FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Geração Automática de Testes a partir de Especificações Algébricas

Francisco Ricardo Pinto da Silva

Mestrado Integrado em Engenharia Informática e Computação

Orientador: João Pascoal Faria (Dr.)

18 de Junho de 2012

Geração Automática de Testes a partir de Especificações Algébricas

Francisco Ricardo Pinto da Silva

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Doutor Raul Fernando de Almeida Moreira Vidal
Arguente: Doutor José Francisco Creissac Freitas Campos
Orientador: Doutor João Carlos Pascoal de Faria

---

18 de Junho de 2012

Resumo

A conformidade de uma implementação para com os requisitos à qual ela pretende corresponder é o objetivo primordial no desenvolvimento de software. Entretanto, para verificar essa conformidade existem várias abordagens que procuram gerar automaticamente testes a partir de especificações abstratas do sistema, como as especificações algébricas.

No entanto, as abordagens existentes têm uma lacuna importante em comum, providenciam um fraco apoio a tipos de dados genéricos, o que contribui para que o uso de especificações algébricas para descrever os sistemas, não seja muitas vezes visto como vantajoso.

Assim, com o objetivos de colmatar essa lacuna e procurar fomentar mais o uso de especificações algébricas, no âmbito do projeto QUEST, foi desenvolvida uma abordagem que permite a geração de testes para tipos de dados genéricos a partir da sua especificação algébrica. A abordagem é concretizada para testes em JUnit para testar implementações na linguagem Java, a partir de especificações na linguagem ConGu. A abordagem, utiliza uma ferramenta de instanciação de modelos – AlloyAnalyzer, para exercitar os axiomas que definem o comportamento do sistema e gerar um conjunto de testes, de acordo com um método de decomposição dos axiomas em mintermos. Esta abordagem já estava em desenvolvimento antes desta dissertação, pelo que o trabalho efetuado foi o de extensão dessa mesma aplicação, no sentido de colmatar algumas lacunas que a mesma continha.

Assim, nesta dissertação, foi desenvolvida uma extensão à abordagem, por forma a satisfazer especificações que não são satisfazíveis por modelos finitos. Foram também implementadas outras extensões à ferramenta, nomeadamente o mapeamento entre a especificação e a implementação, extensão para tipos de dados que tenham construtores com restrições de domínio e/ou conter como parâmetros tipos complexos.

Este método de geração de testes tem um elevado potencial, principalmente, nas áreas de Teste e Qualidade de Software e Métodos Formais de Engenharia de Software, esse potencial aumenta com as novas extensões aplicadas.

Abstract

The conformity of an implementation with the requirements to which it seeks to respond is the main goal in software development. To verify that conformity, there are several approaches that seek to generate automatically tests from abstract specifications of the system as the algebraic specifications.

However, existing approaches have an important gap in common: they provide a poor support to generic data types, which contributes to the algebraic specifications to describe systems, it isn't often considered advantageous.

Thus, with the objective of bridging that gap and seek to encourage a wider use of algebraic specifications, under the QUEST project, we developed an approach that allows the generation of tests for generic data types from their algebraic specification. The approach is made for testing in JUnit to test implementations in the Java language from the ConGu specification language. The approach uses a model finding tool –AlloyAnalyzer, to exercise axioms that define the behavior of the system, generating a set of tests, according to a method for decomposition axioms in minterms. This approach was already in development before this dissertation, so the work done was the extension of this same application in order to fill some gaps.

So, in this thesis, was developed an extension to the approach in order to generate tests from specifications that are not satisfiable by finite models. We also implemented some extra extensions to the tool, including the mapping between specification and implementation, extension for data types that have constructors with domain restrictions and / or contain complex types as parameters.

This method of test generation has a high potential, especially in the areas of Testing and Software Quality and Formal Methods in Software Engineering; this potential increases with the new extensions applied.

Agradecimentos

A realização desta dissertação não teria chegado a bom porto sem a ajuda e apoio de algumas pessoas e instituições, às quais gostaria agradecer.

Em primeiro lugar, à minha família, em especial aos meus pais, Fernando e Maria Rosa, por todo o apoio e incentivo que me deram desde sempre, e com especial ênfase durante a minha vida académica.

Um obrigado à Faculdade de Engenharia da Universidade do Porto, por todo conhecimento que me fez ganhar nos últimos anos. Ao meu orientador nesta tese Professor João Pascoal Faria, pelo seu apoio e boa orientação, sempre com boas perspectivas para os problemas que foram surgindo. Um obrigado aos restantes membros do projeto, em particular à Professora Ana Paiva.

Um agradecimento especial aos meus amigos Alexandre Perez, João Santos e Tiago Loureiro pelos momentos de descontração, apoio e motivação que proporcionaram ao longo deste projeto, bem como para os restantes elementos que foram passando pelo Laboratório de Engenharia de Software. Queria também agradecer de forma especial aos restantes colegas que me acompanharam neste percurso académico, com especial referência ao Luís Silva, Tiago Monteiro e Fábio Costa.

Francisco Ricardo Pinto da Silva

Conteúdo

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Abreviaturas e Símbolos

ADT	*Abstract Data Type* / Tipo Abstrato de Dados
GADT	*Generic Abstract Data Type*
QUEST	*A Quest for Reliability in Generic Software Components*
FEUP	Faculdade de Engenharia da Universidade do Porto
FCUL	Faculdade de Ciências da Universidade de Lisboa
FCT	Fundação para a Ciência e Tecnologia

Capítulo 1

Introdução

As especificações algébricas têm sido um meio efetivo para a especificação formal de tipos abstratos de dados – ADTs, de uma forma axiomática. Assim sendo, elas têm sido alvo de várias abordagens no contexto da engenharia de software, tendo por objetivo diversos fins, sendo que o conceito base para essas abordagens é genericamente o mesmo, a abstração que as especificações formais obrigam e as suas vantagens. Ou seja, quando se pretende utilizar este tipo de técnicas, é exigido um elevado grau de abstração em relação ao sistema, desconsiderando a sintaxe e a semântica das operações, descrevendo-se apenas o comportamento das mesmas.

No sentido, de aproveitar toda esta abstração do funcionamento requerido ao sistema em relação à implementação, muitas abordagens têm incidido na extração automática de casos de teste a partir das especificações algébricas, podendo assim conferir maior confiança à implementação de forma automatizada. No entanto, as soluções existentes apresentam em comum uma importante limitação: não garantem a geração adequada de casos de teste para tipos abstratos de dados genéricos – GADTs.

1.1 Enquadramento

O trabalho de pesquisa, especificação e implementação desenvolvido nesta dissertação, enquadra-se num projeto maior denominado QUEST [paCeT12]. Este projeto contou com o financiamento do FCT [dEeC12], do Ministério da Educação e Ciência, envolvendo uma parceria entre a FCUL, e a FEUP.

A proposta do projeto QUEST consiste no desenvolvimento e integração de um conjunto de técnicas eficazes para uma análise automática da confiança dos componentes de um software Java, implementados a partir de uma dada especificação abstrata de dados. Em particular, pretende-se que as referidas técnicas sejam também aplicáveis a componentes genéricos, que possam ser utilizados em situações diferentes, especificados em linguagens de especificação orientadas às

propriedades, apoiando a análise em casos onde o código fonte está ausente ou presente, ajudando a localizar os erros.

Na parte destinada à FEUP, o objetivo centra-se em testar a conformidade da implementação, em Java, em relação a especificações algébricas em causa. Ou seja, pretende-se extrair automaticamente baterias de testes automatizáveis, testes unitários, que testem a funcionalidade do código, tendo como objetivo garantir a sua conformidade para com a especificação.

Como se pode observar na figura 1.1, o ponto de partida para o desenvolvimento do projeto, baseia-se em três médulos principais: a especificação dos tipos de dados, parametrizados e parâmetro, da qual se pretendem extrair os testes; a implementação dos tipos de dados especificados, pode ser apenas uma implementação parcial – *Method Stubs*; e um mapa de refinamento que estabelece a ponte entre os dois componentes anteriores.

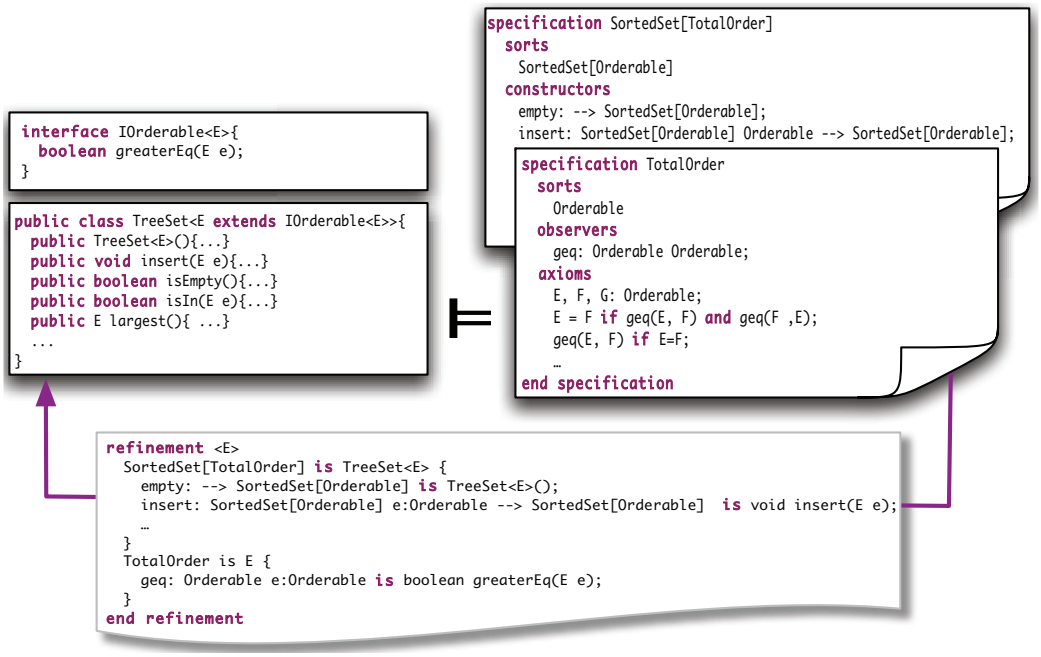

Figura 1.1: Ponto de partida para a geração de testes no presente contexto

Entretanto o ponto de partida desta dissertação não é o ponto de partida do projeto, pois a nível de abordagem geral já tinha sido feito algum trabalho, consistindo na utilização de uma representação intermédia na linguagem de lógica de 1ª ordem –*Alloy*. Esta representação tem por objetivo utilizar o *Alloy Analyzer* [Jac12b] para construção de modelos que por sua vez serão convertidos em casos de teste, como se pode observar de forma detalhada no capítulo 3 bem como nos artigos [AFPL11b, AFPL11a]. Grande parte da arquitetura geral bem como algumas caraterísticas que se pretendiam que a ferramenta possui-se, já se encontravam definidas, como se pode consultar no artigo [AFP11a].

1.2 Motivação e Objetivos

A reutilização de código, tem vindo a ganhar muita relevância no desenvolvimento de software, sendo que uma forma de reutilização consiste na utilização do mesmo tipo de dados em contextos diversos, ou seja, pretende-se o mesmo comportamento do tipo de dados (*e.g.* classes) mas em contextos diferentes. Por exemplo, uma Stack pode ser instanciada com *integer*, *strings*, *dates*, etc. Assim sendo, torna-se relevante e também interessante o uso de especificações algébricas para estes casos. Acrescendo-se, que nos casos onde o código possa ser reutilizado, sejam fornecido mais instrumentos que confiram confiança a essas implementações.

Atendendo à fraca exploração da área de geração de casos de teste a partir de especificações algébricas, em especial para GADT, ou seja, os casos onde se pretende a reutilização de código descrita anteriormente, serve de motivação a criação de uma solução que permita colmatar esta lacuna. Consequentemente, fomenta a reutilização de especificações algébricas, no sentido de utilizar especificações existentes como ponto de partida para novas.

Nos termos desta dissertação em específico, os principais objetivos propostos são:

- Atualização e implementação de novas regras de tradução nos vários patamares de funcionamento da aplicação.
- Estudo e conceção de uma solução para o problema da geração de testes a partir de especificações de tipos de dados não satisfazíveis por modelos finitos, problema esse derivado da utilização do *Alloy Analyser*.
- Efetuar o mapeamento dos testes abstratos gerados a partir da especificação para testes concretos numa linguagem específica e para uma implementação específica do ADT, recorrendo ao mapa de refinamento do ConGu.
- Avaliação e refinamento da aplicação para diversos tipos de ADTs com características específicas.
- Definição automática de limites de pesquisa do AlloyAnalyzer.

1.3 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 6 capítulos. No capítulo 2, são descritos conceitos base e ferramentas utilizadas e também trabalhos efetuados na área de enquadramento desta dissertação. No capítulo 3, é apresentada a abordagem seguida no projeto em desenvolvimento e o estado de desenvolvimento. No capítulo 4, encontra-se descrito o trabalho desenvolvido ao longo da dissertação. No capítulo 5, está apresentada a experimentação que serve de validação ao trabalho efetuado. No capítulo 6, estão as conclusões e o trabalho futuro a desenvolver na ferramenta.

Capítulo 2

Conceitos Base e Estado da Arte

Neste capítulo, são apresentados os conceitos essenciais desta dissertação, seguidos da apresentação de alguns trabalhos já realizados na área de enquadramento da mesma. Inicialmente é feita uma breve descrição de ADT's, seguida por descrição de especificações algébricas com particular foco na linguagem a ser usada, ConGu. Após a descrição dos conceitos base, apresenta-se alguns estudos e ferramentas de anteriores trabalhos que se assemelham a variadas fases do trabalho.

2.1 Tipos Abstratos de Dados

Um tipo abstrato de dados pode ser definido como um modelo matemático, formado por um conjunto de valores e operações, de um determinado tipo de dados, especificando o tipo de objetos que esse mesmo tipo suporta, bem como as funções que sobre si operam. Tudo isto de forma abstraída da sua implementação. Ou seja, quando se define tipos simples como *int*, *double* ou *real*, à partida já sabemos que tipo de operações suportam, somas, subtrações, divisões, multiplicações e outras. Sabemos que tipo de dados de entrada e saída cada operação terá, e para definir que tipo de operações um tipo de dados suporta, não necessitamos de saber dados da sua implementação. O mesmo sucede com tipos mais complexos, como é o caso de estruturas de dados, como exemplo *arrays*, onde se pode definir também os tipos de dados que suporta, bem como o seu comportamento e funções associadas, tudo isto de forma igualmente abstraída da sua implementação.

Um ADT, pode ser generalizado para vários tipos de dados, por exemplo, uma lista ordenada de dados, pode ter como tipo parâmetro qualquer tipo de dados que seja ordenável, ou seja, pode ser instanciada com tipos que possam ser comparáveis, nestes casos estes ADTs denominam-se tipos a abstratos de dados genéricos (GADT).

2.2 Especificações Algébricas em ConGu

Na engenharia de software, dá-se o nome de especificações algébricas à técnica que consiste em especificar formalmente o comportamento de um sistema, de uma determinada forma. Essa especificação dá-se através da definição formal dos tipos de dados, bem como expressões matemáticas que definem o comportamento dos tipos de dados, ou seja, as operações que sobre eles operam. Neste tipo de especificações, pretende-se descrever o que o sistema deverá fazer, não interessando a forma como o faça. Assim sendo, o que se pretende na construção de especificações algébricas, é que estas sejam feitas com um elevado grau de abstração em relação à linguagem de implementação ou mesmo à própria implementação.

Este tipo de abordagem tem por objetivo o desenvolvimento de programas mais corretos e rigorosos, pois a abstração exigida na construção da especificação obriga a um maior cuidado com a definição do que o sistema deve fazer, bem como a própria especificação é uma ferramenta importante de apoio para a verificação do sistema desenvolvido.

Apesar de toda a abstração exigida na construção da especificação, existem formas de refinar a especificação para, mantendo todo o processo de abstração, seja possível a aproximar do desenho concreto do sistema e sua implementação.

Para concretizar este tipo de especificações existem várias linguagens de especificações algébricas. No presente projeto é utilizada uma dessas linguagens, o *ConGu [NV09]*.

2.2.1 Linguagem de Especificação ConGu

A linguagem ConGu possibilita a especificação de vários tipos de géneros (*sorts*): simples, subgéneros e géneros parametrizados. Neste caso, têm maior interesse os géneros parametrizados, visto que são os que nos possibilitam especificar GADTs. Para ajudar a perceber, temos um exemplo de um género parametrizado na figura 2.1 e o respetivo género parâmetro na figura 2.2.

A estrutura de uma especificação em ConGu tem por base um conjunto de cláusulas e uma estrutura [CNV07]. Podem-se dividir as operações em três grupos: **constructors, observers** e **others**. Os primeiros restringem-se a um pequeno conjunto de operações que possibilitam a construção de qualquer instância do género em causa, e podem ser divididos em dois grupos: **criadores** e **transformadores**, distinguindo-se pela presença ou não de um argumento do género em causa, sendo os transformadores que têm esse mesmo argumento. De seguida, os *observers*, servem para analisar as instâncias do género em causa. Tanto *constructors* como os *observers* podem ser regidos por restrições de domínio – *domains*, que controlam as suas condições de utilização. Por fim, as outras operações que são derivadas das anteriores ou operações de comparação.

2.2.2 Mapa de Refinamento

A linguagem ConGu foi desenvolvida com o intuito de colmatar uma lacuna, no que diz respeito à ponte entre as especificações algébricas e as implementações [NV09]. Nesse sentido, o

```
specification SortedSet[TotalOrder]
 sorts
  SortedSet[Orderable]
 constructors
  emptyOp: --> SortedSet[Orderable];
  insertOp: SortedSet[Orderable] Orderable --> SortedSet[Orderable];
 observers
  isEmptyOp: SortedSet[Orderable];
  isInOp: SortedSet[Orderable] Orderable;
  largestOp: SortedSet[Orderable] -->? Orderable;
 domains
  S: SortedSet[Orderable];
  largestOp(S) if not isEmptyOp(S);
 axioms
  E, F: Orderable;  S: SortedSet[Orderable];
  isEmptyOp(emptyOp());
  not isEmptyOp(insertOp(S, E));
  not isInOp(emptyOp(), E);
  isInOp(insertOp(S,E), F) iff E = F or isInOp(S, F);
  largestOp(insertOp(S, E)) = E if isEmptyOp(S);
  largestOp(insertOp(S, E)) = E if not isEmptyOp(S) and geqOp(E, largestOp(S));
  largestOp(insertOp(S, E)) = largestOp(S) if not isEmptyOp(S) and not geqOp(E, largestOp(S));
  insertOp(insertOp(S, E), F) = insertOp(S, E) if E = F;
  insertOp(insertOp(S, E), F) = insertOp(insertOp(S, F), E);
end specification
```

Figura 2.1: Especificação de um *SortedSet* em ConGu

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

Figura 2.2: *TotalOrder* em ConGu

ConGu está habilitado com um mapa de refinamento, com um formato semelhante ao presente na figura 1.1.

Esta funcionalidade do ConGu tem como requisitos iniciais a existência de uma especificação, quer dos géneros parametrizados quer dos géneros parâmetros, bem como a implementação em Java dessas mesmas especificações. Cumpridos os requisitos iniciais, já é possível criar um ficheiro de refinamento (*e.g.* figura 1.1 onde o ficheiro de refinamento faz uma correspondência entre a especificação do *SortedSet* e a implementação *TreeSet*) reconhecido pelo compilador do ConGu (**.rnf*). Esse ficheiro conterá a correspondência entre a especificação e a implementação, essencialmente a nível de nomenclatura de classes e métodos, tipo de retorno dos métodos e indicação dos géneros parâmetro e possível hierarquia entre eles. O compilador, para além de

validar as correspondências sugeridas no ficheiro de refinamento, analisa e disponibiliza informação relativa aos parâmetros das classes genéricas, nomeadamente as interfaces que são necessárias implementar para um objeto ser aceite pela classe genérica.

2.3 Testes de Software

O teste de software é uma atividade do ciclo de vida do desenvolvimento de software de vital importância para garantir a funcionalidade e qualidade do software. Dada toda essa importância, ao longo dos anos têm sido desenvolvidas diversas técnicas de teste. Nesta secção, é feita uma breve descrição de algumas destas técnicas que são focadas no presente projeto.

2.3.1 Testes de Caixa Preta e de Caixa Branca

Os testes de software podem ser divididos em dois grande grupos: *black-box* e *white-box*.

Os testes *black-box* (caixa preta), são criados sem conhecimento da estrutura interna do programa. O seu desenvolvimento é normalmente feito com a intenção de testar as funcionalidades requeridas. Por outro lado, os testes *white-box* (caixa branca), são desenvolvidos tendo em conta o conhecimento da estrutura interna do programa, sendo normalmente criados para garantir a exercitação de todo o código.

No contexto do problema abordado por esta dissertação, o método de testes usado é *black-box*, uma vez que os testes são gerados tendo em conta apenas a especificação servindo para testar a conformidade de uma implementação. No entanto, o processo de geração de testes usa técnicas *white-box*, para garantir a cobertura da especificação.

2.3.2 Testes Unitários

Os testes unitários são testes realizados ao nível de unidades individuais de software, ou de grupos de unidades relacionados. O principal intuito deste método de testes é verificar o correto ou incorreto funcionamento de cada unidade através de asserções entre valores devolvidos pelo bloco em teste e valores previamente esperados. Normalmente essas unidades são métodos ou funções, podendo no entanto ser blocos maiores, se assim fizer sentido. Quando se pretende testar unidades que dependem de outras, o método de funcionamento continua igual, ou seja, testam-se as unidades de forma independente, usando-se por exemplo *mock objects*, que são implementações parciais das unidades de que depende a unidade em teste, que visam simular o funcionamento dessas mesmas unidades. Este método de testes pode ser ***black-box*** ou ***white-box***, dependendo da forma como são desenvolvidos, se antes ou depois da implementação das unidades testadas pelos casos de teste.

Este método de teste trás diversas vantagens no processo de desenvolvimento de software, uma vez que facilita a mudança do código, garantindo que a funcionalidade continua intata depois da mudança. Facilita também a integração, dado que estando cada unidade testada individualmente, o teste da soma dessas unidades é feito com maior grau de confiança. Por fim, serve também

como uma espécie de documentação do sistema, pois permite perceber que funcionalidade são fornecidas, sem necessidade de analisar o código fonte.

2.4 Geração Automática de Testes

Um dos focos de estudo na área da Engenharia de Software é a geração automática de testes, procurando assim facilitar o processo de teste durante o ciclo de desenvolvimento de software. Para que essa geração ser bem sucedida, existem algumas abordagens, no sentido de tornar este processo fiável e de forma a produzir valor para os utilizadores desses mesmos sistemas.

Há várias formas de geração automática de testes, mas genericamente as formas existentes baseiam-se em modelos que procuram representar os requisitos do sistema. Dentro destes modelos encontramos para além das especificações algébricas onde se descrevem as propriedades do sistema, também modelos onde se descrevem as pré e pós-condições das gerações do sistema, outros onde se especifica a máquina de estados, entre outros tipos de modelos.

A seguir são brevemente descritas algumas ferramentas de geração automática de testes, representativas de alguns tipos de abordagens diferentes.

2.4.1 Spec Explorer

O *Spec Explorer* [Mic12] é uma ferramenta desenvolvida pela *Microsoft* e integrada no *Visual Studio* e que está disponível para todas a linguagens *.Net*.

A geração de testes por parte desta ferramenta, processa-se da seguinte maneira: cria-se um *cord script*, que basicamente é uma especificação do comportamento do sistema sob teste; é necessário também criar modelos dos programas, ou seja, descrever as regras e estados do sistema.

2.4.2 JTORX

O *JTORX*[Bel10], aparece como substituto do *TORX* que foi desenvolvido no âmbito do projeto *Côte de Resyste* [STW12]. O tipo de especificações neste caso é fornecido na forma *Labelled Transition Systems (LTS)*, em formato de autómato. Ele interage com a aplicação de forma direta ou também se pode fornecer a implementação na mesma forma LTS. Para a interação direta é necessário que a implementação esteja construída usando a mesma nomenclatura da especificação. O processo de teste dá-se pela comparação dos modelos da especificação e os gerados pela implementação.

2.4.3 QuickCheck

O *QuickCheck* [Cla04] é uma ferramenta desenvolvida, originalmente, para testar implementações na linguagem de programação Haskell, cujo conceito é gerar testes aleatórios com base em especificações. Ou seja, fornece-se à ferramenta a especificação da propriedade que se pretende testar, de seguida a ferramenta procura gerar testes para a propriedade, notificando se a propriedade é válida ou não. A ferramenta, para além de gerar os testes, aplica-os sobre a implementação,

e retorna o resultado da execução. Este conceito já foi extendido a outras linguagens de programação.

2.4.4 Resumo das Ferramentas

Em suma, as ferramentas apresentadas são bastante interessantes e representam perspetivas diversas em relação à problemática da geração automática de testes.

No entanto, e fazendo também uma analogia com o que se pretende do projeto QUEST, o Spec Explorer, é bastante eficaz e fiável, mas necessita de um tempo de especificação muito superior, pois a construção do modelo não é feita só ao nível do comportamento ou propriedades do sistema, como acontece no QUEST. No caso do JTORX, abordagem tem a desvantagem de não gerar testes automatizáveis, apesar de ser uma ferramenta bastante eficaz na procura de erros no sistema. Por fim o QuickCheck tem a limitação de testar apenas propriedade a propriedade. No que diz respeito a teste de GADTs, são também bastante limitados, sendo o QuickCheck, visto que é desenvolvido para uma linguagem funcional, o que melhor responde a este problema.

2.5 Alloy

O Alloy [Jac12b] é uma linguagem de modelação baseada em lógica relacional de primeira ordem que reúne um conjunto de caraterísticas que até à sua origem eram tidas com incompatíveis, pois a linguagem é declarativa e analisável. É declarativa pois descreve a estrutura dos estados e suas restrições, mas não descreve a forma como os alcançar. E é analisável pois existe uma ferramenta capaz de, através da satisfação dessas restrições, criar modelos de execução, sem ser necessário entrada de dados "concretos", o *Alloy Analyzer*. No entanto, por ser analisável, a linguagem possui limitações, pois assim sendo especificações que resultem em modelos infinitos, não são satisfazíveis.

No contexto do objetivo do projeto QUEST, torna-se interessante a sua utilização, visto que é possível extrair casos de teste abstratos a partir de uma especificação em Alloy, sem fornecer nenhum dado extra à especificação. Ou seja, como o Alloy Analyzer é capaz de criar modelos de execução representativos do sistema, podemos a partir desses modelos criar asserções entre estados. Para utilização desta ferramenta, basta desenvolver uma abordagem eficaz de conversão de uma especificação na linguagem ConGu para Alloy.

No que diz respeito à linguagem, o Alloy admite o seguinte tipo de cláusulas:

- **Assinatura** – *sig* – declara um novo tipo e um novo conjunto;
- **Facto** – *fact* – fórmula que restringe os valores do conjunto de relações;
- **Função** – *fun* – fórmula parametrizável que pode ser chamada a qualquer altura;

e para utilização do Alloy Analyzer, existe os seguintes comandos:

- **Asserção** – *assert* – especifica um teorema;

- **Execução** – *run* – procura um modelo válido para a função;
- **Verificação** – *check* – certifica se determinada asserção é válida, procurando contra-exemplos;

```
open util/boolean as BOOLEAN

one sig start {
   emptyOp : one SortedSet
}
sig Orderable extends Any {
   geqOp : (Orderable) -> one BOOLEAN/Bool
}
sig SortedSet extends Any {
   isInOp : (Orderable) -> one BOOLEAN/Bool,
   insertOp : (Orderable) -> one SortedSet,
   isEmptyOp : one BOOLEAN/Bool,
   largestOp : lone Orderable
}
fact SortedSetConstruction {
   SortedSet in (start.emptyOp).*{x: SortedSet, y:
      {y: x.insertOp[Orderable] | some a1: Orderable | y = x.insertOp[a1]
               and precedes[x,y] and  precedes[a1,y]}}
}
fact domainSortedSet0 {
   all S : SortedSet | S.isEmptyOp != BOOLEAN/True implies one S.largestOp else no S.largestOp
}
fact axiomSortedSet4 {
   all E : Orderable, S : SortedSet | S.isEmptyOp = BOOLEAN/True implies S.insertOp[E].largestOp = E
}
run axiomSortedSet4_1 {
   some E : Orderable, S : SortedSet | (S.isEmptyOp != BOOLEAN/True and S.insertOp[E].largestOp = E)
}for 3
```

Figura 2.3: Especificação em Alloy de um *SortedSet*

Na figura 2.3, é possível observar um exemplo de uma especificação em Alloy. Na figura 2.4, está um modelo de execução de um comando "run"pelo Alloy Analyzer, criado através da funcionalidade de instanciação de modelos (*model-finding*) suportada pelo Alloy, basicamente é criado um ou vários modelos que pretendem simular o comportamento descrito pelo comando. Para que a ferramenta efetue esta pesquisa, é necessário fornecer um valor máximo de instâncias consideradas suficientes no contexto do problema. No que diz respeito à forma como o modelo é apresentado, temos caixas amarelas que correspondem a elementos ou características desses mesmos elementos, as caixas estão ligadas por setas que correspondem a operações que originam as transformações dos elementos ou que verificam determinadas características dos mesmos elementos.

2.5.1 Geração de Testes com Recurso ao Alloy

O Alloy já foi utilizado para geração de testes. A ferramenta TestEra [KYZ+11, KM03] usa especificações em Alloy para testar métodos em programas desenvolvidos em Java. Como

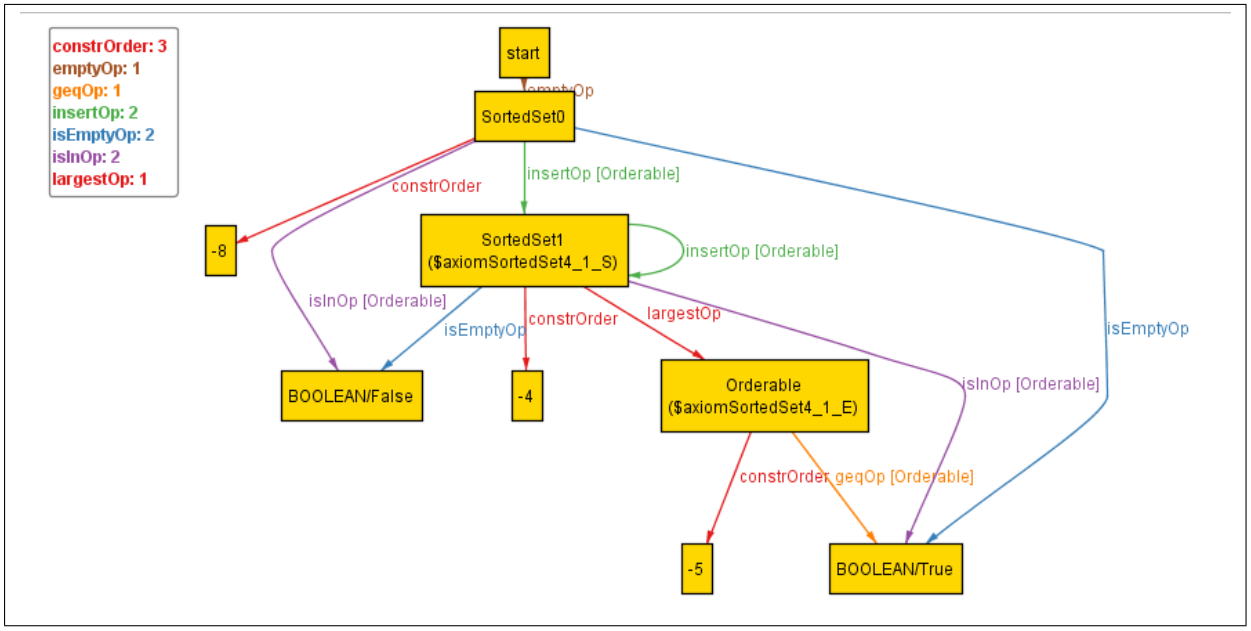

Figura 2.4: Modelo gerado pelo Alloy Analyzer, pela execução do comando *axiomSortedSet4_1* presente na figura 2.3

entradas, o TestEra requer os métodos Java, a especificação formal das pré e pós-condições dos método e os limites e tipos dos casos de teste a serem gerados. Assim, usando as pré-condições e os limites, são gerados casos de teste e aplicados aos métodos e seguidamente os resultados obtidos são comparados com as pós-condições.

Comparativamente com o que se pretende no QUEST, o TestEra tem um objetivo um pouco diferente, mais aproximado ao Spec Explorer, pois são especificadas condições iniciais e finais para cada método em concreto, enquanto no QUEST a especificação visa apenas descrever as propriedades do sistema de forma abstrata e não de forma diretamente relacionada com a implementação.

2.5.2 Modelos Infinitos em Alloy

Uma vez que os modelos especificados em Alloy, são efetuados para avaliação por parte do Alloy Analyzer, leva a que graças à propriedade de instanciação de modelos, eles não possam ser muito complexos, pode levar a uma explosão de estados, bem como não admite modelos infinitos. Este facto é uma grande contrariedade para o uso do Alloy.

Tendo por base esta limitação, foi estudada uma abordagem [KJ05], que basicamente propõe minimizar o problema, criando modelos finitos que satisfaçam a generalidade das caraterísticas do modelo infinito. Essa generalidade que se pretende garantir no modelo finito criado, visa obedecer ao modelo SUA (*Selectors, Uniqueness, Acyclicity*), especificado e demonstrado em [KJ05], que procura garantir que todas as relações do modelo considerado infinito são preservadas no submodelo finito. Este submodelo é gerado efetuando o relaxamento de algumas das suas propriedades, como exemplo a obrigatoriedade de determinado método estar sempre definido.

Esta abordagem é bastante interessante, pois em termos de especificação se conseguirmos um modelo finito que satisfaça um modelo infinito, a lacuna é resolvida. No entanto, é bastante difícil conseguir definir quando é que o modelo finito criado é suficiente, pois as situações e contextos podem variar, tal como as caraterísticas necessárias para especificar a generalidade, ou seja, cumprir o modelo SUA.

Por exemplo, temos a especificação de uma *Tree* em que um nó pode ser nulo, ou então conter um objeto, *Object*, e estar ligado a um nó anterior e outro posterior.

$$datatypeTree = Nil|NodeofTree * Object * Tree$$

$$datatypeObject = Obj1|Obj2|...|ObjN$$

O modelo resultante é infinito, no entanto a abordagem pretende delimitar um conjunto finito que satisfaça a generalidade, como demonstra a figura 2.5.

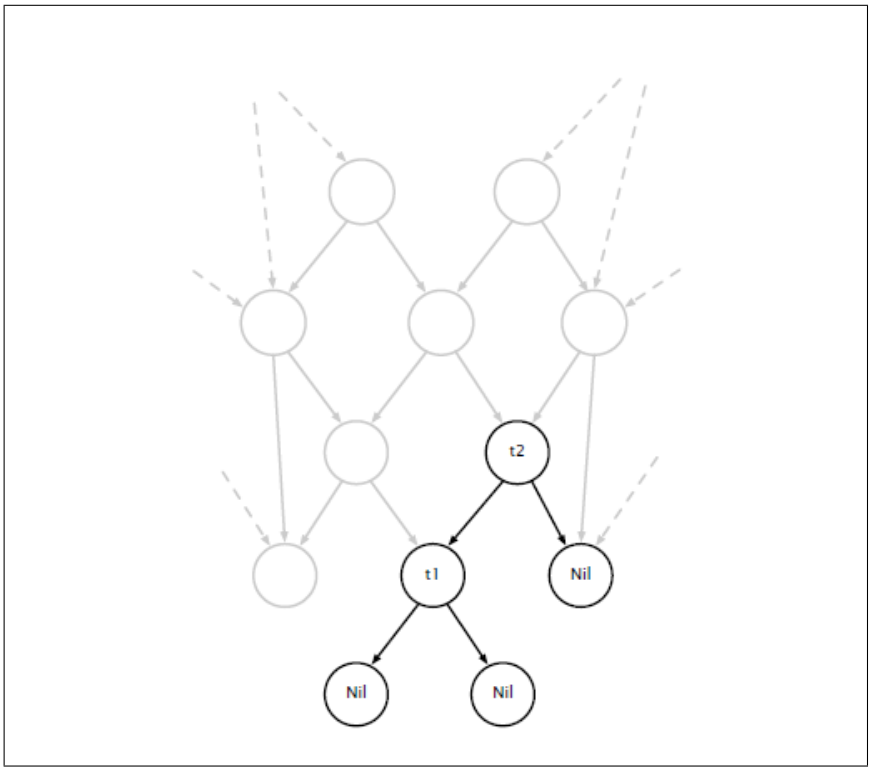

Figura 2.5: Modelo infinito gerado por uma Tree, com realce de um submodelo que satisfaz a generalidade

Capítulo 3

Análise da Versão Anterior do Gerador de Testes (GenT1)

Neste capítulo, será descrito o estado do projecto QUEST [AFP11a, AFPL11a, AFP11b, AFPL11b], até ao início desta dissertação, começando por uma análise a abordagem seguida até esta fase, as ferramentas utilizadas na abordagem seguida, bem como um exemplo elucidativo da utilidade da ferramenta (GenT1). No final do capítulo, será apresentada uma análise, em jeito de conclusão, acerca do projeto desenvolvido nas fases anteriores.

3.1 Abordagem

A abordagem seguida para resolução do problema proposto, geração automática de testes a partir de expressões algébricas escritas na linguagem ConGu, pois é igualmente objetivo deste projeto, tornar a ferramenta do ConGu mais completa.

Como se pode observar na figura 3.1, o sistema desenrola-se a partir de uma especificação algébrica, em ConGu, que através do *parser* do ConGu, fornece uma representação em memória a uma das ferramentas deste projeto, o *Alloy Translator*.

O *Alloy Translator*, é responsável por converter a especificação fornecida em ConGu para uma em Alloy, sendo esta nova especificação submetida ao Alloy Analizer para que este possa gerar modelos representativos da exercitação pretendida dos axiomas da especificação.

No passo seguinte do processo, o Alloy Analyzer retorna os modelos gerados no formato XML, sendo esses modelos processados numa outra ferramenta do presente projeto, o *Test Generator*. Este gerador de testes, gera a partir de cada modelo, um caso de teste, utilizando neste processo também uma representação em memória, fornecida pelo *parser* do ConGu, do mapa de refinamento da especificação algébrica, para fazer a correspondência entre os testes gerados de acordo com a especificação e o mapeamento que é feito para a implementação. Uma outra funcionalidade deste módulo é a geração de *Mock Objects*, para que se possa testar a implementação

do tipo de dados genérico, sem que seja necessário implementar manualmente um tipo parâmetro válido para esse tipo de dados.

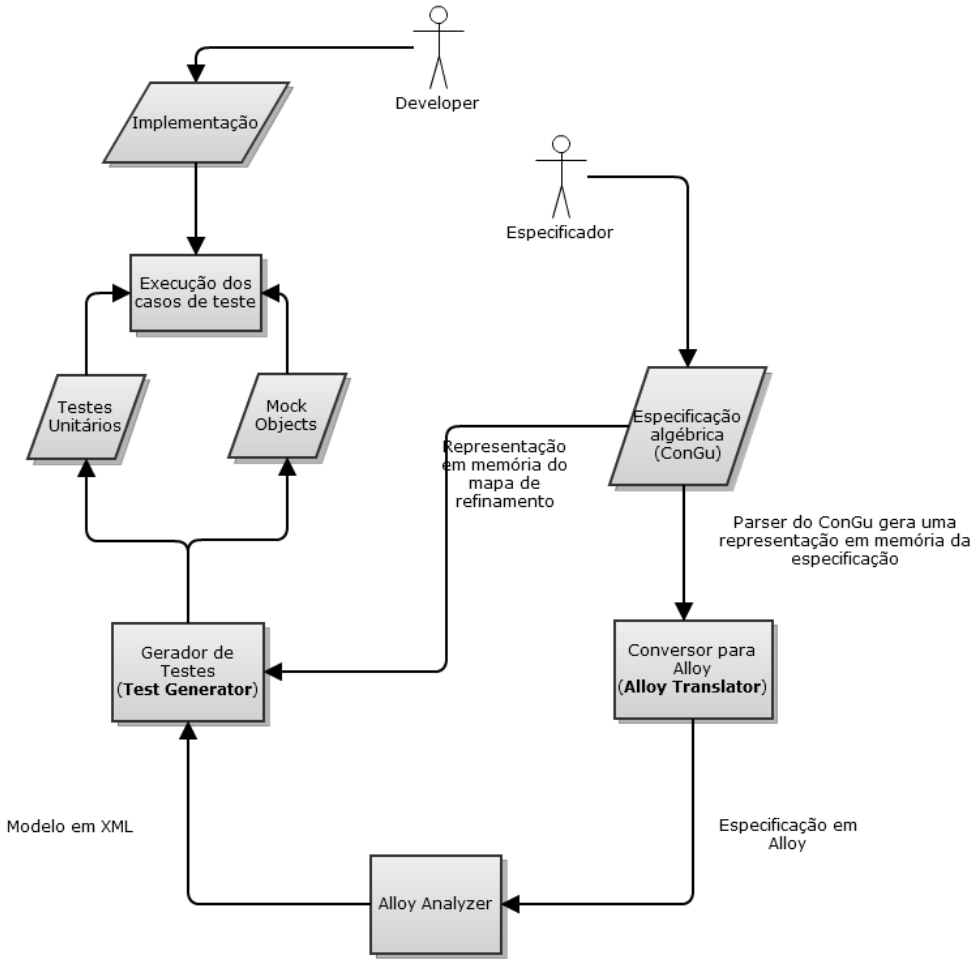

Figura 3.1: Abordagem do Projecto QUEST

3.2 *Alloy Translator*

Como referido anteriormente, um dos componentes criados para este projeto, e com especial relevância, é o Alloy Translator. Este componente tem uma missão crucial na abordagem seguida para resolução do problema, pois pretende fazer uma tradução de uma linguagem de especificação algébrica – ConGu, em outra com uma semântica relativamente diferente – Alloy. Esta tradução tem de ser o mais cuidadosa possível para que se mantenha a conformidade para com a especificação original. Para efetuar essa tradução foram criadas algumas regras, como mostram as tabelas 3.1 e 3.2.

No entanto, este módulo não serve apenas para converter diretamente uma especificação de uma linguagem para outra. Uma vez que o objetivo deste passo intermédio, conversão para Alloy,

Regra	ConGu	Alloy
R1. Género Simples	**sorts** s	**sig** s
R2. Género Parametrizado	s[p]	*s (in sign. decl. and usage)*
R3. Operação Total	op: s –>t'	op: **one** t'
(exceto construtor)	op: s t_1 ... t_n –>t'	op: (t_1 -> ... -> t_n) ->**one** t'
R4. Predicado Total	pr: s	pr: **one** BOOLEAN/Bool
	pr: s t_1 ... t_n	pr: (t_1 ->... ->t_n) ->**one** BOOLEAN/Bool
R5. Operação Parcial	–>?	**lone** em vez de **one**
R6. Instância inicial (*start*)	*não se aplica*	**one sig** start
R7. Construtor	cr: t_1 ... t_n –>s	cr: (t_1 ->... ->t_n) ->**one** s *(in sig start)*

Tabela 3.1: Regras de conversão de ConGU para Alloy – Sintaxe

Constraint (CONGU)	Fact (Alloy)

k^{th} **axiom** in sort s:
$v_1 : s_1; \ldots ; v_n : s_n;$
$formula(v_1, \ldots, v_n);$
	fact axioms_k {
	all $v_1 : s_1, \ldots, v_n : s_n \mid$
	$formula'(v_1, \ldots, v_n)$
	}

k^{th} **domain** restriction in sort s:
$v_1 : s_1; \ldots ; v_n : s_n;$
$op(v_1, \ldots, v_n)$ **if**
$cond(v_1, \ldots, v_n);$
	fact domains_k {
	all $v_1 : s_1, \ldots, v_n : s_n \mid$
	$cond'(v_1, \ldots, v_n)$
	implies one $op'(v_1, \ldots, v_n)$ **else no** $op'(v_1, \ldots, v_n)$
	}

Tabela 3.2: Regras de conversão de ConGU para Alloy – Axiomas e restrições de domínio

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)

Tabela 3.3: Regras de decomposição de axiomas em mintermos

é a utilização da capacidade de procura de modelos do Alloy Analyzer, este módulo é igualmente responsável por criar comandos de execução para os axiomas, para serem exercitados pelo Alloy Analyzer. Esses comandos são gerados para procurar satisfazer o critério de cobertura que consiste em gerar um teste para cada mintermo de cada axioma decomposto na sua forma normal disjuntiva completa –FDNF. Sendo um mintermo, um termo conjuntivo em que cada variável Booleana aparece uma única vez, possivelmente negada. Essa decomposição faz-se de acordo os exemplos da tabela 3.3.

3.3 *Test Generator*

Este componente da aplicação, é responsável pela extração de testes a partir dos modelos gerados pelo Alloy Analyzer, bem como pela extração de *Mock Objects*.

A forma de extração de testes consiste em construir asserções entre estados e sequências de operações aplicados a estados, ou seja, um estado inicial ao qual são aplicadas um conjunto de operações é comparado com o estado que é suposto atingir com essas operações. Na figura 3.2, temos um exemplo de um modelo gerado pelo Alloy Analyzer, e na figura 3.3 temos um excerto do caso de teste, em JUnit, que é gerado. É importante realçar que no caso de teste da figura 3.3, foi efetuado um mapeamento dos nomes de forma manual, para que estes correspondessem com os da implementação.

No que diz respeito aos *Mock Objects*, são implementações parciais de géneros parâmetros, através da suas especificação e dos modelos gerados, e que visam servir de parâmetro para os testes gerados para classes genéricas.

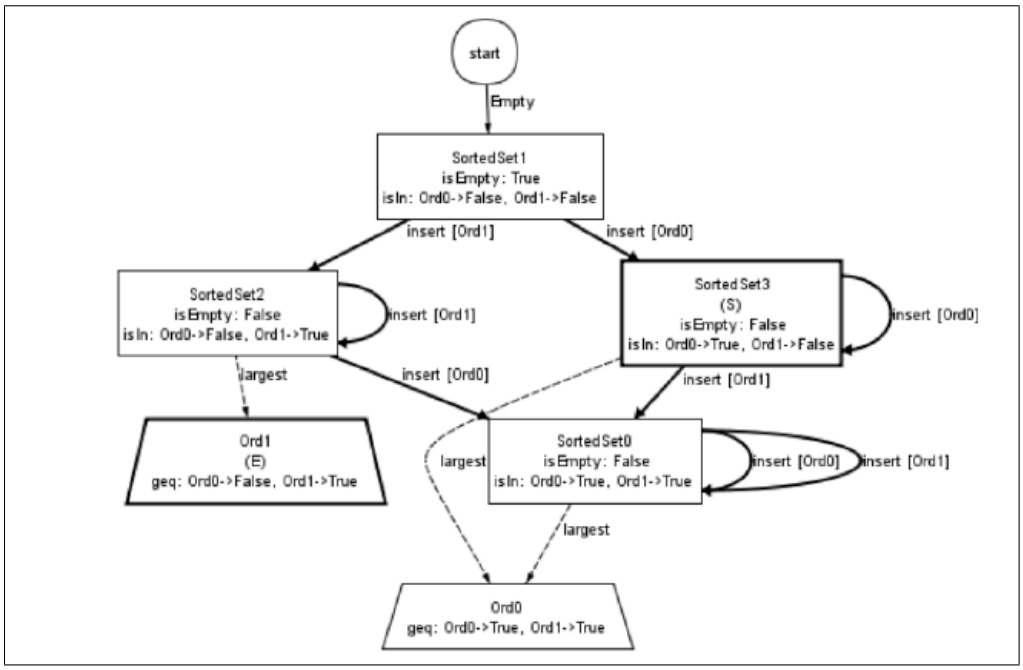

Figura 3.2: Modelo representativo de um SortedSet, gerado pelo Alloy Analizer

```
private interface Factory<T> {T create();}
private void axiomSortedSet4Tester(Factory<TreeSet<OrderableMock>> sFact,
            OrderableMock e) {
      TreeSet<OrderableMock> s_0 = sFact.create();
      TreeSet<OrderableMock> s_1 = sFact.create();
      if(s_0.isEmpty()) {
            s_1.insert(e);
            assertTrue(s_1.largest().equals(e));
      }
}
@Test public void test0_axiomSortedSet4_1(){
      // mock objects for the parameter
      final OrderableMock ord0 = new OrderableMock();
      final OrderableMock ord1 = new OrderableMock();
      ord0.add_greaterEq(ord0, true);
      ord0.add_greaterEq(ord1, false);
      ord1.add_greaterEq(ord0, true);
      ord1.add_greaterEq(ord1, true);
      // factory objects for the axiom var's of parameterized type
      Factory<TreeSet<OrderableMock>> sFact =
         new Factory<TreeSet<OrderableMock>>() {
            public TreeSet<OrderableMock> create() {
                  TreeSet<OrderableMock> s = new TreeSet<OrderableMock>()
                  s.insert(ord0);
                  return s; }
         };
      // checking the axiom
      axiomSortedSet4Tester(sFac, ord1);
}
//… other axioms and test cases
```

Figura 3.3: Excerto do caso de teste JUnit gerado a partir do modelo da figura 3.2

3.4 Limitações

Atendendo ao objetivo nuclear deste projeto, a geração automática de testes para implementações de ADTs genéricos, especificados a partir de especificações algébrica, sendo essas mesmas especificações o instrumento a utilizar para essa mesma geração, a abordagem descrita ao longo deste capítulo é capaz de responder positivamente a grande parte do objetivo. Ou seja, o sistema é capaz, com esta abordagem, gerar testes unitários a partir de especificações algébricas. Os testes gerados são representativos, sendo capazes de validar a conformidade da implementação para com a especificação a um nível de confiança elevado.

No entanto, existem alguma limitações, e apesar dos módulos 3.3 e 3.2 já apresentarem um bom nível de evolução, faltava:

- integração dos módulos que compõem a ferramenta;
- algumas regras de conversão entre módulos, ConGu para Alloy e Alloy para Junit;
- atualização para versão mais recente do ConGu;
- mapeamento automático entre os testes abstratos gerados e a implementação em Java;
- falta igualmente uma abordagem para o caso das especificações não satisfazíveis por modelos finitos.

Interessa também referir que o GenT1 ainda não tinha sido testado para muitos casos, e os casos para que foi testado eram bastantes simples, sendo por isso necessário testá-lo para casos que propusessem problemas diferentes e relevantes, à ferramenta.

Capítulo 4

Conceção da Nova Versão do Gerador de Testes (GenT2)

Neste capítulo, é apresentada a forma como se respondeu aos problemas propostos tanto no capítulo 1 como algumas limitações identificadas na ferramenta descrita no capítulo 3. De tal forma que ao longo deste capítulo, pode ser encontrada: a nova estruturação da ferramenta; a solução para algumas limitações da ferramenta a nível de funcionalidades, nomeadamente a extensão para especificações não satisfazíveis por modelos finitos, o suporte de construtores com argumentos não primitivos e a utilização do mapeamento entre a especificação e a implementação para geração de testes; a forma de definição automática dos limites de pesquisa por parte do AlloyAnalyzer; e forma como são identificados e tratados os testes gerados que não respeitam as pré-condições das operações e dos axiomas.

4.1 Estruturação da Aplicação

Partindo da abordagem descrita no capítulo 3 e da ferramenta já desenvolvida em fases anteriores do projeto, foi efetuada uma restruturação da ferramenta, sendo ela agora organizada de acordo com o esquema da figura 4.1. Basicamente foi feita uma integração dos componentes principais do projeto, a conversão para a Alloy e a geração de testes. Assim, todo o processo de geração de testes em JUnit a partir de especificações em ConGu é ligado de forma automática.

Deste modo, agora todos os elementos resultantes da fase de conversão para Alloy que são necessários para gerar os testes, são automaticamente providenciados. Bem como se poupa a execução de algumas operações que seriam comuns a ambas as fases, principalmente a compilação do código da especificação e refinamento (que será abordado em maior detalhe no subcapítulo 4.4).

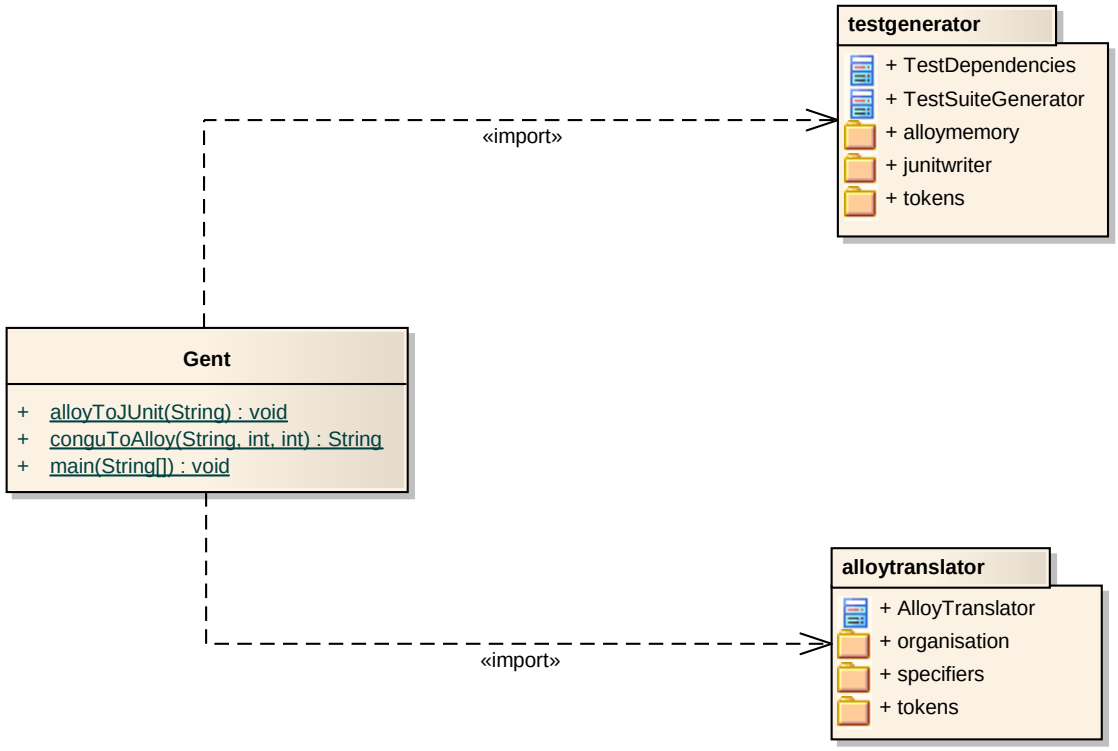

Figura 4.1: Esquema geral da Ferramenta desenvolvida

Para além da estruturação efetuada, foram também atualizadas, para a versão mais recente, as duas principais ferramentas que servem de auxílio, o AlloyAnalyzer [Jac12a] e o ConGu [FCU12].

4.2 Extensão para Especificações não Satisfazíveis por Modelos Finitos

O principal problema proposto para esta dissertação é a geração de testes para especificações que não são satisfazíveis por modelos finitos. Este problema advém da utilização do AlloyAnalyzer como passo intermédio, uma vez que ele procura, modelos finitos que satisfaçam a especificação. Assim, quando a especificação se refere a estruturas de dados de tamanho indefinido, teoricamente infinito, como nos exemplos das figuras 4.2 e 4.3 o modelo a gerar pelo AlloyAnalyzer será também ele infinito, logo não é possível a sua geração.

De modo a exemplificar o problema, vamos nos focar no caso de uma pilha de dados – *Stack*, para tal é apresentado o seguinte excerto da especificação de uma *Stack*:

$$\text{op } push : Stack\ Elem \rightarrow Stack$$
$$\text{op } pop : Stack\ Elem \rightarrow ?Stack$$
$$\text{ax } \forall S : Stack, E : Elem \bullet pop(push(S,E)) = S$$

Podemos observar nesta especificação, dois operadores: o **push**, que é uma função total, ou seja, está sempre definido para qualquer argumento, sendo possível utilizá-lo em qualquer momento; e o **pop**, que é uma função parcial, pois não está definido quando a *stack* está vazia. Está

```
specification Queue[Element]
        sorts
                Queue[Element]
        constructors
                make: --> Queue[Element];
                enqueue: Queue[Element] Element --> Queue[Element];
        observers
                front: Queue[Element] -->? Element;
                dequeue:  Queue[Element] -->? Queue[Element];
                isEmpty: Queue[Element];
        domains
                Q: Queue[Element];
                front(Q) if not isEmpty(Q);
                dequeue(Q)  if not isEmpty(Q);
        axioms
                Q: Queue[Element];
                E: Element;
                front(enqueue(Q, E)) = E when isEmpty(Q) else front(Q);
                dequeue(enqueue(Q, E)) = Q when isEmpty(Q) else enqueue(dequeue(Q), E);
                isEmpty(make());
                not isEmpty(enqueue (Q, E));
end specification
```

Figura 4.2: Especificação de uma *Queue* em ConGu

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

Figura 4.3: Especificação de uma pilha de dados em ConGu

também representado um axioma, que indica que se fizer um *push* seguido de um *pop*, o resultado será igual à *stack* original. Uma vez que temos o *push* que está sempre definido, mesmo sendo, por exemplo, uma *stack* que seja só instanciada por um dado tipo de elementos, vai evoluir para

um modelo infinito, como podemos observar na figura 4.4

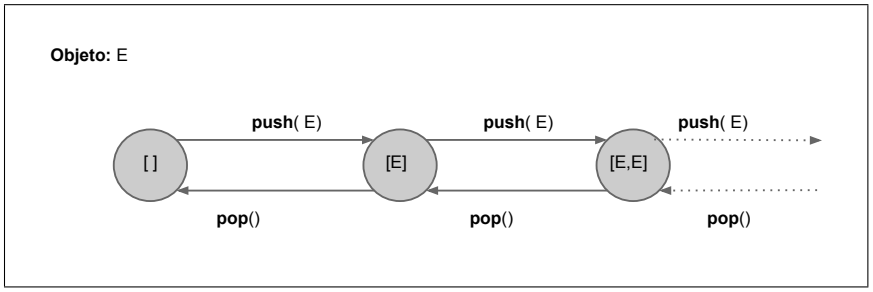

Figura 4.4: Modelo gerado por uma Stack instanciada por objetos de um tipo "E"

A forma encontrada para solucionar este problema consiste em procurar um modelo finito que satisfaça a generalidade das propriedades do modelo infinito, como proposto em [KJ05]. Ou seja, a solução encontrada passa por efetuar um relaxamento da especificação, alterando-a para que seja satisfazível pelo AlloyAnalyzer. Nesse sentido, os construtores que são operações totais (*e.g.* operação *push* no caso da *Stack*) passam a operações parciais. São ainda introduzidas variáveis de guarda de modo a garantir a integridade da especificação.

No caso apresentado como exemplo, a especificação seria convertida na seguinte:

$$\text{op } push : Stack\ Elem \rightarrow \textbf{?}Stack$$

$$\text{op } pop : Stack\ Elem \rightarrow ?Stack$$

$$\text{ax } \forall S : Stack, E : Elem \bullet \textbf{defined(push(S,E))} \Rightarrow pop(push(S,E)) = S$$

Estas alterações são introduzidas durante a conversão para Alloy, e resultam nas modificações da especificação em Alloy representadas nas figuras 4.6 e 4.5, para o exemplo da *Stack*.

Axioma Original	**Axioma Reescrito**
lhs = *rhs*	**guard**(*lhs*) ⇒ *lhs* = *rhs*
lhs = *rhs* ***if*** *cond*	**guard**(*cond*) ∧ **guard**(*lhs*) ∧ *cond* ⇒ *lhs* = *rhs*
lhs = *rhs* ***when*** *cond*	**guard**(*cond*) ∧ **guard**(*lhs*) ∧ *cond* ⇒ *lhs* = *rhs*
lhs ***or*** *rhs*	**guard**(*lhs*) ∧ **guard**(*rhs*) ⇒ *lhs* ***or*** *rhs*
lhs ***and*** *rhs*	**guard**(*lhs*) ∧ **guard**(*rhs*) ⇒ *lhs* ***and*** *rhs*

Tabela 4.1: Reescrita dos axiomas

Tipo de Guarda	**Forma de Guarda**
guard(*constr(exp-args)*)	**defined**(*constr(exp-arg1)*) *and ... and* **defined**(*constr(exp-argn)*)
guard(*non-constr(exp-args)*)	**guard**(*exp-arg1*) *and ... and* **guard**(*exp-argn*)
guard(*var*)	*true*

Tabela 4.2: Forma de guarda dos axiomas

Na figura 4.5, na definição da operação *push* deixa de ser usado o **one**, que no contexto significa que a operação existe em todos os estados que possam ser atingidos, passando-se a usar o **lone**, que indica que a operação pode ou não existir. Por sua vez, na figura 4.6, temos um exemplo de uma condição de guarda introduzida num facto da especificação em Alloy, semelhante às que são introduzidas também nos comandos de execução *run*. Essas condições, consistem em garantir que o lado esquerdo da expressão, que contém um desses construtores que foi modificado, esteja definido. Na tabela 4.1, está representada a forma de reescrita que é aplicada aos axiomas nos diversos casos, sendo de realçar a obrigatoriedade de guarda do lado esquerdo das expressões. Por sua vez, na tabela 4.2 está representada a forma de guarda aplicada, sendo que daí pode-se observar que apenas são aplicadas operações aos construtores.

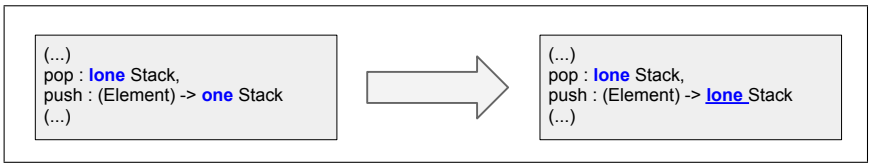

Figura 4.5: Passagem de operações totais a parciais em *Alloy*

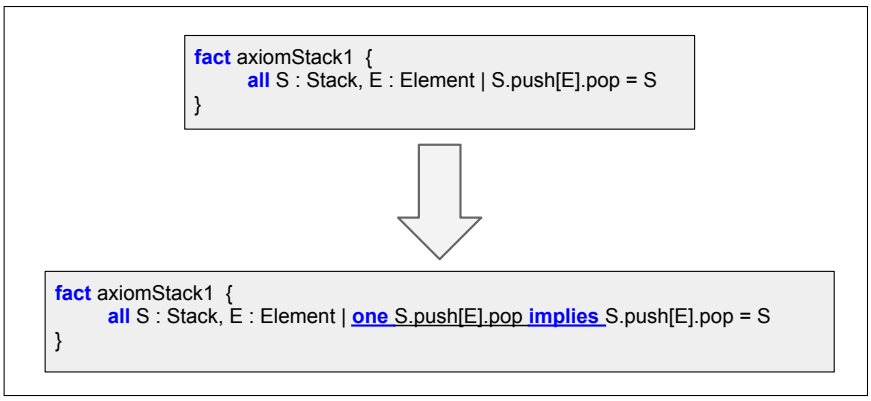

Figura 4.6: Variáveis de guarda na especificação *Alloy*

Assim, o modelo a ser gerado já é finito, como apresenta a figura 4.7, e neste caso em concreto, não se perde as propriedade do modelo infinito para a geração de testes.

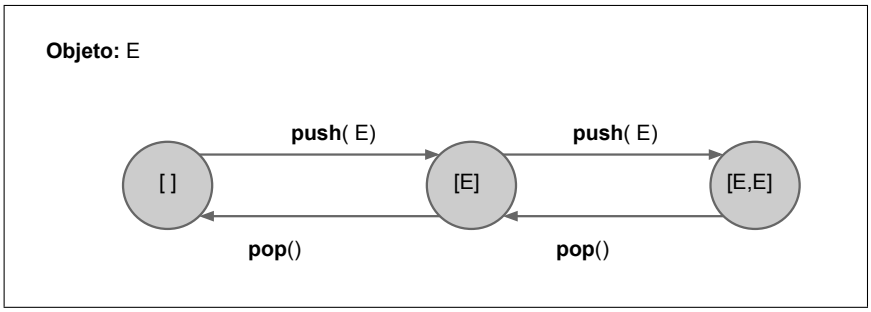

Figura 4.7: Modelo gerado por uma Stack reduzida a um modelo finito

No capítulo 5, encontra-se a experimentação que serve de validação a esta abordagem.

```
refinement<E>
        Element is E{
        }
        Stack[Element] is Stack<E> {
                empty: Stack[Element] is boolean isEmpty();
                push: Stack[Element] Element --> Stack[Element] is void push(E e);
                peek: Stack[Element] -->? Element is E peek();
                pop: Stack[Element] -->? Stack[Element] is void pop();
        }
end refinement
```

Figura 4.8: Mapa de refinamento da especificação da figura 4.3 para a implementação

```
/***** Axiom axiomStack1 *****/
private<E extends java.lang.Object> void axiomStack1Tester(CoreVarFactory<Stack<E>>
S_Factory, E e, String testId) {
   Stack<E> S3 = S_Factory.create();
   S3.push(e);
   S3.pop();
   Stack<E> S4 = S_Factory.create();

   assertTrue(S3.equals(S4));
}

@Test
public void test2_axiomStack1_0() {
   //Mocks' setup
   final ObjectMock Element_0 = new ObjectMock();

   //Factories core var setup
   CoreVarFactory<Stack<ObjectMock>> S_Factory =
      new CoreVarFactory<Stack<ObjectMock>>(){
         @Override
         public Stack<ObjectMock> create(){
            Stack<ObjectMock> __var__0 = new Stack<ObjectMock>();

            return __var__0;
         }
   };

   //Test the Axiom
   axiomStack1Tester(S_Factory, Element_0, "test3_axiomStack1_0");
}
```

Figura 4.9: Testes gerados para um axioma da especificação da Stack(figura 4.3)

4.3 Construtores com Argumentos Não Primitivos

Existem casos onde as operações de construção têm como argumentos tipos não primitivos, ou seja, os argumentos necessitam também de serem construídos. Assim sendo, a conversão para Alloy necessita de tratar estes casos de forma especial, uma vez que o modelo gerado pelo AlloyAnalyzer terá de possuir a informação necessária para a construção do argumento não primitivo.

Um exemplo do problema descrito, encontra-se na figura 4.10, onde temos uma especificação de um *Set* com três tipos de construtores: o primeiro – *empty* – sem argumentos; o segundo – *singleton*– com um argumento primitivo do tipo *Element*; e um terceiro – *union* – que recebe

como argumento dois objetos que não são primitivos, dois *Sets* do mesmo tipo da especificação. Assim, para conversão para Alloy, foi seguido a especificação representada na tabela 4.3, no que se refere à geração do facto que limita a construção do objeto, sendo que o resultado desta conversão para o exemplo dado pode-se observar na figura 4.11.

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

Figura 4.10: Especificação de uma estrutura de dados com um construtor com parâmetros construídos

```
fact SetConstruction {
    Set in (start.empty + {x: start.singleton[Element] | some a1:Element |
        x=start.singleton[a1] and precedes[a1,x]}).* {x: Set, y: {y: x.union[Set] |
        some a1: Set | y=x.union[a1] and precedes[x,y] and precedes[a1,y]}}
}
```

Figura 4.11: Facto de construção em Alloy gerado para o Set da figura 4.10

Importa ainda referir, como foi proposto em [AFP11a], foi adicionado uma assinatura abstrata na geração do ficheiro Alloy, representada na figura 4.12, que serve de raiz para todas outras assinaturas. Basicamente, este assinatura serve para indicar a ordem de construção dos objetos, através do predicado *precedes*. Desta forma, torna-se possível construir caminhos de construção, na geração dos testes, que sejam válidos, impedindo que um construtor utilize um argumento que ainda não foi construído.

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

Figura 4.12: Assinatura raiz das assinatura geradas em Alloy

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

Tabela 4.3: Regras de conversão de ConGU para Alloy – factos de construção [AFP11a]

4.4 Mapeamento entre a Especificação e a Implementação na Geração de Testes

Nesta dissertação procedeu-se também ao mapeamento entre a especificação algébrica e a implementação em Java, de modo a que os testes gerados cumprissem os requisitos propostos pela especificação e simultaneamente utilize a sintaxe utilizada na implementação. Para tal recorreu-se ao mapa de refinamento suportado pelo ConGu, descrito em 2.2.2.

A utilização deste componente, compilado pelo ConGu, processou-se de forma bastante simples e direta. Para efetuar a correspondência de nomes, quer de operações/funções quer de *sorts*/ classes, criaram-se estruturas de dados auxiliares que na sua construção efetuavam a pesquisa da sua correspondência direta (*e.g.* para corresponder à estrutura *Field* foi criado um *FieldRefined* que contém todas as características do *Field* normal, acrescidas do valor da variável na implementação).

```
private<E extends IOrderable<E>> void axiomSortedSet2Tester(E e, String testId) {
    TreeSet<E> __var__1 = new TreeSet<E>();

    assertTrue((__var__1.isIn(e) != true));
}
```

Figura 4.13: Caso de teste em *Junit*

Na figura 4.13, está destacado outro uso dado ao mapa de refinamento, a introdução da limitação tipo genérico aceite pela função, que é semelhante ao declarado na classe genérica, que é

encontrado pelo compilador, quando este verifica o mapeamento. Também relacionada com esta funcionalidade do mapa de refinamento, está a sua utilização para a criação das *Mock Classes*, pois indica de que tipos elas têm de descender para serem aceites pela classe genérica.

4.5 Descoberta Automática de Limites de Procura em Alloy

Um elemento fundamental na construção de cada comando em Alloy, para ser executado pelo AlloyAnalyzer, é o campo de pesquisa, ou seja, o número de instâncias a utilizar na pesquisa. Por defeito o valor utilizado pelo AlloyAnalyzer é três, sendo assim necessária a redefinição deste valor. Neste sentido, estão implementadas duas formas de definição do campo de pesquisa.

```
por cada comando run
    começar com número de instâncias = 1

    tentar correr no AlloyAnalyzer
        se exceder tempo ou memória máxima disponível
            considerar comando  não satisfazível
            passar ao comando seguinte
    verificar resultado
        se é satisfazível
            passar ao comando seguinte
        se não é
            se já atingiu o número máximo de instâncias
                considerar o comando não satisfazível
                passar ao comando seguinte
            se ainda não
                incrementar o número de instância e voltar a "tentar correr no AlloyAnalyzer"
```

Figura 4.14: Pseudo-código da definição de limites de procura em Alloy

Um dos métodos utilizados para o fim sugerido pelo tópico, consiste em ser o próprio utilizador a fornecer esses dados, algo que pode ser utilizado com alguma facilidade em sistemas mais simples em que é possível à partida ter uma ideia do número de elementos necessários para satisfazer os comandos de execução e gerar um modelo válido.

No entanto, na maioria dos casos é bastante complexo conhecer um valor que satisfaça todos os comandos *run* exequíveis e simultaneamente não faça com que seja um valor demasiado exagerado para alguns comandos, fazendo com que a execução destes seja mais lenta, bem como sejam geradas muitas instâncias que serão desnecessárias na geração dos testes, criando "lixo"no código de teste. Com o intuito de responder a esta questão, quando não é especificado pelo utilizador nenhum limite, é corrido cada comando individualmente com um determinado valor inicial para esse mesmo limite, valor esse que é incrementado até o AlloyAnalyzer conseguir gerar um modelo para esse comando, sendo essa busca limitada por um valor máximo de procura e por um certo tempo de execução, como pode ser verificado pela figura 4.14. Nos casos onde é definido um valor pelo utilizador, se o comando não for satisfazível, é feito o mesmo procedimento, sendo que o ponto de partida é o valor definido pelo utilizador.

4.6 Violação de Pré-condições

Na abordagem seguida até ao momento, ainda não são considerados testes fora do domínio da especificação, entretanto, pelos métodos de geração de testes utilizados, são gerados testes que podem não cumprir as pré-condições para testes dentro do domínio do problema, pelo que é necessário tratar esses casos de modo a que não crie problemas de interpretação dos resultados nem origine erros e exceções inesperadas.

Este tipo de testes que viola as pré-condições, pode aparecer vindo de dois caminhos diferentes: pode ser associado a restrições de domínio, definidas para construtores parciais, através de um campo opcional do ConGu – *domains*; pode também ser originado a quando a aplicação da regras da tabela 3.3 para a decomposição do axioma em mintermos, podendo ser originados mintermos que não cumpram as pré-condições de axiomas condicionais.

```
/***** Axiom axiomSortedSet4 *****/
  private<E extends IOrderable<E>> void axiomSortedSet4Tester(E e, CoreVarFactory<TreeSet<E>>
S_Factory, String testId) {
            TreeSet<E> S3 = S_Factory.create();

            if((S3.isEmpty() == true)) {
               TreeSet<E> S4 = S_Factory.create();
               S4.insert(e);

               assertTrue((S4.largest() == e));
            }else{
               System.out.println(testId+": Axiom precondiction not met");
            }

  }
```

Figura 4.15: Caso de teste gerado para o axioma "*largestOp(insertOp(S, E)) = E if isEmptyOp(S)*"da especificação da figura 4.17

Assim, nos casos de teste correspondentes a cada axioma, as pré-condições são convertidas em expressões condicionais que no caso de não serem cumpridas, notificam o teste, que exercita esse axioma, em que ocorreu essa violação, sem fazer com que o teste falhe. Como se pode verificar na figura 4.15, na especificação existe uma pré-condição tal que o caso referido no axioma só se verifica quando a estrutura de dados é vazia, assim no código de teste correspondente, é gerado, como está realçado na referida figura, uma condição que verifica isso mesmo.

Para além da verificação que se faz só no caso de teste que corresponde a cada axioma, também é efetuada em todos os testes, que exercitam esses axiomas, quando é chamado uma operação que está especificada como sendo um construtor e que possui restrições de domínio, é verificado se esse construtor pode ser aplicado, ou seja, se não será violada nenhuma pré-condição. Na figura 4.16, é apresentado um exemplo em que uma operação que contém uma restrição de domínio, quando no código é chamada, verifica-se se as condições em que está a ser aplicada para só ser executada se essas condições forem verificadas.

```
public MySSet<ObjectMock> create(){
        MySSet<ObjectMock> __var__0 = new MySSet<ObjectMock>();
        if((__var__0.isIn (Element_0) != true) || (__var__0.isIn (Element_0) != true)){
                __var__0.insertOneOf (Element_0, Element_0);
        }
        else{
                System.out.println("Invalid");
        }
        if((__var__0.isIn (Element_1) != true) || (__var__0.isIn (Element_0) != true)){
                __var__0.insertOneOf (Element_0, Element_1);
        }
        else{
                System.out.println("Invalid");
        }
        return __var__0;
}
```

Figura 4.16: Verificação da restrição de domínio especificada por "*insertOneOf(S, E, F) if not isIn(S, E) or not isIn(S, F)*"

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

Figura 4.17: Especificação de uma estrutura de dados com construtor sub-especificado

Capítulo 5

Experimentação

Neste capítulo, depois de se apresentar o modo de utilização do GenT2, são apresentados alguns resultados experimentais.

5.1 Modo de Utilização

A ferramenta desenvolvida têm vários modos de funcionamento. A execução da ferramenta dá-se através da linha de comandos, executando o seguinte comando:
java -jar Gent.jar [opt] [dir] [maxBound] [exactBound] [-wr]

- *opt* → operação que se pretende executar dentro das opções: *conguToJunit, conguToAlloy, alloyToJunit*;
- *dir* → diretório contento os ficheiros das especificações, o mapa de refinamento e as classes compiladas da implementação (**.class*), no caso em que o parâmetro *opt* é *conguToJunit* ou *conguToAlloy*; ficheiro Alloy, quando *opt* é *alloyToJunit*.
- *maxBound* → indica o número máximos de instâncias a utilizar na execução dos comandos em Alloy, é um parâmetro opcional e só é válido quando o parâmetro *opt* é *conguToAlloy* ou *conguToJunit*;
- *exactBound* → indica o número exato de instâncias do tipo principal a usar na execução dos comandos *run*, é um parâmetro opcional e só é válido quando o parâmetro *opt* é *conguToAlloy* ou *conguToJunit*;
- *-wr* → *flag* a utilizar no caso de se pretender usar a abordagem especificada na secção 4.2, é um parâmetro opcional e só é válido quando o parâmetro *opt* é *conguToAlloy* ou *conguToJunit*.

Importa referir que, a geração de testes tendo como ponto de partida um ficheiro com o modelo já convertido para Alloy, utilizando a opção *alloyToJunit*, não leva em consideração o mapa de refinamento, podendo os testes gerados não serem completamente compatíveis, em termos de sintaxe, com a implementação.

5.2 Exemplos de Teste e Matriz de Rastreabilidade

	Exemplos de teste				
	Estrutura ordenada (figura 2.1)	Estrutura com parâmetros complexos (figura 4.10)	Estrutura com restrições de domínio (figura 4.17)	Pilha de dados (figura 4.3)	Fila de espera (figura 4.2)
Mapa de Refinamento (secção 4.4)	√		√	√	√
Não satisfazível por modelos finitos (secção 4.2)				√	√
Construtores com argumentos não primitivos (secção 4.3)		√			
Restrições de domínio para os construtores (secção 4.6)			√		
Violação de Pré-condições (secção 4.6)	√	√	√		
Pesquisa automática de limites no alloy (secção 4.5)	√	√	√	√	√

Tabela 5.1: Matriz de rastreabilidade dos testes efetuados

Na tabela 5.1, está a matriz de rastreabilidade que indica os tipos de funcionalidades, descritas ao longo deste documento, que são cobertas pelos diversos exemplos de especificações, mais relevantes, utilizadas na experimentação do projeto desenvolvido e apresentados nas figuras referidas.

Na secção 4.2, temos um exemplo completo dos dados de entrada e dados de saída do funcionamento da ferramenta, assim na figura 4.3 temos a especificação, na figura 4.8 temos o mapa de refinamento e na figura 4.9 um excerto do código de teste produzido, a juntar-se a estes elementos é ainda utilizado os *bytecodes* das classes implementadas.

5.3 Resultados

Nas tabelas 5.2 e 5.3, estão os resultados da experimentação. Na tabela 5.2, temos os resultados para a abordagem base, ou seja, aquela que faz uma conversão direta da especificação em

	Estrutura ordenada (figura 2.1)	Estrutura com parâmetros complexos (figura 4.10)	Estrutura com restrições de domínio (figura 4.17)	Pilha de dados (figura 4.3)	Fila de espera (figura 4.2)
Linhas de especificação ConGU	25	24	24	22	23
Número de Axiomas	9	10	7	4	4
Tempo de execução do Alloy	8122 ms	3300 ms	284607 ms	2458 ms	3158 ms
Comandos de execução gerados	28	21	105	4	6
Comandos de execução não satisfeitos	7	0	37	3	5
Comandos de execução não satisfazíveis teoricamente	7	0	37	3	5
Testes gerados	21	21	68	1	1
Testes que não cumprem condições de guarda dos axiomas	9	2	58	0	0
Testes que não cumprem restrições de domínio	0	0	1	0	0
Cobertura de Código	93.5%	95.5%	100%	22.6%	10.4%
Mutation Testing Score	75%	90%	83%	17%	23%

Tabela 5.2: Resultados da experimentação utilizando a abordagem inicial

ConGu para a especificação em Alloy, e na tabela 5.3 está os resultados obtidos para a abordagem desenvolvida tendo em vista a resolução do problema que era proposto acerca das especificações que não são satisfeitas por modelos finitos.

Toda experimentação foi realizada num computador portátil com CPU de 64 bits Intel Core i7-2670QM 2.20GHz com 6 GB de RAM, a correr com Windows 7 da Microsoft. A cobertura de código é fornecida pelo Eclemma [Ecl] e a pontuação da *Mutation Testing* é atribuída pelo Jumble [Jum], sendo neste caso utilizados todos os tipos de mutações possibilitadas pela ferramenta (condicionais, valores de retorno, constantes, etc.).

Começando por fazer uma análise comparativa das duas abordagens, podemos verificar que

	Estrutura ordenada (figura 2.1)	Estrutura com parâmetros complexos (figura 4.10)	Estrutura com restrições de domínio (figura 4.17)	Pilha de dados (figura 4.3)	Fila de espera (figura 4.2)
Linhas de especificação ConGu	25	24	24	22	23
Número de Axiomas	9	10	7	4	4
Tempo de execução do Alloy	10358 ms	3059 ms	290242 ms	1246 ms	2044 ms
Comandos de execução gerados	28	21	105	4	6
Comandos de execução não satisfeitos	7	0	37	0	0
Comandos de execução não satisfazíveis teoricamente	7	0	37	0	0
Testes gerados	21	21	68	4	6
Testes que não cumprem condições de guarda dos axiomas	9	2	57	0	0
Testes que não cumprem restrições de domínio	0	0	1	0	0
Cobertura de Código	93.5%	95.5%	100%	75.8%	71.3%
Mutation Testing Score	79%	90%	91%	64%	56%

Tabela 5.3: Resultados da experimentação utilizando a abordagem adaptada para modelos infinitos

nos exemplos em que a especificação conduzia a um modelo infinito, os comando que não eram satisfazíveis com a abordagem base, com a nova abordagem já são satisfazíveis. Podemos também verificar que quer a cobertura de código quer a pontuação do *Mutation Testing* aumenta para valores semelhantes aos das especificações satisfazíveis por modelos finitos na abordagem base. É importante também salientar, que com esta nova abordagem, os exemplos de especificações satisfazíveis por modelos finitos não perdem qualidade nos testes gerados.

Os valores de *Mutation Testing* e de cobertura de código são afetados pelo facto do método *equals* ser necessário estar implementados e não ser testado em específico, o que leva a que parte do seu código não seja coberto, bem como certas modificações não levem a que sejam detetadas

no processo de *Mutation Testing*, isso faz com que em especificações simples que conduzem a implementações curtas, logo influencie bastante estas métricas.

Outro dado interessante de observar, refere-se à quantidade de comandos de execução gerados para o exemplo da estrutura de dados com construtor sub-especificado, especificação da figura 4.17, e isso deve-se à forma de geração de mintermos descrita anteriormente, que leva a que quando uma especificação tem muitas condições, a sua decomposição leva a que sejam gerados muitos casos diferentes, sendo que, como também é possível observar, apenas dez testam realmente dentro do domínio e cumprem as pré-condições dos axiomas. Isto leva a que como há muitos comandos para procurar o limite, limite esse que em muitos casos não é encontrado, o tempo de execução do AlloyAnalyzer, nestes casos, aumenta muito.

Na procura dos limites de pequisa no AlloyAnalyzer, para esta experimentação, foi definido como teto máximo os seis elementos e cinco minutos de tempo máximo de espera até considerar insatisfazível nessas condições. Nos casos referidos nenhum comando foi considerado insatisfazível por atingir estes limites, pois eram todos insatisfazíveis mesmo em termos teóricos.

Capítulo 6

Conclusões

Neste capítulo, é feito um resumo do que foi feito ao longo desta dissertação, sendo apresentadas a conclusões acerca dos resultados obtidos, tal como acerca de limitações e trabalho a desenvolver no futuro para colmatar essas limitações.

6.1 Resultados

Os objetivos propostos inicialmente, foram atingidos com sucesso, uma vez que todos os tópicos que se pretendiam desenvolver nesta dissertação, apresentados em 1, foram cumpridos e sobre os quais os testes aplicados revelaram resultados experimentais bastante positivos.

Ao longo desta dissertação, houve um confrontamento com vários desafios que correspondiam a limitações da ferramenta desenvolvida até à fase descrita no capítulo 3 e outras limitações originadas pela escolha das tecnologias a utilizar, nomeadamente as especificações não satisfazíveis por modelos finitos, problema criado pelo uso do AlloyAnalyzer. Neste sentido, foi desenvolvida uma abordagem que é capaz de colmatar esse problemas (secção 4.2) que, como se pode verificar pelos resultados obtidos da experimentação, resolve o problema, pois não só consegue gerar testes para modelos que teoricamente não seriam satisfazíveis, bem como não perde eficiência nos casos que já eram tratados pela abordagem previamente desenvolvida.

Foi também concluída com sucesso a extensão da ferramenta para outros casos mais específicos, como os casos de especificações que continham construtores com restrições de domínio (ver secção 4.6) e construtores que recebem elementos construídos como parâmetro (ver secção 4.3).

A aplicação foi também munida de uma forma automática de procura de limites de pesquisa dos comandos de execução do alloy 4.5, bem como uma eficaz utilização do mapa de refinamento do ConGu, para o mapeamento, entre a especificação e a implementação, na geração de testes.

Assim, pode-se afirmar que esta ferramenta pode ter um impacto importante na área de Testes e Qualidade de Software e Métodos Formais de Engenharia de Software, uma vez que utiliza uma abordagem inovadora para a utilização de especificações algébricas nos testes de software,

utilizando como passo intermédio a ferramenta de geração de modelos através da satisfação de restrições –AlloyAnalyzer. Permitindo assim, efetuar uma simulação do sistema de forma genérica, fazendo com que seja possível a extração de testes para sistemas genéricos, bem como geração de *mock objects* para testar a implementação sem ser necessário fornecer nenhum implementação de tipos de parâmetros desses sistemas. Com o acréscimo das funcionalidades descritas anteriormente, o leque de tipos de sistemas suportados pela ferramenta aumenta significativamente.

6.2 Trabalho Futuro

Apesar do cumprimento de todos os objetivos propostos e dos resultados positivos obtidos, continuam ainda muitos problemas em abertos.

Os testes gerados dependem da correta implementação dos métodos *clone* e *equals*, implementação essa que tem de ficar à responsabilidade do utilizador sem que a ferramenta seja capaz de detetar incorreções.

A decomposição dos axiomas em mintermos gera muitos comandos de execução que acabam por ser inválidos, uma lacuna da abordagem, mas que apenas são considerados inválidos após exaustiva pesquisa dentro de vários limites, o que provoca um aumento considerável do tempo de execução e recursos computacionais utilizados pelo processo, tornando assim necessário eliminar esses casos sem os tentar executar no AlloyAnalyzer.

A mesma decomposição, mas efetuada às restrições de domínio, gera também casos que são inválidos para testes no domínio do problema mas que poderiam ser utilizados para geração de testes fora de domínio.

O número de testes gerados para cada comando executado pelo AlloyAnalyzer é de apenas um, no entanto, esse número podia ser aumentado, sendo necessária, no entanto, a identificação dos casos em que era interessante e os testes gerados não seriam iguais.

Outro problema que fica em aberto também para trabalho futuro, é o problema da escalabilidade, pois a atual abordagem em casos de sistemas muito complexos pode-se tornar bastante pesada e lenta em termos computacionais, devido à forma de funcionamento do AlloyAnalyzer. Sendo que para este caso seria interessante aplicar uma abordagem que detetasse automaticamente que um comando é executável e em caso afirmativo o número de instâncias necessárias, como por exemplo a abordagem referida no artigo [NJDFK], poupando imenso tempo e recursos computacionais na utilização do AlloyAnalyzer.

Por fim, a extração dos *mock objects*, atualmente é bastante limitada, funcionando apenas para casos simples, estando ainda pouco explorada.

Referências

[AFP11a] Francisco R De Andrade, P Faria e Ana C R Paiva. Specification-driven Unit Test Generation for Java Generic Classes. Technical Report i, 2011.

[AFP11b] Francisco Rebello De Andrade, João Pascoal Faria e Ana C R Paiva. TEST GENERATION FROM BOUNDED ALGEBRAIC. 2011.

[AFPL11a] Francisco R De Andrade, P Faria, Ana C R Paiva e Antónia Lopes. Geração de Testes a partir de Especificações Algébricas. 2011.

[AFPL11b] Francisco R De Andrade, P Faria, Ana C R Paiva e Antónia Lopes. Specification-driven Test Generation for Java Generic Classes. pages 1–17, 2011.

[Bel10] Axel Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In Javier Esparza e Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[Cla04] Koen Claessen. QuickCheck: Specification-based Random Testing. 2004.

[CNV07] Alexandre Caldeira, Isabel Nunes e Vasco T Vasconcelos. ConGu Checking Java Classes Against Property-Driven Algebraic Specifications. 2007.

[dEeC12] Ministério da Educação e Ciência. Fundação para a ciência e tecnologia, 2012.

[Ecl] Java code coverage for eclipse, year =.

[FCU12] FCUL. Congu download page, 2012.

[Jac12a] D. Jackson. Alloy analyzer 4.2 download page, 2012.

[Jac12b] D. Jackson. Alloy analyzer's website, 2012.

[Jum] Jumble, year =.

[KJ05] Viktor Kuncak e Daniel Jackson. Relational analysis of algebraic datatypes. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, page 207, 2005.

[KM03] Sarfraz Khurshid e Darko Marinov. TestEra A Novel Framework for Testing Java Programs. *Automated Software Engineering*, (Ase 2001), 2003.

[KYZ+11] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov e Sarfraz Khurshid. TestEra- A Tool for Testing Java Programs Using Alloy Specifications. In *Proc ASE*, pages 608–611, 2011.

[Mic12] Microsoft. Spec explorer 2010, 2012.

[NJDFK] Timothy Nelson, Daniel J. Dougherty, Kathi Fisler e Shriram Krishnamurthi. Toward a More Complete Alloy.

[NV09] Isabel Nunes e Vasco T Vasconcelos. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. *Order A Journal On The Theory Of Ordered Sets And Its Applications*, pages 115–131, 2009.

[paCeT12] Fundação para a Ciência e Tecnologia. A quest for reliability in generic software components, 2012.

[STW12] Dutch Technology Foundation STW. Côte de resyste, 2012.