TEST GENERATION FROM BOUNDED ALGEBRAIC SPECIFICATIONS USING ALLOY

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

Francisco Rebello de Andrade[1], João Pascoal Faria[1,2], Ana C. R. Paiva[1]
Department of Informatics Engineering - Faculty of Engineering of the University of Porto[1], INESC Porto[2],
Rua Dr. Roberto Frias, s/n, 4200-465, Porto, Portugal
francisco.andrade@fe.up.pt, jpf@fe.up.pt, apaiva@fe.up.pt

Keywords: Test case generation, Algebraic specifications, Abstract data types, Alloy analyzer.

Abstract: Algebraic specification languages have been successfully used for the formal specification of abstract data types (ADTs) and software components, and there are several approaches to automatically derive test cases that check the conformity between the implementation and the algebraic specification of a software component. However, existing approaches do not assure the coverage of conditional axioms and conditions embedded in complex axioms. In this paper, we present a novel approach and a tool to automatically derive test cases from bounded algebraic specifications of ADTs, assuring axiom coverage and of all minterms in its full disjunctive normal form (FDNF). The algebraic specification is first translated into the Alloy modelling language, and the Alloy Analyzer tool is used to find model instances for each test goal (axiom and minterm to cover), from which test cases in JUnit are extracted.

1 INTRODUCTION

Our society is increasingly dependent on the correct functioning of software systems, so the software industry should strive to deliver essentially defect free software, by using more effective and efficient defect prevention and detection techniques than are in common use today. The automatic generation of test cases from formal specifications should play an important role in that effort, because of the higher rigor, automation and thoroughness that is introduced in the testing process, when compared to manual test case generation [1-3].

Amongst the existing formal specification languages, algebraic ones are particularly well suited for the generation of black-box tests, because the syntax and semantics of the operations provided by a software component are specified irrespective of how its state is represented and manipulated internally, contrarily to what happens with other formal specification languages. A simple example of an algebraic specification of an abstract data type (ADT) [4] is shown in Figure 1. The semantics of operations is defined through axioms that relate different operations, without any assumption about how the state is represented internally.

However, this very-high level of abstraction also poses additional challenges for test case generation. In fact, several approaches exist to automatically derive test cases from algebraic specifications [1, 2, 5-9], but they do not assure the coverage of conditional axioms and conditions that are part of complex Boolean expressions, as explained in more detail in the state of the art section of this paper.

```
1:  Sort
2:    Stack
3:  Operations
4:    newStack: -> Stack
5:    push: Stack  Int -> Stack
6:    pop: Stack -> Stack
7:    ...
8:  Axioms
9:    Stack S, Int E: S.push(E).pop() = S
10:   ...
```

Figure 1. Excerpt of the Stack algebraic specification.

To overcome such limitations, we use the Alloy Analyzer tool and its constraint satisfaction capabilities [10-13]. The main idea is to translate the algebraic specification into a satisfiable Alloy

model, according to a set of rules. Then, the Alloy Analyzer is used to find model instances that exercise specified axiom cases, from which test cases are finally extracted using a refinement mapping to the target implementation language.

With this method, it is possible to generate tests that guarantee coverage of all the minterms in the axiom's full disjunctive normal form (FDNF). The method also allows checking the consistency of the algebraic specification by examining the model instances found by Alloy Analyzer.

The work presented in this paper is part of a larger project aimed at improving the reliability of software components [14], using ConGu [15-17] as the algebraic specification language and Java as a target implementation language.

The rest of the paper is organised as follows. Section 2 presents the state of the art. Section 3 gives an overview of the test case generation approach. Section 4 gives an overview of the algebraic specification language used and the refinement mapping to Java types. In section 5, the translation rules and decisions made to convert the algebraic specification modules into Alloy specifications are explained. Section 6 describes how test cases can be produced from the model instances. Section 7 presents some conclusions and future work. A simple running example is used to illustrate the approach.

2 STATE OF THE ART

Three main techniques were found in the literature to generate test cases based on algebraic specifications [1-3, 5-9, 18]: manual scripting, term rewriting and variable substitution.

2.1 Manual scripting

In manual scripting [3, 18] the user supplies the values and terms to exercise for each free variable, and a tool substitutes each possible combination of values in the axioms. This approach involves too much manual work, is prone to errors and omissions, and may cause a combinatorial explosion of test cases.

2.2 Term rewriting

Term rewriting proposes that permissible term expressions be generated at random using the methods and operations of an algebraic specification, and then rewritten into their necessarily unique normal form, using the algebraic specification's axioms as rewriting rules [9]. This way, one may build test cases by checking if the legal terms generated and the normal form terms are equivalent. Considering the push and pop operations and the axiom in Figure 1, the three steps involved are illustrated in Figure 2.

Step 1: Generate term expression
newStack.push(3).push(7).pop()

Step 2: Reduce to normal form using axioms as rewriting rules
newStack.push(3).push(7).pop()→ newStack.push(3)

Step 3: Produce assertion
newStack.push(3).push(7).pop() = newStack.push(3)

Figure 2. Test case generation using term rewriting.

One of the problems to overcome with this method is how to generate the initial term expressions in an automated way, when operation domains and conditional axioms are present. Another problem, which is put up by the author of [9], is how to deal with these initial term expressions when they do not hold a unique normal form equivalent – in a set you may have the axiom *set1.insert(a).insert(b) = set1.insert(b).insert(a).* A possible solution to this problem may be to generate several test assertions.

2.3 Variable substitution

Variable substitution suggests going through each axiom and substituting its variables with randomly generated type instances and term expressions made up of constructive operations only – the same as saying normal form term expressions [1, 2, 5-8]. Figure 3 shows an example considering the same algebraic specification excerpts as in Figure 1.

Step 1: Pick axiom
Stack S, Int E: S.push(E).pop() = S

Step 2: Generate expressions and primitives for variables
S = newStack.push(7)
E = 3

Step 3: Produce assertion
newStack.push(7).push(3).pop()= newStack.push(7)

Figure 3. Test case generation using variable substitution.

Although this method has the advantage that each test case generated exercises a well identified axiom, the random generation process may be unable to generate combinations of values that satisfy conditions in conditional axioms, multi-conditional axioms (*if-then-else* and *if-and-only-if*), and complex Boolean expressions.

2.4 Automatic test case generation with Alloy

TestEra [19, 20] is a tool which generates input values to test Java programs from pre-conditions given as first-order relational formulas in the Alloy modelling language. First, it generates all non-isomorphic instances to find the possible inputs for a Java method, using the available Alloy pre-conditions for a given bound, and converts these inputs to Java – concretisation translation. Afterwards, it runs the Java method with these input values and converts the outputs obtained back to Alloy – abstraction translation – to verify the correctness of each input/output pair by evaluating a formula that represents the method post-condition.

Although the concretisation translation is quite interesting, and quite similar to the problem to tackle in this paper, the goal is to translate the whole instances found by Alloy to Java as test cases – sequences of operations with corresponding input values – and not only input variables.

3 METHOD AND TOOL OVERVIEW

This section gives an overview of the approach proposed in this paper. The workflow diagram is shown in Figure 4.

First of all, the ConGu algebraic specification and the refinement mapping file – explained later on in section 4 – are inputs of the ConGu's Parser which creates an in-memory object representation of the algebraic specification and refinement mapping. This representation is translated into Alloy by the new Alloy Translator tool, according to a set of translating rules. A set of run commands are automatically generated by the tool to exercise all minterms in the full disjunctive normal form (FDNF) of each axiom. More detail is provided in section 5.

Afterwards, Alloy Analyzer reads the outputted Alloy specification and executes each of the run commands at a time. Each model instance found by Alloy Analyzer satisfying a run command is exported into a XML format. The test generator tool receives the XML representation of the model instances found and the in-memory representation of the refinement mapping as input, and generates corresponding JUnit test cases. More in-depth explanation of this step can be found in section 6.

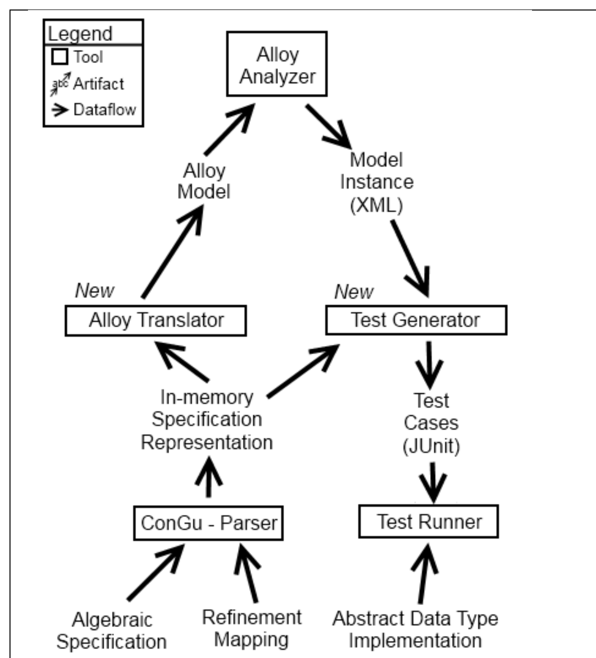

Figure 4. Overview of the test generation process.

4 ALGEBRAIC SPECIFICATION AND REFINEMENT MAPPING WITH CONGU

This section explains how the algebraic specification and the refinement mapping to Java classes, and interfaces, are organised in ConGu [17].

ConGu supports sub-sorting, i.e., the specification of sorts that extend other sorts. Implicitly, all sorts are ultimately a sub-sort of the implicitly existent ConGu sort Element.

In a ConGu specification there are three types of operations [21]: **constructors**, **observers** and **others**. Constructors consist of a minimal number of operations needed to build any possible value of the sort, while observer operations are operations used to analyze the value of a sort. As for other operations, these are defined as being derived from the first two previously defined operations or comparing operations. The constructors can also be

divided into two types: the **creators**, that do not have parameters of the sort type they instantiate, and **transformers**, that are the remaining constructors that have at least the first parameter of the same sort type as its output, called **self argument**. All non-constructor operations have the self argument.

After the declaration of the several types of operations, there is a section called “domains” to restrict the domain of partial operations.

An example of a ConGu specification for a bounded stack of integers (with limited size) is shown in Figure 5. The size limitation makes the corresponding Alloy model satisfiable by finite instances.

```
1:    specification BStackInt
2:      sorts
3:        BStackInt
4:      constructors
5:        make: int --> BStackInt; //Creator
6:        push:  BStackInt int --> BStackInt; //Transformer
7:      observers
8:        peek:    BStackInt -->? int;
9:        pop:     BStackInt -->? BStackInt;
10:       size:    BStackInt --> int;
11:       maxSize: BStackInt --> int;
12:     others
13:       empty: BStackInt;
14:     domains
15:       S: BStackInt;
16:       E: int;
17:       peek(S) if not empty(S);
18:       pop(S)  if not empty(S);
19:       push(S, E) if size(S) < maxSize(S);
20:     axioms
21:       S: BStackInt;
22:       E, N: int;
23:       peek(push(S, E)) = E  if size(S) < maxSize(S);
24:       pop(push(S, E)) = S if size(S) < maxSize(S);
25:       size(make(N)) = 0;
26:       size(push(S, E)) = 1+size(S) if size(S)<maxSize(S);
27:       empty(S) iff size(S) = 0;
28:       maxSize(make(N))  = N;
29:       maxSize(push(S,E)) = maxSize(S)
30:           if size(S) < maxSize(S);
31:   end specification
```

Figure 5. ConGu algebraic specification of a bounded stack of integers.

For mapping sorts to Java classes, there are files called refinement mapping files, in charge of associating each sort and its predicates and operations to the respective Java class and methods. Figure 6 is the possible Java class – StackInt – the BStackInt sort would correspond to.

So, having the ConGu specification and knowing the corresponding types desired, the refinement mapping would be the one represented in Figure 7.

```
1:    public class StackInt{
2:      public StackInt(int max){...}
3:      public void push(int e){...}
4:      public int peek(){...}
5:      public void pop(){...}
1:      public int size(){...}
2:      public int maxSize(){...}
3:      public boolean isEmpty(){...}
4:      ...
5:    }
```

Figure 6. Skeleton of a Java implementation of a Stack of integers.

```
1:    refinement
2:     BStackInt is StackInt {
3:      make: n:int --> BStackInt is StackInt(int n);
4:      push:BStackInt e:int-->BStackInt is void push(int e);
5:      peek: BStackInt -->? int is int peek();
6:      pop: BStackInt -->? BStackInt is void pop();
7:      size: BStackInt --> int is int size();
8:      maxSize: BStackInt --> int is int maxSize();
9:      empty: BStackInt is boolean isEmpty();
10:    }
```

Figure 7. Stack of integers’ refinement mapping.

5 TRANSLATION TO ALLOY

This section presents the rules followed by the developed tool to translate algebraic specifications, written in ConGu, to Alloy specifications. We assume the reader is familiar with Alloy. For an introduction please see [10].

Requirements to take into consideration before going further are:

- The resulting Alloy specification should be satisfiable by finite models in order to enable Alloy Analyzer to find model instances;
- The resulting Alloy specification should be consistent with the algebraic specification;
- Sorts should include, at least, two constructors: one **creator** and one **transformer**. These constructors should exist in order to extract state transitions of the sort type from the Alloy model instances and generate test cases.

The first two points may conflict with each other. The first one requires the manipulation of the

algebraic specification in order to generate finite Alloy specifications. This was performed for the stack of integers by imposing a limit on its size. Additional constraints introduced in the Alloy specification will be described later on.

In the sequel it will be described how sorts, operation domains and axioms are translated from the algebraic specification to Alloy, and how the test goals are specified as run commands in Alloy.

5.1 Signatures

Each sort of the algebraic specification is translated into a signature in Alloy, with its operations and predicates as relations, according to a set of rules summarized in Table 1, found in the appendix, and illustrated in this section through the examples. Figure 8 presents the translation to Alloy of the syntactic part of the algebraic specification of the bounded Stack of integers, according to the rules in Table 1.

```
1:   sig Element{}
2:
3:   sig BStackInt extends Element{
4:      push: Int -> lone BStackInt,
5:      peek: lone Int,
6:      pop: lone BStackInt,
7:      size: one Int,
8:      empty: one BOOLEAN/Bool,
9:      maxSize: one Int
10:  }
11:
12:  one sig start{
13:      make: Int -> lone BStackInt,
14:      pushInt0: set Int,
15:      makeInt0: set Int
16:  }
```

Figure 8. Alloy signatures generated for the bounded stack of integers.

Sorts. All non-primitive signatures extend the Element signature, represented in line 1 in Figure 8, which has no relations and represents the Element sort in ConGu (see rules R1, R2 and R3 in Table 1).

Operations (except creator constructors). All operations (except creator constructors) are represented as fields (relations) of the signature corresponding to the original sort. Since the self argument represents the sort the operation is applied to, it does not appear as an argument.

An operation that only has the self argument as an argument, in Alloy, becomes a single relation to an instance of the signature that represents the output sort of the operation (R4 in Table 1). An example of this case, in the bounded Stack of Integers example, is the peek operation.

An operation that has more arguments than the self argument requires a multirelational field, relating the signature instances of the arguments of the operation with its output parameter (R4 in Table 1).

Predicates. Predicates obey these same rules except that the outputs of the resulting relations are always a Boolean signature instance (R5 in Table 1). An example, in the Stack example, is the empty predicate.

Partial operations and predicates. Relations translated from operations or predicates that have a restricted domain, create lone (one or none) type relations (R6 in Table 1). The reason to generate lone type relations from an operation with a restricted domain is to only allow relations to exist within that domain. An example of an operation with a restricted domain is pop.

Creator constructors. Creator constructors generate a lone relation in the signature named start, which has one instance of itself in every model instance generated and is the source of all the signature instances (R7 and R8 in Table 1). An example of a creator constructor is make.

Argument sorts of constructor operations. In order to guarantee that the Alloy specification is satisfiable by finite model instances, the domain of each argument of each constructor (except the self argument) is constrained to belong to a finite set. Those sets are declared as fields of the start signature (rule R9 in Table 1), and facts are added to restrict the arguments' domains to those sets. Domain restriction conditions are implicitly applied to all the axioms that refer those constructors (to be described in the sequel). Examples for the push and make constructors are shown in lines 14-15 in Figure 8.

5.2 Signature restraining facts

Next, basic signature restraining facts to ensure that model instances are consistent with the original algebraic specification will be described.

In the algebraic specification, the instances of the sorts are implicitly constrained to the ones that may be constructed by a constructor term expression, starting with a creator constructor operation. When translating the algebraic specification to Alloy, this assumption must be made explicit by defining signature binding facts to avoid generating

inconsistent instances wrt the original algebraic specification. For example, the instances of the **core signature** – signature correspondent to the sort to be tested – reached only by using at least one observer operation – like the pop operation – could turn the model instance inconsistent wrt the original algebraic specification. So, starting from the start signature instance, all instances of core signatures must be reached through constructor relations.

The fact to impose this constraint, in the case of the bounded stack of integers, is shown in lines 1-5 in Figure 9, following the rule in Table 2 in the appendix. This fact imposes that all instances of the stack are generated by using the creator constructor make, followed by 0 or more applications of the transformer constructor push.

```
1:   fact BStackIntConstruction{
2:     BStackInt in
3:       (start.make[Int]).*
4:         {x:BStackInt, y:x.push[Int]}
5:   }
6:
7:   fact ElementUsedVariables {
8:     Element in (BStackInt)
9:   }
```

Figure 9. Signature restraining facts generated for the bounded stack of integers.

In order to remove from the equation model instances with unrelated instances of non-core signatures – Element in the example –, another fact is written. In this fact, either a non-core signature instance is being used as the input or output of a core signature's relation or the instance at hand is an instance of a sub-signature of this signature, as can be seen in Table 3, in the appendix. Lines 7-9 in Figure 9 show the fact generated for the Element signature.

5.3 Axioms and Domains

Now we will explain the translation of the semantic part of the algebraic specification, i.e., axioms and domains, to facts.

Axioms. Each axiom is translated to a fact in Alloy. First, the free variables used in the axiom originate universally quantified variables over the corresponding signatures in Alloy. Then, if the axiom expression involves constructors with non-self arguments, pre-conditions are introduced to restrict their domains to the finite sets declared in the start signature (see section 5.1). Done this, the axiom expression is laid down. The rule is described in Table 4, in the appendix, and an example is presented in Figure 10.

```
1:   fact axiomBStackInt1{
2:     all E:Int, S:BStackInt |
3:       (E in start.pushInt0) implies (
4:         (S.size < S.maxSize) implies (S.push[E].pop = S))
5:   }
```

Figure 10. Alloy fact generated for the axiom in line 24 of Figure 5.

As can be seen, conditional axioms are treated as implications. As for the else and iff tokens, of the ternary conditional and biconditional axioms, they remain unaltered in Alloy since these tokens exist in Alloy and mean the same as they do in ConGu.

Domains. Each domain restriction (for a partial operation) is translated also to a fact in Alloy. The variables used in a domain fact are declared in the same manner as with axioms. When the declared pre-condition in the domain declaration evaluates to false, or the non-self arguments of the constructors appearing in that pre-condition are outside the domain declared in the start signature, the corresponding lone relation in Alloy becomes nonexistent. Otherwise, the relation must exist.

The rule is described in Table 4 and an example is presented in Figure 11.

```
1:   fact domainBStackInt2{
2:     all E:Int, S:BStackInt |
3:       (S.size < S.maxSize and E in start.pushInt0)
4:         implies one S.push[E] else no S.push[E]
5:   }
```

Figure 11. Alloy fact generated for the domain restriction in line 19 of Figure 5.

5.4 Generation of run commands for exercising axioms

Our default test coverage criterion is to generate a test case for each minterm in the FDNF representation of each axiom (see Table 5 in the appendix). Although not all minterms are necessarily satisfiable, at least one minterm should be satisfiable for each axiom.

For each axiom and minterm, in order to find bindings of the axiom's free variables that satisfy the minterm, a run command is generated in a straightforward way as illustrated in Figure 12.

There is one configurable complexity variable that may be defined when performing a run command with the developed tool. This variable, called **max**, defines the maximum allowed number

of instances for each signature in a model instance. In this example, this variable was set to 7 (seven).

```
1:    run run_axiomBStackInt4_0{
2:      some S: BStackInt |
3:        (S.empty = BOOLEAN/True) and  (S.size = 0)
4:    } for 7
5:
6:    run run_axiomBStackInt4_1{
7:      some S: BStackInt |
8:        (S.empty = BOOLEAN/False) and (S.size != 0)
9:    } for 7
```

Figure 12. Run commands generated for exercising axiomBStackInt4 (see line 27 of Figure 5).

A graphical representation of a model instance found by Alloy Analyzer when a run command is executed is shown in Figure 13. The diagram shows clearly how the free variables are instantiated (S in this example) and how the axiom is exercised.

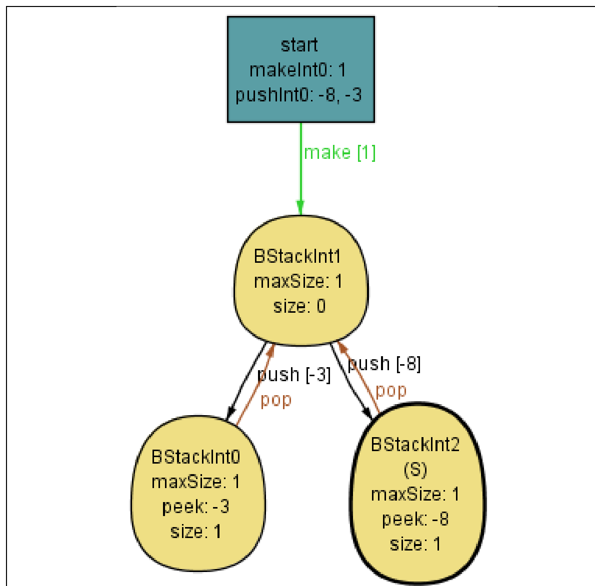

Figure 13. Model instance found by Alloy Analyzer when the run_axiomBStackInt4_1 command is executed.

6 EXTRACTION OF JUNIT TEST CASES FROM ALLOY

This section describes how test cases in JUnit [22] are extracted from the model instances found by Alloy Analyzer.

As explained before, for each minterm of each axiom, it is generated a run command that, when executed by Alloy Analyzer, will find a model instance and bindings for the axiom's free variables that satisfy the minterm. The model instance found is exported to XML and subsequently interpreted (as a Finite State Machine) by the test extractor tool, to find shortest paths for constructing the axiom variables.

The refinement mapping from the algebraic specification to Java (see section 4) is used to produce a proper encoding of the test cases in Java.

An example of a test case extracted from the model instance in Figure 13 is shown in Figure 14.

The test code generated has two parts: a setup part, where the variables involved in the axiom are constructed (according to the model instance found by Alloy Analyzer), and an axiom verification part, where the specific minterm of the axiom is checked (as specified in the run command). To construct each variable, it is followed a shortest path in the model instance, from the start signature instance to the node bound to that variable. In the axiom verification part, it is generated a separate assertion for each operand of the minterm (conjunction) being exercised, for better fault localization.

```
1:    @Test
2:    public void test_axiomBStackInt4_1() {
3:      // setup
4:       StackInt BStackInt1 = new StackInt(1);
5:       StackInt BStackInt2 = BStackInt1.push(-8);
6:       StackInt S = BStackInt2;
7:       // axiom verification
8:       assertTrue(!S.isEmpty());
9:       assertTrue(S.size() != 0);
10:   }
```

Figure 14. JUnit test case corresponding to the second run command of Fig. 12 and the model instance in Fig. 13, using the refinement mapping in Figure 7.

Equality axioms, such as pop(push(S,E))=S, are checked with assertEquals, which in turn uses the equals method in Java, so the approach relies on the correct implementation of equals.

7 CONCLUSIONS

An approach was described to automatically generate JUnit test cases from algebraic specifications of ADTs, using an intermediate translation to Alloy. In this approach, a test case is generated for each minterm of the FDNF representation of each axiom. Reasoning about the Alloy model allows also checking the consistency of the algebraic specification itself. Although, for space constraints, the example presented in the paper is

very simple, the method has been successfully applied to several other ADTs, such as the Sorted Set and Priority Queue.

A common limitation of Alloy Analyzer is scalability, because of the time required to find model instances in complex cases. However, we did not found this to be a problem for testing ADTs in our approach. On a 32 bit Intel Core 2 Duo T6600 @ 2.2 GHz with 3 GB of RAM, running Windows 7, Alloy Analyzer took around 100 seconds to run the 17 axiom cases of the bounded stack. Times of the same magnitude were obtained for other ADTs.

As future work, we intend to consolidate the test case extractor tool, integrate the complete tool suite into the ConGu's plug-in for Eclipse [23, 24], experiment the approach with more ADTs, demonstrate the adequacy of the test cases generated by mutation testing, and support unbounded ADTs by finding automatically safe bounds applicable.

REFERENCES

1. Bo, Y., et al. *Testing Java Components based on Algebraic Specifications*. in *International Conference on Software Testing, Verification, and Validation*. 2008. Washington, DC, USA: IEEE Computer Society.
2. Chen, H.Y., et al., *In black and white: an integrated approach to class-level testing of object-oriented programs*, in *ACM Trans. Softw. Eng. Methodol.* 1998, ACM. p. 250-295.
3. Hughes, M. and D. Stotts, *Daistish: systematic algebraic testing for OO programs in the presence of side-effects*, in *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*. 1996, ACM: San Diego, California, United States. p. 53-61.
4. Guttag, J.V., *Abstract data types, then and now*, in *Software pioneers: contributions to software engineering*. 2002, Springer-Verlag New York, Inc. p. 442-452.
5. Chen, H.Y., T.H. Tse, and T.Y. Chen, *TACCLE: a methodology for object-oriented software testing at the class and cluster levels*, in *ACM Trans. Softw. Eng. Methodol.* 2001, ACM. p. 56-109.
6. Dan, L. and B.K. Aichernig, *Combining Algebraic and Model-Based Test Case Generation*. 2005. p. 250-264.
7. Bernot, G., M.C. Gaudel, and B. Marre, *Software testing based on formal specifications: a theory and a tool*, in *Softw. Eng. J.* 1991, Michael Faraday House. p. 387-405.
8. Kong, L., H. Zhu, and B. Zhou, *Automated Testing EJB Components Based on Algebraic Specifications*, in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*. 2007, IEEE Computer Society. p. 717-722.
9. Doong, R.-K. and P.G. Frankl, *The ASTOOT approach to testing object-oriented programs*, in *ACM Trans. Softw. Eng. Methodol.* 1994, ACM. p. 101-130.
10. Jackson, D. *Alloy Analyzer's website, http://alloy.mit.edu/*. 2011 11-06-2010]; Available from: http://alloy.mit.edu/.
11. Jackson, D. *Alloy Analyzer's API, http://alloy.mit.edu/alloy4/public/*. 2011; Available from: http://alloy.mit.edu/alloy4/public/.
12. Cunha, A., *'An introduction to Alloy' slides*. 2009.
13. Anastasakis, K., BehzadBordbar, and J.M. Kuster, *Analysis of model transformations via Alloy*. 2008.
14. FCT, *A Quest for Reliability in Generic Software Components*. 2009.
15. Abreu, J., et al., *ConGu v.1.50 The Specification and the Refinement Languages*. 2007.
16. Reis, L.S., *ConGu v.1.50 User's Guide*. 2007.
17. Nunes, I., A. Lopes, and V. Vasconcelos, *Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming*. 2009. p. 115-131.
18. Mcmullin, P.R., *Daists: a system for using specifications to test implementations*. 1982, University of Maryland at College Park. p. 131.
19. Khurshid, S. and D. Marinov, *TestEra: A Novel Framework for Testing Java Programs*. 2003.
20. Khurshid, S. and D. Marinov, *TestEra: Specification-based Testing of Java Programs Using SAT*. 2004.
21. Abreu, J., et al., *Congu, Checking Java Classes Against Property-Driven Algebraic Specifications*. 2007.
22. Beck, K., E. Gamma, and D. Saff. *JUnit's project homepage,*

http://junit.sourceforge.net/. Available from: http://junit.sourceforge.net/.

23. Eclipse Foundation, I. *Eclipse's website*, http://www.eclipse.org/. 2010; Available from: http://www.eclipse.org/.
24. Vasconcelos, V.T., et al., *Monitoring Java Code Using ConGu*. 2008.

APPENDIX

Table 1. Translation rules from ConGu (syntax) to Alloy.

Rule	Algebraic Spec (ConGu)	Alloy
R1. Root type	*(implicit)*	**sig** Element { }
R2. Type	**sorts** *S*	**sig** *S* **extends** Element
R3. Subtype	**sorts** $S < S'$	**sig** *S* **extends** S'
R4. Total operation (except creator constructors)	o: S --> t' o: S x t_1 x...x t_n-->t'	o: **one** t' o: (t_1->...-> t_n)->**one** t'
R5. Total predicate	p : S p : S x t_1 x ... x t_n	p: **one** BOOLEAN/Bool p: (t_1 -> ... -> t_n) -> **one** BOOLEAN/Bool
R6. Partial operation or predicate (except creator constructors)	*(with domain restriction)*	*Same as R4 and R5, with* **lone** *instead of* **one**
R7. Start instance	*(not defined)*	**one sig** start
R8. Creator constructor	c : t_1 x ... x t_n -->S	c: (t_1->...-> t_n)->**lone** *S* (*inside* **sig** start)
R9. Non-self arguments of constructor operations	*Transformer:* c: Sxt_0x...xt_n-->S *Creator:* c : t_0 x ... x t_n -->S	ct1: **set** t_0 ... ctn: **set** t_n (*inside* **sig** start)

Table 2. Rule for the construction fact.

Algebraic specification
Core sort *S* with creator constructors c_i: s_{i1} x ... x $s_{i\,k_i}$ --> S (i=1, ..., n) and transformer constructors t_j: S x s'_{j1} x ... x s'_{jw_j} --> S (j=1,..., m).
Alloy construction fact

fact *S*Construction {
S in (start.$c_1[s_{11}]$ ... $[s_{1k_1}]$ + ... + start.$c_n[s_{n1}]$ ... $[s_{nk_n}]$).*
{x: *S*, y: x.$t_1[s'_{11}]$ ... $[s'_{1w_1}]$ + ... + x.$t_m[s'_{m1}]$ ... $[s'_{mw_m}]$}
}

Table 3. Rule for usage fact.

Algebraic specification
Non-core sort *S* (including root sort *Element*) with *n* occurrences as input or output parameter of operations of core signatures f_i: s_{i1} x ... x $s_{i\,w_i-1}$ --> s_{iw_i} defined in sort S_i (i=1, ..., n) where *S* or a supertype of *S* occurs in position k_i of the input/output parameter list, with $1 \leq k_i \leq w_i$. Additionally, the non-core S sort may have direct subtypes T_1, ..., T_m. Predicates are treated as operations with output type BOOLEAN/Bool.
Alloy usage fact

fact *S*UsedVariables {
S in ($S_1.f_1[s_{11}]$ ... $[s_{1\,k_1-1}].s_{1w_1}.s_{1\,w_1-1}.$ ... $.s_{1\,k_1+1}$ + ... +
$S_n.f_n[s_{n1}]$ ... $[s_{n\,k_n-1}].s_{nw_n}.s_{n\,w_n-1}.$ ... $.s_{n\,k_n+1}$
+ T_1 + ... + T_m)
}

Table 4. Rules for axiom and domain facts.

<table>
<tr><th>Constraint (ConGu)</th><th>Fact (Alloy)</th></tr>
<tr><td>k^{th} axiom in sort S:
v_1: S_1; ... ; v_n: S_n;
formula(v_1, ..., v_n);</td><td>fact axiomSk {
all v_1: S_1, ... , v_n: S_n |
argTesting(formula'))
implies formula'(v_1, ..., v_n)
}</td></tr>
<tr><td>k^{th} domain in sort S:
v_1: S_1; ... ; v_n: S_n;
op(v_1, ..., v_n) if
cond(v_1, ..., v_n);</td><td>fact domainSk {
all v_1: S_1, ... , v_n: S_n |
(cond'(v_1, ..., v_n) and
argTesting(op'))
implies one op'(v_1, ..., v_n)
else no op'(v_1, ..., v_n)
}</td></tr>
</table>

Table 5. Cases to exercise in conditional axioms and constituent Boolean expressions.

Axiom or constituent Boolean expression	Cases to exercise (minterms of the FDNF)
Conditional axiom: B if A	A and B not A and B not A and not B
Logical disjunction: A or B	A and B A and not B not A and B
Biconditional axiom: A iff B	A and B not A and not B
Ternary conditional: X = Y when A else Z	Previous rules for the pair: X = Y if A X = Z if not A