Monitorização da Correcção de Classes Genéricas *

Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos

Faculdade de Ciências, Universidade de Lisboa,
Campo Grande, 1749–016 Lisboa, Portugal,
{pedro.crispim,mal,vv}@di.fc.ul.pt

Resumo Dado que os genéricos se tornaram muito populares nas linguagens de programação OO, o facto de um método formal não suportar genéricos limita extraordinariamente a sua utilidade e eficácia. De forma a ultrapassar este problema no CONGU, uma abordagem à monitorização da correcção de programas Java face a especificações algébricas, propusemos recentemente uma noção de *mapa de refinamento* que permite definir uma correspondência entre especificações paramétricas e classes genéricas. Baseados nestes mapas, definimos uma noção de correcção de programas face a especificações. Neste artigo, propomos uma forma de monitorizar, em tempo de execução, esta noção de correcção e apresentamos a solução de desenho que suporta este processo na versão 2 da ferramenta CONGU.

1 Introdução

A especificação formal é uma actividade importante no processo de desenvolvimento de software já que, por um lado, auxilia a compreensão e promove a reutilização e, por outro, possibilita a utilização de ferramentas que analisam automaticamente a correcção das implementações face ao especificado. Uma das formas de proceder à analise automática da fiabilidade de componentes de software é através da verificação em tempo de execução (*runtime verification*). Esta abordagem envolve a monitorização e análise das execuções do sistema; à medida que o sistema executa é testada a correcção do comportamento dos componentes relativamente ao que foi especificado.

Apesar dos genéricos se terem tornado muito populares em linguagens como o Java e o C#, as técnicas e ferramentas disponíveis para verificar a correcção de implementações face a especificações não são utilizáveis quando há classes genéricas envolvidas. Este era também o caso do CONGU, uma abordagem à monitorização da correcção de programas Java face a especificações algébricas que temos vindo a desenvolver [7,13]. A ferramenta CONGU que apoia esta abordagem é, há vários anos, intensivamente usada pelos nossos alunos na disciplina de Algoritmos e Estruturas de Dados. Os alunos recebem especificações formais dos tipos de dados que têm de implementar e recorrem à ferramenta para ganhar confiança relativamente à correcção das classes que produzem. Uma vez que os genéricos são extremamente úteis na implementação de tipos de dados em Java, a partir do momento em que passámos a fazer uso dos genéricos na disciplina, o facto do CONGU não suportar genéricos tornou-se um problema. Decididos a ultrapassá-lo, começámos por desenvolver uma noção de *mapa de refinamento*

* Este trabalho foi financiado parcialmente pela FCT através do projecto QUEST, PTDC/EIA-EIA/103103/2008.

que permite definir uma correspondência entre especificações paramétricas e classes genéricas [12]. Baseados nestes mapas, definimos uma noção de correcção de implementações face a especificações mais abrangente. Este trabalho preparou o terreno para a extensão da abordagem CONGU que se apresenta neste artigo. Discute-se ainda a forma como foi concretizada esta extensão na nova versão da ferramenta.

A solução para o problema da monitorização de programas Java que foi desenvolvida de forma a acomodar os genéricos, e que é descrita neste artigo, é substancialmente diferente da usada anteriormente no CONGU [13]. Anteriormente, a estratégia passava por trocar as classes originais por classes *proxy* que usavam classes geradas automaticamente, anotadas com contractos monitorizáveis, escritos em JML [10]. Os principais aspectos inovadores da solução adoptada no CONGU2 são: (i) a introdução de mecanismos que permitem lidar com as classes genéricas e os seus parâmetros e verificar que estão correctas face ao especificado; (ii) o facto de se ter abandonado a geração de classes *proxy* (que traziam problemas quando as classes originais faziam uso de certas primitivas da linguagem Java, como classes internas ou anotações) e se ter passado a utilizar a instrumentação dos binários Java; (iii) o facto de se ter abandonado o JML (o qual não suporta genéricos) e se ter passado a gerar código que, recorrendo a asserções Java, trata directamente da monitorização das propriedades especificadas.

O artigo está estruturado da seguinte forma. A secção 2 fornece uma visão abrangente do CONGU. A secção 3 apresenta a noção de correcção de programas e discute a forma como as propriedades de objectos induzidas pelas especificações podem ser monitorizadas. A solução que foi concretizada no CONGU2 é apresentada na secção 4. O artigo termina apresentando as conclusões na secção 5.

2 Visão geral da abordagem CONGU

O CONGU suporta a monitorização da correcção de programas Java face a especificações algébricas. Nesta secção, recorrendo a um exemplo simples, fornece-se uma visão geral da abordagem, focada essencialmente em aspectos que são visíveis aos seus utilizadores: as especificações, os módulos e os mapas de refinamento.

O exemplo escolhido foram as *listas com fusão*, um tipo de listas (i) cujos elementos são "fundíveis" e (ii) que têm uma operação — mergeInRange — que funde os elementos da lista num determinado intervalo de posições $i...j$. A figura 1 mostra os três elementos envolvidos na especificação deste tipo de dados: ListWM, Mergeable e LWM. ListWM é um exemplo de uma especificação paramétrica, enquanto que Mergeable é um exemplo de uma especificação simples (usada como parâmetro na primeira).

Cada especificação introduz um único género. No exemplo, Mergeable define o género simples com o mesmo nome enquanto que ListWM introduz um género composto ListMW[Mergeable]. O género **int** é primitivo na linguagem.

Cada especificação declara um conjunto de operações e predicados, classificando-os como constructors, observers ou others. As operações classificadas como constructors são aquelas com as quais se podem construir todos os valores do género. As restantes operações e predicados permitem obter informação adicional acerca de valores do género ou oferecem formas alternativas de os construir e, portanto, recebem como argumento um valor do género (por convenção, no primeiro). A classificação das operações

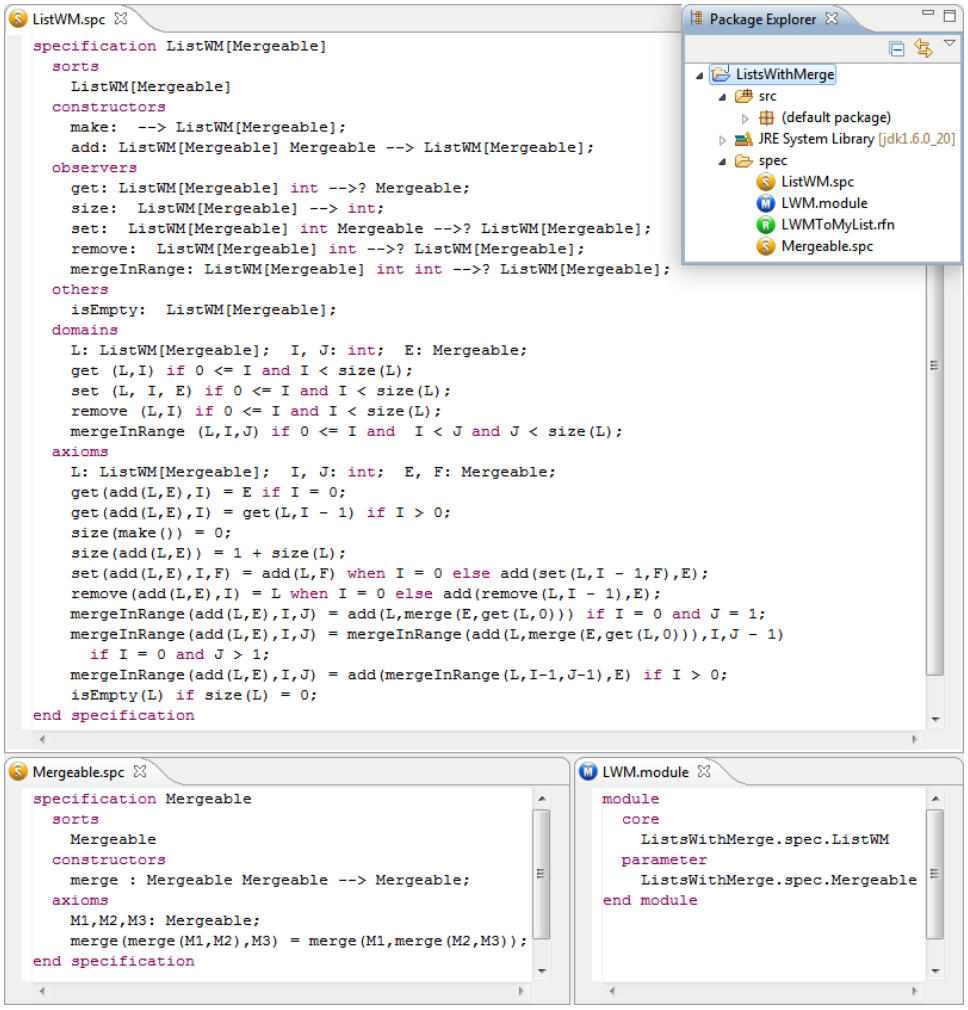

Figura1. Os três elementos envolvidos na especificação do tipo de dados *listas com fusão*.

como observers ou others apenas se reflecte na estrutura sintáctica dos axiomas que podem ser usados para definir as suas propriedades. As propriedades dos observers têm de ser expressas usando construtores no primeiro argumento e variáveis nos restantes, enquanto que no caso dos others podem ser usados variáveis em todos os argumentos.

Porque as operações podem ser parciais, cada especificação define as situações em que operação parcial tem de estar definida — a chamada *condição de domínio*. Por exemplo, em ListWM, a operação get é declarada como sendo parcial e é definido que get(L, I) tem de estar definida se I é um índice da lista L.

A ligação entre diferentes especificações é realizada através de um elemento adicional — os *módulos* (LWM, no nosso exemplo). Nestes módulos, as especificações identificadas como *core* definem os tipos de dados que têm de ser implementados en-

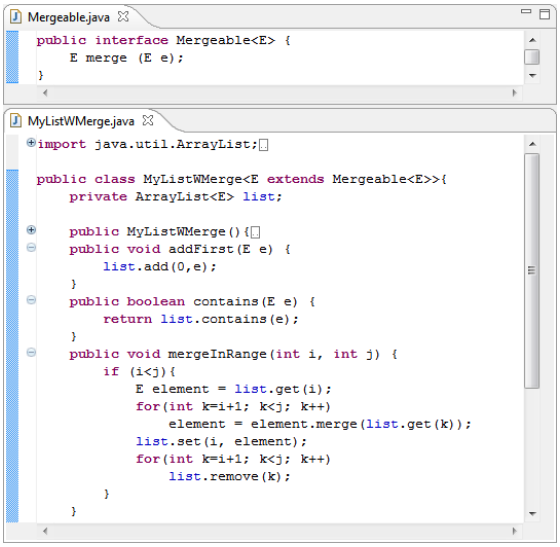

Figura2. The interface `Mergeable<E>` e an excerpt of the Java class `MyListWMerge<E>`.

quanto que as *parameter* não carecem de implementação; servem apenas para impor limitações à instanciação de certos tipos.

Suponha-se que temos uma implementação candidata para o módulo LWM e que gostaríamos de verificar a sua correcção. Primeiro, é necessário estabelecer uma correspondência entre o género definido por cada especificação *core* S de LWM e um tipo Java T definido por uma das nossas classes. Mais ainda, precisamos de estabelecer uma correspondência entre as operações e predicados de S e os métodos e construtores de T. No CONGU, esta correspondência é definida através de um *mapa de refinamento*. Este permite ainda ligar as especificações parâmetro com as variáveis de tipo das classes genéricas: cada género definido por uma especificação parâmetro S é ligada a uma variável de tipo E e cada operação/predicado de S é ligada a uma assinatura de método.

Suponha-se que a implementação candidata para o módulo LWM consiste na classe `MyListWMerge` e no interface `Mergeable` apresentados na figura 2. Na figura 3 é estabelecida uma correspondência entre LWM e esta implementação. Esta faz corresponder o género ListWM[Mergeable] ao tipo definido por `MyListWMerge` e as operações e predicados do primeiro aos métodos e construtores do segundo. Por exemplo, o make é enviado para o construtor `MyListWMerge()` e o add é enviado para o método `addFirst`.

O primeiro argumento de uma operação corresponde sempre ao objecto **`this`** e, portanto, uma operação é enviada sempre para um método com menos um argumento. Apenas as operações declaradas como constructors podem ser enviadas em construtores Java. Os predicados são sempre enviados para métodos cujo tipo de retorno é **`boolean`**. As operações que produzem elementos do género definido pela especificação são enviadas para métodos que tanto podem ser **`void`** como retornar um elemento do tipo correspondente. Desta forma é possível considerar tanto implementações com objectos mutáveis como imutáveis. No nosso exemplo, a classe `MyListWMerge` for-

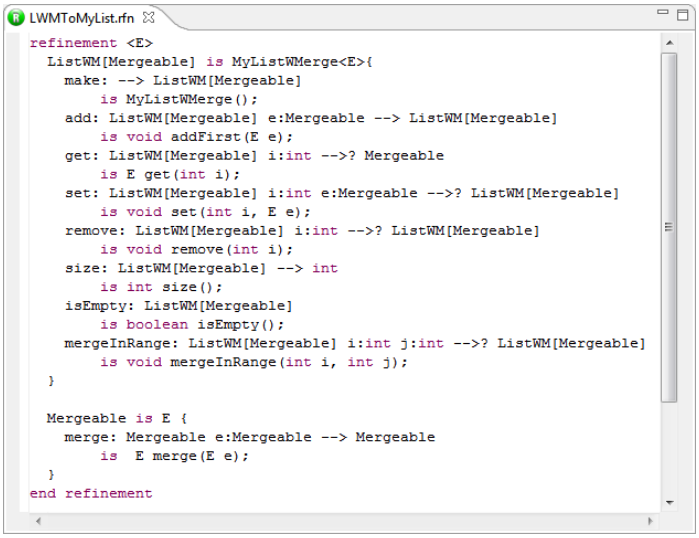

Figura3. Um mapa de refinamento de LWM para {`MyListWMerge<E>`, `Mergeable<E>`}.

nece uma implementação de listas mutáveis e, portanto, as operações que produzem valores do género ListWMerge[Mergeable] são todas enviadas para métodos **`void`**.

O refinamento apresentado na figura 3 também estabelece a correspondência entre o género Mergeable e a variável de tipo `E` e define que a operação merge corresponde a um método com assinatura `E merge(E e)`.

Uma vez definido o refinamento, o CONGU instrumenta `MyListWMerge.class` de forma a que, durante a execução de qualquer programa que use a classe `MyListWMerge`, a correcção do comportamento da classe seja verificada. Adicionalmente, o comportamento das classes que no programa instanciam `MyListWMerge<E>` é também verificado face ao especificado em Mergeable. Por exemplo, se o nosso programa inclui a classe `Color` que implementa `Mergeable<Color>` e, uma outra classe que cria e manipula objectos do tipo `MyListWMerge<Color>`, o comportamento de `Color` é verificado face ao especificado em Mergeable.

3 Correcção de Programas face a Especificações

Nesta secção apresentamos a noção de correcção de programas que é considerada no CONGU2 e discutimos os aspectos mais importantes da abordagem CONGU à monitorização desta correcção.

3.1 Propriedades de objectos induzidas por especificações

A correcção de um programa Java face a um módulo é definida tendo por base um mapa de refinamento estabelecendo uma ligação entre os dois lados. Na secção anterior foram mencionadas algumas das condições que a ligação de operações e predicados a métodos e construtores Java tem de obedecer. A ligação entre especificações e classes Java está sujeita a outras condições, capturadas na definição de mapa de refinamento.

Um *mapa de refinamento* consiste num conjunto V (variáveis de tipo) equipadas com uma pre-ordem $<$ e uma função $\mathcal{R}$ que envia: (1) cada especificação simples e *core* numa classe não genérica; (2) cada especificação paramétrica e *core* numa classe genérica com a mesma aridade; (3) cada especificação *core* que define um género $s < s'$, para uma subclasse de $\mathcal{R}(S')$, onde S' é a especificação que define s'; (4) cada especificação parâmetro numa variável de tipo V; (5) cada operação de uma especificação *core* num método da correspondente classe; (6) cada operação de uma especificação parâmetro numa assinatura de um método. Adicionalmente, (7) se uma especificação parâmetro S' define um sub-género do género definido na especificação S, então $\mathcal{R}(S') < \mathcal{R}(S)$; (8) se S é uma especificação paramétrica com parâmetro S', então tem de ser possível assegurar que qualquer tipo C que possa ser usado para instanciar $\mathcal{R}(S)$ tem métodos com as assinaturas definidas por $\mathcal{R}$ para a variável de tipo $\mathcal{R}(S')$ depois de trocar todas as ocorrências de $\mathcal{R}(S')$ por C.

Note-se que, no nosso exemplo, a condição (8) verifica-se porque `MyListWMerge` impõe que `E extends Mergeable<E>`. Uma vez que o interface `Mergeable` declara o método `E merge(E e)`, `E` só pode ser instanciado com classes `C` que implementem `Mergeable<C>` e, portanto, está assegurado que `C` tem o método `C merge(C e)`.

Seja $\mathcal{R}$ um mapa de refinamento entre um módulo $\mathcal{M}$ e um programa Java $\mathcal{J}$. Para $\mathcal{J}$ estar correcto face a $\mathcal{M}$, (i) as propriedades especificadas em $\mathcal{M}$ e (ii) as propriedades algébricas da noção de igualdade têm de ser verdadeiras em todas as possíveis execuções de $\mathcal{J}$.

As propriedades de uma especificação *core* S restringem o comportamento dos objectos do tipo $T_S = \mathcal{R}(S)$, enquanto que as propriedades de uma especificação S que é parâmetro, por exemplo $S'[S]$, restringem o comportamento dos objectos dos tipos T_S em $\mathcal{J}$ que sejam usadas para instanciar a variável de tipo $\mathcal{R}(S')$. As restrições impostas por axiomas e condições de domínio são diferentes:

Axiomas. Cada axioma de uma especificaçãoS define uma propriedadespara os objectos do tipo T_S que tem de ser verdadeira em todos os estados do objecto que sejam visíveis para o cliente.
Considere-se, por exemplo, o primeiro axioma do get em ListWM[Mergeable]. Se `lwm` é um objecto do tipo `MyListWMerge<C>`, está sujeito à seguinte propriedade: para todo o `e` de tipo `C`, se `e!=null`, então após a execução de `lwm.addFirst(e)`, `lwm.get(0).equals(e)` é verdadeira.

Domínios. Cada condição de domínio ϕ de uma operação op de uma especificação S define que, para todo o objecto do tipo T_S, sempre que ϕ é verdadeiro, a invocação de $\mathcal{R}(op)$ tem de retornar normalmente, sem levantar qualquer excepção.

As propriedade de objectos induzidas pelos axiomas e domínios que foram apresentadas definem uma noção de correcção. O CONGU usa, por omissão, uma noção mais forte que também impõe restrições aos clientes da classe T_S, quando invocam o método $\mathcal{R}(op)$. É exigido a estas classes que não passem o valor `null` como argumento a $\mathcal{R}(op)$ e que só invoquem o método quando a sua condição de domínio é verdadeira.

3.2 Monitorização das propriedades dos objectos

A estratégia usada no CONGU para monitorizar a correcção de um programa consiste em verificar os invariantes induzidos pelos axiomas no final de métodos específicos, de-

terminados pela estrutura dos axiomas. No caso dos axiomas em que a operação/predicado tem como primeiro argumento uma operação construtora, o invariante é verificado no final do método que refina a referida operação construtora. Por exemplo, o invariante induzido pelo primeiro axioma de get é verificado no final de **void** `addFirst(E e)` através da execução do seguinte código, onde `eOld` é uma cópia de `e` obtida à entrada do método.

```
if (eOld != null) {
  E e2 = this.clone().get(0);
  assert(e2 != null && e2.equals(eOld));
}
```

Note-se que todas as invocações que são realizadas de forma a verificar uma propriedade são realizadas sobre clones, se possível. De outra forma, os efeitos colaterais destes métodos afectariam os objectos monitorizados. Se o clone não é suportado, assume-se que os objectos da classe são imutáveis. De forma semelhante, o segundo axioma de get é verificado pelo seguinte código:

```
if (eOld ! = null)
  for (int i: rangeOfInt)
    if (i>0 && i<this.clone().size()) {
      E e2 = this.clone().get(i);
      assert((i-1)>=0 && (i-1)<thisOld.clone().size());
      E e3 = thisOld.clone().get(i-1);
      assert(e2!=null && e3!=null && e2.equals(e3));
    }
```

onde `rangeOfInt`, de tipo `Collection<Integer>`, é preenchido com os inteiros que são usados como argumento ou valores de retorno de algum dos métodos da classe. Apesar de neste caso, por se tratar de um tipo de dados primitivo, ser fácil arranjar outras formas de obter os valores do domínio alvo da quantificação universal, isto não acontece em geral. A estratégia adoptada, que passa por coleccionar os elementos do domínio que atravessam a fronteira da classe, é uma forma relativamente simples e leve de ter uma população representativa do domínio em causa.

Por outro lado, as propriedades induzidas por axiomas que têm uma variável como primeiro argumento da operação são verificadas no final do método que refina essa operação. Por exemplo, o último axioma de ListWM, que descreve uma propriedade de isEmpty, é verificado no final do método **boolean** `isEmpty()` pelo seguinte código, onde `result` é o valor de retorno de **boolean** `isEmpty()`.

```
if (result) assert(thisOld.clone().size()==0);
```

A igualdade entre inteiros é convertida em comparações com == enquanto que a igualdade de um género não primitivo, s, é convertida na invocação do método `equals` da classe T_S. É assim essencial que todas as classes envolvidas definam uma implementação apropriada deste método que, em particular, deve considerar dois objectos iguais apenas se têm comportamentos equivalentes quando se consideram os métodos que refinam alguma das operações de s (i.e., são *equivalentes do ponto de vista do seu comportamento*).

A correcção do `equals` é monitorizada no final deste método. Por exemplo, a monitorização de **boolean** `equals(Object other)` em `MyListWMerge` envolve:

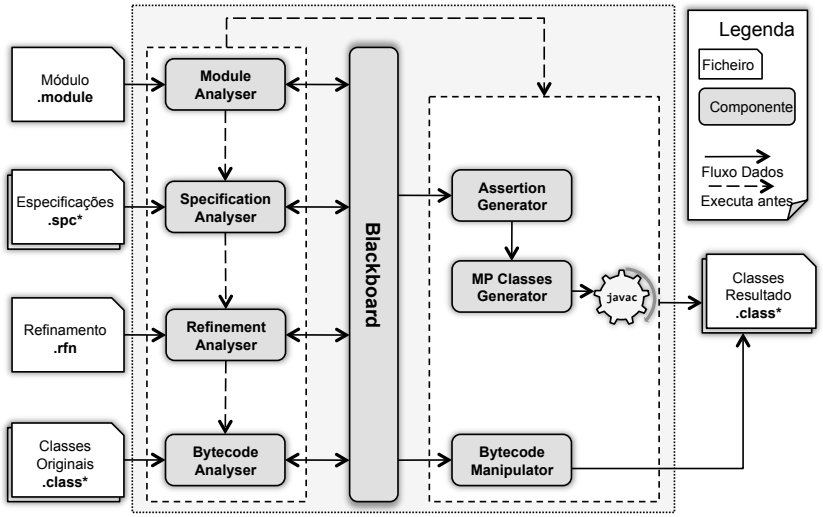

Figura4. A arquitectura do CONGU2.

```
if (result)
  for (int i: rangeOfInt)
    if (i>=0 && i<thisOld.clone().size()
             && i<otherOld.clone().size()) {
      E e1 = thisOld.clone().get(i);
      E e2 = otherOld.clone().get(i);
      assert(e1!=null && e2!=null && e1.equals(e2));
    }
```

Finalmente, a verificação de que as classes clientes não invocam um método que refina uma operação quando a condição de domínio é falsa, nem passam o valor **null** como argumento, pode ser facilmente realizada no início do método. Por exemplo, no caso do método **void** `set(`**int** `i,E e)`, isto é verificado por:

```
assert(e != null && i >= 0 && i < this.clone().size());
```

4 CONGU2

A versão 2 da ferramenta CONGU implementa a monitorização da correcção de programas Java descrita na secção anterior. Como mostrado na Figura 4, a ferramenta recebe um módulo, especificações, um mapa de refinamento e os binários de um programa Java. O programa é então transformado de forma a que, quando é executado com o CONGU, seja monitorizada a sua correcção. Isto é alcançado através da intercepção de todas as chamadas a métodos que refinam alguma operação/predicado, e redireccionado-as para classes monitoras de propriedades.

A ferramenta executa duas tarefas principais: a análise das diferentes fontes de *input* e a geração das classes de *output*. Na tarefa de análise, os maiores desafios colocados pela extensão da abordagem a especificações paramétricas e classes genéricas surge na análise dos refinamentos. Em 4.1 discute-se como foram endereçados estes desafios.

Apresentam-se depois os aspectos chaves da tarefa de geração: em 4.2, a instrumentação dos binários e, em 4.3, a geração das classes monitoras das propriedades.

4.1 Análise do mapa de refinamentos

A extensão do CONGU levanta vários desafios ao nível da análise dos refinamentos já que a verificação de várias condições a que os refinamentos têm de obedecer exige investigar os binários das classes referidas. Isto é alcançado recorrendo às capacidades de reflexão oferecidas pelo pacote `java.lang.reflection` do *API* do Java.

O processo de análise dos refinamentos tem duas fases: a primeira focada nas classes que refinam os géneros das especificações *core* e a segunda na verificação de condições relacionados com os géneros das especificações parâmetro.

A verificação de que um tipo não genérico tem os métodos mencionados no refinamento é bastante simples, já que basta obter o método especificado e depois verificar que o seu tipo de retorno é o esperado. A situação complica-se no caso de tipos genéricos por causa do mecanismo conhecido como *type erasure*, o qual torna um tipo genérico num *raw type* (substituindo todas as ocorrências das variáveis de tipo pelos seus limites superiores [8]). Por esta razão, é preciso uma estratégia mais sofisticada para verificar que uma classe genérica tem os métodos mencionados num refinamento: primeiro é preciso obter todos os métodos da classe e, para cada um destes, obter os seus tipos de parâmetros e o seu tipo de retorno genéricos e, depois, é preciso verificar se algum destes métodos é o esperado (um processo que tem de ser recursivo na estrutura dos tipos).

Na segunda fase da análise dos refinamentos são endereçadas as condições relacionadas com os géneros das especificações parâmetro. Recorde-se que, neste caso, os géneros não são refinados em tipos concretos e o que é preciso é assegurar que as classes que podem instanciar a correspondente variável de tipo têm os métodos necessários. Por exemplo, no nosso caso, nesta fase é verificado que toda a classe `C` que pode instanciar `E` em `MyListWMerge<E>` tem um método com a assinatura `C merge(C e)`. Isto é conseguido recorrendo aos limites superiores de `E` em `MyListWMerge` (no nosso caso há só um, mas em geral podem ser vários). Os métodos cuja assinatura envolve o `E` só precisam de ser pesquisados nos limites que também dependem de `E` (no nosso caso, o método `E merge(E e)` é procurado em `Mergeable<E>`).

4.2 Instrumentação dos binários

A monitorização dos programas Java assenta na intercepção das chamadas aos métodos relevantes por classes clientes. No CONGU2, esta intercepção é realizada através da instrumentação dos binários. As chamadas interceptadas são traduzidas em chamadas de um método correspondente numa classe monitora de propriedades, gerada pela ferramenta. Os principais desafios que este processo coloca são os seguintes:

Chamadas Internas. Como interceptar apenas as chamadas externas? As chamadas internas não podem ser monitorizadas, caso contrário o programa não termina.

Chamadas de super-classes. As chamadas de dentro de super-classes também não podem ser interceptadas.

Construtores. Como interceptar e redireccionar as chamadas aos construtores?

Clone e equals. O que fazer quando estes métodos não são redefinidos na classe?

A estratégia adoptada foi renomear todo o método *m* que precisa de ser interceptado (passa a chamar-se *m_Original*) e substituí-lo por um método com o mesmo interface, que redirecciona a chamada para `ClassPMonitoring.m(this, ...)`. Por outro lado, as chamadas internas são trocadas por chamadas ao método na sua forma renomeada. De forma semelhante, as chamadas a este método nas super-classes são também trocadas por chamadas ao método renomeado, o qual também é acrescentado nestas classes. Relativamente aos construtores, a sua intercepção é realizada de uma forma idêntica à dos métodos, apenas com a salvaguarda de que os construtores exigem chamadas de inicialização, as quais são removidas do método renomeado e inseridas no método que o substitui. O mesmo é feito para o `equals` e `clone`. Se a classe não redefine o `equals`, então é primeiro criado este método, o qual delega a chamada na super-classe. Se a classe não anuncia implementar `Cloneable` ou não redefine o método como público, então é gerado um método `clone_Original` que simplesmente retorna **`this`** (recorde-se que neste caso se assume que os objectos da classe são imutáveis). O método gerado é para exclusivo uso do processo de monitorização.

A implementação desta estratégia para instrumentação dos binários recorre à biblioteca de manipulação de bytecode ASM [4]. O ASM é uma biblioteca leve e eficiente, que, disponibilizando um API simples e bem documentado, suporta completamente o Java 6 e é distribuído sob uma licença *open-source* que possibilita a conveniente inclusão no pacote da ferramenta CONGU2.

4.3 Geração das Classes Monitoras de Propriedades

A monitorização das propriedades de objectos descrita na sub-secção 3.2 é executada por classes geradas pela ferramenta, a que chamámos *classes-MP*. Para cada classe C cujo comportamento precisa de ser monitorizado é gerada uma classe-MP, cujo nome resulta de juntar o sufixo `PMonitoring` ao nome de C.

Cada método que é interceptado tem um equivalente na respectiva classe-MP, na forma de um método de classe com o mesmo nome e tipo de retorno e cujos argumentos são os do método original mais um argumento *callee* com o tipo C (uma referência do objecto alvo da invocação original) e outro *flag* com o tipo Booleano (indicando se deve ser realizada a monitorização de propriedades).

A estrutura do corpo destes métodos segue o seguinte padrão: (1) guardar à entrada do método os valores dos vários argumentos; (2) verificar que a condição de domínio é verdadeira (apenas no caso da noção de correcção forte) e os argumentos não são **`null`**; (3) chamar o método original sobre o *callee* e guardar o valor de retorno, (4) verificar as propriedades requeridas e (5) retornar o valor de retorno original. Os passos (2) e (4), que são apenas executados se a *flag* é verdadeira, usam dois métodos de classe auxiliares, respectivamente, *mPre* e *mPos*. Estes testam as propriedades do objecto $callee$ de acordo com o descrito em 3.2 mas, em vez de chamarem os métodos originais, chamam os equivalentes na respectiva MP-classe, com a *flag* a falso (ver figura 5).

No passo (3), além de ser invocado o método original, é verificado que se a condição de domínio é verdadeira, então o método retorna normalmente. A chamada ao método original é envolvida por um *try-catch*, de forma a poder assinalar uma violação

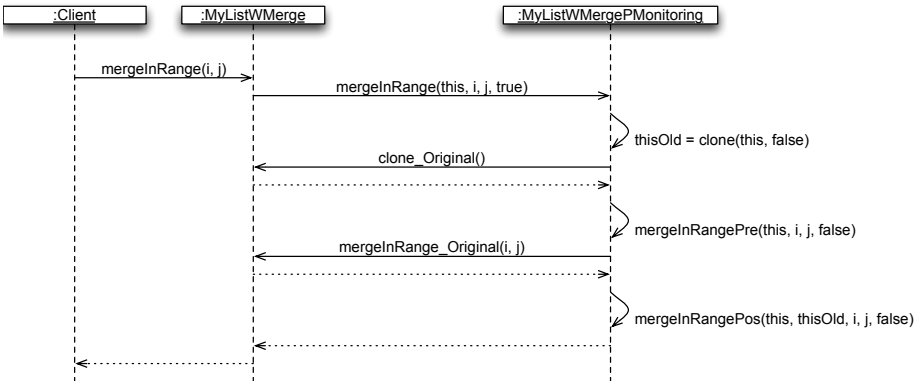

Figura5. Processo desencadeado por uma chamada externa a `mergeInRange(int,int)`.

da correcção do programa sempre que a condição do domínio é verdadeira e o método original levantar uma excepção. Se, pelo contrário, a condição de domínio for falsa, qualquer excepção apanhada é relançada.

Tudo o que foi até aqui descrito aplica-se a classes que refinam géneros *core* ou são usadas para instanciar uma classe genérica que refina um género *core*. No entanto, a monitorização das propriedades das especificações parâmetro exige um nível adicional de indirecção. Seja C uma classe genérica com parâmetro E que refina uma especificação $S[S']$. Apesar de uma classe usada para instanciar E em C ter uma correspondente classe-MP, o código gerado para monitorizar as propriedades de S que envolve invocar um método de E, não se pode comprometer com uma classe-MP específica. A solução adoptada foi gerar uma classe *dispatcher* para cada variável de tipo do mapa de refinamento. Esta classe, com os mesmos métodos que uma classe-MP, apenas serve para resolver a que classe-MP deve ser entregue cada chamada de um método, baseando-se para isso no tipo concreto do objecto *callee*. Na classe-MP de C, sempre que a verificação de uma propriedade exigir invocar um método de E, a chamada é feita sobre o correspodente *dispatcher*.

5 Conclusões

A importância de ferramentas que suportam a verificação da correcção de implementações face a especificações formais tem sido largamente reconhecida. Na última década, várias abordagens foram desenvolvidas que permitem monitorizar a fiabilidade de implementações em linguagens OO (ex., [1,3,5,6,9,11]). No entanto, e apesar da popularidade dos genéricos nestas linguagens, as abordagens existentes ainda não os suportam. Esta era também uma limitação do CONGU que foi superado com o CONGU2.

Neste artigo mostramos como a abordagem e a ferramenta CONGU foram estendidas de forma a apoiar a especificação de tipos de dados genéricos e sua implementação em termos de classes genéricas. A extensão da linguagem de especificação de forma a permitir a descrição de tipos de dados genéricos foi relativamente simples. Dado que o CONGU depende especificações baseadas em propriedades, essencialmente foram adoptadas especificações paramétricas semelhantes às disponíveis em várias linguagens

de especificação algébricas. A fim de colmatar o fosso entre especificações paramétricas e classes genéricas propusemos uma nova noção de mapa de refinamento em torno da qual foi definida uma nova noção de correcção de programas face a especificações. Tanto quanto sabemos, este aspecto não foi endereçado noutros contextos. Existem outras abordagens que lidam com especificações arquitectónicas envolvendo especificações paramétricas, como por exemplo [2], mas têm como alvo programas ML. Por outro lado, as relações entre especificações algébricas e programas OO de que temos conhecimento consideram exclusivamente especificações simples e sem estrutura.

O CONGU2 assegura a monitorização desta noção de correcção mais abrangente, que, no caso dos tipos de dados genéricos, envolve verificar que tanto a classe que implementa o tipo de dados como as classes usadas para instanciá-lo estão em conformidade com o que foi especificado. Com o CONGU2, a verificação da correcção de um programa em tempo de execução passa a ser aplicável a um conjunto de situações em que o suporte automático para a detecção de erros é ainda relevante: os genéricos são reconhecidamente complicados de dominar e, portanto, a correcção de implementações com genéricos mais difícil de atingir.

Referências

1. S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.
2. D. Aspinall and D. Sannella. From specifications to code in CASL. In *Proc. Algebraic Methodology and Software Technology (AMAST) 2002*, volume 2422 of *LNCS*, pages 1–14. Springer, 2002.
3. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
4. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proc. ACM SIGOPS France Journées Composants 2002: Systemes a composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.
5. F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2005*, volume 3440 of *LNCS*, pages 546–550, 2005.
6. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proc. International Conference on Software Engineering Research and Practice (SERP 2002)*, pages 322–328, 2002.
7. Contract Based System Development. http://gloss.di.fc.ul.pt/congu/.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 06 2005.
9. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP 2003*, volume 2743 of *LNCS*, pages 431–456, 2003.
10. G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–208, 2005.
11. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997.
12. I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Runtime Verification*, pages 115–131, 2009.
13. I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L.S. Reis. Checking the conformance of Java classes against algebraic specifications. In *ICFEM'06*, volume 4260 of *LNCS*, pages 494–513. Springer, 2006.