs defined in terms of Java generic classes, that comprise automatically generated mock classes and mock objects that can be used to instantiate the generic classes' 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, as defined for instance in [6], are supported by several specification languages. In contrast, refinement mappings were defined in [16] for the specifications supported by CONGU, a tool-based approach to runtime conformance checking of Java programs against algebraic specifications [15,3]. Herein, we revisit this notion and reformulate it in a more general setting.

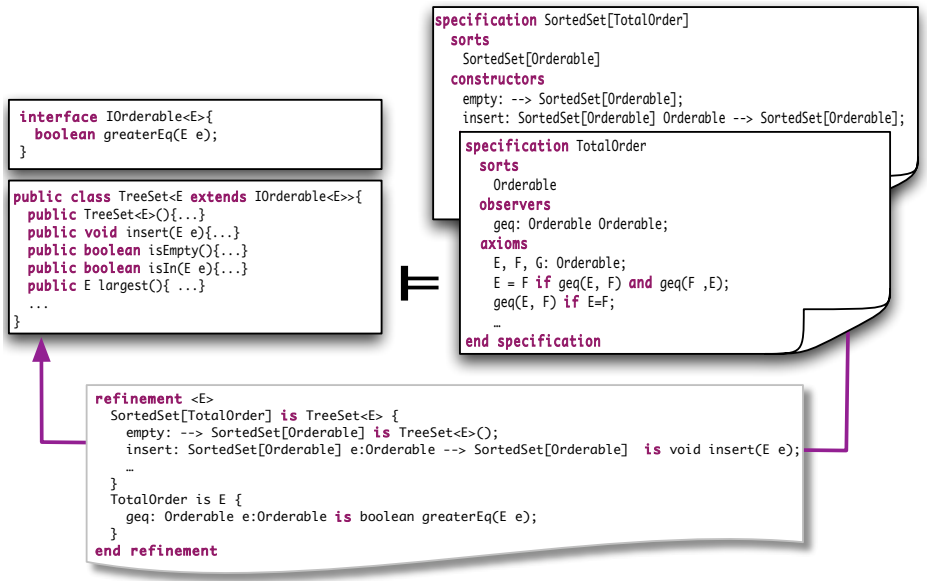

Fig. 1. The aim is the generation of unit tests for checking if a set of generic classes correctly implements an ADT.

Following the tradition of specification-based testing, the technique we propose 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 fixing an evaluation for some variables into a specific model of the parameter specification. For the generation of this type 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 sorts and Java types and between operations and methods defined by the given refinement mapping between the specification and its implementation.

The organisation of the report 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, Sec. 6 presents additional examples to illustrate some features and applicability of the approach proposed, and Sec. 7 concludes the report and discusses future work.

2 Specifications of Generic Data Types

In algebraic specification, the description of ADTs that admit different versions – which we designate by generic data types – 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., [6,1,7]) with significant variations in terms of syntax and semantics. In this section, we discuss the specifications considered in our approach and their semantics.

Table 1 presents a summary of some preliminary notation and definitions.

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:

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.

Table 1. Summary of preliminary notation and definitions

Notation	Description	Example
$\Sigma = \langle S, F, P \rangle$	Σ - many-sorted signature S - set of sorts F - set of operation symbols P - set of predicate symbols	$S = \{Orderable\}$ $F = \{\}$ $P = \{geq\}$
$Spec = \langle \Sigma, Ax \rangle$	$Spec$ - specification Ax - set of axioms described by formulas in first-order logic (with equality)	$TotalOrder$ (see Fig. 1) $Ax = \{\forall e, f : Orderable.geq(e, f) \wedge geq(f, e) \Rightarrow e = f, ...\}$
$PSpec = \langle Param, Body \rangle$	$PSpec$ - parameterized spec $Param$ - formal parameter spec $Body$ - body spec (must include $Param$)	$SortedSet[TotalOrder]$ (see Fig. 2) $Param = TotalOrder$ $Body = SortedSet.$
$Body - Param$	Sorts, operations, predicates and axioms in $Body$ but not in $Param$	$\{SortedSet[Orderable]\}$, $\{empty, insert, largest\}$, $\{isEmpty, isIn\}$, {several axioms}
$Spec + Spec_s$	The extension of $Spec$ with an increment centred on sort s	
$Spec_{s_1} + ... + Spec_{s_n}$	Sequence of increments.	
$Term_\Sigma$	Set of ground terms built over Σ	
$CTerm_\Sigma$	Set of canonical ground terms built over Σ (i.e., terms defined exclusively in terms of constructors)	
$Term_\Sigma(X)$	Terms built over a set X of variables typed by sorts in Σ	
$CTerm_\Sigma(X)$	Canonical terms built over a set X of variables typed by sorts in Σ	
$Form_\Sigma(X)$	Formulas built over a set X of variables typed by sorts in Σ	
$A = \langle \{A_s\}_{s \in S}, \mathcal{F}, \mathcal{P} \rangle$	A - Σ-algebra A_s - carrier set of sort s $\mathcal{F}$ - interpretation of operation symbols as partial functions $\mathcal{P}$ - interpretation of predicate symbols as relations	
$[\![t]\!]^A$	Interpretation of a ground term t in a Σ-algebra A	
ρ	Assignment of X into A, i.e., a function that assigns a value in A_s to each variable $x{:}s$ in X.	
$[\![t]\!]^{A,\rho}$	Interpretation of a term t in $Term_\Sigma(X)$ with an assignment ρ of X into A.	

```
specification SortedSet[TotalOrder]
  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));
    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));
    ...
end specification
```

Fig. 2. Specification of a sorted set in CONGU.

These elements define *an increment*, which can define a specification by itself or rely on sorts, operations and predicates available in the base specification $Spec$ (see the notation for representing increments and sequences of increments in Table 1). For the body of a parameterized specification $PSpec$, it is also required 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.

It is not difficult to confirm that specifications $TotalOrder$ and $SortedSet$ fulfil 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 what concerns the axioms, we assume they have one of the following forms:

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

where $op : s_1, ..., s_n \rightarrow s$ 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 while the second type of axioms supports the definition of a domain condition of an operation (also called definedeness condition), i.e., the condition under which the operation must be defined. Domain conditions 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: $\forall s : SortedSet[Orderable].\, \neg isEmpty(s) \Rightarrow \mathit{defined}(largest(s))$.

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

Definition 1. *$defined^*(t)$ is the formula defined inductively in the structure of term t as follows:*

1. *$defined^*(x) = true$ if x is a variable,*
2. *$defined^*(op(t_1, ..., t_n)) = defined^*(t_1) \wedge \ldots \wedge defined^*(t_n) \wedge \psi_{op}[t_1/x_1, \ldots t_n/x_n]$ if $op : s_1, ..., s_n \rightarrow s$ is an operation with domain condition $\forall x_1 : s_1 \ldots \forall x_n : s_n.\ \psi_{op} \Rightarrow defined(op(x_1, ..., x_n))$.*

2.2 Semantics

Specifications are interpreted in terms of Σ-algebras (see definition and notation of Σ-algebras in Table 1). Given that operation symbols can be interpreted by partial functions, the interpretation of a term $[\![t]\!]^{A,\rho}$ might not be defined. In fact, $[\![t]\!]^{A,\rho}$ is defined if and only if $A, \rho \vDash defined(t)$. 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 (as defined in [1]).

There are various forms of semantic construction in the algebraic approach to specification of ADTs [5]. For the purpose at hand, the appropriate construction is loose semantics. This construction associates to a specification $Spec = \langle \Sigma, Ax \rangle$ the class of all Σ-algebras which satisfy its set of 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.

Proceeding by induction on the structure of terms it is easy to prove that the formula $defined^*(t)$ indeed defines sufficient conditions for the term t to be defined:

Proposition 1. *For every Spec-algebra A and term $t \in Term_\Sigma(X)$,*

$$A, \rho \vDash defined^*(t) \Rightarrow defined(t)$$

In what concerns parameterized specifications, loose semantics associates to $PSpec = \langle Param, Body \rangle$ the class of functions $\mathcal{T}_{Body}$ which assign to each $Param$-algebra A, a $Body$-algebra $\mathcal{T}_{Body}(A)$ that coincides with A when restricted to $Param$. In this way, 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

As mentioned before, 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 (Σ-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 idea can be easily generalised to parameterized specifications but the result is not particularly interesting as specifications used as parameters are not expected to have creators and, in this situation, 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, for every* $\mathcal{T}_{Body}$ *in the semantics of PSpec,*

$$\mathcal{T}_{Body}(A), \rho_P \vDash \rho^*_B(\phi)$$

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

1. a syntactic replacement of variables in X_B by terms and
2. 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 particularly interested in the generation of tests that result from the instantiation of axioms of the form $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n \;.\; \phi$. Because closed tests may involve the replacement of variables by terms and the interpretation of these terms in a specific Σ_{Body}-algebra might be undefined, this instantiation needs to be conditioned by the definedeness of these terms. Because the formula $defined^*(t)$ provides a sufficient condition for the term t to be defined, we can use the formula $\wedge_{x \in X_B} defined^*(\rho_B(x)) \Rightarrow \phi$. The proposition below expresses this idea.

Proposition 2. *Let* $PSpec = \langle Param, Body \rangle$ *be a parameterized specification and* $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n \;.\; \phi$ *an axiom in* $Body$*. If* A *is a* $Param$*-algebra,* X *is a set of variables including* $\{x_1{:}s_1, \ldots, x_n{:}s_n\}$*,* 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*

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

is a closed test for PSpec.

Proof. Let $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n \,.\, \phi$ be an axiom in $Body$ and

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

a tuple as defined in the proposition statement. Then, what we need to prove is that, for every $\mathcal{T}_{Body}$ in the semantics of $PSpec$:

$$\mathcal{T}_{Body}(A), \rho_P \vDash \rho_B^*((\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x))) \Rightarrow \phi)$$

Let $M = \mathcal{T}_{Body}(A)$. Then, because the terms $\rho_B(x)$ do not involve variables in X_B and $\mathit{defined}^*(t)$ is a term that only involves variables used in t (see Def. 1), this is equivalent to prove that

$$M, \rho_P \vDash (\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x))) \Rightarrow \phi[\rho_B(x)/x : x \in X_B] \quad (*)$$

Because M is a $Body$-algebra,

$$M \vDash \forall x_1 : s_1 \ldots \forall x_n : s_n .\, \phi$$

Hence, for every assignment ρ of X into M

$$M, \rho \vDash \phi \quad (**)$$

Consider a variable $x \in X$, a term t in $Term_\Sigma(Y)$ with Y disjoint from X. From (**), it follows that, for every assignment ρ^* of $X \cup Y$ into M we also have

$$M, \rho^* \vDash \mathit{defined}(t) \Rightarrow \phi[t/x]$$

This happens because $[\![t]\!]^{M,\rho^*}$ is defined if and only if $M, \rho^* \vDash \mathit{defined}(t)$. By Prop. 1, then we have

$$M, \rho^* \vDash \mathit{defined}^*(t) \Rightarrow \phi[t/x]$$

By applying this reasoning successively to each variable in X_B we can conclude $(*)$. □

As an example, let us consider the axiom $\forall s : SortedSet[Orderable].\, \forall e : Orderable.\, \neg isEmpty(insert(s, e))$ of $SortedSet$. As a result of Prop. 2, we have that

- 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 $SortedSet[TotalOrder]$. According to Def. 1, $\mathit{defined}^*(empty())$ is the domain condition of $empty()$, which is is $true$.

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 implementation under test. 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, following a partition testing strategy. In general, this amounts to partition each axiom into a finite set of (possibly non-disjoint) "cases", either by successively unfolding the premises of equational axioms (axioms of the form $\psi \rightarrow t_1 = t_2$) 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

$$\forall x_1 : s_1 \ldots \forall x_n : s_n .\ \phi$$

in $Body{-}Param$ that do not express a domain condition and start by converting the axiom to FDNF. Assuming that the result is

$$\forall x_1 : s_1 \ldots \forall x_n : s_n .\ \phi_1 \vee \ldots \vee \phi_k$$

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 (i.e., M_s is finite, for every sort s);
2. M satisfies sort generation constraints for sorts in $Body{-}Param$, i.e., 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 $Body{-}Param$: $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n .\ \psi_{op} \Leftrightarrow \mathit{defined}(op(x_1, ..., x_n))$;
4. M satisfies $\exists x_1 : s_1 \ldots \exists x_n : s_n .\ \phi_i$.

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 that hold in the implementation under test (the other axioms concern properties that are expected to hold in actual parameters). Although domain conditions are not considered directly, they are subsumed in the generated tests for the other axioms because of condition 3.

In what concerns the constraints imposed on the search of the $Body$-algebra: 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 $M|_{Param}$; 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 $X{=}\{x_1{:}s_1, \ldots, x_n{:}s_n\}$ 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 allow us to define the following closed test for $PSpec$:

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

- $M|_{Param}$ 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 correction of the proposed technique is an immediate result of Prop. 2.

Proposition 3. *The tuple $\langle M|_{Param}, X', \phi', \rho_P, \rho_B\rangle$ is a closed test for PSpec.*

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

$$\forall s : SortedSet[Orderable].\forall e : Orderable.\, isEmpty(s) \Rightarrow largest(insert(s, e)) = e$$

One minterm of the corresponding FDNF is $\neg isEmpty(s){\wedge}\neg largest(insert(s, e)){=}e$ and, hence, the application of the technique just described would involve to use Alloy Analyzer to obtain a $SortedSet$-algebra satifying the four conditions described before. Fig. 3 presents an example of one of these algebras (referred as SS^2 in the sequel), represented as an Alloy model instance. In fact, this instance also defines an assignment

$$\rho : \{e : Orderable \mapsto Ord1,\ s : SortedSet[Orderable] \mapsto SortedSet3\}$$

into SS^2. 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

$$\rho_s : \{f : Orderable \mapsto Ord0\}$$

As a result, we obtain the closed test for $SortedSet[TotalOrder]$

$$\langle TO^2, X', \phi', \rho_P, \rho_B\rangle$$

where

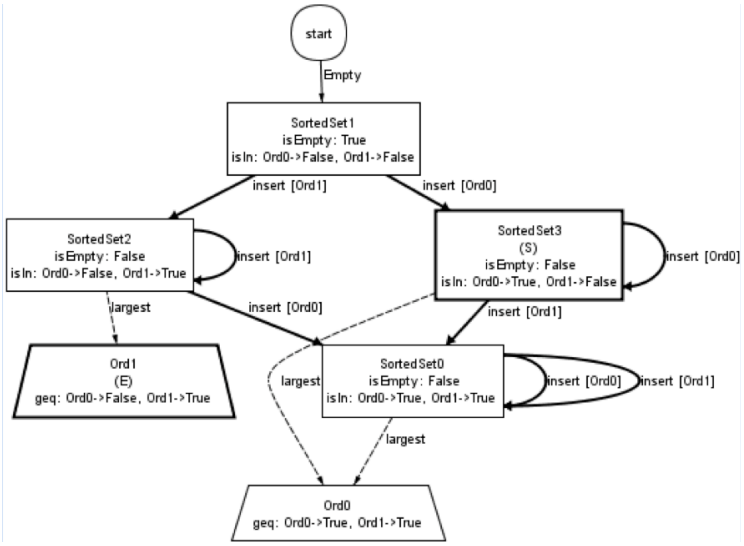

Fig. 3. A model instance defining a $SortedSet$-algebra and an assignment to variables e and s.

- X'=$\{s$:$SortedSet[Orderable], e$:$Orderable\} \cup Y_s$ with $Y_s = \{f$:$Orderable\}$
- ϕ' is $\mathit{defined}^*(insert(empty(), f)) \Rightarrow (isEmpty(s) \Rightarrow largest(insert(s, e))=e)$, with $\mathit{defined}^*(insert(empty(), f)) = true \wedge true \wedge true$
- 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 a minterm of the 4th axiom presented in Fig. 2 that involves $isEmpty(s)$ and the term $largest(s)$.

3.3 From Algebraic Specifications to Alloy and Back

The technique just described requires the ability to generate Alloy models from a parameterized specification $PSpec$=$\langle Param, Body\rangle$, to generate 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, has to ensure conditions 2 and 3 of Sec. 3.2: there is no junk in the parameterized sorts and partial operations are defined iff their domain condition holds.

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

```
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/True implies (S.insert[E].largest = E))
}
// … other axioms of Orderable and SortedSet
run axiomSortedSet4_1 {
    some E : Orderable, S : SortedSet |
        (S.isEmpty ! = BOOLEAN/True and (S.insert[E].largest != E))
} 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 $SortedSet[TotalOrder]$.

by Alloy Analyzer (see Fig. 3), holding fields corresponding to the creators of all sorts (s.a. $empty$). 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 (s.a. $isEmpty$), predicates are handled as operations of return type $Boolean$. Partial operations (s.a. $largest$) 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 (s.a. $SortedSetConstructionFact$ 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 have also to be constructed, it is also 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 (see $axiomSortedSet4$ in Fig. 4). Tables 2, 3 and 4 present a summary of the translation rules.

Table 2. Translation rules from CONGU to Alloy - syntax

Rule	**CONGU**	**Alloy**
R1. Simple sort	**sorts** s	**sig** s
R2. Parameterized sort	s[p]	s *(in sign. decl. and usage)*
R3. Total operation	op: s –>t'	op: **one** t'
(except creator)	op: s t_1 ... t_n –>t'	op: (t_1 ->... ->t_n) ->**one** t'
R4. Total predicate	pr : s	pr: **one** BOOLEAN/Bool
	pr : s t_1 ... t_n	pr: (t_1 ->... ->t_n) ->**one** BOOLEAN/Bool
R5. Partial operation	–>?	***(lone** instead of **one**)*
R6. Start instance	*(not defined)*	**one sig** start
R7. Creator	cr : t_1 ... t_n –>s	cr: (t_1 ->... ->t_n) ->**one** s *(in sig start)*

Table 3. Translation rules from CONGU to Alloy - construction facts

Algebraic specification
Parameterized sort s with creator constructors
$c_i : s_{i1},\ ...,\ s_{ik_i} \rightarrow s\ \ (i = 1,\ ..., n)$
and other constructors
$t_j : s,\ s'_{j1},\ ...,\ s'_{jw_j} \rightarrow s\ \ (j = 1,\ ..., m)$
Alloy construction fact
fact sConstruction { s **in** $(\gamma_1 + \ ... \ + \gamma_n). * \{x : s,\ y : \tau_1 \ + \ ... \ + \tau_m\}$ }
whith
$\gamma_i = start.c_i[s_{i1}]$ ... $[s_{ik_i}]$, if none of the $s_{i1},\ ...,\ s_{ik_i}$ is a constructed type
$\gamma_i = \{x : start.c_i[s_{i1}]$ ... $[s_{ik_i}]$ | **some** $a_1 : s_{i1}, ..., a_{k_i} : s_{ik_i}$ | $x = start.c_i[a_1]...[a_{k_i}]$ **and** precedes$[a_1, x]$ **and** ... **and** precedes$[a_{k_i}, x]\}$, otherwise
and
$\tau_j = x.t_j[s'_{j1}]$ ... $[s'_{jw_j}]$, if none of the $s'_{j1},\ ...,\ s'_{jw_j}$ is a constructed type
$\tau_j = \{y : x.t_j[s'_{j1}]$ ... $[s'_{jw_j}]$ | **some** $a_1 : s'_{j1}, ..., a_{w_j} : s'_{jw_j}$ | $y = x.t_j[a_1]...[a_{w_j}]$ **and** precedes$[x, y]$ **and** precedes$[a_1, y]$ **and** ... **and** precedes$[a_{w_j}, y]\}$, otherwise
where
precedes$[x, y]$ is a predicate that checks if x precedes y in a partial ordering (to be determined by Alloy Analyzer) of all generated instances.

Generation of model finding commands. In order to find a model instance that satisfies each minterm of the FDNF representation of each axiom in $Body-Param$, a "run"

Table 4. Translation rules from CONGU to Alloy - axiom and domain restriction facts

Constraint (CONGU)	Fact (Alloy)
k^{th} **axiom** in sort s:	**fact** axioms_k {
$v_1 : s_1;\ ... ; v_n : s_n;$	**all** $v_1 : s_1, ..., v_n : s_n \mid$
$formula(v_1,\ ..., v_n);$	$formula'(v_1,\ ..., v_n)$
	}
k^{th} **domain** restriction in sort s:	**fact** domains_k {
$v_1 : s_1;\ ... ; v_n : s_n;$	**all** $v_1 : s_1, ..., v_n : s_n \mid$
$op(v_1,\ ..., v_n)$ **if**	$cond'(v_1,\ ..., v_n)$
$cond(v_1,\ ..., v_n);$	**implies one** $op'(v_1, ..., v_n)$ **else no** $op'(v_1, ..., v_n)$
	}

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 manually 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 (s.a. S:$SortedSet$ in Fig. 3) is obtained by following a path from the $start$ node to the node assigned to that variable (s.a. $SortedSet3$ in Fig. 3). 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 this example, the extracted Alloy expression would be $start.empty.insert[Ord0]$. Since, by definition, a canonical term can contain variables but not elements of carrier sets, values of parameter sorts ($Ord0$ in this case) are replaced by variables in the expression of the canonical term, and their values are recorded in a separate function (ρ_P).

Prop. 2 ensures the correction 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, we have in fact a stronger result: all model instances of the generated Alloy model define a $Body$-algebra (the proof of this result is out of the scope of this report).

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 implementation under test, 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 [16], 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

$$\mathcal{B} = Spec_{s_1} + \ldots + Spec_{s_n} + Spec_p + Spec_{t_1} + \ldots + Spec_{t_k}$$

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[4] 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 $s \in \{s_1, \ldots, s_n, t_1, \ldots, t_k\}$, 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 and such that, for every $1 \leq i \leq k$, 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 the type variable V by K.*

[4] Although this type variable is not strictly needed here because we are only considering a single parameter sort, it helps to make the bridge with what can be defined in CONGU refinement language used in the example, which supports not only several parameters but also subtype relations between them.

```
import  java.util.HashMap;

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 Figure 1.

Fig. 1 partially shows an example of 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 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 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.

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 $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n \,.\, \phi$ in $Body{-}Param$ that does not express a domain condition is encoded as a method (to be reused by all test methods generated for that axiom) with the axiom variables as parameters and a body that evaluates and checks (with assertTrue) 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.

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 — *Sorted Set*.

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

```
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()) {
            s_1.insert(e);
            assertTrue(s_1.largest().equals(e));
      }
}
@Test public void test0_axiomSortedSet4_1(){
      // 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.

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 5.

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 that were not adequately exercised, due to the use of an partially unspecified and manually crafted implementation of *equals*, 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 Additional Examples

This section presents additional examples to illustrate the features and applicability of the proposed technique.

6.1 Example with Less Trivial Parameters – POSet

In the example of the generic type used throughout the report ($SortedSet$), one could argue that a standard Java type with a natural ordering, like String or a wrapper class,

Table 5. Experimental results for the SortedSet example

Item	Sorted Set
Size of algebraic specification ($Body - Param$)	25 lines [(1)]
Total number of axioms	9 (5/4)
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 $equals$.

could be used to instantiate the parameter of the generic type and adequately test the generic type (according to the FDNF criteria), with no need for generating mock objects. To illustrate a situation where such an approach would not yield adequate testing, we present here a slightly different generic type - a partially ordered set ($POSet$).

Fig. 7 presents the specification of this data type. The main differences with respect to the $SortedSet$ example are the removal of the totality axiom in $POrderable$ when compared with $Orderable$, and the replacement of the $largest$ operation in $SortedSet$ by the $maximal$ operation in $POSet$, returning the set of maximal elements in the $POSet$. An element x is a maximal element of a $POSet$ s if s does not contain any other element y greater than x.

Fig. 8 presents an excerpt of the corresponding encoding in Alloy, including a run command that generates the model instance shown in Fig. 9. If a standard Java type with a natural ordering, like String or a wrapper class, was used directly to instantiate the parameter of $POSet$, this run command would not be satisfiable, because all pairs of elements would be comparable. By relying on Alloy Analyzer for finding an appropriate instantiation for the parameter of the generic type under test, such a problem does not arise, assuring adequate testing of the type under test.

Another example of a generic class with non trivial parameters is the generic class

```
specification PartialOrder
 sorts
   POrderable // Partially Orderable
 observers
   geq: POrderable POrderable;
 axioms
   E, F, G: POrderable;
   E = F if geq(E,F) and geq(F,E);
   geq(E,E);
   geq(E,G) if geq(E,F) and geq(F,G);
end specification

specification POSet[PartialOrder]
 sorts
   POSet[POrderable] // Partially Ordered Set
 constructors
   empty: --> POSet[POrderable];
   insert: POSet[POrderable] POrderable --> POSet[POrderable];
 observers
   isEmpty: POSet[POrderable];
   isIn: POSet[POrderable] POrderable;
   maximal: POSet[POrderable] --> POSet[POrderable];
   hasComparable: POSet[POrderable] POrderable; // auxiliary
 axioms
   E, F: POrderable;
   S: POSet[POrderable];
   isEmpty(empty());
   not isEmpty(insert(S,E));
   not isIn(empty(),E);
   isIn(insert(S,E),F) iff E = F or isIn(S,F);
   insert(insert(S,E),E) = insert(S, E);
   insert(insert(S,E),F) = insert(insert(S,F),E);
   maximal(empty()) = empty();
   maximal(insert(S,E)) = insert(S,E) if isEmpty(S);
   maximal(insert(insert(S,E),F))) = maximal(insert(S,F)) if geq(F,E);
   maximal(insert(insert(S,E),F))) = maximal(insert(S,E)) if geq(E,F);
   maximal(insert(insert(S,E),F))) = insert(maximal(insert(S,E)),F)
     if not hasComparable(insert(S,E),F);
   not hasComparable(empty(),E);
   hasComparable(insert(S,E),F)
     iff geq(E,F) or geq(F,E) or hasComparable(S,F);
end specification
```

Fig. 7. Specification of a partially ordered set in CONGU.

```
open util/boolean as BOOLEAN

one sig start {
   empty: one POSet
}

sig POrderable {
   geq: POrderable -> one BOOLEAN/Bool
}
// … encoding of axioms of POrderable omitted

sig POSet {
   insert: POrderable -> one POSet,
   isEmpty: one BOOLEAN/Bool,
   isIn: POrderable ->one BOOLEAN/Bool,
   maximal: one POSet,
   hasComparable: POrderable -> one BOOLEAN/Bool
}
// … encoding of other axioms and construction fact of POSet omitted
fact axiomPOSet10 {
   all E, F: POrderable, S: POSet |
      S.insert[E].hasComparable[F] = BOOLEAN/False  implies
         S.insert[E].insert[F].maximal = S.insert[E].maximal.insert[F]
}
// … other run commands omitted
run run_axiomPOSet10_1{
   some E, F: POrderable, S: POSet |
      S.insert[E].hasComparable[F] = BOOLEAN/False and
         S.insert[E].insert[F].maximal  = S.insert[E].maximal.insert[F]
} for 4
```

Fig. 8. Excerpt of the encoding in Alloy of the specification of the partially ordered set.

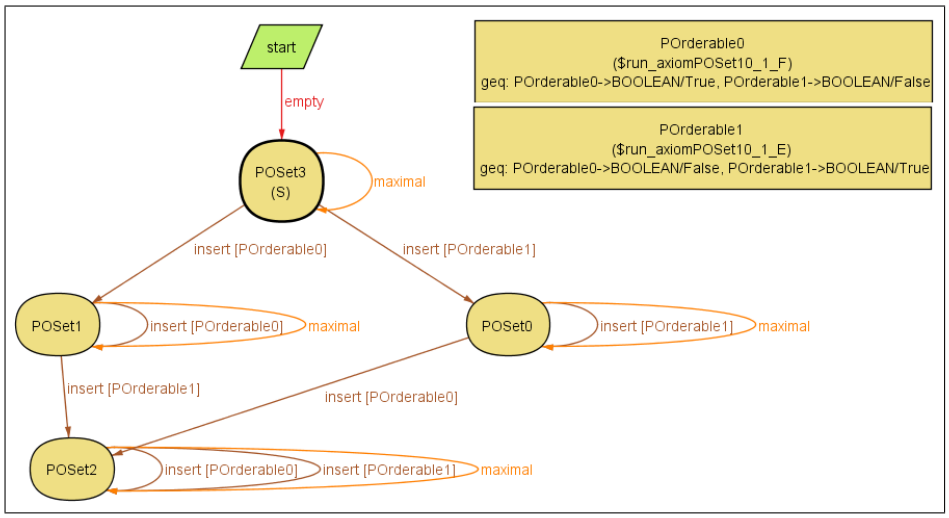

Fig. 9. Model instance generated by the run command in Fig. 8.

class IntervalTree<E **extends** Comparable<E>,T **extends** HasInterval<E>>

available in the Stanford JavaNLP API (http://nlp.stanford.edu/nlp/javadoc/javanlp/). It admits as second parameter a type that extends the interface

interface HasInterval<E **extends** Comparable<E>>

which defines a method

Interval <E> getInterval()

No implementation of this interface, nor of Interval <E>, is readily available in standard Java types nor in the framework.

6.2 Example with Underspecified Constructors – Set

The next example illustrates the need, in the general case, to check the definedeness of the terms used to replace variables in axiom expressions (see Sec. 3.1), as well as the need to check the complete axiom expression instead of minterm expressions (see Section 3.2). This need arises in special cases as will be shown in a fabricated case.

The example is a generic Set whose elements are inserted with a special constructor $insertOneOf(S, E, F)$. This constructor inserts in set S one of the elements E or F, with the restriction that it can only be applied if at least one of these elements is not yet in S. If none of the elements exist in set S, the implementation is free to choose one of the values. Fig. 10 presents the specification of this data type in CONGU.

Assume that we generate tests based on models in which the first element takes precedence (i.e., in case neither E nor F exists in S, E is inserted), whilst the implementation under test behaves the opposite way. More concretely, the implementation under test in case neither E nor F exists in S, inserts F; if one already exists raises an exception. For example, consider one generates the following expression (a valid expression under the assumption above stated) to replace some variable S in an axiom under test:

$$insertOneOf(insertOneOf(empty(), Elem0, Elem1), Elem1, Elem1)$$

A direct translation of this expression into Java code (assuming the same names are used in the implementation under test) would be:

```
Set<Element> s = new Set<Element>();
s.insertOneOf(elem0, elem1);
s.insertOneOf(elem1, elem1);
```

But, if the implementation under test behaves as stated above, the last instruction will cause an exception, because elem1 already exists in the set. So, a safe encoding requires checking the domain condition before applying any operation with a defined domain condition and issuing a warning in case it does not hold (see Sec. 4.2):

```
Set<Element> s = new Set<Element>();
if (s.isIn(elem0) and s.isIn(elem1)) {
    warn("DomainConditionNotMet:insertOneOf.");
```

```
    return;
}
s.insertOneOf(elem1, elem1);
if (s.isIn(elem0) and s.isIn(elem1)) {
    warn("DomainConditionNotMet:insertOneOf.");
    return;
}
s.insertOneOf(elem1, elem1);
```

Similar reasons justify the need to verify the axiom expression instead of minterm expressions. Assume one wants to test the minterm

$$isIn(insertOneOf(S, E, F), E) \ \wedge \ (isIn(S, E) \mathrel{!}= isIn(S, F))$$

of the FDNF expression of the axiom

$$isIn(insertOneOf(S, E, F), E) \ \ if \ \ isIn(S, E) \mathrel{!}= isIn(S, F)$$

using the following assignment (satisfying the minterm, according to our assumptions):

$E \mapsto Elem0$
$F \mapsto Elem1$
$S \mapsto insertOneOf(empty(), Elem0, Elem2)$

Since the set S in the implementation under test will contain $Elem2$ instead of $Elem0$ (according to the above stated assumptions), and the result of $insertOneOf(S, E, F)$ is F to be inserted in the set, none of the clauses of the minterm will hold in the implementation under test. However, the axiom still holds (by definition, it holds for any variable assignment). This is the justification for verifying the axiom expression instead of the minterm expression in the generated test case (see Sec. 4.2).

Hence, in both situations, the approach followed ensures that the generated test cases are correct (i.e., do not fail in the presence of a correct implementation), even if test coverage is less guaranteed. Nevertheless, the potential negative impacts in test coverage can be mitigated by generating multiple test cases for each minterm.

6.3 Example of a Constructor with Constructed Arguments – Set Union

The next example illustrates the features designed in the Alloy encoding and abstract test extraction to properly handle constructors that have arguments that have themselves to be constructed (see Sec. 3.2).

Fig. 11 shows the specification of a generic Set with the $empty$ and $singleton$ creators, and a $union$ transformer to generate sets with more than one element (taking as arguments two sets). Fig. 12 shows the corresponding encoding in Alloy. The main differences wrt the normal encoding are underlined. Since the arguments of the constructor $union$ are themselves of a constructed type, the more complex form of the construction fact has to be used, to ensure that all instances can be constructed in an acyclic way

```
specification Set[Element]
 sorts
   Set[Element]
 constructors
   empty: --> Set[Element];
   insertOneOf: Set[Element] Element Element -->? Set[Element];
 observers
   isEmpty: Set[Element];
   isIn: Set[Element] Element;
 domains
   E, F: Element;
   S: Set[Element];
   insertOneOf(S, E, F) if not isIn(S, E) or not isIn(S, F);
 axioms
   E, F, G: Element;
   S: Set[Element];
   isEmpty(empty());
   not isEmpty(insertOneOf(S, E, F));
   not isIn(empty(), E);
   isIn(insertOneOf(S,E,F),G) = isIn(S,G)
        if G != E and G != F and (not isIn(S, E) or not isIn(S, F));
   isIn(insertOneOf(S,E,F),E) if isIn(S, E) != isIn(S, F);
   isIn(insertOneOf(S,E,F),F) if isIn(S, E) != isIn(S, F);
   isIn(insertOneOf(S,E,F),E) = (E=F or !isIn(insertOneOf(S,E,F),F))
        if not isIn(S, E) and not isIn(S, F);
end specification
```

Fig. 10. Specification of a generic set in CONGU with an "underspecified" constructor.

(see Table 3). The construction fact imposes 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. Such partial ordering is provided by the signature Any and predicate $precedes$.

Fig. 13 shows a model instance found by Alloy Analyzer when the run command in the bottom of Fig. 12 is executed. In this case, the canonical term to be assigned to each variable x in X_B (in this example, $S1$, $S2$ and $S3$) is obtained by following a path from the $start$ node to the node assigned to that variable, ignoring all the edges $s \rightarrow t$ with $constrOrd(s) \geq constrOrd(t)$ or that involve some argument a with $constrOrd(a) \geq constrOrd(t)$. Hence, a possible assignment (written using Alloy notation) is:

$S1 \mapsto start.singleton[Element0]$
$S2 \mapsto start.singleton[Element1]$
$S3 \mapsto start.singleton[Element0].union[Set2]$, with $Set2 = singleton[Element1]$

6.4 Example of Generation of Actual Method Parameters – SortedSet with Comparator

This example illustrates how the presented approach can also be applied to automatically generate actual parameters for methods where the formal parameters are of interface types. The example is a $SortedSet$ in which the elements are not compared by natural order, but instead by using a comparator that is passed as parameter to the

```
specification Set[Element]
 sorts
   Set[Element]
 constructors
   empty: --> Set[Element];
   singleton: Element --> Set[Element];
   union: Set[Element] Set[Element] --> Set[Element];
 observers
   isEmpty: Set[Element];
   isIn: Set[Element] Element;
 axioms
   E, F: Element;
   S, S1, S2, S3: Set[Element];
   union(S, empty()) = S; // neutral element
   union(S, S) = S; // idempotence
   union(S1, S2) = union(S2, S1); // commutativity
   union(union(S1, S2), S3) = union(S1, union(S2, S3)); // associativity
   isEmpty(empty());
   not isEmpty(singleton(E));
   isEmpty(union(S1, S2)) iff isEmpty(S1) and isEmpty(S2);
   not isIn(empty(), E);
   isIn(singleton(E), F) iff E = F;
   isIn(union(S1,S2), E) iff isIn(S1,E) or isIn(S2, E);
end specification
```

Fig. 11. Specification of a generic Set with a $union$ constructor in CONGU.

constructor (creator) – a possibility commonly available in the Java Collections Framework.

Fig. 14 presents the specification of this data type. Fig. 15 presents an excerpt of the corresponding encoding in Alloy, including a run command that generates the model instance shown in Fig. 16. Figs. 18 and 17 show the corresponding test code generated from the model instance in Fig. 16.

6.5 Example of an ADT that does not admit finite models - Stack

The fact that Alloy Analyzer only performs model-finding over restricted scopes consisting of a user-defined finite number of objects imposes a limitation of the approach presented so far: the inability to generate tests for ADTs that do not admit finite models. A simple example is the case of an unbound $Stack$: since the domain of $push$ is true, it is always possible to create a bigger stack.

To overcome this problem, we have defined a modified set of translation rules from CONGUto Alloy, that encompasses transforming constructors into partial functions in Alloy and inserting definedeness guard conditions in the axioms that use those constructors. The translation rules from Alloy to JUnit are not changed.

Fig. 19 presents a specification of a generic $Stack$. Fig.20 presents the corresponding encoding in Alloy. The differences with respect to the normal translation rules (underscored in Fig. 20) are:

```
open util/boolean as BOOLEAN
open util/integer as INTEGER

abstract sig Any { constrOrder : one Int }
pred precedes [x : Any, y : Any] { x.constrOrder < y.constrOrder }

one sig start {
   empty: one Set,
   singleton : Element -> one Set
}

sig Element extends Any {}

sig Set extends Any {
   union : Set -> one Set,
   isEmpty: one BOOLEAN/Bool,
   isIn: Element -> one BOOLEAN/Bool
}
fact SetConstruction {
   Set in (start.empty + {x: start.singleton[Element] | some a1:Element |
                              x=start.singleton[a1] and precedes[a1,x]}).*
    {x: Set, y:  {y: x.union[Set] | some a1: Set |
                  y=x.union[a1] and precedes[x,y] and precedes[a1,y]}}
}
fact axiomSet0{ all S:Set | S.union[start.empty] = S }
fact axiomSet1{ all S:Set | S.union[S] = S }
fact axiomSet2{ all S1, S2:Set | S1.union[S2] = S2.union[S1] }
fact axiomSet3{
   all S1, S2, S3:Set | (S1.union[S2]).union[S3] = S1.(union[S2].union[S3])
}
fact axiomSet4{ start.empty.isEmpty = BOOLEAN/True }
fact axiomSet5{ all E: Element | start.singleton[E].isEmpty = BOOLEAN/False}
fact axiomSet6{
   all S1, S2: Set | S1.union[S2].isEmpty = BOOLEAN/True
      iff (S1.isEmpty = BOOLEAN/True and S2.isEmpty = BOOLEAN/True)
}
fact axiomSet7{ all E:Element |  start.empty.isIn[E] = BOOLEAN/False }
fact axiomSet8{ all E,F: Element | start.singleton[E].isIn[F]=BOOLEAN/Trueiff (E=F)}
fact axiomSet9{
   all S1, S2: Set, E: Element | S1.union[S2].isIn[E] = BOOLEAN/True
      iff (S1.isIn[E] = BOOLEAN/True or S2.isIn[E] = BOOLEAN/True)
}
// ... other run commands omitted
run run_axiomSet3_0{
   some S1, S2, S3:Set | (S1.union[S2]).union[S3] = S1.(union[S2].union[S3])
            and (S1 != S2) and (S2 != S3) and (S1 != S3)
} for 8 but exactly 2 Element
```

Fig. 12. Encoding in Alloy of the specification of the generic Set with a $union$ constructor.

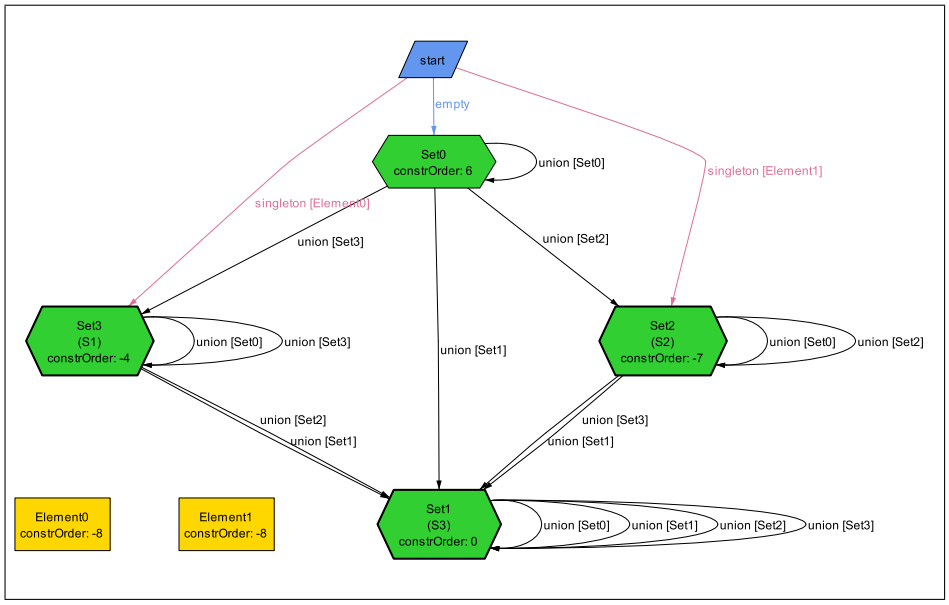

Fig. 13. Model instance generated by the run command in Fig.12.

- constructors ($push$ and $make$) have multiplicity $lone$ (zero or one) instead of one;
- constructor definedeness guard conditions are introduced in the axioms that use the constructors (in the case of equality axioms, only constructor references in the left-hand side are protected to reduce the possibility of generating inadequate model instances and improve test coverage);
- constructor definedeness conditions appear also in the generated run commands.

The drawback of this approach is that, for some specifications the finite models found with Alloy Analyzer are not guaranteed to be subsets of a (infinite) model of the initial specification (we are still investigating a characterization of these specifications). As a consequence, the test data extracted from those finite models for exercising a given axiom, may turn out to not be adequate to fully exercise the axiom (according to the FDNF criteria). Since we verify the original axiom in the test cases generated (and axioms must hold for any variable assignment), there is no danger of generating *unsafe* test cases (i.e., test cases that can fail when testing a correct implementation), but only the danger of generating *unsound* test cases (i.e., test cases in which the *assert* statements are not reached). Anyway, this problem can be mitigated by generating multiple test cases for each axiom or axiom case.

Even for specifications satisfiable by finite models, relaxing the definedeness obligation for constructors has the advantage of leading to potentially smaller model instances.

```
specification Comparator[Element]
  sorts
    Comparator
  others
    geq: Comparator Element Element;
  axioms
    C: Comparator;
    E, F, G: Element;
    E = F if geq(C, E, F) and geq(C, F ,E);
    geq(C, E, F) if E = F;
    geq(C, E, F) if not geq(C, F, E);
    geq(C, E, G) if geq(C, E ,F) and geq(C, F, G);
end specification

specification SortedSet[Element]
  sorts
    SortedSet[Element]
  constructors
    empty: Comparator[Element] --> SortedSet[Element];
    insert: SortedSet[Element] Element --> SortedSet[Element];
  observers
    isEmpty: SortedSet[Element];
    isIn: SortedSet[Element] Element;
    largest: SortedSet[Element] -->? Element;
    comparator: SortedSet[Element] --> Comparator[Element];
  domains
    S: SortedSet[Element];
    largest(S) if not isEmpty(S);
  axioms
    E, F: Element;
    C: Comparator[Element];
    S: SortedSet[Element];
    isEmpty(empty(C));
    not isEmpty(insert(S, E));
    not isIn(empty(C), 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(comparator(S), E, largest(S));
    largest(insert(S, E)) = largest(S)
        if not isEmpty(S) and not geq(comparator(S), E, largest(S));
    insert(insert(S, E), E) = insert(S, E);
    insert(insert(S, E), F) = insert(insert(S, F), E);
    comparator(empty(C)) = C;
    comparator(insert(S,E)) = comparator(S);
end specification
```

Fig. 14. Specification of a $SortedSet$ with a $Comparator$ in CONGU.

```
open util/boolean as BOOLEAN

one sig start{
   Empty: Comparator -> one SortedSet
}

sig Element{}

sig Comparator  {
   geq: Element -> Element -> one BOOLEAN/Bool
}
fact axiomComparator0{
   all C: Comparator, E, F: Element | E = F implies C.geq[E][F] = BOOLEAN/True
}
// …other axioms omitted

sig SortedSet  {
   isEmpty:one BOOLEAN/Bool,
   isIn:Element ->one BOOLEAN/Bool,
   largest:lone Element,
   insert:Element -> one SortedSet,
   comparator: one Comparator
}
fact SortedSetConstruction{
   SortedSet in (start.Empty[Comparator]).*({x:SortedSet, y:x.insert[Element]})
}
// …other axioms omitted
fact axiomSortedSet3{
   all E:Element, S:SortedSet|
       (S.isEmpty = BOOLEAN/False
        and S.comparator.geq[E][S.largest]=BOOLEAN/False)
           implies S.insert[E].largest = S.largest
}
// …other run commands omitted
run run_axiomSortedSet3_0{
   some E:Element, S:SortedSet|
      S.isEmpty = BOOLEAN/False
      and S.comparator.geq[E][S.largest] = BOOLEAN/False
      and S.insert[E].largest = S.largest
} for 4
```

Fig. 15. Excerpt of the encoding in Alloy of the specification of $SortedSet$ with $Comparator$.

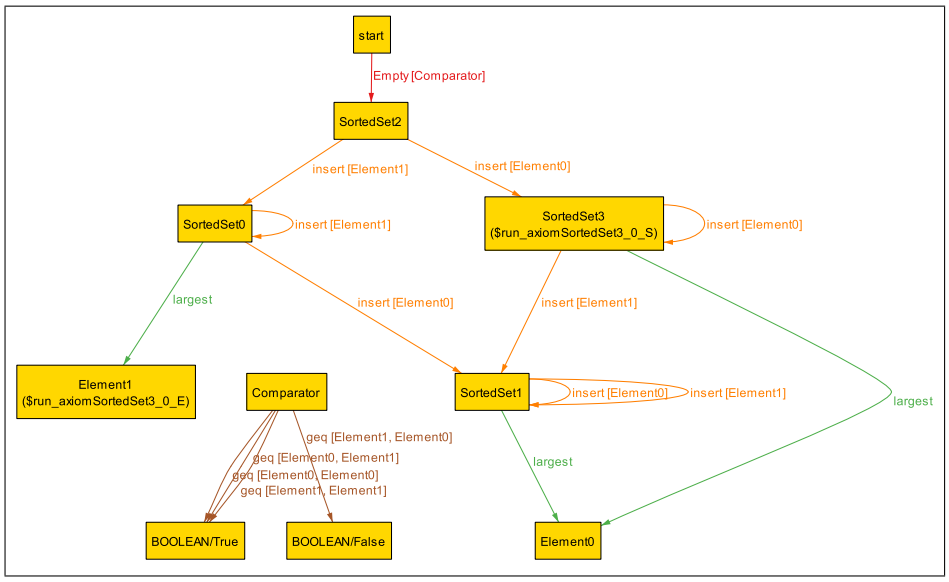

Fig. 16. Model instance generated by the run command in Fig.15.

```
1 import org.junit.*;
3 public class SortedSetTest {
4     private interface Factory<T> { T create(); }
5
6     private void axiomSortedSet3(Factory<TreeSet<ElementMock>> sFact,  ElementMock e) {
7         TreeSet<ElementMock> s_0 = sFact.create();
8         TreeSet<ElementMock> s_1 = sFact.create();
9         if (!s_0.isEmpty() && !s_0.comparator().greaterEq(e, s_0.largest())) {
10            s_1.insert(e);
11            assertTrue(s_1.largest().equals(s_0.largest()));
12        }
13    }
14
15    @Test public void test0_axiomSortedSet3_0() {
16        // mock objects for the parameter
17        final ElementMock element0 = new ElementMock();
18        final ElementMock element1 = new ElementMock();
19        final ComparatorMock<ElementMock> comparator = new ComparatorMock<ElementMock>();
20        comparator.add_greaterEq(element0, element0, true);
21        comparator.add_greaterEq(element0, element1, true);
22        comparator.add_greaterEq(element1, element1, true);
23        comparator.add_greaterEq(element1, element0, false);
24        // factory objects for the axiom var's of parameterized type
25        Factory<TreeSet<ElementMock>> sFact =
26          new Factory<TreeSet<ElementMock>>() {
27            public TreeSet<ElementMock> create() {
28                TreeSet<ElementMock> s = new TreeSet<ElementMock>(comparator);
29                s.insert(element0);
30                return s;
31            }
32        };
33        // axiom verification
34        axiomSortedSet3(sFact, element1);
35    }
36 }
```

Fig. 17. JUnit test case generated from the model instance in Fig. 16.

```java
import  java.util.HashMap;
import java.util.LinkedList;

public class ComparatorMock<E>  implements Comparator<E> {
    private HashMap<LinkedList<Object>, Boolean> greaterEqMap =
        new HashMap<LinkedList<Object>, Boolean>();
    public boolean greaterEq(E x, E y) {
        LinkedList<Object> params = new LinkedList<Object>();
        params.add(x);
        params.add(y);
        return greaterEqMap.get(params);
    }
    public void add_greaterEq(E x, E y, boolean result) {
        LinkedList<Object> params = new LinkedList<Object>();
        params.add(x);
        params.add(y);
        greaterEqMap.put(params, result);
    }
}
```

Fig. 18. Mock class generated from the *Comparator* signature.

```
specification Stack[Element]
  sorts
    Stack[Element]
  constructors
    make:   --> Stack[Element];
    push:    Stack[Element] Element --> Stack[Element];
  observers
    peek:   Stack[Element] -->? Element;
    pop:    Stack[Element] -->? Stack[Element];
    empty: Stack[Element];
  domains
    S: Stack[Element];
    peek(S) if not empty(S);
    pop(S)  if not empty(S);
  axioms
    S: Stack[Element];
    E: Element;
    peek(push(S, E)) = E;
    pop(push(S, E)) = S;
    empty(make());
    not empty(push(S, E));
end specification
```

Fig. 19. Specification of a *Stack* in CONGU.

```
open util/boolean as BOOLEAN

one sig start{ make: lone Stack }

sig Element{}

sig Stack  {
   push: Element -> lone Stack,
   peek: lone Element,
   pop: lone Stack,
   empty: one BOOLEAN/Bool
}
fact StackConstruction{
   Stack in (start.make).*({x:Stack, y:x.push[Element]})
}
fact domainStack0{
   all S:Stack | S.empty != BOOLEAN/True implies one S.peek else no S.peek
}
fact domainStack1{
   all S:Stack | S.empty != BOOLEAN/True implies one S.pop else no S.pop
}
fact axiomSortedSet0{
   all E:Element, S:Stack | one S.push[E] implies (S.push[E].peek = E)
}
fact axiomSortedSet1{
   all E:Element, S:Stack | one S.push[E] implies (S.push[E].pop = S)
}
fact axiomSortedSet2{
   one start.make implies  (start.make.empty = BOOLEAN/True)
}
fact axiomSortedSet3{
   all E:Element, S:Stack | one S.push[E] implies (S.push[E].empty=BOOLEAN/False)
}

run run_axiomSortedSet0_0{
   some E:Element, S:Stack | one S.push[E] and (S.push[E].peek = E)
}
run run_axiomSortedSet1_0{
   some E:Element, S:Stack | one S.push[E] and (S.push[E].pop = S)
}
run run_axiomSortedSet2_0{
   one start.make and (start.make.empty = BOOLEAN/True)
}
run run_axiomSortedSet3_0{
   some E:Element, S:Stack | one S.push[E] and (S.push[E].empty=BOOLEAN/False)
}
```

Fig. 20. Encoding in Alloy of the specification of the $Stack$.

7 Conclusions and Future Work

Although testing from algebraic specifications has been thoroughly investigated, existent approaches are based on flat specifications. In this technical report, we discussed testing from parameterized specifications and put forward a notion of 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 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. The approach presented can also be applied to automatically generate actual parameters for methods where the formal parameters are of interface types. A tool that completely automates the approach presented is currently under development.

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 unspecified behaviours.

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, s.a. unbounded stacks. 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.

As future work, we also intend to extend the approach to automatically rule out by static analysis unsatisfiable axiom cases and generate tests outside operations' domains and for properties not explicitly included in specifications s.a. 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. Bidoit, M., Mosses, P.: CASL User Manual, LNCS, vol. 2900. Springer (2004)
2. 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)
3. 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)

4. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. 3, 101–130 (1994)
5. Ehrig, H., Mahr, B., Orejas, F.: Introduction to algebraic specification. part 2: from classical view to foundations of system specifications. Comput. J. 35, 468–477 (October 1992)
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., Abreu, J., , Reis, L.: Checking the conformance of Java classes against algebraic specifications. In: Proc. of Eighth International Conference on Formal Engineering Methods 2006. LNCS, vol. 4260, pp. 494–513. Springer-Verlag (2006)
16. 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)
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)