FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Generation of Test Cases Derived from Algebraic Specifications

Francisco Xavier Richardson Rebello de Andrade

Master in Informatics and Computing Engineering

Supervisor: João Carlos Pascoal de Faria (Professor)

Co-Supervisor: Ana Cristina Ramada Paiva Pimenta (Professor)

28th June, 2010

Automatic Generation of Test Cases Derived from Algebraic Specifications

Francisco Xavier Richardson Rebello de Andrade

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Rui Filipe Lima Maranhão de Abreu (Professor)

External Examiner: Manuel Alcino Cunha Pereira (Professor)

Supervisor: João Carlos Pascoal de Faria (Professor)

28th June, 2010

Abstract

The aim of this dissertation is to find a way to automatically generate test cases for generic abstract data types derived from algebraic specifications, completely offline from any implementation.

The reported work in this dissertation was done in the context of the QUEST research project funded by FCT. Some of the demands of the QUEST Project's group are the use of the algebraic specification language utilized by a certain runtime conformance checking tool named ConGu, and the use of Java as the output language of the generated test cases.

After some research and analysis of the state of the art of test case generation based on algebraic specifications, no methods or tools were found that could fully satisfy the described requisites, and so there was need to create a new method and supporting tool.

In this report, one can find the layout of the newly developed method to tackle this problem and satisfies all the requisites. This method consists on translating the algebraic specifications into Alloy specifications – modelling language based on first-order relational logic -, and then using the Alloy Analyzer to find model instances with this new specification. After this, all that needs to be done is to extract from these models the test cases and the behavior of the objects used as generic variables.

During this dissertation, a tool was developed that currently translates algebraic specifications into Alloy specifications successfully. An added value of this method and the developed tool is that it allows consistency checking of algebraic specifications. This is done just by looking at the model instances generated.

This new method has great potential and brings a scientific contribution, mainly in the areas of Software Quality and Testing and Formal Methods of Software Engineering.

Resumo

Esta dissertação foca-se em arranjar uma solução para a problemática de geração automática de casos de testes para tipos abstractos de dados genéricos a partir de especificações algébricas e de uma forma totalmente independente de qualquer implementação.

O trabalho relatado nesta dissertação foi desenvolvido no contexto do projecto de investigação QUEST, financiada pela FCT. Algumas das exigências do grupo do projecto QUEST que requisitou esta solução incluem usar a linguagem de especificação de uma ferramenta de verificação de conformidade em tempo de execução chamada ConGu, e que os casos de teste sejam escritos em Java.

Após ter investigado e analisado o estado da arte de geração de Casos de Teste a partir de especificações algébricas, não havia nenhum método ou ferramenta que conseguisse satisfazer na totalidade os requisitos descritos, e portanto houve a necessidade de criar um novo método e ferramenta de suporte.

Neste relatório, pode-se encontrar uma planificação da nova solução desenvolvida para este problema e que satisfaz todos os requisitos. Esta consiste em traduzir as especificações algébricas em especificações de Alloy – linguagem de modelação baseada em lógica relacional de primeira ordem -, e depois usa o Alloy Analyzer para encontrar instanciações modelo com esta nova especificação. No final, resta apenas extrair os casos de teste, e os comportamentos dos objectos usados como variáveis genéricas, destes modelos.

Foi desenvolvida uma ferramenta que actualmente traduz as especificações algébricas em especificações de Alloy com sucesso. Um valor acrescido deste método e da ferramenta desenvolvida é que permite verificar a consistência de especificações algébricas. Isto é feito apenas por olhar para as instanciações modelo geradas.

Este novo método tem bastante potencial e trás um contributo cientifico, principalmente, nas ares de Teste e Qualidade de Software e Métodos Formais de Engenharia de Software.

Acknowledgements

I would like to thank everyone, in general, that helped, supported and put up with me during this long, and a lot of the times exhausting, battle to achieve these results and success. But nevertheless, there are few people that stuck out of the crowd and would like to mention.

To my supervisor, Professor João Pascoal Faria, I would like to thank his availability to meet up and discuss the project, and his invaluable and intelligent input.

To my co-supervisor, Professor Ana Paiva, I would like to thank the different points of view, and fresh pair of eyes, on the matter.

To Pedro Crispim and Professor Antónia Lopes, part of the FCUL ConGu team, I thank the promptness in helping with the setting up of the ConGu tool.

To my Family, I mostly thank the understanding of my absence during this last semester.

To my friends, I thank the good and relaxing moments (lunches and teas included) to unwind from work and get the dissertation off my mind.

To Mariana and António I thank for all the good reasons.

And finally to everyone that made my stay in FEUP, as a student of MIEIC, so enjoyable. Not only during the last semester, but the last 5 years.

Francisco Rebello de Andrade

Contents

List of Figures

Abbreviations

ADT	Abstract Data Type
AS	Algebraic Specification
FEUP	Faculty of Engineering of the University of Porto
FCUL	Faculty of Science of the University of Lisbon
IDE	Integrated Development Environment
TC	Test Case
JML	Java Modelling Language
FSM	Finite State Machine
AA	Alloy Analyzer

1 Introduction

1.1 Context

The investigation and programming, that was performed during the execution of this dissertation, are part of a larger project called *A Quest for Reliability in Generic Software Components* [1], also referred to as QUEST for short.

The two institutions involved in this project are the Faculty of Science of the University of Lisbon – FCUL - and the Faculty of Engineering of the University of Porto - FEUP. Its mission, described in [1], can be read below:

> *"QUEST aims at the development and integration of a set of effective techniques for the automated reliability analysis of Java software components that implement given, specified, data abstractions. Particularly, it must be applicable to both raw and generic Java modules, specified in terms of property-driven specifications, and it must support the analysis in the absence of the source code. Furthermore, in cases where source code is available, it should help to interpret failures and locate errors in the code that may have caused them."*

Given this goal and these objectives, and after the deliberation of the researching team, the project [1] was divided into five major tasks:

1. Elaborate an approach to check runtime conformance of Java generic classes against Algebraic Specifications – ASs - of generic data abstractions;
2. Define and investigate specification-based techniques for unit and integration testing that make use of runtime conformance checking;
3. Develop techniques that interpret failures, and locate errors, when the implementation's source code is available;
4. Use the researched techniques to implement a tool integrated in a popular Java integrated development environment - IDE;
5. Elaborate some case studies that evaluate the produced solutions.

The QUEST project's goal is to develop over the already existing ConGu [2-9] application and extend it, which was implemented by the same FCUL researchers, and already checks conformance at runtime and possesses an AS language with generic compatibility. ConGu is also working on a plug-in for the Eclipse IDE [4, 10]. More about ConGu and its language will be added in chapter 2.

1.2 Motivation

The main motivation for this project is the development of an approach to generate test cases – TCs - for generic abstract data types – ADTs - based on ASs in an automated way, and the implementation of a tool with this approach. The TC generation based on ASs field ought to be more looked into and explored because these types of specifications are easy to produce, and it's not as tedious compared to other types of formal specifications. This can motivate people to write more ASs and test their programs.

Another motivation to follow through with the objectives is that then it will be possible to reuse ASs, since we will start developing generic ASs, saving time and making specifications evermore complete and safe.

By the end, we hope to help make ConGu a more complete tool.

1.3 Objectives

The main objectives for this dissertation are:

- To investigate and define which strategies and algorithms to use in the automatic generation of unit TCs based on ASs, for non-generic and generic ADTs implemented in Java.
- Initiate the development of a tool that can generate these TCs, without requiring the presence of any Java implementation of the type that we wish to test - offline -, using the previously defined strategies and algorithms. Other QUEST tasks should be considered when making implementation decisions.
- To write a thesis documenting this produced approach.

For starters, the implemented tool must run independently from ConGu's runtime conformance checker, out of interest for an independent generic ADT unit TC generator. Later, it will be adapted to accomplish the second major task of the QUEST project, mentioned in section 1.1, which falls under the FEUP's responsibility.

1.4 Report's structure

The rest of the dissertation is divided into 6 parts going from chapter 2 to 5, since this introduction chapter is the first of this report, and the Appendix A.

The second chapter aims at presenting some concepts of strong presence in this dissertation, along with a description of the ConGu tool and language.

In the third chapter, a description of the methods to automatically generate TCs based on ASs is presented along with a conclusion of their aptitude to resolve the problem of this dissertation. Here we can find the studied state of the art methods and two new engineered ones.

Key points to consider when evaluating these methods as possible solutions to the problem at hand are also specified

In Chapter 4, we can find the method that was opted to elaborate and implement to fulfil the dissertation's objectives. Here we can also find a description of the developed application that generates test case opportunities, and its architecture, plus the plan to extract TCs from them.

Chapter 5 is where the conclusions and future plan ideas can be found.

The Final Chapter, chapter 6, lists all the bibliographical references for this report.

Finally, the Appendix A presents some tested ConGu modules specifying generic ADTs and the results produced by the developed application.

2 Algebraic Specifications for Generic Abstract Data Types

In this chapter, we present two definitions of two fundamental elements of this dissertation. First a description of ADTs is presented followed by a description of ASs. After consolidating the definition of these terms, the ConGu tool is explained along with ConGu's AS language which is used in the developed application described later on in this report.

2.1 Abstract Data Types

Since midst 1970s, and as predicted by John Guttag [11], there has been huge progress in the way people think about software programming – new concepts, methodologies, designs and architectures-, and ADTs are an example of this.

An ADT [11] is a set of values and operations - possessing a certain interface - that have an invariant specification independent from the unknown implementation – black-box. Using the Stack data structure example, the interface would be the operations that one can perform with it, like *push* and *pop*, and the invariant specifications of these operations would be adding an element to the Stack, in the case of the *push* operation, and removing the element added last from the Stack, in the case of the *pop* operation.

Although the ADT concept is frequently used in programming, it's seldom given its due value. Initially, their use was considered by many unnecessary, or with a small amount of application cases. But then again, the people back then had a more fractal view of the software creating process – the code was developed the same way at each layer – as opposed to a more architectural view – with different possible structural designs and algorithms, to consider, for each layer.

One of the assumptions that support the ADTs' use is that knowing too much about a specific implementation may be as harmful, in the long run, as knowing too little. This may cause a person to fiddle and ruin something that was considered reliable, and that obeyed a certain specification. The use of ADTs also provides a safer implementation by not allowing direct access to inside information, making the management of available operations on the type's interface, a crucial decision.

This valuable engineering concept also eases the maintenance of software due to the partition characteristics that it gives to the developed software, allowing the substitution of one part easier, therefore making each part more independent from the remaining ones. And due to its abstraction, which eliminates detail, it makes the type reusable and more reliable with each use.

A related, and derived, concept used by several programming languages – like Java – is the interface abstract type. These interfaces have proven themselves to be very useful to develop software and facilitate design and communicating protocols – like Java's remote method invocation [12].

2.2 Algebraic Specifications

ASs appeared in 1970 [13, 14] and, over that past three decades, it has grown into a mature formal method [14, 15].

As mentioned in the article [16], ASs of an implementation are composed of a syntax declaration followed by a semantic declaration. The former one declares sorts and lists the sorts' operations, along with their input parameters and return types. The latter declaration defines rules, sequencing and relating several operations of the involved sorts – black-box –, forming axioms that must come out true.

Using the previously used Stack example, the syntax part would be defined something like this:

*push: Stack * Element -> Stack*

pop: Stack -> Stack

top: stack -> Element

And two examples of axioms, part of the semantic component, are:

$$\forall Stack\ s,\ s.push(\_).pop() = s$$

$$\forall Stack\ s,\ Elem\ x,\ s.push(x).top() = x$$

As a standard followed by most AS languages, including ConGu, in equality axioms the right hand side expression is a more, or equally, normalised form[1] of the left hand side expression. As once said by Clarke and Wing in [17], "One current trend is to integrate different specification languages, each able to handle a different aspect of a system". As far as black-box testing goes, using AS based testing has proven to be a popular way [14, 16, 18-22], and this project intends to expand this area.

2.3 ConGu

In order to successfully create TCs for generic ADTs based on ASs, we must first be able to specify the genericity in an AS language, and then be able to map it to the desired implementation interface that one wishes to test.

[1] Normal form terms are the simplest and most standard way of representing equivalent terms. E.g. with a stack: *newStack().push(x).pop() = newStack()*. The right term is the normal form of the left term.

In the article [23], which was written about ConGu, the authors state that they are the first to bridge the gap between ASs and generic object-orientated programming. After some research was done, it is believed that not only it is true what they say, but also that they are the only ones to do so up until now. The approach talked about in that paper is an extension from ConGu's previous mapping technique described in article [5], written by the same people along with others, which does not allow genericity nor sub-sorting.

Next, a description of the mapping process and structure of both, specification and mapping, languages presented in article [5] are given. After that, a summary of the adjustments made when generics and sub-typing were introduced into the equation, as depicted in the article [23], is presented.

2.3.1 Algebraic Specifications and Mapping

In the article [5] the authors present, and discuss, ConGu's approach to check the conformance of the implementations of Java classes, at run-time, using ASs. This is done by applying two extra buffer layers of Java classes on top of the original Java class which, ultimately, allows one to verify the conformance each time one of the methods is called by generating JML contract annotations, derived from the provided AS module.

JML [24] is a specification language for the programming language Java using the design by contracts paradigm – specifying pre and post-conditions and invariants of an implementation. In this language, all conditions and invariants of a specific method are defined by annotations in comments, placed before it. The annotation '@ requires' is used to denote a pre-condition and '@ ensures' a post-condition.

This conformance check approach can be resumed to the following graphic:

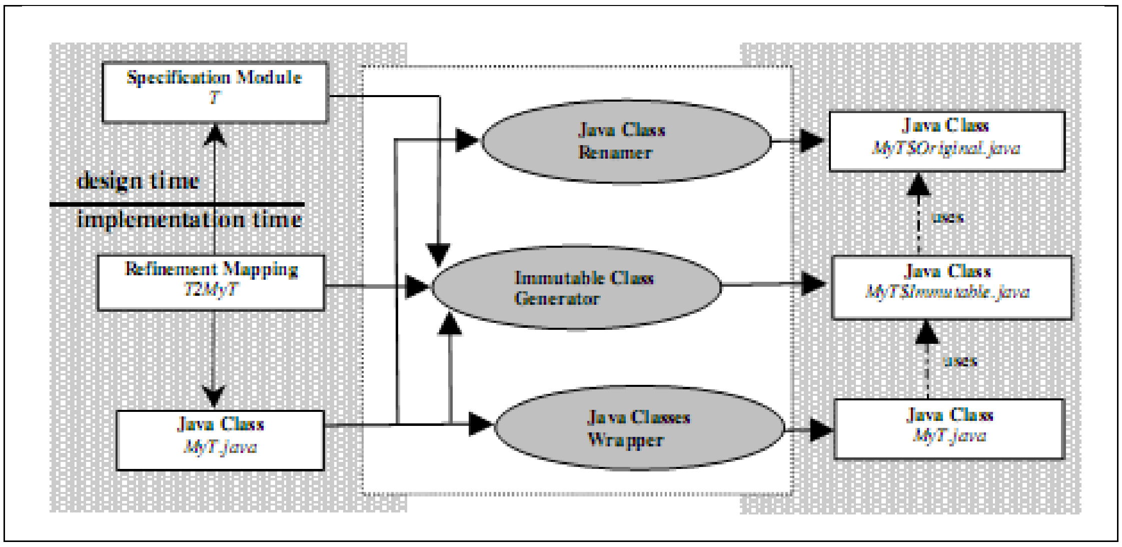
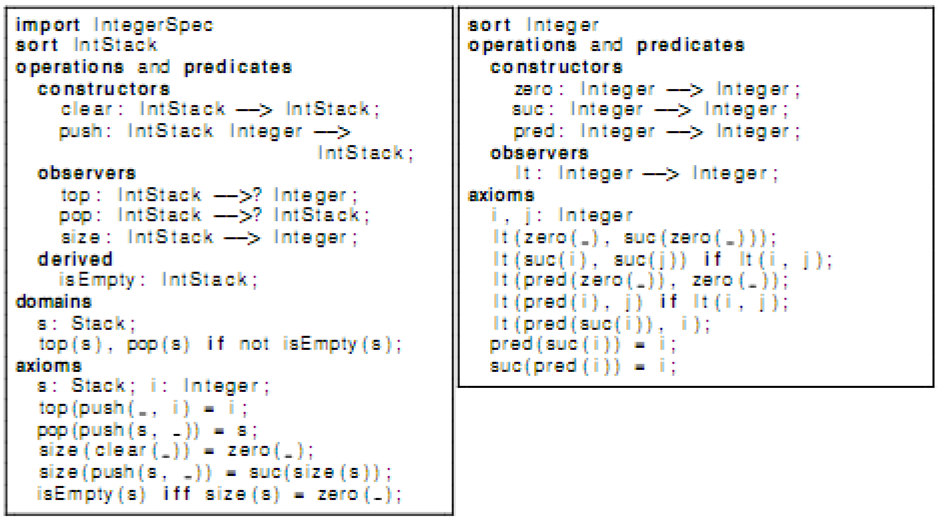


Fig. 1 Conformance check at runtime overview

On the left, we have the files that need to be provided by the user – specification module, refinement mapping and implemented Java class - and, on the right, we have the main generated Java files that constitute the new implementation with conformance checking at runtime – *original class*, *immutable class* and *wrapper class*. The file *MyT.java*, on the left, is renamed to *MyT$Original.java* , producing the first file on the right the new implementation.

The file *MyT$Immutable.java*, on the right, is used to ensure the conformance of the implementation that is being generated. This class has only static functions, whose format is similar to the operations in the specification module *T* - immutable functional notation -, with defined pre and post-conditions. These contracts are written in JML and are derived from the domains and axioms defined in *T*. The methods in class *MyT$Original.java* are paired up to the operations in file *T*, so that the corresponding functions in *MyT$Immutable.java* may know which method to call. This is achieved with the aid of a refinement mapping file *T2MyT*, on the left.

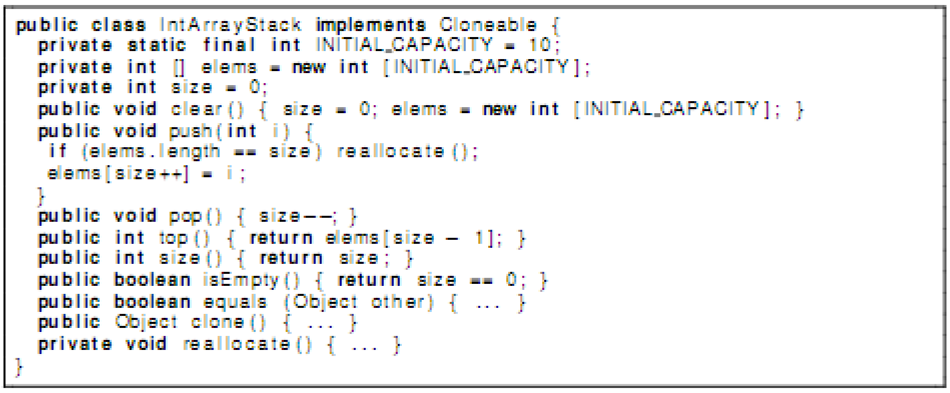
Here are examples of a specification module, an implemented Java class and a refinement mapping, of a *stack of integers'*, for better understanding:



```
import IntegerSpec
sort IntStack
operations and predicates
  constructors
    clear: IntStack —> IntStack;
    push: IntStack Integer —>
                        IntStack;
  observers
    top: IntStack —>? Integer;
    pop: IntStack —>? IntStack;
    size: IntStack —> Integer;
  derived
    isEmpty: IntStack;
domains
  s: Stack;
  top(s), pop(s) if not isEmpty(s);
axioms
  s: Stack; i: Integer;
  top(push(_, i) = i;
  pop(push(s, _)) = s;
  size(clear(_)) = zero(_);
  size(push(s, _)) = suc(size(s));
  isEmpty(s) iff size(s) = zero(_);
```

```
sort Integer
operations and predicates
  constructors
    zero: Integer —> Integer;
    suc: Integer —> Integer;
    pred: Integer —> Integer;
  observers
    lt: Integer —> Integer;
axioms
  i, j: Integer
  lt(zero(_), suc(zero(_)));
  lt(suc(i), suc(j)) if lt(i, j);
  lt(pred(zero(_)), zero(_));
  lt(pred(i), j) if lt(i, j);
  lt(pred(suc(i)), i);
  pred(suc(i)) = i;
  suc(pred(i)) = i;
```

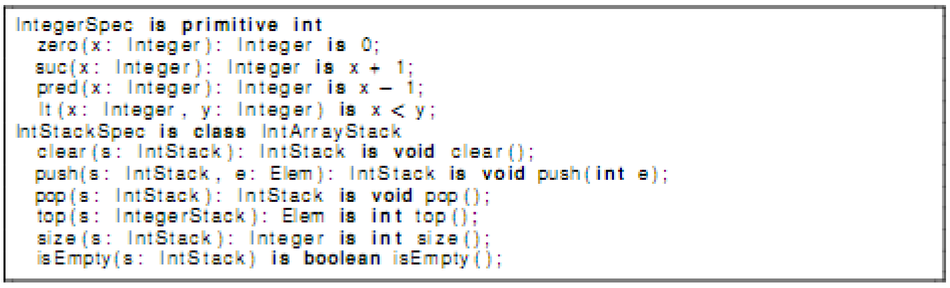
Fig. 2 ASs of the stack of integers and the primitive integer in ConGu [5] [2]



```
public class IntArrayStack implements Cloneable {
  private static final int INITIAL_CAPACITY = 10;
  private int [] elems = new int [INITIAL_CAPACITY];
  private int size = 0;
  public void clear() { size = 0; elems = new int [INITIAL_CAPACITY]; }
  public void push(int i) {
    if (elems.length == size) reallocate();
    elems[size++] = i;
  }
  public void pop() { size--; }
  public int top() { return elems[size - 1]; }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public boolean equals (Object other) { ... }
  public Object clone() { ... }
  private void reallocate() { ... }
}
```

Fig. 3 Java class of the stack of integers

[2] In Fig. 2 the Integer specification has two errors: the 'lt' observer, in the 'operations and predicates', should have one extra input parameter of the type Integer and the axiom 'lt(pred(suc(i)), i)' should be 'lt(i, suc(i))'.



```
IntegerSpec is primitive int
  zero(x: Integer): Integer is 0;
  suc(x: Integer): Integer is x + 1;
  pred(x: Integer): Integer is x - 1;
  lt(x: Integer, y: Integer) is x < y;
IntStackSpec is class IntArrayStack
  clear(s: IntStack): IntStack is void clear();
  push(s: IntStack, e: Elem): IntStack is void push(int e);
  pop(s: IntStack): IntStack is void pop();
  top(s: IntegerStack): Elem is int top();
  size(s: IntStack): Integer is int size();
  isEmpty(s: IntStack) is boolean isEmpty();
```

Fig. 4 Refinement mapping file for the stack of integers [3]

As it can be seen, the specification module is structured in three main sections: *operations and predicates*, *domains* and, finally, *axioms*. The first part, which indicates solely the name and the input and output parameters of the operations and predicates of the sort being defined, is also divided into three parts: *constructors* – operations from which all possible states of the sort being specified can be reached -, *observers* – operations that provide fundamental information about the state of the sort at hand- and *derived* – operations that provide redundant, but potentially useful information about this same sort. The second part of the module specifies domains of the input parameters for certain operations – the operations declared with the operator -->*?* -, serving as the pre-condition contracts of these operations in the JML annotations. The last part specifies the actual axioms that should hold true in the implementation, if the operations used respect their domains, generating post-condition contracts in the JML annotations.

The contracts are formed using the following rules:

- Specified domain restrictions of a certain operation are used as pre-conditions of the method that implements that operation.
- Axioms that relate a constructor operation with another operation, generate post-conditions to the method that implements that constructor operation.
- Axioms that specify the result of an observer operation applied to a constructor operation, create post-conditions to the method that implements this last operation.
- Axioms that specify the result of a derived operation/predicate on simple instances of the sort, develop into post-conditions for the method that implements that operation.
- Equality axioms generate post-conditions for the equals method – used to compare two objects of the implementation being tested.

In the refinement mapping figure - Fig. 4 -, it can be seen how the several operations and predicates, from the specification modules of the Stack and the Integer, are related to the Java methods. The specification Integer and its functions are mapped to Java as the primitive data type *int* and its respective operations. The specification of the *stack of integers*, and its operations and predicates, are mapped to an already existing implemented Java class and its methods. As it can be seen, operations and predicates translate to methods that cease to have the Stack itself as the first input variable, and the ones that have the *stack of integers* as the

[3] In Fig. 4, there are two mistakes. The push and top operations should be accepting Integer sorts instead of Elem sorts.

returning type tend to change it to void, although some might return useful information – the function *pop* could return the Integer it removed from the Stack.

Said this, the *immutable class* in our previously given example, would look something like this:

```
public class IntArrayStack$Immutable {
  //@ ensures size(\result ).value == 0;
  static public IntArrayStack$Original clear (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.clear();
    return aClone;
  }
  //@ ensures size(\result ).value == size(s).value + 1;
  //@ ensures top(\result ).value == i;
  //@ ensures equal(pop(\result ).state , s).value;
  static public IntArrayStack$Original push (IntArrayStack$Original s, int i) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.push(i);
    return aClone;
  }
  //@ requires ! isEmpty(s).value;
  static public IntArrayStack$Original pop (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.pop();
    return aClone;
  }
  static public int$Pair size (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    return new int$Pair (aClone.size(), aClone);
  }
 //@ ensures (* See Section 6 *);
    static public boolean equals (IntArrayStack$Original s, Object t) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    return new boolean$Pair (aClone.equals(t), aClone);
  }
  ...
}
```

Fig. 5 Stack of integers’ immutable class

```
public class int$Pair {
  public final int value;
  public final IntArrayStack$Original state;
  public int$Pair (int value, IntArrayStack$Original state) {
    this.value = value; this.state = state;
  }
}
```

Fig. 6 Auxiliary class and state pairing class

As can be seen, the functions in the specification module that originally returned types other than the new state of the Stack, in the *immutable class* return an auxiliary class. This is necessary for cases where a state is changed and useful information is also given – for example, the *pop* function mentioned previously. The auxiliary class pairs up this returned information from the operation with the Stack’s new state. This allows the *immutable class*’s operation at hand to test the new state, verifying any post-condition with JML annotations, and return it, along the useful information, to the *wrapper class* – explained below. In these situations, the extra useful information that is returned is not tested.

Lastly, the wrapper class file *MyT.java*, on the right hand side of Fig. 1, consists of the same name and public method interface as that of the *original class*. Each method calls their corresponding static contracted function from the *immutable class*, utilising and updating the instance of the *original class* hidden within this *wrapper class*.

Ahead, the *wrapper class* of the *stack of integers* example is displayed:

```
public class IntArrayStack implements Cloneable {
  private IntArrayStack$Original stack = new IntArrayStack$Original();
  public void clear() { stack = IntArrayStack$Immutable.clear(stack); }
  public void push(int i) { stack = IntArrayStack$Immutable.push(stack, i); }
  public void pop() { stack = IntArrayStack$Immutable.pop(stack); }
  public int size() {
    int$Pair pair = IntArrayStack$Immutable.size(stack);
    stack = pair.state;
    return pair.value;
  }
  public int top() {
    int$Pair pair = IntArrayStack$Immutable.top(stack);
    stack = pair.state;
    return pair.value;
  }
  ...
}
```

Fig. 7 Stack of integers' wrapper example

Congu uses the approach described above to generate a new implementation, of a simple non-generic Java type, that monitors its conformance at runtime. Several comparisons were performed in order to test the efficiency of the implementations developed by the tool [5]. In some cases, the time spent using an implementation that checks its conformance was ten times higher than the one that doesn't.

2.3.2 Sub-sorting and Parameterized Sorts

Now we will talk about article [23] and the extensions that were made to the technique mentioned previously in order to add generic data types into the equation.

In this paper the authors differentiate three kinds of specifications: Simple, with Sub-sorting and Parameterized. The simple specifications are the ones that are originally supported by ConGu and described in the previous section 2.3.1. Sub-sorting allows one to specify sorts that extend from other sorts – super-sorts. And finally, the parameterized specifications are definitions of compound sorts that, in their specification, use other specified sorts that are presented as parameter sorts in the form of *sortname[paramsort1,..., paramsortm]* – parameter sorts are specified in parameter specifications, and these cannot be parameterized specifications. This is useful to make specifications reusable at this level and, in this case, as means of specifying generic data types. Ahead we will see three examples of these three types of specifications:

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

Fig. 8 Specification of a total order (simple)

In the simple specification of Fig. 8, a sort Orderable is defined along with its observer predicate *geq* that verifies whether an instance of that sort is greater or equal than another instance of that same sort.

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

Fig. 9 Specification of a total order with a successor operation (sub-sort) [4]

In this specification, sort Successorable is defined as being the sub-sort of the sort Orderable defined previously. Notice that the new operation *suc* is specified algebraically, in the axioms section, with the aid of the previously defined predicate *geq*.

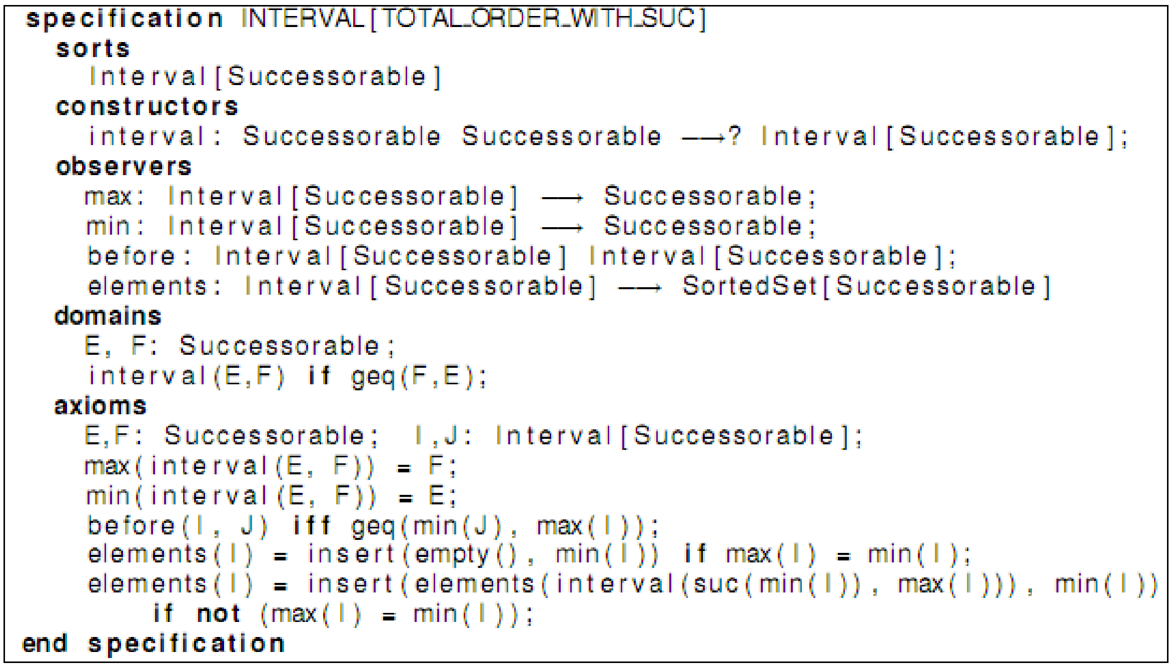
```
specification SORTED_SET[TOTAL_ORDER]
  sorts
    SortedSet[Orderable]
  constructors
    empty: ⟶ SortedSet[Orderable];
    insert: SortedSet[Orderable] Orderable ⟶ SortedSet[Orderable];
  observers
    isEmpty: SortedSet[Orderable];
    isIn: SortedSet[Orderable] Orderable;
    largest: SortedSet[Orderable]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

Fig. 10 Specification of a Sorted Set (parameterized)

In the parameterized specification of Fig. 10, we have the specification of a Sorted Set using the simple specification TOTAL_ORDER, previously defined, so that the sort Orderable along with predicate *geq* may be utilized. This way, we theoretically allow any sort that sub-sorts Orderable to be utilized in this specification. This method, for simplicity sake, does not support explicit instantiation of sub-sorts.

The article gives a more complex and elaborate definition of a specification module for this case. All the involved and needed specifications used to define a certain specification are its module. Take the parameterized specification of Fig. 11 as an example.

[4] In Fig. 9, there is a mistake. The last axiom does not exist.



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

Fig. 11 Specification of an Interval (parameterized)

This INTERVAL specification allows one to generate a Sorted Set of all Successorable elements in between two defined Successorable elements, with the operation *elements*. In this case, the module would consist of TOTAL_ORDER, TOTAL_ORDER_WITH_SUC, SORTED_SET and INTERVAL. Since the role of the first two specifications is clearly different from the last two, specification modules are defined as a pair of sets of specifications - <core, parameter>. The core represents the specifications of the implementation of the data, while parameter represents the specifications that impose constraints to the elements used by the core specifications. In Java, these will be interpreted as classes and interfaces, respectively. Here are the pairs of sets <core, parameter> of the TOTAL_ORDER, SORTED_SET and INTERVAL specification modules:

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

Fig. 12 Some ConGu generic modules

The interpretation of these Modules, in terms of algebras, is important in order to establish the considerations that should be taken towards each specification when mapping them to Java. There are several existing constraints, between the implemented Java classes and their corresponding specification module, that need to be checked in order to guarantee the correct mapping. These constraints come in different levels. A list of structural constraints regarding specification types follows:

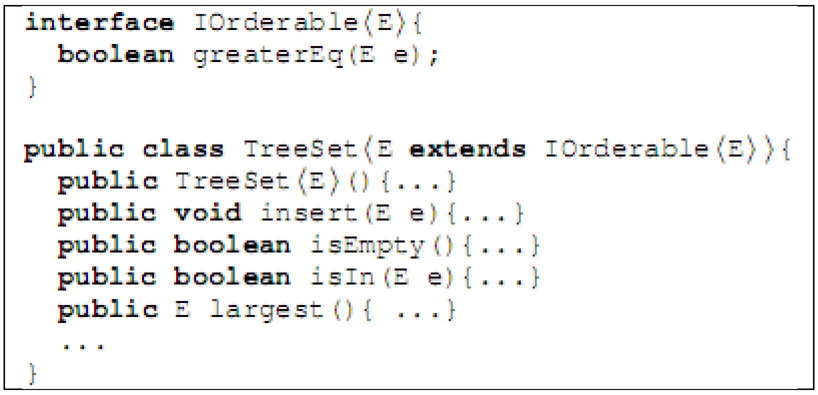
- Core specifications represent Java classes, being these generic if parameterized – at this moment ConGu only allows one core specification per module.
- When dealing with specifications that make use of Sub-sorting, the induced type hierarchy must be enforced by the implementations.
- Sorts specified by simple core specifications correspond to a simple non-generic class.

- Generic sorts specified by parameterized core specifications correspond to a Java generic type with the same arity of generics
- Sub-sorts specified by sub-sorting core specifications correspond to a subtype of the Java type that matches the super-sort of the specification.

There are also constraints to be taken into consideration over the structure of classes and interfaces. These deal with the constraints that every defined operation and predicate of a specification has over the public methods of the corresponding Java class or interface, namely:

- Every operation and predicate of a specification must have a corresponding public method in the Java type that matches that specification.
- The arity of the matching method of an operation or predicate is reduced by one due to being unnecessary to include the current state object (**this**) as the first parameter. An exceptional case is when dealing with a zero-ary operation, which corresponds to a zero-ary constructor of the corresponding class – only zero-ary Java constructors can be specified in ConGu.
- Every predicate matches a Boolean method. Any operation of a specification that returns the sort of the same specification, matches a Java method with any return type, void included – like the case of the *pop* method that could also *top*, see section 2.3.1. All operations that don't, match a method that returns the type that corresponds to the returned sort.
- The type of the i-th parameter of a Java method must correspond to the sort in the (i+1)-th parameter of the matching specified operation.
- If a Java class is used as the n-th generic variable of a generic class, then this class must correspond to the n-th parameter sort of the parameterized specification that maps to that generic class.

For better understanding of these constraints, an example of a Java class and interface – TreeSet and IOrderable - that complies with the module SS – SORTED_SET and TOTAL_ORDER - is presented:



```
interface IOrderable<E>{
  boolean greaterEq(E e);
}

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

Fig. 13 An excerpt of a Java implementation of a sorted sort

So in this case, *SortedSet[Orderable]* corresponds to the type *TreeSet<E extends IOrderable<E>>* guaranteeing that any class that instantiates this generic type has the needed operations since it must implement the interface *IOrderable<E>*, i.e., implements a method *greaterEq(E e)*.

Refinement mapping is also used in this approach. Although similar to the one used in the previously mentioned article [5], a few modifications were made to adjust to the generic needs. Along with respecting the already declared constraints, here are a few new added constraints to the mapping process - type variables are equipped with a pre-order:

- If a parameter specification defines a sub-sort of a sort in another parameter specification, then the mapped out Java type of the first must be a subtype of the Java type that corresponds to the second.
- The Java type - interface - correspondent to a parameter specification must possess all methods that correspond to the operations and predicates of its specification.

Here is an example of a refinement mapping between the SS module and the Java types {*TreeSet<E>*, *IOrderable<E>*}:

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

Fig. 14 Refinement mapping for a Sort Set

Next a more complex refinement mapping example will be described. This mapping is between the ITV module and the Java types *TreeSet<E>*, *IOrderable<E>* and *MyInterval<E>*, *ISuccessorable<E>*. An excerpt of the Java implementation is also presented, for better comprehension:

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

Fig. 15 Refinement for an interval

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

public class MyInterval⟨E extends ISuccessorable⟨E⟩⟩{
  public MyInterval⟨E⟩(E e1, E e2){...}
  public E fst(){...}
  public E snd(){...}
  public before (MyInterval⟨E⟩ i) {...}
  public TreeSet⟨E⟩ elems(){...}
  ...
}
```
Fig. 16 Excerpt of a Java implementation of an interval

This case is good to show how a refinement mapping deals with sub-sorts, in this case *F extending E*. Since Successorable is a sub-sort of Orderable, we must also confirm that the matching subtype ISuccessorable declares the functions and methods of the supertype IOrderable, in this case just *boolean greaterEq(E e)*.

As for the domains and axioms, these are handled just like in the original ConGu approach [5]. We must also guarantee that all Java types that correspond to a parameter specification of a module, also maintain axiom coherence.

3 Automatic Test Case generation based on Algebraic Specifications

In this chapter, we talk about various methods on how to automatically generate TCs based on AS and their aptitude to fulfil the needs of the project.

First off, we describe a few aspects that should be considered when evaluating a method as a possible solution for the problem. After that, a presentation of the state of the art methods found in several articles of this area, followed by new methods developed and proposed to satisfy the project's key points of interest.

3.1 Key points of method evaluation

Out of the candidate methods that will be presented later on in this chapter, a need to choose the one that's most adequate, and best adapts to the needs of the project, is crucial.

Bellow, you can find two aspects used to determine the method that most fits the needs of the project at hand.

3.1.1 QUEST requisites

A very important aspect when considering a method as a possible solution are the project QUEST's needs.

Two main requisites of the QUEST project are taken into consideration during this selection phase. The first, which is one of the objectives of this dissertation, is to allow the implemented application to generate TCs for generic ADTs without needing the Java implementation or generic parameters being specified – offline. The second is a future functionality of the QUEST project that consists of locating the erroneous operation, in an implementation being tested, that provokes TCs to fail.

3.1.2 Test adequacy and coverage criteria

In order to verify if the generated TCs are adequate enough, standards had to be established to evaluate these TCs. Since the ideal would be to cover as many unique situations as possible, both unconditional and conditional axioms should be thoroughly tested within the defined domains. Outside of the established domains, there is no specification of how the implementation reacts, so no tests should be done there. For instance, take the following Sorted Set AS ConGu module – example commonly used by ConGu's developers for testing –, composed from the core sort SortedSet – same as Fig. 10 - and parameter sort TotalOrder – same as Fig. 8:

```
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));
        not isIn(empty(), E);
        isIn(insert(S,E), F) iff E = F or isIn(S, F);
        largest(insert(S, E)) = E if isEmpty(S);
        largest(insert(S, E)) = E if not isEmpty(S) and geq(E, largest(S));
         largest(insert(S, E)) = largest(S) if not isEmpty(S) and not geq(E, largest(S));
        insert(insert(S, E), F) = insert(S, E) if E = F;
        insert(insert(S, E), F) = insert(insert(S, F), E);
end specification
```

Fig. 17 SortedSet: AS of a Sorted Set sort in ConGu – Core Sort

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

Fig. 18 TotalOrder: AS of an Orderable sort in ConGu - parameter sort

The axiom *largest(insert(S, E)) = largest(S) if not isEmpty(S) and not geq(E, largest(S))* should not be tested if the Sorted Set's instance *S* is empty, because the domain *largest(S) if not isEmpty(S)* makes *not geq(E, largest(S))*, and *largest(S)*, impossible to determine.

Verification of the coverage of the Boolean expressions should also be taken into account. For instance, TCs should be generated in order to cover, in all the different possible ways, the Boolean conditions of the conditional axioms. For example, using the Sorted Set example, when dealing with a logical conjunction – and - in an axiomatic condition like *not isEmpty(S) and* *geq(E, largest(S))*, both *not isEmpty(S)* and *geq(E, largest(S))* terms must be true. But in the case of a logical disjunction – or -, like *E = F or isIn(S, F)*, there are 3 possible situations to turn the Boolean expression true:

- *E = F* is **true** and *isIn(S, F)* is **true**;
- *E = F* is **true** and *isIn(S, F)* is **false**;
- *E = F* is **false** and *isIn(S, F)* is **true**;

In logical disjunction cases, all three situations must hold unless there is incompatibility between the left and right expressions, in any of these combinations. For instance, sometimes both expressions may not be able to come out as true at the same time - exclusive disjunction.

Ternary conditional and biconditional axioms – when-else and iff - should also be covered. In these cases, if the condition is false another expression is implied. In the ternary conditional axiom, if the condition is false then the else expression is implied. For example, consider the axiom from the specification of a List – AS not represented -, *getLast (addFirst (L, E)) = E when isEmpty(L) else getLast(L)* where *L* is a List and *E* an Element. In this case, the bellow should be covered:

- *isEmpty(L)* is **true** and *getLast (addFirst (L, E)) = E* is **true**;
- *isEmpty(L)* is **false** and *getLast (addFirst (L, E)) = getLast(L)* is **true**;

In the biconditional axioms, when the condition is false, the remaining axiom expression should also be false. So in the case of a very common axiom amongst data structures, *isEmpty (L) iff size (L) = 0*, the following should be considered:

- *isEmpty (L)* is **true** and *size (L) = 0* is **true**;
- *isEmpty (L)* is **false** and *size (L) = 0* is **false**;

The easier it is for a TC generating method to control and provoke these cases, the better the test coverage is, and more adequate.

3.2 State of the Art

After reading several prominent articles on generating TCs based on ASs, patterns were found, and it was possible to characterise the techniques used in these articles, as specific cases of one of these three macro methods: manual scripted tests, term rewriting and variable substitution.

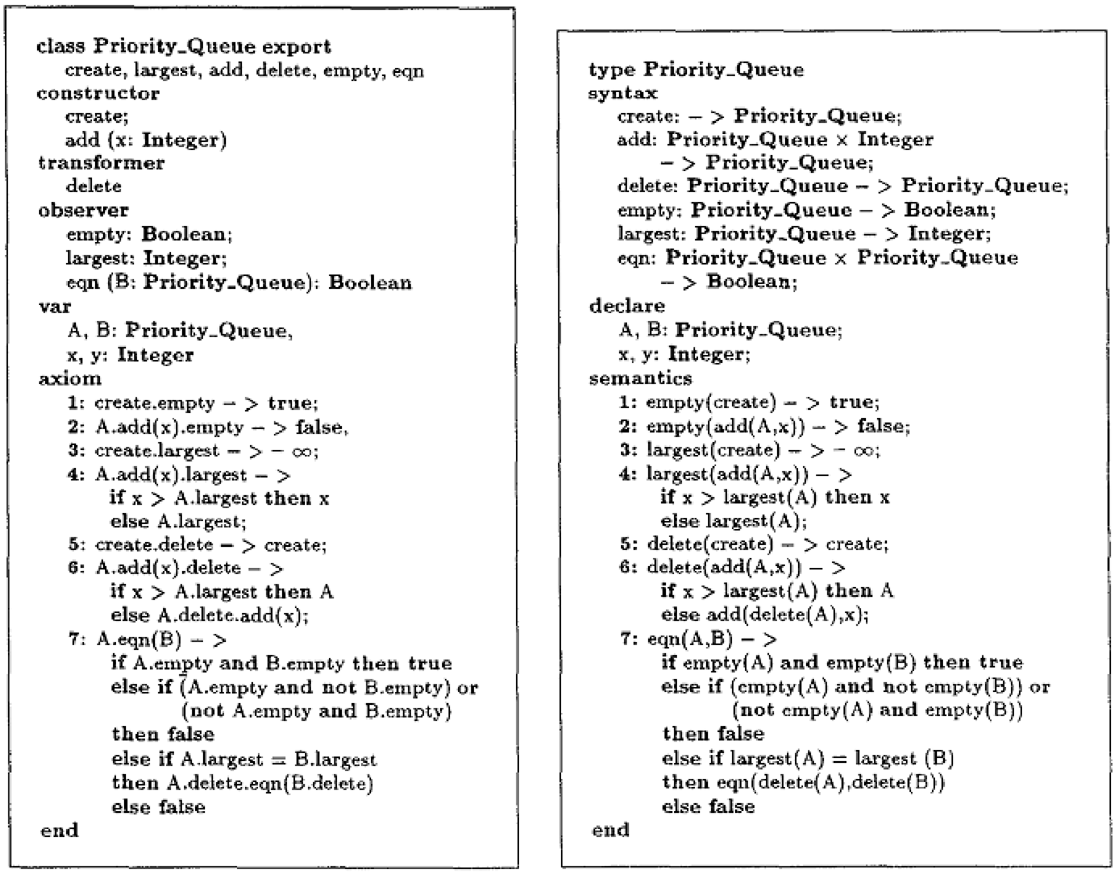
The manual scripted tests method, represented in such works as [25, 26], was found as of no interest for the case at hand because it involves too much manual labour on the behalf of the user. Not only does it go against the objectives of the project, but also makes the general testing of a data type very unreliable and error prone.

This being said, the ideology behind each of the remaining macro methods - term rewriting and variable substitution - will be described, referencing specific cases of these methods reported in articles.

3.2.1 Term Rewriting method

This method proposes that legal terms[5] be generated from a specification and then rewritten into their necessarily unique normal form terms, using the specification's axioms as rewriting rules. This way one may form TCs by checking if the legal terms generated and the normal form terms are equivalent. A specific case of this method will be detailed next for better understanding.

A representative article that reports an approach and tool for TC generating based on AS using this method is [22]. This article is considered of great importance to this AS based TC generating field due to reporting the first case using this method, and a tool developed using this specific approach. The specification language used in this approach is LOBAS. Next, a specification of a Priority Queue in LOBAS is demonstrated, along with the same specification in a functional notation, so that one may see the syntactic difference:



```
class Priority_Queue export
   create, largest, add, delete, empty, eqn
constructor
   create;
   add (x: Integer)
transformer
   delete
observer
   empty: Boolean;
   largest: Integer;
   eqn (B: Priority_Queue): Boolean
var
   A, B: Priority_Queue,
   x, y: Integer
axiom
   1: create.empty - > true;
   2: A.add(x).empty - > false,
   3: create.largest - > - ∞;
   4: A.add(x).largest - >
        if x > A.largest then x
        else A.largest;
   5: create.delete - > create;
   6: A.add(x).delete - >
        if x > A.largest then A
        else A.delete.add(x);
   7: A.eqn(B) - >
        if A.empty and B.empty then true
        else if (A.empty and not B.empty) or
                 (not A.empty and B.empty)
        then false
        else if A.largest = B.largest
        then A.delete.eqn(B.delete)
        else false
end
```

(a) Specification in LOBAS

```
type Priority_Queue
syntax
   create: - > Priority_Queue;
   add: Priority_Queue × Integer
        - > Priority_Queue;
   delete: Priority_Queue - > Priority_Queue;
   empty: Priority_Queue - > Boolean;
   largest: Priority_Queue - > Integer;
   eqn: Priority_Queue × Priority_Queue
        - > Boolean;
declare
   A, B: Priority_Queue;
   x, y: Integer;
semantics
   1: empty(create) - > true;
   2: empty(add(A,x)) - > false;
   3: largest(create) - > - ∞;
   4: largest(add(A,x)) - >
        if x > largest(A) then x
        else largest(A);
   5: delete(create) - > create;
   6: delete(add(A,x)) - >
        if x > largest(A) then A
        else add(delete(A),x);
   7: eqn(A,B) - >
        if empty(A) and empty(B) then true
        else if (empty(A) and not empty(B)) or
                 (not empty(A) and empty(B))
        then false
        else if largest(A) = largest (B)
        then eqn(delete(A),delete(B))
        else false
end
```

(b) Specification in functional notation

Fig. 19 Comparing LOBAS to a generic functional specification

Operations in LOBAS are divided, as it can be seen above, into three types: constructors, transformers and observers. The generated terms, which are afterwards rewritten and used to produce equivalent terms, are composed of constructors, transformers and, possibly, an observer at the end. This term generation is done in a random sort of fashion. By the end of the term rewriting exercise, only constructors - and primitive variables, such as integers and strings, which could be also considered constructors - should be present, explaining the difference between these and transformers. Transformers, in a normalised point of view, merely transform

[5] Terms are sequences of operations performed to a sort instance of an AS.

a sequence of constructors into another sequence of constructors. Observer operations return a class different from the class that possesses this observer operation helping to inspect the objects state. Here is an example of the term rewriting technique applied to a generated term using the previously presented specification:

$$\boxed{\begin{array}{l} \qquad \text{create.add(5)}\textbf{.add(3).delete()} \\ \xRightarrow{\text{6th axiom}} \text{create}\textbf{.add(5).delete()}\text{.add(3)} \\ \xRightarrow{\text{6th axiom}} \text{create.add(3)} \end{array}}$$

Fig. 20 Rewrite example in a Priority Queue

This rewriting transition is possible using the 6th axiom. The rewriting exercises that are performed in this technique are left-to-right, meaning that one should be looking for the left term of each axiom, in the generated term, and substituting it with the right term – normal form - of that found axiom. Axioms are first searched from the left side to the right.

The way of two objects verifying their equivalence is by checking if they are observationally equivalent[6]. This should be simulated, according to this article, by running the same sequence of operations – normally transformers – to each object, ending with an observer operation – this is called an observation. If the return values of both objects are considered equal, then more observations are made. If all observations check out correctly, we consider these objects observationally equivalent. This simulated observational equivalence is arranged by a Boolean operation *eqn* defined in the specification, as can be seen in the example of Fig. 19.

The TCs are derived from this observationally equality relation of two objects. TCs are presented as all 3-tuples (S_1, S_2, Tag) where S_1 and S_2 are the terms and Tag is the relation. Examples of TCs using the Priority Queue specification:

(create.add(5) .add(3).delete, create. add(3), equivalent)
(create.add(5) .add(3).delete, create.add(5), not-equivalent)
(create.add(5).add(3).delete.largest, 3 , equivalent)

Although this article brings a very interesting approach to the table, it has a few issues. For instance, take this axiom that could be considered a valid Priority Queue axiom:

A.adds(x).adds(y) -> A.adds(y).adds(x)

This axiom would have presented some problems due to the existing loop - recursiveness - in the term rewriting exercise – no unique normal form. Plus, a TC like the following one would not be generated:

(create.add(5).add(3), create.add(3).add(5), equivalent)

To resolve this problem one could, either add some additional manually produced TCs, write alternative axioms that contour the recursive axiom or support branching when reducing terms, but even then there could be problems. This reported case does not consider domains and uses conditional axioms to counter this need. For example, instead of saying *create.delete* is out of the domain, in the case of the Priority Queue, *create.delete* does not alter the *create* state.

In this specific case of this method, conditional TCs may cause errors to occur since the conditions are not processed properly. For example, having a Priority Queue with a limit of elements may turn two terms, which normally would be considered equivalent, not-equivalent. Take, for instance, a Priority Queue that has a condition applied to his *add* operation limiting

[6] Observationally equivalent objects are those that are in the same “abstract state”, i.e., looking at the objects as black-boxes, they respond in the same way when applied the same operations to both.

the number of elements in the Priority Queue by 2. The *eqn* operation in this case would consider the following TC right when actually it is not:

(create.add(5).add(3).add(2).delete, create.add(3).add(2), equivalent)

Better interpretation of the conditions would be required or a more complex *eqn* operation, which is not defined in ConGu's case.

Having said this, we can conclude that the Term Rewriting method, although good to test complex terms with several axioms, lacks the ability, originally, to test axioms that are recursive and, if not properly managed, axiomatic conditions may cause the tests to fail, giving out the idea that the implementation is wrong when it is not. Also, since this method consists of creating several terms and then rewriting them - term orientated –, it is difficult to ensure that all combinations of the Boolean expressions of the axioms are covered, as mentioned earlier in this chapter – see section 3.1.2.

Being this method term orientated can make it more difficult to locate errors – axiom violated - in implementations since several axioms might have been used to rewrite the term that generated the TC that failed. As for a possible offline generation, this seems to be difficult due to the generic variables, and their interfaces, that obey a certain parameter specification. For example, in order to test the Sorted Set in Fig. 17, the elements which are introduced, should implement an interface than translates the method greater-or-equal-than, respecting the axioms from the specification in Fig. 18, so that the Sorted Set may know how to deal with them. So in this case, real tested classes that implement the parameter sort's interface must be specified, and told how to instantiate, at the beginning in order to be able to use them and test the Priority Queue.

3.2.2 Variable Substitution method

This method suggests that one generates terms and primitive type instances, substituting the variables of the axioms that are being tested with these. This will turn the axioms into TCs.

Exemplifying with an axiom of a Stack, a Stack term and an element like these ones:

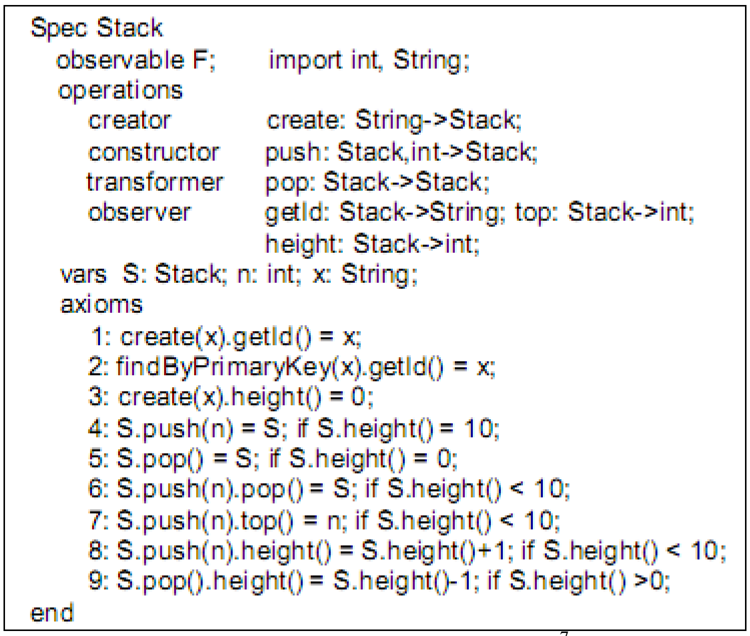
Axiom: Stack s, Elem e; s.push(e).pop() = s;
Stack s: newStack.push(1);
Elem e: 3;

We could end up with a TC like this:

Test case: newStack.push(1).push(3).pop() = newStack.push(1);

There were several articles researched, namely [14, 16, 18-21], that use this method. Next, a specific case of this method, used in the paper [14], will be described. This case was chosen mainly since it is the most recent, tests ADTs and is the most automated. Not so important is the fact that it already produces the TCs in Java.

This approach uses a language called CASOCC – Common Algebraic Specifications of Components and Classes – and the representation of the axioms is very similar to LOBAS's notation – talked about in the previous section 3.2.1. This defined language allows the importation of primitive types like byte, short, int, long, float, double, char, String and Boolean. Here is an example of a Stack, with a maximum allowed size of 10, specified in this language, for better visualization:



```
Spec Stack
  observable F;      import int, String;
  operations
    creator          create: String->Stack;
    constructor      push: Stack,int->Stack;
    transformer      pop: Stack->Stack;
    observer         getId: Stack->String; top: Stack->int;
                     height: Stack->int;
  vars  S: Stack; n: int; x: String;
  axioms
    1: create(x).getId() = x;
    2: findByPrimaryKey(x).getId() = x;
    3: create(x).height() = 0;
    4: S.push(n) = S; if S.height() = 10;
    5: S.pop() = S; if S.height() = 0;
    6: S.push(n).pop() = S; if S.height() < 10;
    7: S.push(n).top() = n; if S.height() < 10;
    8: S.push(n).height() = S.height()+1; if S.height() < 10;
    9: S.pop().height() = S.height()-1; if S.height() >0;
end
```

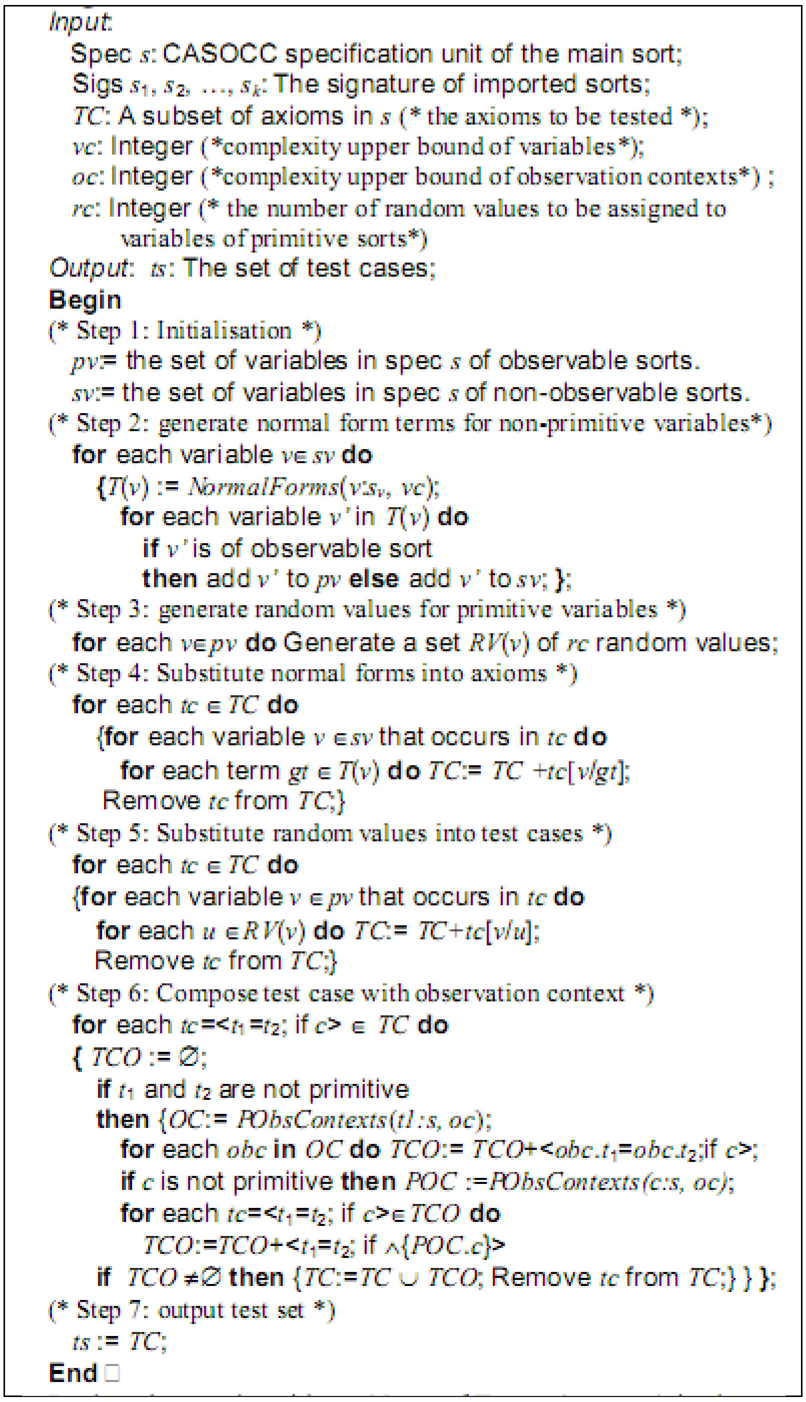
Fig. 21 Stack in CASOCC [7]

As it can be seen, the operations era divided into four types, unlike the language from the previous technique. In this case, the considered constructors in the previous Term Rewriting method are here divided into 2 categories: creators and constructors. Explanation of these groups follows:

- Creators: creates and initialises instances of the main sort. They must not contain any parameters of the type of the sort being specified, but return this type. All terms start with one of these operations.
- Constructors: constructs the data structure adding new data elements to it. They must contain a parameter of the type of the sort being specified and return the same type. May occur in normal form terms.
- Transformers: manipulates the data in the data structure without adding new elements to the data. They must contain a parameter of the type of the main sort and return the same type, just like the constructors. Does not occur in normal form terms.
- Observers: allows the internal state of the data structure to be observed. It must contain a parameter of the type of the main sort and return an imported sort.

Next, the algorithm used by the article will be presented, followed by a detailed description of it. Due to practical reasons of instantiating primitive variables in an axiom, some random testing is introduced. Here is the algorithm:

[7] In Fig. 21, the operation 'findByPrimaryKey' is not defined. At a syntax level, this operation accepts a String variable as input and returns a Stack.



```
Input:
  Spec s: CASOCC specification unit of the main sort;
  Sigs $s_1, s_2, \ldots, s_k$: The signature of imported sorts;
  TC: A subset of axioms in s (* the axioms to be tested *);
  vc: Integer (*complexity upper bound of variables*);
  oc: Integer (*complexity upper bound of observation contexts*) ;
  rc: Integer (* the number of random values to be assigned to
      variables of primitive sorts*)
Output: ts: The set of test cases;
Begin
(* Step 1: Initialisation *)
  pv:= the set of variables in spec s of observable sorts.
  sv:= the set of variables in spec s of non-observable sorts.
(* Step 2: generate normal form terms for non-primitive variables*)
  for each variable $v \in sv$ do
    {T(v) := NormalForms(v:$s_v$, vc);
      for each variable v' in T(v) do
        if v' is of observable sort
        then add v' to pv else add v' to sv; };
(* Step 3: generate random values for primitive variables *)
  for each $v \in pv$ do Generate a set RV(v) of rc random values;
(* Step 4: Substitute normal forms into axioms *)
  for each $tc \in TC$ do
    {for each variable $v \in sv$ that occurs in tc do
      for each term $gt \in T(v)$ do TC:= TC +tc[v/gt];
     Remove tc from TC;}
(* Step 5: Substitute random values into test cases *)
  for each $tc \in TC$ do
  {for each variable $v \in pv$ that occurs in tc do
    for each $u \in RV(v)$ do TC:= TC+tc[v/u];
    Remove tc from TC;}
(* Step 6: Compose test case with observation context *)
  for each tc=<$t_1$=$t_2$; if c> $\in$ TC do
  { TCO := $\varnothing$;
    if $t_1$ and $t_2$ are not primitive
    then {OC:= PObsContexts(t1:s, oc);
      for each obc in OC do TCO:= TCO+<obc.$t_1$=obc.$t_2$;if c>;
      if c is not primitive then POC :=PObsContexts(c:s, oc);
      for each tc=<$t_1$=$t_2$; if c>$\in$TCO do
        TCO:=TCO+<$t_1$=$t_2$; if $\wedge${POC.c}>
    if TCO $\neq\varnothing$ then {TC:=TC $\cup$ TCO; Remove tc from TC;} } };
(* Step 7: output test set *)
  ts := TC;
End □
```

Fig. 22 Variable substitution's algorithm [14, 21]

The complexity of the generated TCs are defined by the variables **vc**, **oc** and **rc**. In order to demonstrate the transformation of the variables of this algorithm, we will take as an example, the Stack specification mentioned above. Considering all complexity values as 2, at the beginning we would have:

TC = { create(x).getId() = x,

findByPrimaryKey(x).getId() = x,

create(x).height() = 0,

if S.height() = 10 then S.push(n) = S,

if S.height() = 0 then S.pop() = S,

if S.height() < 10 then S.push(n).pop() = S, ...};

vc = 2; oc = 2; rc = 2;

ts = {};

First, variables and normal forms are generated and prepared. In step 1, two sets are created: **pv** with the observable variables – primitive - of the specification and **sv** with the non-observable variables. In the example, we would have:

pv = {n,x}; sv = {S};

In step 2, for each variable **v** of **sv**, a set **T(v)** with all the normal forms, formed with a number of operations in between 0 and **vc**, are generated. All the variables, existent in these forms are added to **pv** and **sv** depending on whether they are observable or not. In the previous example, the state would be:

T(S) = {create(str1),

create(str2).push(int1),

create(str3).push(int2).push(int3)};

pv = {n, x, str1, str2, int1, str3, int2, int3};

sv = {S};

In step 3, for each observable sort **v** in **pv**, a set **RV(v)** is generated with **rc** random values. In the demonstration, we would have:

RV(n) = {0, 1}; RV(x) = {"a", "b"};

RV(str1) = {"c", "d"}; RV(str2) = {"e", "f"}; RV(int1) = {2, 3};

RV(str3) = {"g", "h"}; RV(int2) = {4, 5}; RV(int3) = {6, 7};

After this, substitutions are made to the variables of the axioms and normal forms created in step 2, in order to generate TCs. In step 4, for each non-observable variable **v** of the set **sv**, that occurs in an axiom **tc** in the set **TC**, new and complex axioms are generated and added to **TC** by replacing the non-observable variable **v** with the normal forms - **gt** - that one gets from the set **T(v)**. After a simple axiom **tc** generates these new complex axioms, **tc** is removed. Exemplifying, we would have:

TC = {create(x).getId() = x,

findByPrimaryKey(x).getId() = x,

create(x).height() = 0,

if create(str1).height() = 10 then create(str1).push(n) = create(str1),

if create(str2).push(int1).height() = 10 then create(str2).push(int1).push(n) = create(str2).push(int1), …};

In step 5, for each observable variable **v** of the set **pv**, that occurs in an axiom **tc** in the set **TC**, generated TCs are added to **TC** by replacing the observable variable **v** with the values - **u** -

that are found in set **RV(v).** After a complex axiom **tc** generates these TCs, it is removed. Example:

TC = { create("a").getId() = "a",

create("b").getId() = "b",

…,

create("b").height() = 0,

if create("c").height() = 10 then create("c").push(0) = create("c"),

if create("c").height() = 10 then create("c").push(1) = create("c"), …};

Thirdly, observation contexts[8] are applied to both terms of the equations of the TCs as a way of checking if the axiom *<t_1 = t_2 ; if c>*[9] , that generated that TC, stands. In step 6, for each TC *tc = <t_1 = t_2 ; if c>* in the set **TC** if t_1 and t_2 are not primitive types then:

- The set **OC** becomes a set with all the observation contexts, of complexity in between 0 and **oc**, for sorts of the same type as t_1 - if it's not a primitive type.
- For each observation context **obc** of the set **OC**, the TC *<t_1 .obc = t_2 .obc; if c>* is added to **TCO**.
- If **c** isn't of a primitive type, then the variable **POC** is instantiated with all the observation contexts, of complexity in between 0 and **oc**, for sorts of the same type as **c**.
- For each TC *<t_3 = t_4 ; if cond>* of the set **TCO**, the TC *<t_3 = t_4 ; if Λ{ cond .POC }s>* is added to the set **TCO**.
- If **TCO** isn't empty then it is added to **TC** and **tc** is removed from **TC**.

Example:

TC = { create("a").getId() = "a",

…,

if create("c").height() = 10 then create("c").push(0).top() = create("c").top(),

if create("c").height() = 10 then create("c").push(0).height() = create("c").height(),

if create("c").height() = 10 then create("c").push(0).pop().getId() = create("c").pop().getId()

, if create("c").height() = 10 then create("c").push(0).pop().top() = create("c").pop().top(), …};

At last, the set of TCs are returned in step 7. Final demonstration would be:

Ts = TC;

In conclusion, when compared to the previous method, the Variable Substitution method does not produce complex TCs with several axioms but it does allow recursion, plus conditional

[8] Observation contexts are terms made up of a sequence of transformers and ending with a call of an observer operation. The complexity of these contexts defines the number of transformers in the resulting term.

[9] t_1 = t_2 should be true if *c* is true and false if *c* is false.

axioms cause no problems. This method is axiom orientated in the sense that each test tests one specific axiom, making error locating in a Java implementation easier. With this method there is no way of controlling the adequacy and coverage of the tests generated, since there is no guarantee that all the combination of Boolean expressions are covered – see section 3.1.2.

Although this method is one of the best to detect the fault in an implementation, since it only exercises one axiom per TC, it fails in the ability of offline generation. This is so because, like in the previous method, it requires at least one class that implements the parameter sort's interface, and instantiations of it, to automatically generate TCs.

3.3 New methods

In this section we present, to our knowledge, two new methods thought out to tackle the problem at hand.

The first method of the two makes use of the CafeOBJ [27] system's reduce operation, while the second method takes advantage of Alloy Analyzer's [28-30] – AA - model instancing capabilities.

3.3.1 CafeOBJ Reducing method

CafeOBJ [27, 31, 32] is one of the most famous algebraic languages and the direct successor of OBJ programming language introducing some new major enhancements in AS theory and practice. Some of these improvements include embedding new paradigms such as behavioural concurrent specification and rewriting logic. CafeOBJ allows one to support the development process of systems at several levels, including prototyping, specification, and formal verification.

Although many techniques are mentioned about using CafeOBJ to generate TC templates [33, 34], these require manual labour to complete them and none produce "ready to use", and satisfyingly adequate, TCs. The method planned out was to create a finite state machine – FSM - from the AS we would like to base the tests in, using CafeOBJ's reduce command [32]. The reduce command takes a term and applies a sophisticated term rewriting system in a similar fashion as the left-to-right Term Rewriting method described earlier – see section 3.2.1 –, but possessing a better conditional accepting rewriting method. This methods approach is more state-orientated, unlike the Term Rewriting method which is term orientated.

In these FSM, nodes represent the "abstract state" of the sort being tested and the connections represent the transitions between these nodes when *constrops*[10], or *transops*[11], are applied and the resulting terms reduced to a normal form.

In this method, there would be a complexity variable that would define the size of this FSM. Here is an example of a generated FSM for a Stack with complexity 2:

[10] *Constrop* is a term, created in the scope of this thesis, to indicate a ConGu AS operation that categorizes as a constructor in the CASOCC language described in section 3.2.2.

[11] *Transop* is a term, created in the scope of this thesis, to indicate a ConGu AS operation that categorizes as a transformer in the CASOCC language described in section 3.2.2

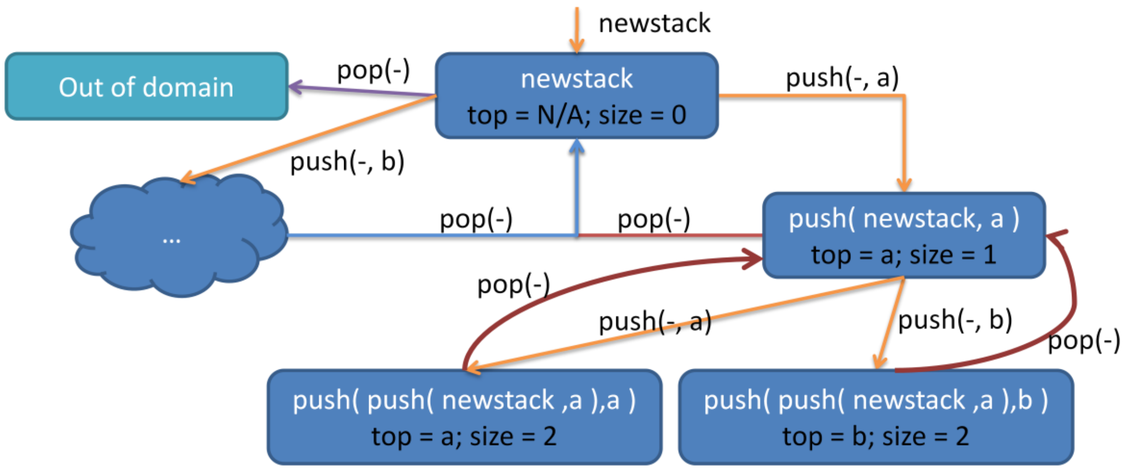


Fig. 23 FSM of a Stack from CafeOBJ's reducing method

TCs would be extracted by leading different paths of the FSM - applying *constrops* and *transops* – and checking if we are in the correct "abstract state". Here are 2 examples of TCs using the previously generated Stack FSM:

Stack implState = newStack.push(a).push(b).pop().pop();

Stack cafeOBJState = newStack;

int cafeOBJStateSize = 0;

Assert(ObservationallyEquals(implState, cafeOBJState))

Assert(implState.size() == cafeOBJStateSize)

The **cafeOBJState** and **cafeOBJStateSize** variables are created by looking at the **implState** variable and making it go through the state machine.

This method has a few of foreseen issues. One of them is, since the reduce command is a black-box, knowing which of the axioms have been covered by the term rewriting system is difficult, meaning one cannot know if the collection of TCs generated are adequate or gives us a good coverage – there is a need to modify the reduce command to monitor which axioms are applied. Another issue is the fact that, since it's a more state orientated method, and not so much axiom orientated, it will make it difficult to, later on, develop a way to pinpoint the error – axiom violated - in an implementation, if it doesn't pass the tests. Furthermore, this solution does not allow a completely offline TC generation since it would be required to specify classes, and instantiate objects, which contain the interfaces specified by the parameter sorts used by the module being tested to employ in the generation of the FSM and TCs – same problem as the previous two methods.

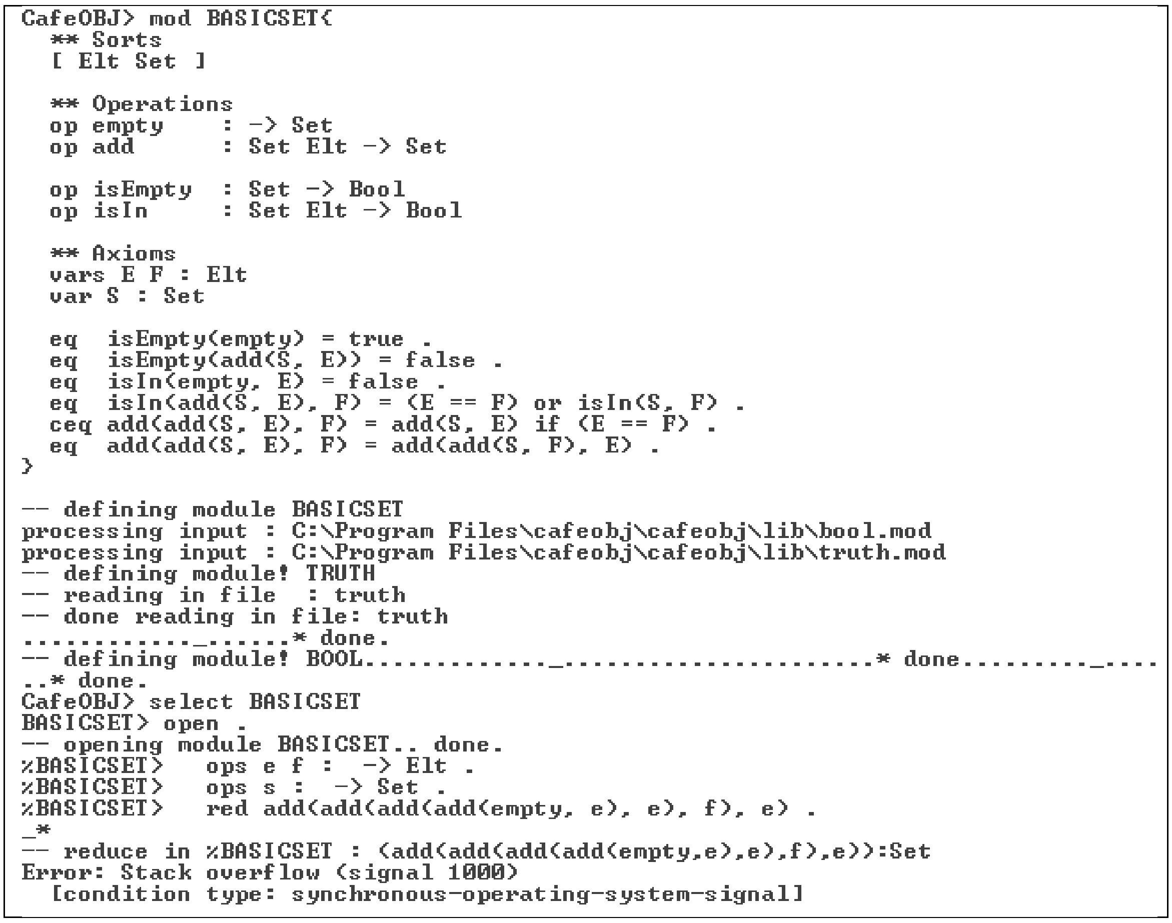
Finally, there was also a problem found when testing this approach with recursive axioms, like with the Term Rewriting method in section 3.2.1. For instance, CafeOBJ was tested with the following specification of a Set:

```
mod BASICSET{
    ** Sorts
    [ Elt Set ]

    ** Operations
    op empty    : -> Set
    op add      : Set Elt -> Set
    op isEmpty  : Set -> Bool
    op isIn     : Set Elt -> Bool

    ** Axioms
    vars E F : Elt
    var S : Set

    eq  isEmpty(empty) = true .
    eq  isEmpty(add(S, E)) = false .
    eq  isIn(empty, E) = false .
    eq  isIn(add(S, E), F) = (E == F) or isIn(S, F) .
    ceq add(add(S, E), F) = add(S, E) if (E == F) .
    eq  add(add(S, E), F) = add(add(S, F), E) .
}
```
Fig. 24 AS of a simple Set in CafeOBJ

The *eq add(add(S, E), F) = add(add(S, F), E)* axiom , which helps to complete the idea behind the *ceq add(add(S, E), F) = add(S, E) if (E == F)* axiom, provokes a loop when it tries to rewrite two operations *add* that follow each other. As an example, the following expression was given to CafeOBJ to reduce:

add(add(add(add(empty, e), e), f), e) .

Just like in the Term Rewriting method, this solution does not react well. Here is a screen printing of the test:



```
CafeOBJ> mod BASICSET{
  ** Sorts
  [ Elt Set ]

  ** Operations
  op empty     : -> Set
  op add       : Set Elt -> Set

  op isEmpty   : Set -> Bool
  op isIn      : Set Elt -> Bool

  ** Axioms
  vars E F : Elt
  var S : Set

  eq  isEmpty(empty) = true .
  eq  isEmpty(add(S, E)) = false .
  eq  isIn(empty, E) = false .
  eq  isIn(add(S, E), F) = (E == F) or isIn(S, F) .
  ceq add(add(S, E), F) = add(S, E) if (E == F) .
  eq  add(add(S, E), F) = add(add(S, F), E) .
}

-- defining module BASICSET
processing input : C:\Program Files\cafeobj\cafeobj\lib\bool.mod
processing input : C:\Program Files\cafeobj\cafeobj\lib\truth.mod
-- defining module! TRUTH
-- reading in file  : truth
-- done reading in file: truth
............_......* done.
-- defining module! BOOL............._.......................* done........._....
..* done.
CafeOBJ> select BASICSET
BASICSET> open .
-- opening module BASICSET.. done.
%BASICSET>   ops e f :  -> Elt .
%BASICSET>   ops s :  -> Set .
%BASICSET>   red add(add(add(add(empty, e), e), f), e) .
_*
-- reduce in %BASICSET : (add(add(add(add(empty,e),e),f),e)):Set
Error: Stack overflow (signal 1000)
  [condition type: synchronous-operating-system-signal]
```
Fig. 25 Recursivity error in CafeOBJ

As seen above, a *Stack overflow* error occurs due to the continuous rewriting exercise.

3.3.2 Alloy Analyzer's Modelling method

Alloy [28, 30, 35] is a textual, first-order relational logic based, declarative modelling language. Alloy specifications consist of:

- Signatures: Entities of the system being specified;
- Relations: Relations between the signatures;
- Facts: Constraints to signatures and relations that always apply;
- Predicates and Functions: Constraints to signatures and relations that apply when used;

Alloy Analyzer [35] – AA - is a finite model finding tool that fully interprets and analyses Alloy specifications, providing two main functionalities: simulation and checking. The simulation functionality generates a random finite model instance, conforming to the specification at hand, helping to ensure that the specifications created are not inconsistent. As for the checking functionality, this one is used to verify if an assertion – constraint to signatures and relations – is verified in the given specification, generating a random finite counterexample instance model if it doesn't. For both of these functionalities one needs to specify the scope to which the generated finite model should be bound to, or the default scope will be used. Ultimately, what AA does is reformulate Alloy specifications into Boolean expressions, so that afterwards they can be analysed by the tool's integrated SAT solvers[12].

In order to best understand Alloy, here is a simple Alloy specification of a Person followed by an explanation:

```
//Signature
sig Person{
        dad: lone Person,
        mum: lone Person
}

//Facts
fact imNotMyAncestors{
        all p:Person | p not in p.^(mum + dad)
}

fact myFemaleAncestorsAreNotMyMaleAncestors{
        all p:Person | no (p.^mum & p.^dad)
}

//Checks and Runs
assert imNotMyOwnMum{
        no p:Person | p = p.mum
}check imNotMyOwnMum

assert myMumIsNotMyDadsMum{
        all p:Person | one p.mum and one p.dad.mum implies p.mum != p.dad.mum
}check myMumIsNotMyDadsMum

run {}
```

Fig. 26 Alloy specification of a Person

[12] SAT solvers try to find values for variables of a Boolean expression that satisfy it. A Boolean expression is satisfied when it returns true.

Both *mum* and *dad* define one or none unidirectional relations between a Person instance and another. The *imNotMyAncestors* fact ensures that every Person instance does not belong to the set of Person instances composed by their own *mum*, *dad* and their *mums* and *dads* and so on and so forth. The *myFemaleAncestorsAreNotMyMaleAncestors* fact, states that for all Person instances the set composing their female ancestors – *mum*, *mum*'s *mum*... - does not intersect the set composed by the male ancestors – *dad*, *dad*'s *dad*....

Now the two asserts presented can be tested using the checking functionality. The *imNotMyOwnMum* assert verifies if, given the specification above, there is no case when a Person instance is its own mother. No counterexamples are generated. But now, when checking the *myMumIsNotMyDadsMum* assert, a counterexample is found because there is no fact that stops a Person instance and its *dad* having a *mum* relation with the same Person instance - *one p.mum and one p.dad.mum* is added to insist that these relations exist. Here is the resulting counterexample model instance generated in AA when checking the *myMumIsNotMyDadsMum* assert:

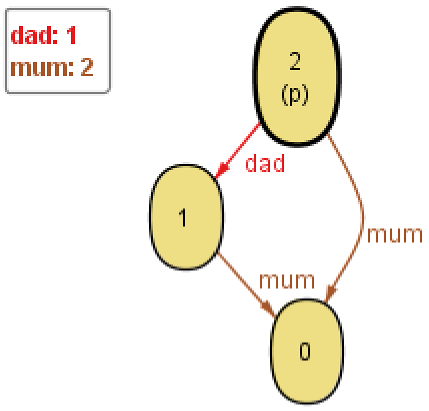


Fig. 27 *myMumIsNotMyDadsMum* assertion check on a Person specification

As you can see Person instance 2 is where the counterexample occurs, for it is represented with a *p* that signifies the *p* variable in the assertion. As for the Simulation functionality, that operates the *run* instructions, this would simply generate unconditional model instances.

Now that the ground has been prepared, the method will be explained. The idea behind this method is to translate the ASs into Alloy, generate some model instances and use these to form TCs – the manner of execution is explained in the next chapter. No article was found about using Alloy as a means to automatically generate TCs derived from ASs. This method is model/state orientated, for tests are made according to the models generated by AA, that end up by being FSMs similar to the ones in the CafeOBJ Reducing method – see section 3.3.1 -, meaning TCs can be extracted the same way.

Although no automatic method was found, with rules to translate ASs into Alloy specifications, article [36] talks about using AA as a means to detect flaws in ASs. Concluding, this Alloy method will also help us verify the consistency of ASs, giving this method an added value.

One of the great challenges of this method was finding the right set of translating rules that made the resulting Alloy specifications conform with the original ASs. Even with a revision of the generated Alloy specifications, some errors were only found after generating several model instances. Plus, not only was it necessary to see if the models made sense, but also had to verify that all unique, and complex, cases inside the domain were tested.

Several examples of Alloy specifications automatically translated from AS can be found in Appendix A. Here is an example of a generated model instance represented graphically from the Alloy specification that was automatically produced from the Sorted Set AS module – composed by Fig. 17 and Fig. 18:

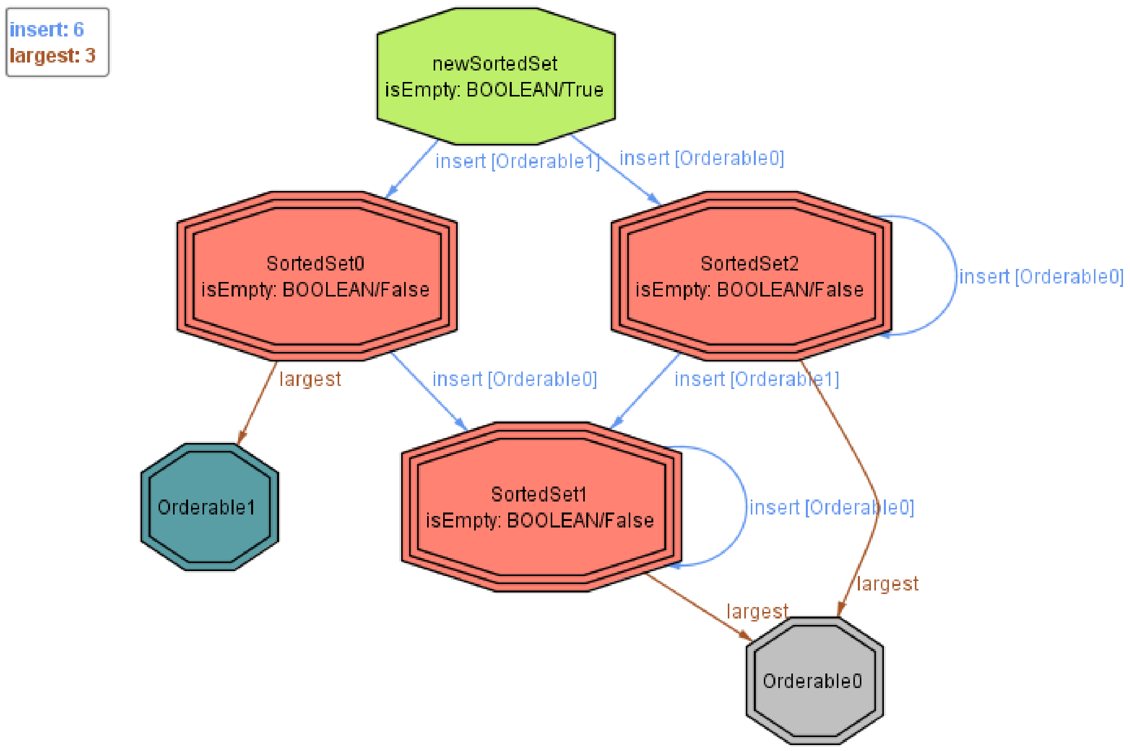


Fig. 28 Model instance of a Sorted Set in AA

Unlike Term Rewriting and CafeOBJ Reducing methods, this method allows recursion in axioms and the TCs can be generated completely offline from any implementation, not needing any instantiating, or specifying, of types that utilise certain interfaces. Once the ConGu's AS parameter sorts are translated into Alloy signatures, AA can generate instances of these signatures in model instances. The idea is to translate each parameter sort into a generated basic Java class, and the generated instances of the Alloy signature that represents that parameter sort, into objects of the basic Java class. Along with the translation of the Alloy signatures instances into objects, the static relations between these signature instances with others are also translated into the Java objects. Then, each basic Java class may be used as the generic variable of the generic Java classes being tested, and use the objects of the basic Java classes in the TCs.

Although this method isn't axiom based, which can make it difficult to localise the error in a Java implementation during a posterior phase, a technique using the checking functionality - counterexample technique - can be used to provoke the axioms we want to test. This technique consists of creating asserts that say the opposite of what the axioms declare, provoking AA to generate counterexample instances where the axiom is verified, giving out a better idea of which axioms are not acting accordingly. So in the case of the axiom *isEmpty(empty())*, we would ask for a counterexample model instance for the assertion *not isEmpty(empty())*. This means someone can choose to test extensively a specific axiom. Plus, the counterexample generation technique can also be used to provoke all the different combinations of the Boolean expressions existent in the axioms, guaranteeing TCs with the full coverage and adequacy defined in section 3.1.2.

This method takes care of all the key points mentioned in section 3.1. This method's only found issue is the time that it may take to generate a model instance, if too many variables are involved, giving it some scalability issues.

4 Test Case Generation from Algebraic Specifications with Alloy

In this chapter we present the method that was chosen to employ in the dissertation and QUEST project, and a description of the developed application and execution decisions.

After considering evaluating all the methods presented in chapter 3, the method considered less limited and better suited to the QUEST project's needs was Alloy Analyzer's Modelling method. This method is the only one that can guarantee a full coverage of the axioms, considering the criteria chosen in section 3.1.2, and a complete offline TC generation.

Said this, an explanation of the developed application, which translates ASs into Alloy, and execution decisions will be given, followed by the plan to extract TCs from the model instances produced from the generated Alloy specifications.

4.1 ConGu to Alloy translation rules

This chapter talks about the conversion rules to obtain an Alloy specification from ASs, of a ConGu module, used in the developed application, along with the decisions made to define these rules. Three examples of Alloy specifications generated with the developed application can be found in Appendix A, for better understanding of these rules. After this, the tool's architecture is briefly described.

Obviously the ASs used are in the ConGu format – check section 2.3 -, and the GonGu's compiler was used to analyse the ASs. Many ways of translation were tested and, by the end, this one was evaluated as the best path and set of rules.

Things to take into consideration before advancing are:

1. The resulting Alloy specification should be satisfiable by finite models in order to enable AA to find model instances;
2. The resulting Alloy specifications should guarantee consistency according to the AS;
3. There should be the least amount possible of unnecessary instances of parameter signatures[13], in the generated Alloy model instances - unused by the core specifications.

[13] Parameter signature is a term, created in the scope of this thesis, to indicate an Alloy signature generated from parameter sort.

Not filtering models with these pointless signature instances will only generate models, which its useful part is isomorphic when compared to the useful part of some other models. Plus, if this constraint is not made, more variables are generated and resources are used up.

4. The core sorts of a ConGu module require a complete set of constructor operations in order to be tested in a Java implementation, as it would be expected. Complete in the sense that a sort should have a way to instantiate itself - *creatop*[14] - and should be able to transition to other states – *constrop*[15]. At least these two types of constructors should exist in order to, later on, extract TCs from the Alloy model instances.

The first two points may conflict if not careful, for one is required to manipulate the AS in order to make finite Alloy Specifications, but one must also consider that if relations between signature instances are loosened up too much, the higher the probability it is of losing consistency.

Next, a description of how the sorts, operation domains and axioms, from ASs of ConGu modules, are translated to Alloy, along with the way the set of checks and run commands are created. After this, the application's architecture is briefly explained.

4.1.1 Signatures

The Sort of an AS is translated into a signature in Alloy, with its operations and predicates as relations. Here is the syntactic specifications of a Sorted Set data type –see Fig. 17 for full AS – and its parameter sort Orderable - see Fig. 18 for full AS - in algebraic and Alloy forms, for comparison:

(...)
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;
(...)
	sig **SortedSet** extends Element{
	   isEmpty:one BOOLEAN/Bool,
	
	   isIn:Orderable -> one BOOLEAN/Bool,
	
	   largest:lone Orderable,
	
	   insert:Orderable -> lone SortedSet
	}
	
	one sig **newSortedSet** extends SortedSet{}

Fig. 29 Sorted Set AS/Alloy syntactic comparison

[14] *Creatop* is a term, created in the scope of this thesis, to indicate a ConGu AS operation that categorizes as a creator in the CASOCC language described in section 3.2.2.

[15] *Constrop* is a term, created in the scope of this thesis, to indicate a ConGu AS operation that categorizes as a constructor in the CASOCC language described in section 3.2.2.

(...)
sorts
    Orderable
others
    geq: Orderable Orderable;
(...)
	
	sig **Orderable** extends Element{
	    geq:Orderable -> one BOOLEAN/Bool
	}

Fig. 30 Orderable AS/Alloy syntactic comparison

As it can be seen, the signatures generated obtain the name of the sort being specified, and all the operations and predicates also give their names to the respective relation, except in the case of the *creatop – empty*, in the example. This case will be explained later. All signatures extend the Element signature which has no relations, is always a parameter signature and represents the Object class in Java.

An operation that only has one parameter – of the sort to which this operation belongs to –, in Alloy becomes a single relation to an instance of the signature that represents the output sort of the operation. An example of this case, in the Sorted Set's AS, is the *largest* operation.

An operation that has more than one parameter becomes a set of relations to instances of the signature that represent the output sort of the operation, each relation defined by the input signature instances. The signatures of input match the parameter sorts of the operation, after the sort that is being translated is subtracted from the parameters. An example of this case, in the Sorted Set's AS, is the *insert* operation.

Predicates obey these same rules except the outputs of the resulting relations are always a Boolean signature instance. An example of the first case, in the Sorted Set's AS, is the *isEmpty* predicate and an example of the second case is the *isIn* predicate.

Another rule when translating the operations is that, *constrops* and other operations that have a limited domain create lone type relations – one or none. The reason to do so for the *constrops* is to turn the specification finite, so that the AA may generate model instances. The reason to generate lone type relations from an operation with a domain is to only allow relations within the domain.

Returning to the case of the operations with zero parameters - *creatops* -, these operations represent by definition, when mapped to Java, class constructors. As mentioned before, only zero-ary Java constructors can be specified in ConGu. They are only to be found in core sorts and only once. This special case of a constructor operation generates a signature with no relations, and extends the previously defined signature – *newSortedSet*, in the example. This new sub-signature has exactly one instance of itself in every model instance generated with AA. This single signature instance is the root – referenced as root - from which all other instances of its super-signature are reached.

4.1.2 Constructive and relation restricting facts

The instances of core signatures[16] need to tighten their relations together in order for the specification to be consistent. Normally, axioms depend on each other to form the real intended outcome of the operations. So, in order to guarantee consistency in Alloy, the instances of a core

[16] Core signature is a term, created in the scope of this thesis, to indicate an Alloy signature generated from core sort.

signature should be connected through constructor relations[17] - fundamental relations - to its root sub-signature instance - *newSortedSet*, in the previous example. Another reason for this is that it is needed to know how to get to each core signature instance, in order to simulate them, later on, in the TCs.

So starting off from the root signature instance, all other instances of the root's super-signature must be reached through the constructor relations. Core signature instances achieved only by using relations that correspond to observer operations – like common *remove* operations - could turn the model instance inconsistent. So to achieve consistency, a fact is stated in the Alloy specification imposing this constraint. Here is the fact - *SortedSetConstruction* - applied to the Sorted Set signature:

```
fact SortedSetConstruction{
    all coreSort:SortedSet | coreSort in newSortedSet.*(
        {coreSort, coreSort':SortedSet | some param0:Orderable |
                coreSort -> param0 -> coreSort' in insert})
}
```

Fig. 31 Sorted Set Alloy constructive fact

In order to remove from the equation model instances with unused instances of parameter signatures, another fact is engineered. In this fact, either a parameter signature instance is being used as the input, or output, of another signature's relation or, the instance at hand, is an instance of a sub-signature of this signature. This fact also applies to the Element signature. Here are the facts - *OrderableUsedVariables* and *ElementUsedVariables* - applied to the Orderable and Element parameter signatures of the previous case:

```
fact OrderableUsedVariables{
    all orderablesort:Orderable |
            (some coreSort:SortedSet | one coreSort.isIn[orderablesort]) or
            (some coreSort:SortedSet | orderablesort = coreSort.largest) or
            (some coreSort:SortedSet | one coreSort.insert[orderablesort])
}
```

Fig. 32 Orederable restricted to relations fact

```
fact ElementUsedVariables{
    all elementsort:Element | elementsort in (Orderable + SortedSet)
}
```

Fig. 33 Element restricted to super-signature fact

As it can be seen, Orderable signature instances only exist if they are the input signature instance for the *isIn* or *insert* relation of at least one SortedSet signature instance. In the Element signature's case, since it is not used directly as the input and output signature of any relation of the SortedSet signature, and since unlike the Orderable signature it's the super-sort of all other signatures, all Element signature instances must either be an Orderable or SortedSet signature instance.

4.1.3 Axiom and domain facts

Axioms and domains – semantic part -, when translated to Alloy from ASs, become facts. Axiom facts will be explained first, followed by domain facts.

[17] Constructor relation is a term, created in the scope of this thesis, to indicate a core signature relation that corresponds to a *constrop* in ASs.

First, the variables used during an axiom fact are declared in a way that makes these variables represent all of the instances of the variable's signature. Then, before stating the axiom's expression, this expression is checked to see if any constructor relations are called upon, since they threaten the finite aspect of the specifications being generated. If there are any of these relations in the axiom, it should be stated that the axiom only applies if these relations actually exist. This is done with the use of an implication over the statement of the expression. Example follows below for an axiom of the SS specification:

	fact **axiomSortedSet0**{
(...)	all E:Orderable, S:SortedSet |
not isEmpty(insert(S, E));	one S.insert[E] implies (
(...)	not (S.insert[E].isEmpty = BOOLEAN/True))
	}

Fig. 34 Comparing a non-conditional axiom in algebraic and Alloy specifications

The implication, in this case, is that if the relation *insert[E]* exists for a SortedSet signature instance *S*, the axiom applies when using *S* and *E* as the SortedSet and Orderable instances. If any other constructor relations existed in this axiom, their conditions of existence would be appended to the *one S.insert[E]* condition of existence, transforming the expression into a logical conjunction – and.

As it can be seen, we are characterizing a relation that originated from a predicate - *isEmpty* -, which means the relation's output is a Boolean instance – True or False. So, in this case, the output of this relation is checked to see if it is equal to the domain of the Boolean's sub-signature True singleton.

A few more considerations should be made when checking for constructor relations. For example, when dealing with an equality expression, an exception is made and only the right side of the expression is checked – the least normalised side. This is done for consistency sake, because if constructor relations of both sides need to exist in order for this axiom to apply to any given situation, this axiom could be ignored when it shouldn't. Here is a good axiom example, of this case, from the Sorted Set module:

	fact **axiomSortedSet2**{
(...)	all E, F:Orderable, S:SortedSet |
insert(insert(S, E), F) = insert(insert(S, F), E);	one S.insert[E].insert[F] implies
(...)	((S.insert[E].insert[F] = S.insert[F].insert[E]))
	}

Fig. 35 Comparing a non-conditional equality axiom in algebraic and Alloy specifications

Yet another consideration that should be made when checking for constructor relations, is when dealing with a logical disjunction expression – or. In these cases, both sub-expressions of the logical disjunction expression are searched for constructor relations and implications are only formed if constructor relations are found in both sub-expressions. If they are, one of the constructor relations must exist in order to for the axiom to hold. No examples of this case exist in ConGu's testing set.

Conditional axioms are treated as implications. Here is an axiom example from the same module as the previous examples:

	fact **axiomSortedSet3**{
(...)	all E:Orderable, S:SortedSet |
largest(insert(S, E)) = E if isEmpty(S);	one S.insert[E] implies
(...)	(S.isEmpty = BOOLEAN/True implies

	(S.insert[E].largest = E)) }

Fig. 36 Comparing a conditional axiom in algebraic and Alloy specifications

As for the *else* and *iff* tokens of the ternary conditional and biconditional axioms, they remain unaltered in Alloy.

Now let's describe the relation domain facts. The variables used in a domain fact are also declared, in the same manner as the axiom facts, so that these variables represent all of the instances of the variable's signature. Now, when the condition expression of the domain fact of a relation evaluates to false, that lone relation becomes nonexistent. But when it evaluates to true, and it is not a constructor relation, that relation exists. When the condition expression of the domain fact of a relation evaluates as true, and it's a constructor relation, the relation might or might not exist – lone –, in order to maintain the finite aspect of the specification being generated. Here is an example of a domain fact applied to the *largest* operation of the Sorted Set data type:

(...)
largest(S) if not isEmpty(S);
(...)
	fact **domainSortedSet0**{
	    all S:SortedSet |
	        not (S.isEmpty = BOOLEAN/True) implies
	            one S.largest else
	            no S.largest
	}

Fig. 37 Comparing an operation's domain in algebraic and Alloy specifications

In this case, the *largest* relation exists for the SortedSet instance *S* if *S*'s *isEmpty* relation equals the domain of the Singleton True, and doesn't exist if the isEmpty relation doesn't equal True's domain, i.e., equals False's domain.

4.1.4 Runs and checks

The runs and checks generate the model instances needed, so that they may be interpreted and turned into TCs later – see more in section 4.2. There are two complexity variables that one may define when running the tool developed with this technique. These are the max and exact variables. The first variable defines the maximum allowed number of signature instances, including the primitive integer signature - int -, in a model instance, while the second variable, which is not always defined, specifies the exact number of instances of each core signature that should exist in a model instance.

The run functionality generates model instances that obey all the previous defined facts and the complexity variables. Next, we have an example of a run command for the Sorted Set module. The max complexity variable is set to 7 and the exact variable to 4:

```
run {} for 7 but 7 int, exactly 4 SortedSet
```

Fig. 38 Simple run command

As for the checks, these are used to generate model instances that, not only obey all the specified facts and complexity values, but also exercise certain axioms and, in some situations, specific cases of axioms. First, the axioms that involve the root instances of the core signatures are checked, for they are the base axioms, and after the other axioms. Since checks attempt to generate a model instance with a counterexample to a given assertion, the assertions should assert the opposite of the axioms. The signature instances that break the assertion in the model instance are marked by AA, as seen previously, making the detection, of the axiom being exercised, easy. This way we can generate model instances that satisfy the whole of the test adequacy plan detailed in section 3.1.2. Considering the following non-conditional axiom fact *A*, the assertion would be:

- *Not A.*

Here is a simple example of an axiom fact and the corresponding assert and check, both generated from the same Alloy specifications as the run example:

```
fact axiomSortedSet2{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E].insert[F] implies (
            (S.insert[E].insert[F] = S.insert[F].insert[E]))
}
```

Fig. 39 A non-conditional equality axiom Alloy fact

```
assert assert_axiomSortedSet2_0{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E].insert[F] implies (
            not (S.insert[E].insert[F] = S.insert[F].insert[E]))
}check assert_axiomSortedSet2_0 for 7 but 7 int, exactly 4 SortedSet
```

Fig. 40 A non-conditional equality axiom Alloy fact's assert and check

In the case of a simple conditional axiom, the assertion should be that, if the condition expression is true then, the resulting expression is false. Exemplifying with the expression *A implies B*:

- *A implies not B;*

Exemplifying with a previously generated axiom fact from the previous Alloy specifications:

```
fact axiomSortedSet4{
    all E:Orderable, S:SortedSet |
        one S.insert[E] implies (
            S.isEmpty = BOOLEAN/True implies (
                S.insert[E].largest = E))
}
```

Fig. 41 A conditional axiom Alloy fact

```
assert assert_axiomSortedSet4_0{
    all E:Orderable, S:SortedSet |
        one S.insert[E] implies (
            (S.isEmpty = BOOLEAN/True) implies not (
                S.insert[E].largest = E))
}check assert_axiomSortedSet4_0 for 7 but 7 int, exactly 4 SortedSet
```

Fig. 42 A conditional axiom Alloy fact's assert and check

When dealing with logical disjunctions - or - in an expression, all three cases mentioned in section 3.1.2 should be checked. For this reason, three different assertions should be checked. These would be, considering the expression *A or B*:

- *not(not A and B);*
- *not(A and not B);*
- *not(A and B) .*

As for ternary conditional and biconditional axioms, the cases mentioned in section 3.1.2 should also be tested. So for a ternary conditional axiom fact, denoted as *A implies B = C else B = D*, the following assertions should be checked:

- *A implies B != C;*
- *not A implies B != D.*

And for a biconditional axiom fact like *A iff B*, the assertions to be made are:

- *not A implies B;*
- *A implies not B.*

Following, a complex axiom fact from the example Alloy specification will be presented along with all the resulting assertions and checks. Since this axiom fact is biconditional that has a logical disjunction in one of its expressions, six asserts are generated to check:

```
fact axiomSortedSet1{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
        S.insert[E].isIn[F] = BOOLEAN/True iff ((E = F) or S.isIn[F] = BOOLEAN/True))
}
```

Fig. 43 A biconditional axiom with a logical disjuntion Alloy fact

```
assert assert_axiomSortedSet1_0{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            (not (S.insert[E].isIn[F] = BOOLEAN/True) implies (
                (E = F) and not (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_0 for 7 but 7 int, exactly 4 SortedSet

assert assert_axiomSortedSet1_1{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            (not (S.insert[E].isIn[F] = BOOLEAN/True) implies (
                not (E = F) and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_1 for 7 but 7 int, exactly 4 SortedSet

assert assert_axiomSortedSet1_2{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            (not (S.insert[E].isIn[F] = BOOLEAN/True) implies (
                (E = F) and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_2 for 7 but 7 int, exactly 4 SortedSet

assert assert_axiomSortedSet1_3{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            ((S.insert[E].isIn[F] = BOOLEAN/True) implies not (
                (E = F) and not (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_3 for 7 but 7 int, exactly 4 SortedSet

assert assert_axiomSortedSet1_4{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            ((S.insert[E].isIn[F] = BOOLEAN/True) implies not (
               not (E = F) and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_4 for 7 but 7 int, exactly 4 SortedSet

assert assert_axiomSortedSet1_5{
    all E, F:Orderable, S:SortedSet |
        one S.insert[E] implies (
            ((S.insert[E].isIn[F] = BOOLEAN/True) implies not (
               (E = F) and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_5 for 7 but 7 int, exactly 4 SortedSet
```
Fig. 44 A biconditional axiom with a logical disjuntion Alloy fact's assert and check

Demonstrating with the expression *A iff B or C*, we would have:

- *not A implies B and not C;*
- *not A implies not B and C;*
- *not A implies B and C;*
- *A implies not(B and not C);*
- *A implies not(not B and C);*
- *A implies not(B and C).*

4.1.5 Tool's architecture

The tool developed during this dissertation time, which translates AS to Alloy specification in the manner explained previously in this chapter, consists of 6 main classes and the ConGu compiler.

AlloyGen is the main class. This is the interface class to all the possible operation of this tool and executes them, with the help of the other 5 classes and the ConGu compiler module. You may choose the ConGu AS module you want to convert to Alloy and set the max and exact complexity variables. This class is connected to ConGu's compiler module and to the remaining 5 classes. When ConGu's Compiler module is utilised by *AlloyGen* to interpret a ConGu AS module, it creates data structures with all the information. In these data structures, the expressions of the axioms and domains are stored as parse trees generated by SableCC parser generator [37].

In order to retrieve information from the parse trees that represent the expressions of the domains and axioms, visitor classes need to be created to go through the tree. In the tool's case there are four visitor classes, and they are:

- *AlloyDomainAxiomVisitor*: These visitors go through the parse tree to form the equivalent Alloy expression to the one represented in this tree.

- *AlloyVariablesVisitor*: These visitors are used before the previous visitor to gather the variables that are used in the axiom and domain expressions, so that they may be declared beforehand.
- *AlloyLimitCrawlerVisitor*: These visitors are also used before the *AlloyDomainAxiomVisitor* visitor to check if there are any future constructor relations that will endanger the finite aspect of the specification. If there are any, these are made sure to condition the axioms or domains being dealt with, as described in section 4.1.3.
- *AlloyAssertVisitor*: These visitor classes visit only axiom expressions, generating the Alloy expressions from these used in the assertions to be checked – see section 4.1.4.

Although these visitor classes may seem too many, and it may seem that too many tree visits are made, the decision of this format was made so that the code would be more comprehensible and structured.

The tool's last class to be mentioned is the *AlloyTokens* singleton class that contains static string variables with Alloy's tokens and keywords - such as "sig", "fact", etc – used to develop the Alloy specifications.

4.2 Test generation and execution

In this section, a description of the plans to generate TCs from the Alloy specifications generated in the previous section is found. A few tests were made with AA's API [35] to get a feel of what it is like to generate instances from the simulation and checking functionalities – runs and checks –, and what is possible to do.

After researching for methods on how to translate Alloy specifications to Java TCs, a tool called TestEra [38, 39] was found which tests single methods of Java systems, online with the implementation, using Alloy. This tool is not so much for ADT testing, which changes the internal state constantly, or for defining relations between methods, like ASs do with their axioms. Basically, what this tool does is generate all the possible inputs for a Java method, using the available Alloy pre-conditions, and converting these inputs to Java - concretisation translation. Then it uses them on the Java method and converts the outputs back to Alloy – abstraction translation - to verify them with the available Alloy post-conditions of the Java method. Although the concretisation translation is quite interesting, and quite similar to the problem at hand, the idea is to translate full instances to Java and not only input variables for an online implementation, so unfortunately there is nothing much here that can be used.

Next, a graphical example of a model instance is presented where the axiom *not isEmpty(insert(S, E))*, of the Sorted Set AS, is provoked with a check. As can be seen below, and mentioned before when explaining Alloy, the variables *S* and *E* are represented automatically with *(S)* and *(E)*, giving a clear idea of where the axiom is being exercised. This is good for when the TCs are extracted. Example follows:

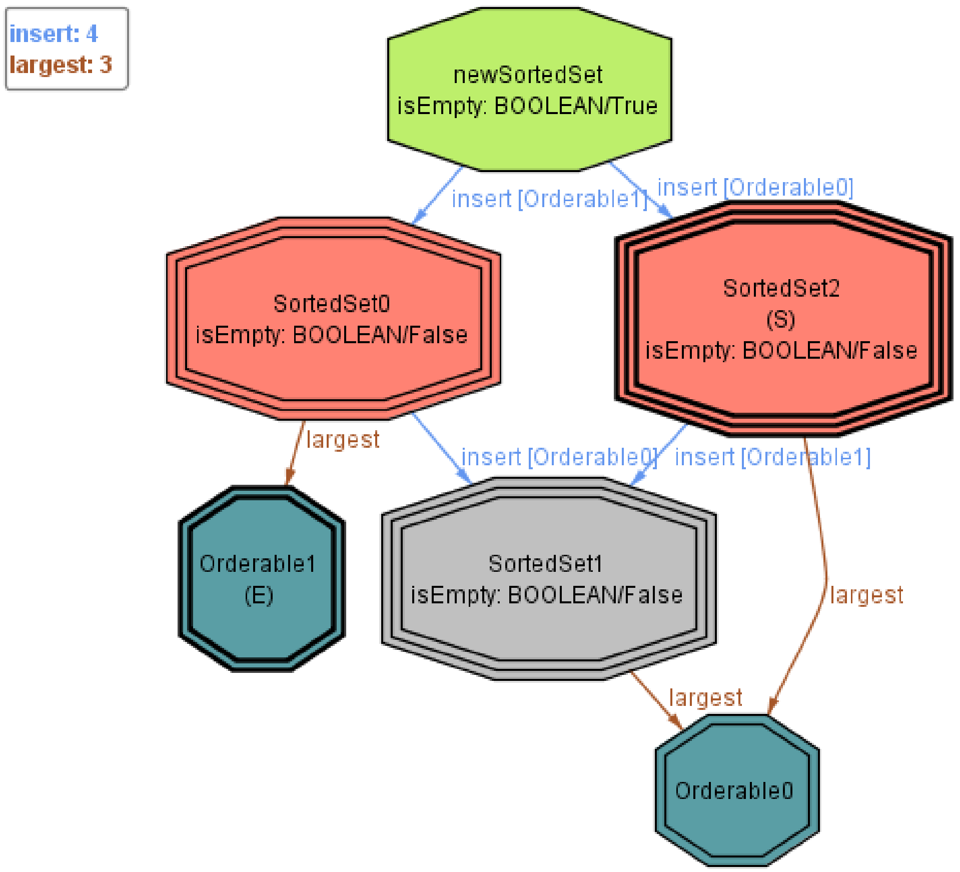


Fig. 45 Model instance with an axiom provoked

In order to generate the Java TCs from the ASs of a module, there will be a need to use the refinement mapping file that maps the ASs operations into the corresponding Java operation – see section 2.3. Using the runs and checks, it will be possible to generate model instances and, using the technique specified in section 3.3.1 by the CafeOBJ Reducing method with its FSMs, it will be possible to test extensively the implementation.

As described in section 3.3.2, a way was thought out to achieve TC generation in a complete offline manor. Next, the TotalOrder specification – described in Fig. 18 – will be used to exemplify how to extract a testing Java class and object instances, from a parameter specification and instances of its Alloy signature, in order to test a parametrized specification. Consider the following refinement mapping that maps the ASs of the Sorted Set ConGu module:

```
import generic.valid.treeSet.TreeSet;

refinement <E>
    SortedSet[TotalOrder] is TreeSet<E> {
        empty: --> SortedSet[Orderable] is TreeSet();
        insert: SortedSet[Orderable] e:Orderable --> SortedSet[Orderable] is void insert(E e);
        isEmpty: SortedSet[Orderable] is boolean isEmpty();
        isIn: SortedSet[Orderable] e:Orderable is boolean isIn(E e);
        largest: SortedSet[Orderable] -->? Orderable is E largest();
    }

    TotalOrder is E {
        geq: Orderable e:Orderable is boolean greaterEq(E e);
    }
end refinement
```
Fig. 46 Refinement mapping of the Sorted Set ConGu module

This will make the Orderable sort, defined in the TotalOrder specification, translate into the interface bellow:

```
public interface Orderable<E>
{
        public Boolean greaterEq(E e);
}
```
Fig. 47 Orderable interface

So given an interface like this, the idea is to generate a simple class that implements the interface and allows the output value of the interface's operations to be defined, given certain input parameters. The output values would be defined according to the model instances generated by the AA. Example:

```
public class OrderableExample implements Orderable<OrderableExample>
{
    private HashMap<OrderableExample, Boolean> map =
    new HashMap<OrderableExample, Boolean>();

    @Override
    public Boolean greaterEq(OrderableExample e)
    {
        return map.get(e);
    }

    public void add_greaterEq(OrderableExample e, Boolean result)
    {
        map.put(e, result);
    }
}
```
Fig. 48 Simple Java test class implementing the Orderable interface

Here we have an object that adds to a Hash Map the return values of the *greaterEq(OrderableExample e)* operation according to the input parameter values, creating a map of static reactions. So if one was to create *OrderableExample* objects and load them with the information of the model instance generated in AA presented in Fig. 49- *Orderable2* < *Orderable0* < *Orderable1* -, it could be done like shown in Fig. 50.

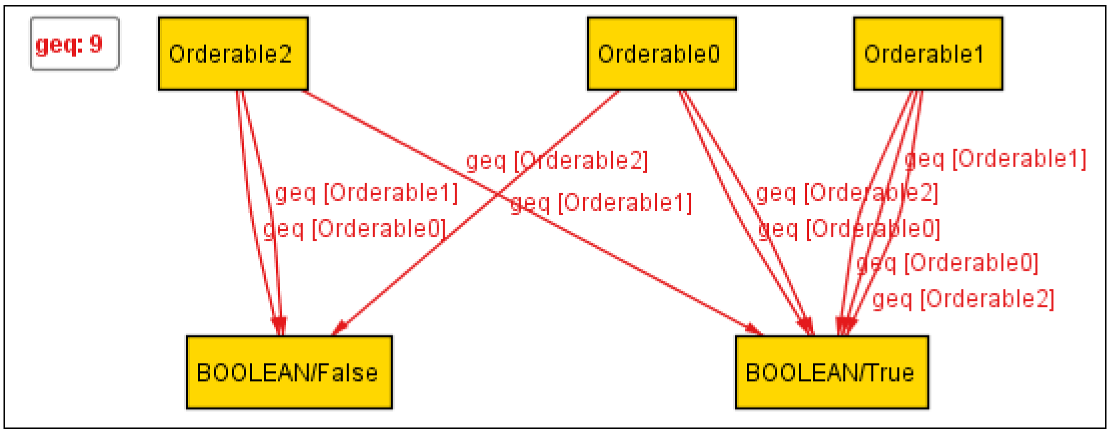


Fig. 49 Orderable interaction model instance

```
(...)
OrderableExample Orderable0 = new OrderableExample();
OrderableExample Orderable1 = new OrderableExample();
OrderableExample Orderable2 = new OrderableExample();

Orderable0.add_greaterEq(Orderable0, true);
Orderable0.add_greaterEq(Orderable1, false);
Orderable0.add_greaterEq(Orderable2, true);

Orderable1.add_greaterEq(Orderable0, true);
Orderable1.add_greaterEq(Orderable1, true);
Orderable1.add_greaterEq(Orderable2, true);

Orderable2.add_greaterEq(Orderable0, false);
Orderable2.add_greaterEq(Orderable1, false);
Orderable2.add_greaterEq(Orderable2, true);
(...)
```

Fig. 50 Instanciating and loading Orderable objects example

To give an idea of how this TC extractor method works, the process of extracting a TC from a model instance of the Alloy specifications, obtained from the Sorted Set module, will be presented. For this, we produced the model instance in Fig. 52 from the check of Fig. 51, made to exercise the axiom *largest(insert(S, E)) = E if not isEmpty(S) and geq(E, largest(S))*.

```
assert assert_axiomSortedSet3_0{
    all E:Orderable, S:SortedSet |
        one S.insert[E] implies (
            ((not ((S.isEmpty = BOOLEAN/True)) and
            (E.geq[S.largest] = BOOLEAN/True)) implies
              not (S.insert[E].largest = E))
}check assert_axiomSortedSet3_0  for 7 but 7 int, exactly 4 SortedSet
```

Fig. 51 Axiom fact used to extract the TC

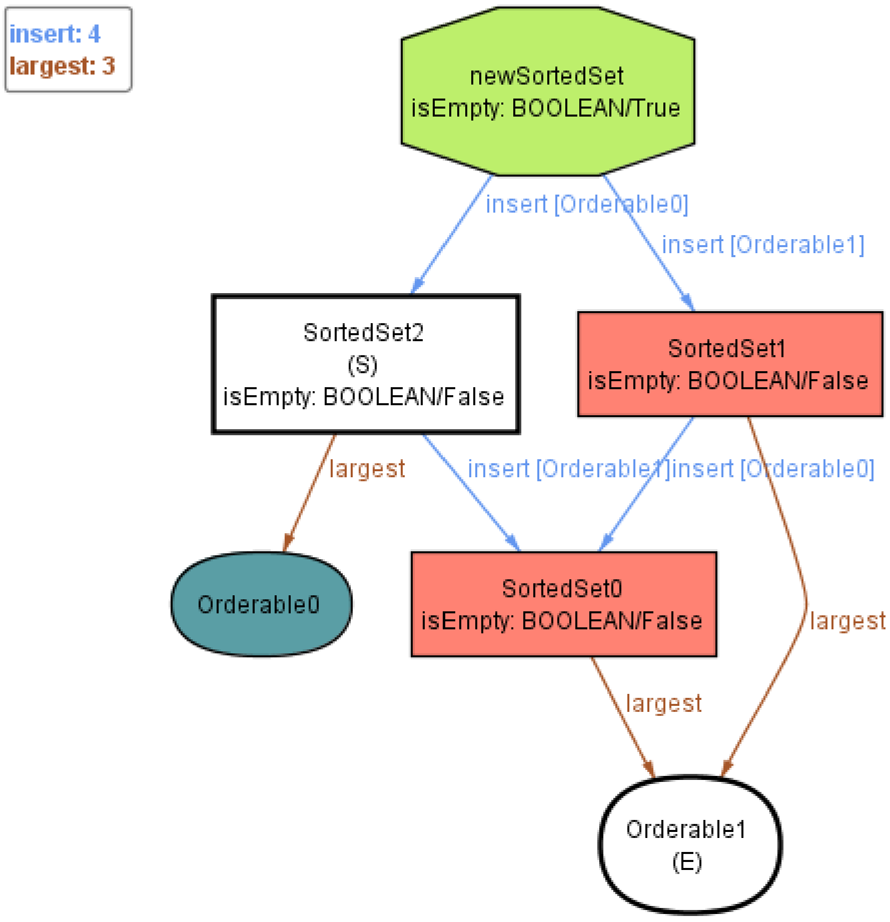


Fig. 52 Model instance used to extract the TC

As it can be seen, the signature instances that represent the variables of the axiom fact are white, to make it easier to distinguish.

Now using the refinement mapping in Fig. 46 and the Java test class in Fig. 48, one can obtain a TC like the mock-up TC that follows:

```
(…)
OrderableExample Orderable0 = new OrderableExample();
OrderableExample Orderable1 = new OrderableExample();

Orderable0.add_greaterEq(Orderable0, true);
Orderable0.add_greaterEq(Orderable1, false);

Orderable1.add_greaterEq(Orderable0, true);
Orderable1.add_greaterEq(Orderable1, true);

TreeSet<OrderableExample> testSortedSet0 = new TreeSet<OrderableExample>();

testSortedSet0.insert(Orderable0);

//Axiom
TreeSet<OrderableExample> testSortedSet1 = testSortedSet0.clone(); //State change
testSortedSet1.insert(Orderable1);

assert(testSortedSet1.largest().equals(Orderable1));
(…)
```

Fig. 53 TC generated from the TC extractor

These TCs may be generated in the JUnit [40] format, for instance, which is a simple unit testing framework. It is believed that all this could be done in an automated way.

By the end, with the generated TCs, it will be possible to fulfil the second task described in section 1.1, stacking these TCs in a JML universal quantifier *\forall*[41].

5 Conclusions

So far, the results of the developed application are excellent and it is still considered the best way to go, for all the reasons denoted in chapter 3.

In this dissertation we managed to conjure a method that allows one to generate Java TCs for generic ADTs from ASs, being completely offline from any Java implementation. Another good thing that this dissertation brings to the table is the generation of Alloy specifications from a black-box specification such as the ASs. This allows one to check the consistency of the actual developed ASs. In a way, it's testing a formal specification.

With this method, extensive TCs for specific axiom or axiom cases can be generated or, if desired, just an overall general TC set generation. This method also allows one to toggle with the quantity of the generated signature instances, in a model instance, with the complexity variables.

Mainly two scientific areas benefit from the method and implementation presented in this report: Software Quality and Testing and Formal Methods of Software Engineering. But there are other areas that could benefit from this, like the Agile Software Development area. In fact, the ability to deal with generics in an offline manor could open a new door for the test-driven development technique, for instance, plus it binds it with formal methods.

An aspect that could be considered a problem is that the AA's model instance generator is not very scalable. Scalable in the sense that when the complexity variables are set high, a model instance may take a few seconds to generate and perhaps use up more CPU resources than expected. The Stack, perhaps due to being the less constricted of the three examples in Appendix A, is the specification that uses up most time and resources to generate model instances.

As for the ConGu project, its big problem is the dependency on such human crafted and unreliable Java methods as *clone* and even *equals* – the second one is not fully tested satisfyingly enough by ConGu.

As for future work, apart from developing the extractor of TCs from the generated Alloy specifications described in section 4.2, we plan to write a scientific article and integrate the developed application into ConGu's under development Eclipse plug-in [4, 10]. Also, we would like to test the AS to Alloy specifications automatic translator even more and with more exoteric cases, together with a theoretical analysis, in order to check if there are any properties of the AS lost in the finite Alloy specifications. Finally, we would like to make, if possible, the application generate Alloy specifications that achieve a better time response on behalf of AA’s instance generation.

6 References

[1] Tecnologia, F. p. a. C. e. a. A Quest for Reliability in Generic Software Components. 2009.

[2] Abreu, J., Vasconcelos, V. T., Nunes, I., Lopes, A., Reis, L. S. and Caldeira, A. ConGu v.1.50 The Specification and the Refinement Languages. 2007.

[3] Vasconcelos, V. T., Lopes, A. and Nunes, I. Specifying and Monitoring Java Classes. 2008.

[4] Vasconcelos, V. T., Nunes, I., Lopes, A., Ramiro, N. and Crispim, P. Monitoring Java Code Using ConGu. 2008.

[5] Nunes, I., Lopes, A., Vasconcelos, V., Abreu, J. and Reis, L. Checking the Conformance of Java Classes Against Algebraic Specifications. 2006.

[6] Nunes, I., Lopes, A., Vasconcelos, V. T., Abreu, J. and Reis, L. S. Testing Implementations of Algebraic Specifications with Design-by-Contract Tools. 2005.

[7] Nunes, I., Lopes, A. and Vasconcelos, V. T. Guiding Specification and OO implementation of Data Types. 2006.

[8] Abreu, J., Caldeira, A., Lopes, A., Nunes, I., Reis, L. S. and Vasconcelos, V. T. Congu, Checking Java Classes Against Property-Driven Algebraic Specifications. 2007.

[9] Reis, L. S. ConGu v.1.50 User's Guide. 2007.

[10] Eclipse Foundation, I. Eclipse's website, http://www.eclipse.org/. 2010.

[11] Guttag, J. V. Abstract data types, then and now. Springer-Verlag New York, Inc., 2002.

[12] Downing, T. B. Java RMI: Remote Method Invocation. IDG Books Worldwide, Inc., 1998.

[13] Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B. Initial Algebra Semantics and Continuous Algebras. 0004-5411, ACM, 1977.

[14] Bo, Y. Testing Java Components based on Algebraic Specifications. 2008.

[15] Sannella, D. and Tarlecki, A. Algebraic methods for specification and formal development of programs. 0360-0300, ACM, 1999.

[16] Chen, H. Y., Tse, T. H. and Chen, T. Y. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. 1049-331X, ACM, 2001.

[17] Clarke, E. M. and Wing, J. M. Formal methods: state of the art and future directions. 0360-0300, ACM, 1996.

[18] Chen, H. Y., Tse, T. H., Chan, F. T. and Chen, T. Y. In black and white: an integrated approach to class-level testing of object-oriented programs. 1049-331X, ACM, 1998.

[19] Dan, L. and Aichernig, B. K. Combining Algebraic and Model-Based Test Case Generation. 2005.

[20] Bernot, G., Gaudel, M. C. and Marre, B. Software testing based on formal specifications: a theory and a tool. 0268-6961, Michael Faraday House, 1991.

[21] Kong, L., Zhu, H. and Zhou, B. Automated Testing EJB Components Based on Algebraic Specifications. 0-7695-2870-8, IEEE Computer Society, 2007.

[22] Doong, R.-K. and Frankl, P. G. The ASTOOT approach to testing object-oriented programs. 1049-331X, ACM, 1994.

[23] Nunes, I., Lopes, A. and Vasconcelos, V. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. 2009.

[24] Gary, T. L. and Yoonsik, C. Design by contract with JML. 2003.

[25] Mcmullin, P. R. Daists: a system for using specifications to test implementations. University of Maryland at College Park, 1982.

[26] Hughes, M. and Stotts, D. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. 0-89791-787-1, ACM, San Diego, California, United States, 1996.

[27] Diaconescu, R. and Futatsugi, K. CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. 1998.

[28] Jackson, D. Alloy Analyzer's website, http://alloy.mit.edu/.

[29] Anastasakis, K., BehzadBordbar and Kuster, J. M. Analysis of model transformations via Alloy. 2008.

[30] Cunha, A. 'An introduction to Alloy' slides. 2009.

[31] Diaconescu, R., Futatsugi, K. and Iida, S. Component-based Algebraic Specification and Verification in CafeOBJ. 1999.

[32] Nakagawa, A. T., Sawada, T. and Futatsugi, K. CafeOBJ User's Manual ver. 1.4. 1999.

[33] Doungsa-ard, C. and Suwannasart, T. An Automatic Approach to Transform CafeOBJ Specifications to Java Template Code. 2003.

[34] Nakamura, M. and Seino, T. Generating Test Cases for Invariant Properties from Proof Scores in the OTS/CafeOBJ Method. 2009.

[35] Jackson, D. Alloy Analyzer's API, http://alloy.mit.edu/alloy4/public/.

[36] Dunets, A., Schellhorn, G. and Reif, W. Automated Flaw Detection in Algebraic Specifications. 2008.

[37] team, S. SableCC's homepage, http://sablecc.org/.

[38] Khurshid, S. and Marinov, D. TestEra: A Novel Framework for Testing Java Programs. 2003.

[39] Khurshid, S. and Marinov, D. TestEra: Specification-based Testing of Java Programs Using SAT. 2004.

[40] Beck, K., Gamma, E. and Saff, D. JUnit's project homepage, http://junit.sourceforge.net/.

[41] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Muller, P., Kiniry, J. and Chalin, P. JML Reference Manual. 2007.

7 Appendix A – Generated Alloy Specifications

Appended to this dissertation report, are the ASs used in the three ConGu modules used to test the developed application, their refinement mapping and the resulting Alloy specifications.

These three modules, developed and tested by the ConGu team, specify a Stack, a Sorted Set and a Priority Queue. The Stack was chosen due to being a simple case, the Sorted Set for being complex and the most popular test module amongst ConGu developers, and the Priority Queue for being complex and bringing to the table an observer operation that outputs an instance of the sort it belongs to – *remove*.

Stack

A.Stack - Stack AS

```
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];
    size:     Stack[Element] --> int;

  others
    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;
    size(make()) = 0;
    size(push(S, E)) = 1 + size(S);
    empty(S) iff size(S) = 0;

end specification
```

B.Stack - Refinement mapping

```
import java.util.Stack;

refinement <E>

      Stack[Element] is java.util.Stack<E> {
            make: --> Stack[Element] is Stack();
            empty: Stack[Element] is boolean empty();
            peek: Stack[Element] -->? Element is E peek();
            pop: Stack[Element] -->? Stack[Element] is E pop();
            push: Stack[Element] item:Element --> Stack[Element]
is E push(E item);
            size: Stack[Element] --> int is int size();
      }

      Element is E

end refinement
```

C.Stack - Alloy specifications

```
open util/boolean as BOOLEAN

/*** PARAMETER SORTS ***/

//Sort

sig Element{}

//Make sure all parameter variables are used

fact ElementUsedVariables{
      all elementsort:Element | elementsort in (Stack) or (some
coreSort:Stack | elementsort = coreSort.peek) or (some
coreSort:Stack | one coreSort.push[elementsort])
}

/*** CORE SORTS ***/

//Sort

sig Stack extends Element{
      pop:lone Stack,

      peek:lone Element,

      empty:one BOOLEAN/Bool,

      size:one Int,

      push:Element -> lone Stack
}

//Term constructor

one sig newStack extends Stack{}

fact StackConstruction{
      all coreSort:Stack | coreSort in newStack.*(
            {coreSort, coreSort':Stack | some param0:Element |
coreSort -> param0 -> coreSort' in push})
}

//Domains

fact domainStack0{
      all S:Stack | not (S.empty = BOOLEAN/True) implies one
S.pop else no S.pop
}

fact domainStack1{
      all S:Stack | not (S.empty = BOOLEAN/True) implies one
S.peek else no S.peek
}

//Axioms

fact axiomStack0{
      (newStack.size = 0)
}
```

```
fact axiomStack1{
      all S:Stack | S.empty = BOOLEAN/True iff (S.size = 0)
}

fact axiomStack2{
      all S:Stack, E:Element | one S.push[E] implies
((S.push[E].peek = E))
}

fact axiomStack3{
      all S:Stack, E:Element | one S.push[E] implies
((S.push[E].pop = S))
}

fact axiomStack4{
      all S:Stack, E:Element | one S.push[E] implies
((S.push[E].size = (1 + S.size)))
}

//Asserts and Checks

assert assert_axiomStack0_0{
      not (newStack.size = 0)
}check assert_axiomStack0_0 for 5 but 5 int, exactly 3 Stack

assert assert_axiomStack1_0{
      all S:Stack | (not (S.empty = BOOLEAN/True) implies (S.size
= 0))
}check assert_axiomStack1_0 for 5 but 5 int, exactly 3 Stack

assert assert_axiomStack1_1{
      all S:Stack | ((S.empty = BOOLEAN/True) implies not (S.size
= 0))
}check assert_axiomStack1_1 for 5 but 5 int, exactly 3 Stack

assert assert_axiomStack2_0{
      all S:Stack, E:Element | one S.push[E] implies (not
(S.push[E].peek = E))
}check assert_axiomStack2_0 for 5 but 5 int, exactly 3 Stack

assert assert_axiomStack3_0{
      all S:Stack, E:Element | one S.push[E] implies (not
(S.push[E].pop = S))
}check assert_axiomStack3_0 for 5 but 5 int, exactly 3 Stack

assert assert_axiomStack4_0{
      all S:Stack, E:Element | one S.push[E] implies (not
(S.push[E].size = (1 + S.size)))
}check assert_axiomStack4_0 for 5 but 5 int, exactly 3 Stack

//Run

run {} for 5 but 5 int, exactly 3 Stack
```

Sorted Set

A.Sorted Set - SortedSet AS

```
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));
    not isIn(empty(), E);
    isIn(insert(S,E), F) iff E = F or isIn(S, F);
    largest(insert(S, E)) = E if isEmpty(S);
    largest(insert(S, E)) = E if not isEmpty(S) and geq(E,
largest(S));
    largest(insert(S, E)) = largest(S) if not isEmpty(S) and not
geq(E, largest(S));
    insert(insert(S, E), F) = insert(S, E) if E = F;
    insert(insert(S, E), F) = insert(insert(S, F), E);
end specification
```

B.Sorted Set - TotalOrder AS

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

C.Sorted Set - Refinement mapping

```
import generic.valid.treeSet.TreeSet;

refinement <E>
  SortedSet[TotalOrder] is TreeSet<E> {
    empty: --> SortedSet[Orderable] is TreeSet();
    insert: SortedSet[Orderable] e:Orderable -->
SortedSet[Orderable] is void insert(E e);
    isEmpty: SortedSet[Orderable] is boolean isEmpty();
    isIn: SortedSet[Orderable] e:Orderable is boolean isIn(E e);
    largest: SortedSet[Orderable] -->? Orderable is E largest();
  }

  TotalOrder is E {
    geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
end refinement
```

D.Sorted Set - Alloy specifications

```
open util/boolean as BOOLEAN

/*** PARAMETER SORTS ***/

//Sort

sig Orderable extends Element{
      geq:Orderable -> one BOOLEAN/Bool
}

//Axioms

fact axiomOrderable0{
      all E, F:Orderable | not (F.geq[E] = BOOLEAN/True) implies
E.geq[F] = BOOLEAN/True
}

fact axiomOrderable1{
      all E, F:Orderable | (E = F) implies E.geq[F] =
BOOLEAN/True
}

fact axiomOrderable2{
      all E, G, F:Orderable | (E.geq[F] = BOOLEAN/True and
F.geq[G] = BOOLEAN/True) implies E.geq[G] = BOOLEAN/True
}

fact axiomOrderable3{
      all E, F:Orderable | (E.geq[F] = BOOLEAN/True and F.geq[E]
= BOOLEAN/True) implies (E = F)
}

//Asserts and Checks

assert assert_axiomOrderable0_0{
      all E, F:Orderable | (not ((F.geq[E] = BOOLEAN/True)))
implies not (E.geq[F] = BOOLEAN/True)
}check assert_axiomOrderable0_0 for 7 but 7 int

assert assert_axiomOrderable1_0{
      all E, F:Orderable | (E = F) implies not (E.geq[F] =
BOOLEAN/True)
}check assert_axiomOrderable1_0 for 7 but 7 int

assert assert_axiomOrderable2_0{
      all E, G, F:Orderable | ((E.geq[F] = BOOLEAN/True) and
(F.geq[G] = BOOLEAN/True)) implies not (E.geq[G] = BOOLEAN/True)
}check assert_axiomOrderable2_0 for 7 but 7 int
```

```
assert assert_axiomOrderable3_0{
      all E, F:Orderable | ((E.geq[F] = BOOLEAN/True) and
(F.geq[E] = BOOLEAN/True)) implies not (E = F)
}check assert_axiomOrderable3_0 for 7 but 7 int

//Make sure all parameter variables are used

fact OrderableUsedVariables{
      all orderablesort:Orderable | (some coreSort:SortedSet |
one coreSort.isIn[orderablesort]) or (some coreSort:SortedSet |
orderablesort = coreSort.largest) or (some coreSort:SortedSet |
one coreSort.insert[orderablesort])
}

/*** PARAMETER SORTS ***/

//Sort

sig Element{}

//Make sure all parameter variables are used

fact ElementUsedVariables{
      all elementsort:Element | elementsort in (Orderable +
SortedSet)
}

/*** CORE SORTS ***/

//Sort

sig SortedSet extends Element{
      isEmpty:one BOOLEAN/Bool,

      isIn:Orderable -> one BOOLEAN/Bool,

      largest:lone Orderable,

      insert:Orderable -> lone SortedSet
}

//Term constructor

one sig newSortedSet extends SortedSet{}

fact SortedSetConstruction{
      all coreSort:SortedSet | coreSort in newSortedSet.*(
            {coreSort, coreSort':SortedSet | some
param0:Orderable | coreSort -> param0 -> coreSort' in insert})
}

//Domains

fact domainSortedSet0{
      all S:SortedSet | not (S.isEmpty = BOOLEAN/True) implies
one S.largest else no S.largest
}

//Axioms

fact axiomSortedSet0{
      all E, F:Orderable, S:SortedSet | one S.insert[E].insert[F]
implies ((E = F) implies (S.insert[E].insert[F] = S.insert[E]))
}

fact axiomSortedSet1{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
(S.insert[E].isIn[F] = BOOLEAN/True iff ((E = F) or S.isIn[F] =
BOOLEAN/True))
}

fact axiomSortedSet2{
      all E, F:Orderable, S:SortedSet | one S.insert[E].insert[F]
implies ((S.insert[E].insert[F] = S.insert[F].insert[E]))
}

fact axiomSortedSet3{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
((not (S.isEmpty = BOOLEAN/True) and E.geq[S.largest] =
BOOLEAN/True) implies (S.insert[E].largest = E))
}

fact axiomSortedSet4{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
(S.isEmpty = BOOLEAN/True implies (S.insert[E].largest = E))
}

fact axiomSortedSet5{
      all E:Orderable, S:SortedSet | one S.insert[E] implies (not
(S.insert[E].isEmpty = BOOLEAN/True))
}

fact axiomSortedSet6{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
((not (S.isEmpty = BOOLEAN/True) and not (E.geq[S.largest] =
BOOLEAN/True)) implies (S.insert[E].largest = S.largest))
}

fact axiomSortedSet7{
      newSortedSet.isEmpty = BOOLEAN/True
}

fact axiomSortedSet8{
      all E:Orderable | not (newSortedSet.isIn[E] = BOOLEAN/True)
}

//Asserts and Checks

assert assert_axiomSortedSet8_0{
      all E:Orderable | not (not ((newSortedSet.isIn[E] =
BOOLEAN/True)))
}check assert_axiomSortedSet8_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet7_0{
      not (newSortedSet.isEmpty = BOOLEAN/True)
}check assert_axiomSortedSet7_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet0_0{
      all E, F:Orderable, S:SortedSet | one S.insert[E].insert[F]
implies ((E = F) implies not (S.insert[E].insert[F] =
S.insert[E]))
}check assert_axiomSortedSet0_0 for 7 but 7 int, exactly 4
SortedSet
```

```
assert assert_axiomSortedSet1_0{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
((not (S.insert[E].isIn[F] = BOOLEAN/True) implies ((E = F) and
not (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet1_1{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
((not (S.insert[E].isIn[F] = BOOLEAN/True) implies (not (E = F)
and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_1 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet1_2{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
((not (S.insert[E].isIn[F] = BOOLEAN/True) implies ((E = F) and
(S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_2 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet1_3{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
(((S.insert[E].isIn[F] = BOOLEAN/True) implies not ((E = F) and
not (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_3 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet1_4{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
(((S.insert[E].isIn[F] = BOOLEAN/True) implies not (not (E = F)
and (S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_4 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet1_5{
      all E, F:Orderable, S:SortedSet | one S.insert[E] implies
(((S.insert[E].isIn[F] = BOOLEAN/True) implies not ((E = F) and
(S.isIn[F] = BOOLEAN/True))))
}check assert_axiomSortedSet1_5 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet2_0{
      all E, F:Orderable, S:SortedSet | one S.insert[E].insert[F]
implies (not (S.insert[E].insert[F] = S.insert[F].insert[E]))
}check assert_axiomSortedSet2_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet3_0{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
(((not ((S.isEmpty = BOOLEAN/True))) and (E.geq[S.largest] =
BOOLEAN/True)) implies not (S.insert[E].largest = E))
}check assert_axiomSortedSet3_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet4_0{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
((S.isEmpty = BOOLEAN/True) implies not (S.insert[E].largest =
E))
}check assert_axiomSortedSet4_0 for 7 but 7 int, exactly 4
SortedSet
```

```
assert assert_axiomSortedSet5_0{
      all E:Orderable, S:SortedSet | one S.insert[E] implies (not
(not ((S.insert[E].isEmpty = BOOLEAN/True))))
}check assert_axiomSortedSet5_0 for 7 but 7 int, exactly 4
SortedSet

assert assert_axiomSortedSet6_0{
      all E:Orderable, S:SortedSet | one S.insert[E] implies
(((not ((S.isEmpty = BOOLEAN/True))) and (not ((E.geq[S.largest]
= BOOLEAN/True)))) implies not (S.insert[E].largest =
S.largest))
}check assert_axiomSortedSet6_0 for 7 but 7 int, exactly 4
SortedSet

//Run

run {} for 7 but 7 int, exactly 4 SortedSet
```

Priority Queue

A.Priority Queue - PriorityQueue AS

```
specification PriorityQueue[TotalOrder]
   sorts
       PriorityQueue[Orderable]
   constructors
       make : --> PriorityQueue[Orderable];
       insert : PriorityQueue[Orderable] Orderable -->
PriorityQueue[Orderable];
   observers
       minimum: PriorityQueue[Orderable] -->? Orderable ;
       remove : PriorityQueue[Orderable] -->?
PriorityQueue[Orderable];
       isEmpty : PriorityQueue[Orderable];
   domains
      Q: PriorityQueue[Orderable];
       minimum (Q) if not isEmpty (Q) ;
       remove (Q) if not isEmpty (Q) ;
   axioms
      Q: PriorityQueue[Orderable]; E, F : Orderable ;
       minimum ( insert (Q, E) ) = E when isEmpty (Q) or geq
(minimum (Q), E)
                                    else minimum (Q) ;
       remove ( insert (Q, E) ) = Q when isEmpty (Q) or geq
(minimum (Q), E)
                                   else insert ( remove (Q) ,
E) ;
       isEmpty (make ( ) ) ;
       not isEmpty ( insert (Q, E ) ) ;
end specification
```

B.Priority Queue - TotalOrder AS

```
specification TotalOrder
    sorts
        Orderable
    others
        geq : Orderable Orderable;
    axioms
        E, F , G: Orderable ;

        E = F if geq (E, F) and geq (F, E) ;

        geq (E, F) if not geq (F, E) ;
        geq (E, G) if geq (E, F) and geq (F, G) ;
end specification
```

C.Priority Queue - Refinement mapping

```
import generic.valid.priorityQueue.GenericHeap;

refinement <E, F>
   PriorityQueue[TotalOrder] is GenericHeap<E> {
      make: --> PriorityQueue[Orderable] is GenericHeap();
      insert: PriorityQueue[Orderable] e:Orderable -->
PriorityQueue[Orderable] is void offer(E e);
      minimum: PriorityQueue[Orderable] -->? Orderable is E
element();
      remove: PriorityQueue[Orderable] -->?
PriorityQueue[Orderable] is void remove();
      isEmpty: PriorityQueue[Orderable] is boolean isEmpty();
   }

   TotalOrder is E {
      geq : Orderable e:Orderable is boolean
greaterEq(java.lang.Object e);
   }

end refinement
```

D.Priority Queue - Alloy specifications

```
open util/boolean as BOOLEAN

/*** PARAMETER SORTS ***/

//Sort

sig Element{}

//Make sure all parameter variables are used

fact ElementUsedVariables{
     all elementsort:Element | elementsort in (Orderable +
PriorityQueue)
}

/*** PARAMETER SORTS ***/

//Sort

sig Orderable extends Element{
     geq:Orderable -> one BOOLEAN/Bool
}

//Axioms

fact axiomOrderable0{
     all E, F:Orderable | not (F.geq[E] = BOOLEAN/True) implies
E.geq[F] = BOOLEAN/True
}

fact axiomOrderable1{
     all E, G, F:Orderable | (E.geq[F] = BOOLEAN/True and
F.geq[G] = BOOLEAN/True) implies E.geq[G] = BOOLEAN/True
}

fact axiomOrderable2{
     all E, F:Orderable | (E.geq[F] = BOOLEAN/True and F.geq[E]
= BOOLEAN/True) implies (E = F)
}

//Asserts and Checks

assert assert_axiomOrderable0_0{
      all E, F:Orderable | (not ((F.geq[E] = BOOLEAN/True)))
implies not (E.geq[F] = BOOLEAN/True)
}check assert_axiomOrderable0_0 for 7 but 7 int

assert assert_axiomOrderable1_0{
      all E, G, F:Orderable | ((E.geq[F] = BOOLEAN/True) and
(F.geq[G] = BOOLEAN/True)) implies not (E.geq[G] = BOOLEAN/True)
}check assert_axiomOrderable1_0 for 7 but 7 int

assert assert_axiomOrderable2_0{
      all E, F:Orderable | ((E.geq[F] = BOOLEAN/True) and
(F.geq[E] = BOOLEAN/True)) implies not (E = F)
}check assert_axiomOrderable2_0 for 7 but 7 int

//Make sure all parameter variables are used

fact OrderableUsedVariables{
      all orderablesort:Orderable | (some coreSort:PriorityQueue
| orderablesort = coreSort.minimum) or (some
coreSort:PriorityQueue | one coreSort.insert[orderablesort])
}

/*** CORE SORTS ***/

//Sort

sig PriorityQueue extends Element{
      remove:lone PriorityQueue,

      isEmpty:one BOOLEAN/Bool,

      minimum:lone Orderable,

      insert:Orderable -> lone PriorityQueue
}

//Term constructor

one sig newPriorityQueue extends PriorityQueue{}

fact PriorityQueueConstruction{
      all coreSort:PriorityQueue | coreSort in
newPriorityQueue.*(
           {coreSort, coreSort':PriorityQueue | some
param0:Orderable | coreSort -> param0 -> coreSort' in insert})
}

//Domains

fact domainPriorityQueue0{
      all Q:PriorityQueue | not (Q.isEmpty = BOOLEAN/True)
implies one Q.remove else no Q.remove
}

fact domainPriorityQueue1{
      all Q:PriorityQueue | not (Q.isEmpty = BOOLEAN/True)
implies one Q.minimum else no Q.minimum
}

//Axioms

fact axiomPriorityQueue0{
      newPriorityQueue.isEmpty = BOOLEAN/True
```

```
}

fact axiomPriorityQueue1{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((Q.isEmpty = BOOLEAN/True or Q.minimum.geq[E] = BOOLEAN/True)
implies (Q.insert[E].minimum = E) else (Q.insert[E].minimum =
Q.minimum))
}

fact axiomPriorityQueue2{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
(not (Q.insert[E].isEmpty = BOOLEAN/True))
}

fact axiomPriorityQueue3{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((Q.isEmpty = BOOLEAN/True or Q.minimum.geq[E] = BOOLEAN/True)
implies (Q.insert[E].remove = Q) else (Q.insert[E].remove =
Q.remove.insert[E]))
}

//Asserts and Checks

assert assert_axiomPriorityQueue0_0{
      not (newPriorityQueue.isEmpty = BOOLEAN/True)
}check assert_axiomPriorityQueue0_0 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue1_0{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not ((Q.isEmpty = BOOLEAN/True) and not (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != Q.minimum)))
}check assert_axiomPriorityQueue1_0 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue1_1{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not (not (Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != Q.minimum)))
}check assert_axiomPriorityQueue1_1 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue1_2{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not ((Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != Q.minimum)))
}check assert_axiomPriorityQueue1_2 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue1_3{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((((Q.isEmpty = BOOLEAN/True) and not (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != E)))
}check assert_axiomPriorityQueue1_3 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue1_4{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
(((not (Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != E)))
}check assert_axiomPriorityQueue1_4 for 7 but 7 int, exactly 4
PriorityQueue
```

```
assert assert_axiomPriorityQueue1_5{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((((Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].minimum != E)))
}check assert_axiomPriorityQueue1_5 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue2_0{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
(not (not ((Q.insert[E].isEmpty = BOOLEAN/True))))
}check assert_axiomPriorityQueue2_0 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_0{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not ((Q.isEmpty = BOOLEAN/True) and not (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove !=
Q.remove.insert[E])))
}check assert_axiomPriorityQueue3_0 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_1{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not (not (Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove !=
Q.remove.insert[E])))
}check assert_axiomPriorityQueue3_1 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_2{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((not ((Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove !=
Q.remove.insert[E])))
}check assert_axiomPriorityQueue3_2 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_3{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((((Q.isEmpty = BOOLEAN/True) and not (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove != Q)))
}check assert_axiomPriorityQueue3_3 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_4{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
(((not (Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove != Q)))
}check assert_axiomPriorityQueue3_4 for 7 but 7 int, exactly 4
PriorityQueue

assert assert_axiomPriorityQueue3_5{
      all E:Orderable, Q:PriorityQueue | one Q.insert[E] implies
((((Q.isEmpty = BOOLEAN/True) and (Q.minimum.geq[E] =
BOOLEAN/True)) implies (Q.insert[E].remove != Q)))
}check assert_axiomPriorityQueue3_5 for 7 but 7 int, exactly 4
PriorityQueue

//Run

run {} for 7 but 7 int, exactly 4 PriorityQueue
```