Geração de Testes a partir de Especificações Algébricas de Tipos Genéricos usando Alloy [1]

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

[1] Departamento de Engenharia Informática – Faculdade de Engenharia da Universidade do Porto, [2] INESC Porto
Rua Dr. Roberto Frias, s/n, 4200-465, Porto, Portugal
{francisco.andrade, jpf, apaiva}@fe.up.pt
[3] Universidade de Lisboa – Faculdade de Ciências
Campo Grande, 1749–016 Lisboa, Portugal
mal@di.fc.ul.pt

Resumo. Várias linguagens de especificação algébrica têm sido usadas com sucesso para a especificação formal de tipos de dados abstratos (ADTs). Em alguns casos, as linguagens foram equipadas com abordagens que permitem derivar casos de teste de forma automática para testar a conformidade da implementação de ADTs em relação à sua especificação. No entanto, as abordagens existentes apresentam limitações relativas ao teste de ADTs genéricos (parametrizados). Este artigo apresenta uma nova abordagem para derivar automaticamente casos de teste a partir de especificações algébricas de ADTs genéricos, garantindo a cobertura de axiomas, sem a necessidade de ter presente implementações dos parâmetros do ADT. A especificação algébrica é primeiro traduzida para Alloy, de forma a usar o Alloy Analyzer para encontrar modelos para cada objetivo de teste (caso axiomático a cobrir). Destas instâncias são extraídos casos de teste e implementações finitas dos parâmetros do ADT. A abordagem é suportada pela ferramenta GenT que gera testes em JUnit a partir de especificações algébricas em ConGu.

Palavras-chave: Geração de casos de teste, Especificações Algébricas, Tipos de Dados Abstratos, Alloy.

1 Introdução

Um tipo de dados abstrato (ADT) é um conjunto de valores e de operações – com uma determinada interface – que têm uma especificação invariante independente da sua implementação desconhecida [1]. No caso de ADTs genéricos ou parametrizados (GADTs), os tipos usados para instanciar os ADT são genéricos, podendo estar sujeitos a especificações parciais. Por exemplo, um conjunto ordenado deve aceitar apenas os tipos que contêm uma forma para comparar as suas instâncias.

[1] Trabalho apoiado pela FCT segundo o contrato PTDC/EIA/103103/2008.

As linguagens de especificação algébrica são particularmente adequadas para a especificação formal de ADTs. Uma especificação algébrica compreende uma declaração sintática, dos tipos envolvidos (géneros) e suas operações e predicados, e uma declaração semântica, contendo as regras (axiomas) que relacionam as operações dos géneros envolvidos, conforme exemplificado na Fig. 1. Para especificar GADTs são utilizadas especificações algébricas parametrizadas, conforme se verá na secção 4.

A geração automática de casos de teste (TCs) a partir de especificações formais tem a vantagem de aumentar o rigor, automação e detalhe no processo de teste, comparativamente ao teste manual [2-4]. Existem diversas abordagens para derivar automaticamente TCs a partir de especificações algébricas [2, 3, 5-9], mas nenhuma das abordagens encontradas garante o teste adequado de GADTs.

Para superar as limitações das abordagens atuais, propomos uma abordagem de geração de testes de GADTs a partir de especificações algébricas, suportada pela ferramenta GenT, que tira partido da capacidade de satisfação de restrições do Alloy Analyzer (AA) [10, 11]. Os módulos da especificação algébrica são primeiro traduzidos para especificações em Alloy satisfazíveis por modelos finitos. De seguida, o AA é usado para encontrar modelos que exercitam casos axiomáticos específicos, de acordo com critérios de adequação de teste definidos. Finalmente, dos modelos encontrados, são gerados casos de teste na linguagem de implementação alvo, usando um mapeamento de refinamento entre a especificação e a implementação. A partir desses modelos são também geradas implementações parciais (*mock objects*) dos parâmetros do GADT, permitindo assim testar a implementação do GADT sem depender de qualquer implementação dos seus parâmetros.

Sem perda de generalidade, a linguagem de especificação algébrica usada aqui é a suportada pela ferramenta ConGu [12-14], pelo facto deste trabalho se inserir num projeto maior [15] onde a mesma é usada, e por suportar especificações algébricas parametrizadas e o seu mapeamento para implementações com genéricos em Java.

O resto do artigo está organizado da seguinte forma. Após uma descrição do estado da arte na secção 2 e uma visão geral da abordagem na secção 3, a secção 4 apresenta uma visão geral da linguagem ConGu. Na secção 5 explicam-se as regras de tradução de ConGu para Alloy. A secção 6 descreve a extração de casos de teste dos modelos encontrados pelo Alloy Analyzer. A secção 7 apresenta os resultados experimentais. A secção 8 termina com as conclusões e o trabalho futuro.

2 Estado da arte

2.1 Geração de casos de teste a partir de especificações algébricas

Na literatura existente, identificaram-se três técnicas principais para gerar casos de teste a partir de especificações algébricas: a configuração manual, a reescrita de termos e a substituição de variáveis.

Na técnica de configuração manual, usada em [4, 16], são geradas todas as combinações possíveis dos axiomas para os valores dados pelo utilizador para as variáveis livres, criando testes. Esta técnica envolve muito trabalho manual, é propensa a erros e omissões, e pode causar uma explosão combinatória de casos de teste.

Na técnica de reescrita de termos, são gerados aleatoriamente termos legais, usando as operações da especificação algébrica, que são depois rescritos numa forma normal por aplicação dos axiomas [9]. A verificação de que cada termo legal é equivalente à sua forma normal constitui um caso de teste. Esta técnica é ilustrada no topo da Fig. 2. Um problema com este método é a dificuldade em gerar, de forma automática, termos legais na presença de operações parciais e axiomas condicionais. Surge também um problema quando não há uma única forma normal [9].

Na técnica de substituição de variáveis, as variáveis livres de cada axioma são substituídas por valores (no caso de tipos primitivos) ou termos formados apenas por operações de construção (no caso doutros tipos) gerados aleatoriamente [2, 3, 5-8], conforme ilustrado na Fig. 2 (em baixo). Embora este método tenha a vantagem de identificar o axioma que está a ser exercitado em cada caso, o processo de geração aleatória pode ser incapaz de gerar combinações de valores e expressões que satisfazem as condições em axiomas condicionais e expressões Booleanas complexas.

```
sorts
  Stack
operations
  newStack: --> Stack;
  push: Stack Int --> Stack;
  pop: Stack --> Stack;
  top: Stack --> Int;
axioms
  S: Stack; E: Int;
  pop(push(S,E)) = S;
  top(push(S,E)) = E;
```

Fig. 1. Excerto da especificação algébrica de uma pilha de inteiros.

Passo 1: Gerar termo *Reescrita*
pop(push(push(newStack,3),7))
Passo 2: Reescrever para a forma normal usando axiomas
pop(push(push(newStack,3),7)) -> push(newStack,3)
Passo 3: Produzir asserção
pop(push(push(newStack,3),7)) = push(newStack,3)

Passo 1: Escolher axioma *Substituição*
S: Stack; E: Int; pop(push(S,E) = S;
Passo 2: Gerar valores e expressões para as variáveis
S = push(newStack,3))
E = 7
Passo 3: Produzir asserção
pop(push(push(newStack,3),7)) = push(newStack,3)

Fig. 2. Geração de um caso de teste por reescrita de termos e por substituição de variáveis para o exemplo da pilha.

Além das falhas particulares de cada uma das técnicas, o seu maior problema é não sugerirem uma solução para gerar casos de testes para GADTs.

2.2 Geração automática de testes com Alloy

A ferramenta TestEra [17, 18] permite gerar automaticamente valores de entrada para testar métodos em Java, obedecendo a pré-condições escritas em Alloy. São primeiro gerados e traduzidos para Java (passo de concretização) valores dos parâmetros de entrada satisfazendo as pré-condições. Os resultados da execução do método com essas entradas são traduzidos de volta para Alloy (passo de abstração) para verificar a conformidade com pós-condições escritas também em Alloy. Embora a tradução de concretização seja bastante interessante e semelhante ao problema a resolver no presente artigo, o objetivo aqui é traduzir casos de teste completos de Alloy para Java – sequências de operações com valores de entrada, e não apenas valores de entrada.

3 Visão geral do abordagem da proposta

A abordagem proposta para gerar casos de teste a partir de especificações algébricas de GADTs com a nova ferramenta GenT compreende os seguintes passos (ver Fig. 3):

1) O *parser* do ConGu é utilizado para carregar para memória os módulos da especificação algébrica e o mapeamento de refinamento para Java (ver secção 4).
2) O GenT traduz a especificação algébrica carregada no passo 1 para Alloy, e gera comandos de execução para exercitar cada axioma (ver secção 5).
3) O Alloy Analyzer é utilizado para encontrar modelos que satisfazem cada comando de execução (ver secção 5).
4) O GenT gera casos de teste em JUnit a partir dos modelos obtidos no passo 3 e das especificações e mapeamento carregados no passo 1, incluindo *mock classes* e *mock objects* para os parâmetros do GADT (ver secção 6).

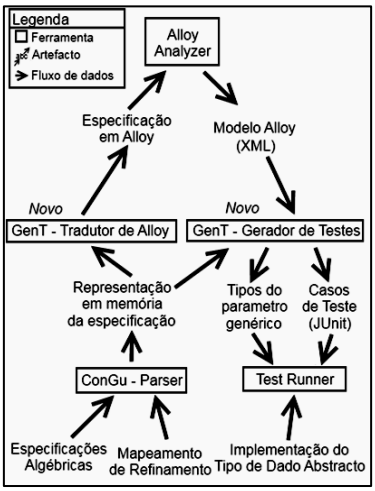

Fig. 3. Visão geral do processo de geração de testes proposto.

4 Especificação algébrica e mapeamento de refinamento em GonGu

Esta secção introduz brevemente a linguagem ConGu e o processo de mapeamento de refinamento para Java, com base num exemplo que será usado no resto do artigo. Mais detalhes sobre ConGu podem ser consultados em [14].

A linguagem ConGu permite especificar de forma modular géneros simples, subgéneros e géneros parametrizados. Estes últimos são os que nos interessam para especificar GADTs. Um exemplo de especificação de um género parametrizado (ou núcleo) e respetivo género parâmetro é apresentando na Fig. 4 e na Fig. 5.

Em ConGu distinguem-se três grupos de operações [19]: **construtores, observadores** e **outras**. Os construtores são um conjunto mínimo de operações que permitem construir qualquer instância do género em causa. Os observadores permitem analisar as instâncias do género em causa. As outras operações são derivadas das anteriores ou são operações de comparação. Os construtores podem-se ainda dividir em **transformadores** ou **criadores**, conforme têm ou não um argumento do género em causa (***self***). Todas as operações não-construtoras têm o argumento *self*. As restrições os domínios das operações parciais são especificadas numa secção *domains*.

A ligação entre o mundo das especificações e do Java é, no ConGu, especificado através de um mapeamento de refinamento. Um exemplo para a especificação da Fig. 4 pode ser encontrado no artigo [14]. Estes mapeamentos associam os géneros, predicados e operações das especificações a tipos Java e correspondentes métodos.

```
specification SortedSet[TotalOrder]
 sorts
  SortedSet[Orderable]
 constructors
  empty: --> SortedSet[Orderable];  //creator
  insert: SortedSet[Orderable] Orderable -->
          SortedSet[Orderable];  //transformer
 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. 4. Especificação em ConGu do género parametrizado (ou núcleo) *SortedSet*.

```
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. 5. Especificação em ConGu do género parâmetro *Orderable* utilizado por *SortedSet*.

```
sig Element {}
sig SortedSet extends Element {
  isEmpty: one BOOLEAN/Bool,
  isIn: Orderable -> one BOOLEAN/Bool,
  largest: lone Orderable,
  insert: Orderable -> one SortedSet
}
sig Orderable extends Element {
  geq:Orderable -> one BOOLEAN/Bool
}
one sig start { empty: lone SortedSet }
fact SortedSetConstruction {
  SortedSet in (start.empty).*
    {x: SortedSet, y: x.insert[Orderable]}
}
fact OrderableUsedVariables {
 Orderable in (SortedSet.isIn.BOOLEAN/Bool
 +SortedSet.largest +SortedSet.insert.SortedSet)
}
fact ElementUsedVariables {
  Element in (Orderable + SortedSet)
}
fact domainSortedSet0 {
   all S:SortedSet | S.isEmpty!=BOOLEAN/True
    implies one S.largest else no S.largest
} (...more domain facts …)
fact axiomSortedSet4{
  all E:Orderable, S:SortedSet |
    (S.isEmpty = BOOLEAN/True
      implies (S.insert[E].largest = E))
} (... more axiom facts ….)
```

Fig. 6. Excerto da especificação em Alloy gerada para o exemplo de *SortedSet*.

```
run run_axiomSortedSet4_0 {
  some E:Orderable, S:SortedSet|
    (S.isEmpty = BOOLEAN/True
     and (S.insert[E].largest=E))
} for 6 but exactly 2 Orderable
run run_axiomSortedSet4_1 {
  some E:Orderable, S:SortedSet|
    (S.isEmpty != BOOLEAN/True
     and (S.insert[E].largest != E))
} for 6 but exactly 2 Orderable
run run_axiomSortedSet4_2 {
  some E:Orderable, S:SortedSet|
    (S.isEmpty != BOOLEAN/True
     and (S.insert[E].largest = E))
} for 6 but exactly 2 Orderable
```

Fig. 7. Comandos de execução gerados para exercitar *axiomSortedSet4* da Fig. 6.

5 Tradução para Alloy

Nesta secção descreve-se a estrutura geral da tradução de especificações algébricas em ConGu para Alloy, bem como a estrutura geral da geração dos comandos de execução que permitem exercitar os axiomas e gerar casos de teste, tendo em conta os seguintes requisitos e pressupostos:

- A especificação Alloy resultante deve ser consistente com a semântica *loose* [20] da especificação algébrica, ou seja, todos os seus modelos devem ser modelos da especificação algébrica alvo.
- A especificação Alloy resultante deve ser satisfazível por modelos finitos a fim de permitir o uso do Alloy Analyzer.
- Os géneros núcleo devem incluir pelo menos um criador e um transformador.

As regras de tradução serão explicadas com base no exemplo da Fig. 6.

5.1 Tradução da parte sintática

Géneros: Os géneros originam assinaturas em Alloy. As assinaturas não-primitivas estendem, em último caso, a assinatura *Element*, que representa a raiz de todos os géneros em ConGu (ver o caso de *SortedSet*).
Operações (exceto criadores): Todas as operações, exceto as definidas como criadores, são traduzidas para campos (relações) da assinatura correspondente em Alloy, omitindo-se o argumento *self*. Operações que têm apenas o argumento *self* originam campos simples (ver o caso de *largest*). Operações com argumentos adicionais originam campos cujo valor é uma relação de valores de argumentos para valores de retorno (ver o caso de *insert*). No caso de serem declaradas como parciais, as operações originam relações *lone* – para 0 ou 1 (ver o caso de *largest*).
Criadores: Estas operações originam campos na assinatura *start*, que tem apenas uma instância e é a raiz de todas as instâncias em cada modelo (ver caso de *empty*).
Predicados: São tratados como operações de retorno Boolean (ver o caso de *isIn*).
Factos de construção: De forma a posteriormente poder usar os modelos da especificação Alloy resultante para gerar testes, estes modelos não podem ter *junk* [20], ou seja, as instâncias dos géneros núcleo são restritas às que podem ser representadas por termos usando apenas construtores (aplicação de um criador, seguido de 0 ou mais transformadores). Para impor esta restrição, são gerados factos correspondentes em Alloy. No caso do *SortedSet*, o facto gerado (ver facto *SortedSetConstruction* em Fig. 6) impõe que todas as instâncias de *SortedSet* são geradas usando o criador *empty* (a partir da única instância da assinatura *start*) seguido por 0 ou mais aplicações do transformador *insert*.
Factos de uso: A fim de evitar a geração de modelos contendo instâncias supérfluas de assinaturas não-núcleo — *Element* e *Orderable* no exemplo —, são gerados factos adicionais, impondo que cada instância de uma assinatura não-núcleo é usada como entrada ou saída de uma relação de uma assinatura núcleo, ou a instância em questão é também uma instância de uma sub-assinatura da assinatura não-núcleo. A Fig. 6 mostra os factos gerados para as assinaturas *Element* e *Orderable* (*ElementUsedVariables* e *OrderableUsedVariables*).

5.2 Tradução da parte semântica

Restrição de domínio (relativas a operações parciais): Estas restrições são traduzidas para factos em Alloy, impondo que o campo correspondente em Alloy esteja definido dentro do domínio, e não definido fora do mesmo pois a sua especificação é desconhecida (ver o caso de *largest*).
Axiomas: Os axiomas são traduzidos para factos em Alloy, com as respetivas variáveis livres quantificadas universalmente. Um exemplo é apresentado na Fig. 6.

5.3 Geração dos comandos de execução para exercitar os axiomas

O critério de cobertura de teste proposto consiste em gerar um caso de teste para cada mintermo de cada axioma na sua forma normal disjuntiva completa (FDNF) (ver Tabela 1). Um mintermo é um termo conjuntivo em que cada variável Booleana aparece uma única vez, possivelmente negada.

Tabela 1. Casos a exercitar em axiomas condicionais e expressões Booleanas

Axioma ou expressão Booleana constituinte	**Casos a exercitar (mintermos de FDNF)**
Axioma condicional simples: **B if A**	A **and** B, **not** A **and** B, **not** A **and not** B
Lógica disjuntiva: **A or B**	A **and** B, A **and not** B, **not** A **and** B
Axioma biconditional: **A iff B**	A **and** B, **not** A **and not** B
Axioma condicional ternário: **X = Y when A else Z**	*Regras prévias para o par:* X = Y **if** A, X = Z **if not** A
Múltiplas variáveis do mesmo tipo: **Expression(A, B)**	A = B **and** *Expression*(A, B) A != B **and** *Expression*(A, B)

Além disso, quando há mais do que uma variável livre do mesmo género num axioma, as combinações de igualdade entre estas também são exercitadas.

Para cada mintermo e combinação de igualdade entre as variáveis livres de cada axioma, é gerado um comando de execução (*run*) que permite encontrar, através do Alloy Analyzer, modelos e valores para as variáveis livres do axioma que satisfazem esse mintermo e combinação de igualdade (ver exemplos na Fig. 7). Embora nem todos os casos exercitados sejam necessariamente satisfeitos, pelo menos um caso deve ser satisfeito por cada axioma.

Duas variáveis configuráveis da ferramenta GenT permitem limitar a exploração: a variável **max** limita o número máximo de instâncias de uma assinatura num modelo; a variável **exact**, que é opcional, especifica o número exato de instâncias das assinaturas núcleo que devem existir num modelo.

Um modelo encontrado pelo Alloy Analyzer em resultado da execução do comando run_axiomSortedSet4_1 é mostrado na Fig. 8. O diagrama mostra como o modelo fornece também uma instanciação para as variáveis livres (*S* e *E* no exemplo).

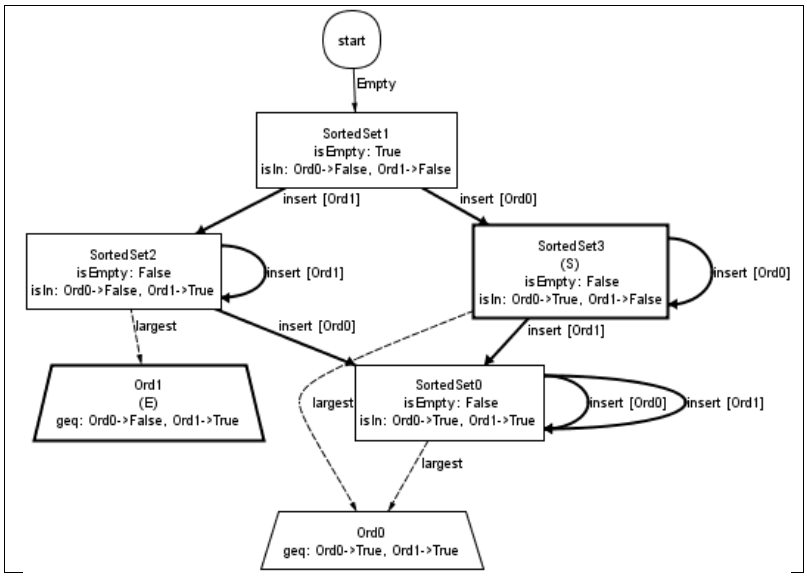

Fig. 8. Modelo encontrado pelo Alloy Analyzer para o comando de execução 4_1 (*Orderable* truncado para *Ord*).

6 Geração de casos de teste em JUnit a partir de Alloy

Esta secção descreve a extração de casos de teste em JUnit [21] a partir dos modelos encontrados pelo AA. Cada modelo encontrado é interpretado pelo extrator de testes para gerar um método de teste em JUnit, incluindo uma implementação finita dos parâmetros dos GADTs. O mapeamento de refinamento, entre a especificação algébrica dos géneros envolvidos em ConGu e a sua implementação em Java (ver secção 4), é usado para produzir uma codificação correta dos casos de teste.

6.1 Geração de *mock classes* para os parâmetros dos GADTs

Implementações finitas dos parâmetros dos GADTs são geradas automaticamente com base nos modelos encontrados pelo AA, compreendendo duas partes: *mock classes*, independentes dos modelos encontrados (ver Fig. 9); e *mock objects*, criados e configurados em cada teste de acordo com os modelos encontrados (ver Fig. 10).

Quanto às *mock classes*, para cada género parâmetro *P* (como *Orderable* na Fig. 5), é gerada uma classe Java *PMock* (como *OrderableMock* na Fig. 9) que implementa a interface que estabelece um limite para a variável de tipo do parâmetro da classe correspondente (neste caso, *Orderable*<*OrderableMock*>). Para cada método *M* de uma interface a implementar, esta classe providencia: um *hash map MMap*, para armazenar os valores de retorno do método para os parâmetros permitidos; um método *add_M*, para configurar os *mock objects* dos testes; e uma implementação de *M* em si, que simplesmente recupera o valor anteriormente armazenado no *hash map*.

6.2 Geração dos métodos de teste

Para cada modelo encontrado pelo Alloy Analyzer é gerado um método de teste com três partes (ver exemplo na Fig. 10):

- **Configuração dos parâmetros do GADT:** Criação e configuração de *mock objects* relativos aos parâmetros do GADT, de acordo com o modelo encontrado.
- **Configuração das variáveis do axioma:** Configuração das variáveis livres do axioma de acordo com o modelo encontrado. Variáveis de um género parâmetro (*E* na Fig. 8) são simplesmente ligadas aos *mock objects* correspondentes. Variáveis de géneros núcleo (*S* na Fig. 8) são construídas seguindo um caminho mais curto no modelo, do nó *start* até ao nó ligado a essa variável. São criadas tantas cópias da variável quantas as ocorrências distintas no axioma.
- **Verificação do axioma:** asserção com a tradução da expressão do axioma, e construção dos termos axiomáticos intermédios utilizados nesta.

```
public class OrderableMock implements
Orderable<OrderableMock> {

   private HashMap<OrderableMock,
Boolean> greaterEqMap = new
HashMap<OrderableMock, Boolean>();

   @Override public  Boolean
greaterEq(OrderableMock e) {
       return greaterEqMap.get(e);
    }

  public void add_greaterEq(
OrderableMock e, Boolean result) {
       greaterEqMap.put(e, result);
  }
}
```

Fig. 9. *Mock class* gerada para o género parâmetro *Orderable*.

```
@Test public void test_axiomSortedSet4_1() {
  // Setup GADT parameters (mock objects)
  OrderableMock Orderable0=new OrderableMock();
  OrderableMock Orderable1=new OrderableMock();
  Orderable0.add_greaterEq(Orderable0, true);
  Orderable0.add_greaterEq(Orderable1, true);
  Orderable1.add_greaterEq(Orderable0, false);
  Orderable1.add_greaterEq(Orderable1, true);
  // Setup axiom variables
  OrderableMock E = Orderable1;
  TreeSet<OrderableMock> S1=
    new TreeSet<OrderableMock>();
  S1.insert(Orderable0);
  TreeSet<OrderableMock> S2=
    new TreeSet<OrderableMock>();
  S2.insert(Orderable0);
  // Axiom verification
  if (S1.isEmpty()) {
    S2.insert(E);      // Intermediate axiom term
    assertTrue(S2.largest().equals(E));
  }
}
```

Fig. 10. Método de teste em JUnit gerado a partir do modelo da Fig. 8.

7 Resultados experimentais

Para avaliar a eficácia (capacidade de deteção de defeitos dos casos de teste gerados) e eficiência (tempo gasto) da abordagem, foi conduzida uma experiência utilizando o exemplo do *SortedSet*, que produziu os resultados sumariados na Tabela 2. A mesma

experiência foi conduzida para a especificação de uma pilha de tamanho limitado (*BoundedStack*), cujos resultados também se encontram na Tabela 2.

Tabela 2. Resultados experimentais

Item	SortedSet	BoundedStack
Tamanho da especificação algébrica (género núcleo) [1]	25 linhas	28 linhas
Número total de axiomas	9	6
Com modelos encontrados em todos os casos axiomáticos	5	3
Com modelos encontrados em alguns casos axiomáticos	4	3
Número total de casos axiomáticos (mintermos)	36	16
Casos em que foram encontrados modelos [2]	20 (56%)	12 (75%)
Casos em que nenhum modelo foi encontrado [2] [3]	16 (44%)	4 (25%)
Tempo gasto pelo AA a encontrar modelos [2]	129 seg	9 seg
Número de casos de teste JUnit gerados [4]	20	12
Tamanho da implementação Java do GADT a ser testado	77 linhas [1]	72 linhas
Número de casos de teste falhados	0	0
Número total de mutantes gerados [5]	41	31
Mortos pelo conjunto de testes inicial	35 (85%)	22 (70%)
Não mortos pelo conjunto de testes inicial	6 (15%)	9 (30%)
Equivalente à implementação original	0	0
Não equivalente à implementação original [6]	6	9
Cobertura da implementação Java do GADT a ser testada [7]	96,9%	86%
Nº de casos de teste adicionados para matar todos os mutantes e cobrir todo o código	3	4

[1] Ignorando comentários e linhas em branco.
[2] Nesta experiência, as variáveis *max* e *exact* estavam definidas como 12 e 3 para o *SortedSet* e 6 e 4 para o *BoundedStack*, respetivamente.
[3] Uma análise manual mostrou que estes casos não eram satisfazíveis.
[4] Apenas um caso foi gerado para cada caso axiomático satisfazível (1º modelo encontrado).
[5] Gerado pelo *Jumble* [22].
[6] Relacionado com invocações de métodos fora do domínio e teste insuficiente do método *equals*.
[7] Medido com *Eclemma* [23].

Foi medido o tempo gasto pelo AA a encontrar modelos para os vários comandos de execução (casos axiomáticos) e o número de comandos para os quais foram encontrados modelos. Para aqueles em que o AA não conseguiu encontrar modelos, uma análise manual foi realizada para determinar se estes poderiam ser teoricamente satisfeitos. Depois disso, os casos de teste foram executados sobre uma implementação (baseada em árvores binárias de pesquisa no caso de *SortedSet*). Posteriormente, foi realizada uma análise de mutação para avaliar a qualidade do conjunto de testes. Os mutantes não mortos pelo conjunto de testes foram inspecionados manualmente para determinar se eram equivalentes ao código original, e foram adicionados casos de teste para matar os não-equivalentes. Uma análise da cobertura dos testes foi também realizada como técnica complementar de avaliação da qualidade dos testes.

A experiência foi realizada num computador portátil com CPU 32 bits Intel Core 2 Duo T6600@2.2 GHz com 3GB de RAM, a correr o Windows 7 da Microsoft.

Em termos de eficiência, concluiu-se que o tempo gasto na busca de modelos (~2 min. no caso de *SortedSet*) não é um obstáculo para a adoção da abordagem proposta. A percentagem de casos axiomáticos sem nenhum modelo encontrado é significativa (44% no caso de *SortedSet*). Uma análise manual mostrou que estes casos não podem ser satisfeitos por nenhum modelo. Grande parte corresponde a casos axiomáticos (mintermos) que violam as restrições de domínio. Como trabalho futuro, pretendemos excluir automaticamente casos axiomáticos deste tipo. A análise de mutações revelou algumas partes da implementação que não estavam a ser devidamente exercitadas, devido a particularidades da implementação de *equals*, e a código de tratamento de invocações fora do domínio (*largest*, neste caso) que não estava especificado e, consequentemente, não foi testado. Como trabalho futuro, pretendemos explorar a geração de casos de teste para valores fora do domínio das operações.

8 Conclusões

A abordagem e a ferramenta apresentadas permitem a geração de testes unitários para ADTs genéricos a partir de especificações algébricas parametrizadas, sem recorrer à implementação prévia dos parâmetros do GADT, porque o código de teste gerado inclui implementações finitas (”mock”) dos mesmos.

A abordagem apresentada baseia-se numa tradução intermediária para Alloy, e na capacidade do Alloy Analyzer encontrar modelos que satisfaçam determinadas condições, que correspondem no nosso caso aos casos axiomáticos a testar (mintermos da representação FDNF).

Nas experiências realizadas, o Alloy Analyzer foi capaz de encontrar instâncias de modelo para todos os casos axiomáticos teoricamente satisfazíveis num tempo aceitável. Os testes de mutação realizados e a análise da cobertura dos testes mostraram que os casos de teste gerados eram de elevada qualidade, pois foram capazes de matar todos os mutantes e cobrir todo o código, exceto no caso de comportamentos que não se encontravam especificados (invocação fora do domínio).

Embora a ferramenta GenT atualmente trabalhe apenas com a linguagem Java como linguagem de saída, outras linguagens podem ser facilmente suportadas.

Uma outra vantagem da abordagem é permitir verificar a coerência da especificação algébrica em si, ao raciocinar sobre os modelos encontrados em Alloy.

Como trabalho futuro, pretendemos ampliar a abordagem para gerar casos de teste fora dos domínios das operações, excluir automaticamente por análise estática mintermos não satisfazíveis, reduzir a dependência sobre a correta implementação do método *equals*, e suportar mais tipos de ADTs não limitados, encontrando automaticamente limites de exploração seguros. Pretendemos também integrar a ferramenta GenT no Plug-in do ConGu para Eclipse, e testá-lo com mais ADTs.

Referências

1. 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.
2. 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.
3. 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.
4. 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.
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*. 2011; Available from: http://alloy.mit.edu.
11. Anastasakis, K., BehzadBordbar, and Kuster, *Analysis of model transformation via Alloy*. 2008.
12. Abreu, J., et al., *ConGu v.1.50 The Specification and the Refinement Languages*. 2007.
13. Reis, L.S., *ConGu v.1.50 User's Guide*. 2007.
14. Nunes, I., A. Lopes, and V. Vasconcelos, *Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming*. 2009. p. 115-131.
15. Tecnologia, F.p.a.C.e.a., *A Quest for Reliability in Generic Software Components*. 2009.
16. Mcmullin, P.R., *Daists: a system for using specifications to test implementations*. 1982, University of Maryland at College Park. p. 131.
17. Khurshid, S. and D. Marinov, *TestEra: A Novel Framework for Testing Java Programs*. 2003.
18. Khurshid, S. and D. Marinov, *TestEra: Specification-based Testing of Java Programs Using SAT*. 2004.
19. Abreu, J., et al., *Congu, Checking Java Classes Against Property-Driven Algebraic Specifications*. 2007.
20. Bidoit, et al., *CASL user manual, Introduction to using the Common Algebraic Specification Language*. Vol. 2900. 2004, Berlin, ALLEMAGNE: Springer. XIII, 240 p.fig.
21. Beck, K., E. Gamma, and D. Saff. *JUnit's project homepage, http://junit.sourceforge.net/*. 2011; Available from: http://junit.sourceforge.net/.
22. Irvine, S.A., et al., *Jumble Java Byte Code to Measure the Effectiveness of Unit Tests*, in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 2007, IEEE Computer Society. p. 169-175.
23. Hoffmann, M.R. *Ecclema: Java code coverage tool for Eclipse, http://www.eclemma.org/*. 2011; Available from: http://www.eclemma.org/.