Pedro Loura Barros

Linguagem Concurrent Contract-Java

Pedro Loura Barros

Linguagem Concurrent Contract-Java

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor Miguel Augusto Mendes Oliveira e Silva do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

António Rui de Oliveira e Silva Borges
Professor Doutor Associado da Universidade de Aveiro

vogais / examiners committee

Miguel Augusto Mendes Oliveira e Silva
Professor Doutor Auxiliar da Universidade de Aveiro (orientador)

Jorge Miguel Matos Sousa Pinto
Professor Doutor Associado do Departamento de Informática da Escola de Engenharia da Universidade do Minho

agradecimentos / acknowledgements

Ao Professor Doutor Miguel Oliveira e Silva, por ter contribuído para a experiência deste trabalho em muito mais do que com orientação técnica.

Resumo

Esta tese estuda a integração de mecanismos de programação concorrente, linguagens orientadas por objectos e programação por contrato.

Propomos assim uma nova linguagem, Concurrent Contract-Java (CCJava), que é uma extensão de Contract-Java, uma linguagem que estende Java com suporte para mecanismos de programação por contrato.

CCJava pretende facilitar, tanto quanto possível, o desenvolvimento de programas concorrentes ao tornar-se estaticamente segura (previne a ocorrência de *race conditions*) e abstracta (os mecanismos de concorrência de baixo nível são dispensados para certas concretizações, como, por exemplo, a escolha dos esquemas de sincronização de objectos partilhados). A linguagem reutiliza ao máximo o significado dos mecanismos Java, dos objectos e da programação por contrato de modo a disponibilizar, da forma mais simples e expressiva quanto possível, mecanismos de concorrência. Todos os aspectos de sincronização de um objecto concorrente e de criação de *threads* são garantidos pela nossa linguagem, recorrendo apenas a duas novas palavras reservadas: `shared` e `remote`.

Como prova de conceito das nossas propostas desenvolvemos um compilador que implementa os aspectos mais importantes da linguagem CCJava.

Abstract

The goal of this thesis is the study on the integration of mechanisms from concurrent programming, objected-oriented languages, and Design-by-Contract.

We propose a new language, Concurrent Contract-Java (CCJava), that's an extension of Contract-Java, a language that has extended Java with Design-by-Contract language mechanisms.

CCJava aims to ease object-oriented concurrent programming, ensuring safety (no race conditions will occur) and abstracting away lower level concurrent realizations such as the choice for shared object synchronization schemes. It reuses the semantics of Java mechanisms, together with the semantics of objects and Design-by-Contract constructs, providing simple and expressive language mechanisms for concurrency. CCJava guarantees all aspects of a shared object's concurrent utilization and thread creation, using only two new language keywords: `shared`, and `remote`.

A compiler was developed implementing the more important language mechanisms, and used as a proof of concept of our proposals.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Blocos de Códigos

Capítulo 1

Introdução

O nosso trabalho estuda a programação concorrente em linguagens orientadas por objectos com o suporte a técnicas de programação por contrato, culminando com a especificação de uma nova linguagem que facilita o desenvolvimento de programas concorrentes.

No seio da ciência da computação, a programação concorrente tem sido um tema amplamente estudado. No entanto, aquando da construção de programas, os programadores tinham a tendência para escolher mecanismos sequenciais em detrimento de mecanismos concorrentes. Tal cenário pode ser explicado, por um lado, pelo facto de até recentemente a capacidade e velocidade dos computadores crescer de forma exponencial em cada ano, confirmando-se, assim, a previsão enunciada na Lei de Moore [1], e, por outro lado, pelo facto do não-determinismo associado à concorrência dificultar o desenvolvimento e a correcção dos programas.

Não obstante, as Unidades Centrais de Processamento evoluem progressiva e recentemente no sentido das arquitecturas concorrentes, o que desde logo contribui para a proliferação da programação concorrente.

A construção de programas, sequenciais ou concorrentes, é uma tarefa difícil. Assegurar que a complexidade dos programas não compromete a correcção dos mesmos pode ser conseguido através da utilização de metodologias adequadas. No contexto do nosso estudo, essas metodologias são a Programação Orientada por Objectos e a Programação por Contrato.

A Programação Orientada por Objectos distingue-se de outros paradigmas de programação, essencialmente devido à possibilidade de minimizar a complexidade de um programa através das suas propriedades de modularidade, reutilização e extensibilidade.

A Programação por Contrato é igualmente uma metodologia com vantagens reconhecidas,

[1] A Lei de Moore enuncia que por cada ano o número de transístores num circuito integrado duplica.

de entre as quais se destacam a maximização da correcção e robustez de um programa, a forte ligação à programação orientada por objectos e o facto de poder ser utilizada como uma ferramenta de especificação e depuração.

A conjugação destes dois mecanismos, quando aplicada num contexto concorrente, pode, porém, originar alguns problemas. Ainda assim, e se todas as propriedades de ambas as metodologias se verificarem, tal conjugação constitui um ponto de partida para o desenvolvimento de soluções que potenciam a construção de programas concorrentes e suavizam a imprevisibilidade que lhes está associada.

Nestes termos, vamos especificar uma nova linguagem, Concurrent Contract-Java (CCJava), que combina mecanismos de programação concorrente, de linguagens orientadas por objectos e de programação por contrato. CCJava é uma extensão de uma outra linguagem, Contract-Java, que já corresponde a um *superset* de Java para suportar mecanismos de programação por contrato[2]. A nossa linguagem disponibiliza mecanismos de alto nível de programação concorrente assentes em contratos e no sistema de tipos de linguagem, que facilitam a construção de programas concorrentes estaticamente seguros e que não obrigam à escolha de esquemas de sincronização de objectos partilhados.

Do trabalho elaborado destacamos as seguintes contribuições:

- Utilização de mecanismos de alto nível simples, expressivos e seguros para integrar concorrência em linguagens orientados por objectos com suporte à programação por contrato;
- Desenvolvimento de um sistema de compilação para a linguagem proposta;
- Sincronização abstracta de objectos partilhados;
- Sincronização automática de objectos partilhados;
- Sincronização segura de objectos partilhados.

[2] Trabalho desenvolvido pelo aluno Pedro Filipe do Amaral Goucha Francisco, como resultado da sua Dissertação de Mestrado em Engenharia de Computadores e Telemática apresentada ao Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, 2012. [1]

1.1 Organização do Documento

O nosso trabalho dividir-se-à da seguinte forma:

No Capitulo 2 apresentaremos diferentes paradigmas da programação sequencial. O objectivo é explicar a interligação entre três paradigmas, começando pela Programação Procedimental e terminando na Programação por Contrato. A par com a programação por contrato, é apresentada a linguagem Contract-Java, que especifica mecanismos para programação por contrato.

No Capítulo 3 analisaremos as características da Programação Concorrente e os desafios por ela colocados, bem como a solução para os mesmos.

No Capítulo 4 será apresentada a nossa linguagem. Explicaremos de que forma CCJava facilita o trabalho do programador na construção de programas concorrentes em linguagens orientadas por objectos com suporte para mecanismos de programação por contrato, recorrendo a exemplos para suportar as explicações apresentadas.

No Capítulo 5 apresentaremos o sistema de compilação da nossa linguagem, dando conta das soluções tomadas para os problemas emergentes.

A conclusão desta tese, onde se enuncia as contribuições da linguagem CCJava e o trabalho futuro a desenvolver, será apresentada no Capítulo 6.

Capítulo 2

Programação Sequencial

A metodologia de desenvolvimento de um programa é tipicamente designada por paradigma da programação. Ao longo dos anos, vários paradigmas foram emergindo e conforme a abordagem programática que os caracteriza podem ser declarativos ou imperativos. No âmbito do nosso trabalho, focar-nos-emos apenas no último grupo, fazendo uma breve referência à programação declarativa, que se distingue da imperativa pelo facto de não contemplar o registo de informação mutável e a noção de estado de um programa.

Assim, o objectivo deste capítulo é sintetizar três paradigmas de programação sequencial: Programação Procedimental, Programação Orientada por Objectos e Programação por Contrato.

2.1 Programação Procedimental

Na programação procedimental, que assenta na estratégia de dividir para conquistar (*divide-and-conquer*), um programa de complexidade considerável deve ser decomposto num conjunto de problemas com menor dimensão e cada um deles com um nível de resolução mais baixo do que o original [2].

Neste tipo de programação, o programador começa por construir um programa no qual identifica os problemas de menor dimensão que precisam de ser tidos em consideração para que o programa principal seja resolvido. Para cada um dos problemas de menor complexidade identificados é aplicada a mesma estratégia, e assim sucessivamente, sempre que se revelar necessário. Ou seja, é feita a decomposição hierárquica do programa principal de acordo com uma abordagem de decomposição de cima para baixo (*top to bottom*).

A construção de um programa de acordo com este paradigma conduz à criação de uma estrutura de dados centralizada e o algoritmo do programa é implementado através de vários subprogramas. Cada um dos subprogramas encapsula uma parte do algoritmo, existindo pois uma abstracção algorítmica que separa a utilização do algoritmo da sua implementação. Fica por isso aberta a possibilidade de reutilizar estes subprogramas noutros programas que não aquele para o qual foram desenvolvidos.

No entanto, quando o nível de complexidade dos problemas é progressivamente mais elevado, a programação procedimental revela-se uma estratégia inadequada. Se forem tomadas más decisões, os efeitos sentidos num nível da hierarquia são propagados para os seguintes, impondo-se ao programador que mais tarde corrija o programa. Ao mesmo tempo, a existência de uma estrutura de dados centralizada levanta problemas de modularidade, uma vez que qualquer alteração na estrutura poderá implicar uma alteração nos subprogramas que dela dependem.

Os problemas apontados a este paradigma são resolvidos no paradigma da programação orientada por objectos.

2.2 Programação Orientada por Objectos

A programação orientada por objectos (OO) é introduzida como outra estratégia para lidar com a complexidade de um problema, ao propor que a análise da sua solução seja feita simultaneamente a partir da perspectiva funcional das operações e da perspectiva composicional das estruturas de dados.

A estratégia agora em estudo tem razão de ser porque, e de uma forma geral, as operações estão bastante relacionadas com estruturas de dados bem definidas - de tal modo que uma alteração nestas estruturas implicará alterações nas operações pelas quais são utilizadas - e porque o comportamento das estruturas de dados é definido pelas operações que as manipulam.

2.2.1 Objecto

Analisar a solução do problema tal como é proposto significa solucioná-lo através de uma decomposição orientada ao objecto, sendo *objecto* a denominação atribuída a uma entidade

que associa estrutura de dados e métodos.

Um objecto é composto por atributos e/ou métodos que concretizam o conjunto de serviços desse objecto. Os atributos, que podem ser variáveis ou constantes, são utilizados para armazenar o estado do objecto. Os métodos representam as operações que podem ser executadas sobre o objecto. Os métodos que podem alterar o estado do objecto são designados por comandos. Já os métodos que retornam propriedades dos objectos são designados consultas de um objecto. Uma consulta diz-se pura se não alterar o estado do objecto.

Regra geral, a maioria das linguagens OO define o comportamento de um objecto em entidades sintácticas denominadas *classes*. Um objecto é uma instância de uma classe.

2.2.2 Encapsulamento de Informação

Num objecto, o encapsulamento de informação pode ser definido como sendo a ocultação de alguns dos seus serviços dos seus clientes externos [3]. Assim, a relação existente entre os objectos e os seus utilizadores externos deverá ter limites bem definidos. São duas as razões que sustentam esta afirmação. A primeira é a necessidade de impedir utilizadores externos de alterar partes internas dos objectos, restringindo-se-lhes o acesso somente à interface que lhes disponibiliza as operações essenciais para resolver o problema. Com efeito, apenas os objectos devem controlar a correcção do seu estado interno, prevenindo usos incorrectos. A segunda prende-se com o facto de todo o conteúdo interno não visível para o exterior poder ser alterado sem afectar os utilizadores, mantendo o comportamento que se espera do objecto.

O encapsulamento de informação é uma característica muito importante das linguagens OO, uma vez que:

- Permite observar um objecto tendo em conta apenas um subconjunto de métodos;
- Os atributos e métodos que não são visíveis para o exterior podem ser livremente modificados sem afectar (directamente) os clientes de um objecto;
- Permite que só o objecto seja responsável pela correcção do seu estado, prevenindo usos incorrectos.
- Permite perspectivar um objecto como uma implementação de um Tipo de Dados Abstracto (conceito que mais à frente abordaremos).

Da análise efectuada sobre esta secção, com enfoque na correcção do estado de um objecto, podemos afirmar que a inexistência de atributos directamente modificáveis pelos clientes é um requisito essencial para garantir que a correcção só dependa da própria classe.

2.2.3 Herança

A herança é uma característica da programação orientada por objectos que permite construir uma classe a partir de outra já existente, reutilizando (e possivelmente redefinindo) métodos ou extendendo a classe com novos atributos e métodos.

A herança permite criar um tipo de dados base para exprimir os aspectos mais comuns do problema, derivando, a partir deste, as restantes classes descendentes. Assim, quando se criam classes derivadas, estas herdam toda a estrutura do seu ascendente, ou seja, quando uma classe A herda de outra classe B, diz-se que A é uma *subclasse* ou *classe descendente* de B.

Esta propriedade permite que qualquer mensagem enviada para a classe base possa também ser enviada para todas as classes derivadas. Uma vez herdada toda a estrutura, o programador tem duas possibilidades: adicionar mais métodos à classe derivada, significando isto que a classe base simplesmente não fazia o que era pretendido, ou alterar o comportamento de um método herdado, sendo esta faculdade vulgarmente conhecida por *overriding*.

2.2.4 Polimorfismo de Subtipo

Polimorfismo significa a capacidade para assumir diferentes formas. Concretamente, polimorfismo de subtipo significa que uma entidade do tipo A pode estar associada a qualquer objecto de outro tipo, por exemplo B, desde que B seja um subtipo de A. Ou seja, em todos os contextos onde é esperado um objecto do tipo A é possível utilizar um objecto do tipo B.

Se o cenário apresentado se verificar, a selecção de uma determinada operação a executar terá de ser feita dinamicamente consoante o tipo de objecto ao qual a entidade está associada, já que o compilador pode não conseguir saber em tempo de compilação qual a operação exacta a executar. Nas linguagens orientadas por objectos, esta escolha é feita pelo objecto, tratando-se de uma situação de encaminhamento dinâmico simples.

Subtipo ou Subclasse?

Uma relação de subclasse permite a reutilização interna de código entre classes. Uma relação de subtipo está dependente da interface pública de um ojecto (especificação) e não da sua implementação.

No entanto, há duas situações que podem ser consideradas equivalentes: uma classe A herdar de uma classe B e uma classe A implementar directamente os serviços de uma classe B. Na primeira situação, estando A a reutilizar os serviços de B, A poderá cumprir os mesmos requisitos impostos pela interface pública da classe B.

Desta forma, uma relação de subclasse tem condições para ser considerada uma relação de subtipo (associação que acontece na linguagem Java).

2.2.5 Tipos de Dados Abstractos

Os Tipos de Dados Abstractos (TDA) constituem o suporte teórico para descrever correctamente um objecto e, juntamente com os quatro secções anteriores, formam o conjunto de características essenciais numa linguagem orientada por objectos [3].

Um TDA pode ser enunciado como uma classe de objectos definidos apenas pelas operações que lhe são aplicáveis e respectiva semântica [4]. Neste sentido, uma classe poderá corresponder a uma implementação parcial ou total de um TDA [4].

Os TDAs são caracterizados pelo encapsulamento de informação, uma vez que disponibilizam uma interface para comunicação com o exterior, mas não evidenciam nenhum detalhe da estrutura de dados interna, nem nenhum detalhe da implementação das operações que sobre ela são efectuadas.

Pelo que fica dito, um programa orientado por objectos será uma colecção estruturada de implementações de tipos abstractos [4].

2.3 Programação por Contrato

A programação por contrato é uma metodologia baseada no estudo da correcção de um programa [5, 6, 7] e os seus mecanismos foram primeiramente implementados na linguagem Eiffel [8].

No desenvolvimento de *software* ignora-se muitas vezes a questão da sua qualidade. O factor de qualidade mais importante é a capacidade do programa para efectuar a tarefa para a qual foi desenhado (correcção) e para lidar com situações anormais (robustez), ou seja, a fiabilidade do *software*. Surge pois a pergunta: como será possível maximizar a fiabilidade de *software* orientado por objectos? A programação por contrato (DbC[1]) é uma resposta a esta questão.

Por isso, nesta secção pretendemos apresentar o conceito sequencial de DbC e os requisitos a assegurar no desenvolvimento de um suporte para os mecanismos que esta metodologia propõe.

2.3.1 Contratos

DbC é uma metodologia que se baseia na ideia simples de utilizar asserções para expressar o significado do código. As asserções são expressões booleanas, frequentemente executáveis, que exprimem uma propriedade do código. Desta forma, são minimizadas as diferenças entre especificação e implementação de um código.

Uma asserção pode ser verdadeira ou falsa. No primeiro caso, a asserção não tem qualquer impacto no programa. Na verdade, se fosse possível assegurar a veracidade, em contexto sequencial, de todas as asserções, então, as suas verificações não seriam necessárias. Já uma asserção falsa é sempre uma manifestação de um erro de um programa.

A designação DbC resulta do facto desta metodologia estabelecer um contrato, especificado por asserções, entre uma entidade de um programa (método, classe, algoritmo) e os seus clientes. A existência de um contrato permite distribuir de forma inequívoca responsabilidades. Assim, quando ocorre uma falha de contrato não há dúvidas sobre quem é o responsável por essa falha.

Os contratos mais importantes são os que contratualizam os módulos essenciais de um programa, ou seja, os métodos e as classes.

Contrato de um Método

O contrato aplicado a um método é especificado através de *pré-condições* e *pós-condições*. A pré-condição de um método deverá ser verdadeira no início da sua execução, pelo que

[1] Na literatura anglo-saxónica o termo Programação por Contrato é designado *Design-by-Contract* e daí o seu acrónimo.

a responsabilidade em garantir a veracidade da asserção é do cliente que invoca o método. A pós-condição deverá ser verdadeira no final da execução do método, pertencendo aquela responsabilidade ao método.

Nestes termos, uma pré-condição restringe clientes e protege entidades do programa, enquanto que uma pós-condição restringe entidades do programa e garante um certo resultado aos clientes.

Contrato de uma Classe

Para completar a especificação de uma classe é utilizado um novo tipo de asserção – *invariante*. O invariante de uma classe deverá ser verdadeiro em todos os estados estáveis dos seus objectos, ou seja, quando são utilizáveis externamente, isto é, quando são criados e quando são utilizados os seus serviços públicos. A responsabilidade para garantir a veracidade do invariante é da própria classe, caso não existam atributos públicos.

Numa classe, a junção dos contratos dos métodos públicos e dos invariantes formam o seu contrato. A Tabela 2.1 sumariza as responsabilidades de um cliente e de uma classe.

	Deveres	**Direitos**
Cliente	Satisfazer a pré-condição de cada serviço requerido.	Tanto o invariante da classe como a pós-condição do método são verdadeiras após a execução do método.
Classe	Garantir que o invariante da classe se verifica nos tempos estáveis. Garantir que no fim da execução de cada um dos seus serviços a respectiva pós-condição se verifica.	Exigir a verificação da respectiva pré-condição, sempre que um dos seus serviços é solicitado.

Tabela 2.1: DbC - Contrato de uma classe

Os invariantes e as pós-condições são propriedades locais a cada classe, isto é, a sua garantia é da inteira responsabilidade do módulo, independentemente do número de módulos existentes.

2.3.2 Mecanismo Disciplinado de Excepções

No contexto da programação por contrato, só existem dois resultados possíveis para a execução de um método: sucesso (o método cumpre o seu contrato) ou insucesso (é lançada uma excepção que simboliza uma falha de contrato), não existindo meio termo. Este enunciado corresponde à Primeira Lei da Contratualização de *Software* [9].

Assim, quando uma excepção é lançada, a mesma nunca poderá ser ignorada. Assim pode acontecer, por exemplo, na linguagem Java que utiliza o mecanismo `try/catch` para lidar com as excepções (no Bloco de Código 3.8 a excepção é ignorada porque o bloco de código `catch` está vazio).

Bloco de Código 2.1: Java - Gestão de Excepções

```
static double sqrt(double x){
   if(x < 0) throw new IllegalArgumentException();
   ...
}
static void calculateSqrt(double x){
   ...
   try{
      sqrt(x);
   }catch(IllegalArgumentException ex){ /*BlocoVazio*/ }
}
```

A Segunda Lei da Contratualização de *Software* enuncia que quando um método não cumpre o seu contrato, o método que o invoca também não cumpre [9].

É por isso importante perceber quais as soluções para lidar com uma excepção, que de acordo com [9], são só duas: pânico organizado ou recomeço. Na primeira solução admite-se que o contrato não pode ser cumprido e, após garantir-se o estado interno do objecto, é reportada uma falha. Esta falha despoleta o lançamento de uma excepção que é sinalizada ao método que invocou o método que falhou, que poderá depois aplicar uma destas duas soluções (e assim sucessivamente). Na segunda solução o método tenta resolver o problema que comprometeu o cumprimento do seu contrato e recomeça a execução do seu corpo. Este tratamento das excepções constitui o princípio do Mecanismo Disciplinado de Excepções.

Desta forma, a abordagem da programação por contrato às excepções está bem definida e é distinta da abordagem do Java, dado que uma excepção contratual não pode nunca ser ignorada.

2.3.3 Contract-Java: Linguagem

Nesta secção faremos uma síntese dos aspectos mais importantes relacionados com a linguagem Contract-Java [1]. Esta linguagem oferece um suporte completo à programação por contrato em Java uma vez que:

- Faz a distinção entre diferentes tipos de contratos;
- Possibilita a definição de contratos junto das entidades (métodos, classes) que contratualizam;
- Possibilita a definição de contratos na interface pública (TDA) de uma classe;
- Possibilita herança de contratos;
- Apresenta um mecanismo disciplinado próprio para lidar com as falhas de contrato.

Contrato de um Método

As palavras reservadas `requires` e `ensures` permitem exprimir as pré-condições e pós-condições de um método, respectivamente – Bloco de Código 2.2.

Bloco de Código 2.2: Contract-Java - Contrato de um Método

```
public class ArrayedQueue<T>{
   ...
   public void in(T e);
      requires !isFull();
   { /*CorpoDoMétodo*/ }
      ensures  !isEmpty() && peekIn() == e ;
   ...
}
```

Os construtores, que à semelhança dos restantes serviços de uma classe também são métodos mas apenas invocados quando um novo objecto é criado, podem igualmente ter pré-condições e/ou pós-condições – Bloco de Código 5.1.

Bloco de Código 2.3: Contract-Java - Contrato de um Método Construtor

```
public class ArrayedQueue<T>{
   ...
   public ArrayedQueue(int maxSize)
      requires maxSize >= 0;
   { /*CorpoDoMétodo*/ }
   ...
}
```

Contrato de uma Classe/Interface

A palavra reservada `invariant` permite que o programador exprima as condições externas que caracterizam uma classe (ou interface) como um todo – Bloco de Código 2.4. Um invariante pode ter quatro níveis de visibilidade: `package`, `public`, `protected` ou `private`.

Bloco de Código 2.4: Contract-Java - Invariante de Classe

```
public class ArrayedQueue<T>{
   public invariant (isEmpty() && size()==0) || (!isEmpty() && size()>0);
   ...
}
```

As interfaces são as entidades que especificam um TDA, sem qualquer tipo de implementação. São as interfaces públicas das classes. Assim, para que um TDA possa ser completamente especificado, os contratos deverão fazer parte da sua semântica – Bloco de Código 2.5.

Bloco de Código 2.5: Contract-Java - TDA

```
public interface QueueInterface<T>
{
   invariant (isEmpty() && size() == 0) || (!isEmpty() && size()>0 );

   public void in(T e);
      requires !isFull() ;
      ensures  !isEmpty();
      ensures  peekIn() == e ;

   public void out() ;
      requires !isEmpty();
      ensures  !isFull();

   public T peek();
      requires !isEmpty();

   public T peekOut();
      requires !isEmpty();
      ensures  !isFull();


   public T peekIn();
      requires !isEmpty();

   public void clear();
      ensures isEmpty();

   public boolean isEmpty();

   public boolean isFull();

   public int size();
}
```

Herança de Contratos

Sendo A uma subclasse de B, o contrato da classe B (contratos dos métodos públicos e invariantes) é herdado pela classe A – Bloco de Código 2.6. Também quando A for subtipo de B, os contratos dos métodos de B, juntamente com os invariantes, serão herdados – Bloco de Código 2.7.

Bloco de Código 2.6: Contract-Java - Herança (subclasse)

```
public abstract class Queue<T>{
   public abstract void in(T e)
      requires !isFull();
      ensures  !isEmpty() && peekIn() == e;
   ...
}
```

```
public class ArrayedQueue<T> extends Queue<T>{
   //Pré-Condição e Pós-Condições Automaticamente Herdadas
   public void in(T e){ /*CorpoDoMétodo*/ }
   ...
}
```

Bloco de Código 2.7: Contract-Java - Herança (subtipo)

```
public interface QueueInterface<T>{
   public void in(T e)
      requires !isFull();
      ensures  !isEmpty() && peekIn() == e;
   ...
}
```

```
public class ArrayedQueue<T> implements QueueInterface<T>{
   //Pré-Condição e Pós-Condições Automaticamente Herdadas
   public void in(T e){ /*CorpoDoMétodo*/ }
   ...
}
```

Mecanismo Disciplinado de Excepções

Na programação por contrato, tal como enunciámos anteriormente (2.3.2), a existência de um mecanismo disciplinado de excepções é essencial para lidar com as falhas contratuais.

De acordo com este mecanismo, que é similar ao mecanismo da linguagem Eiffel, cada método pode ter um bloco de execução `rescue` no qual a execução do método é repetida, possivelmente seleccionando uma nova abordagem para o fazer, caso exista a instrução `retry` ou falha, despoletando a propagação de uma excepção (Bloco de Código 2.8) [9]. Se ocorrer uma nova execução do método, apenas as pós-condições (e eventualmente os invariantes) devem ser asseguradas, uma vez que a existência de pré-condições falsas não permitiria que o método fosse sequer executado da primeira vez. Para os clientes do método, este processo

é completamente transparente. Quando um método não tem o bloco de execução `rescue`, qualquer falha de contrato origina o insucesso da execução do método e a respectiva excepção é propagada para o cliente.

Bloco de Código 2.8: Contract-Java - Mecanismo Disciplinado de Excepções

```
public class ArrayedQueue<T>{
   public void in(T e)
      local int attempt = 1;
   { /*CorpoDoMétodo*/}
   rescue{
      if(attempt < 2){
         attempt++;
         retry ;
      }else{ /* Excepção propagada ao cliente */ }
   }
   ...
}
```

Este mecanismo é independente do mecanismo `try/catch` existente em Java para lidar com excepções. Para que os dois mecanismos funcionem correctamente é necessário fazer a distinção entre as excepções associadas a falhas de contrato (DbC_Failure) e as restantes. O mecanismo `try/catch` só poderá ser utilizado nas excepções que não estão associadas a falhas de contratos (as excepções contratuais não são "apanhadas"mesmo que se usem excepções *Throwable* ou *Error*). No entanto, o mecanismo `rescue`, para além das excepções implícitas DbC_Failure, pode ser utilizado nos restantes tipos de excepções.

Assim, com este mecanismo, que é totalmente independente do mecanismo `try/catch`, nunca será possível ignorar uma falha de contracto.

2.4 Síntese do Capítulo

Neste capítulo sintetizámos três paradigmas da programação sequencial: programação procedimental, programação orientada por objectos e programação por contrato.

No primeiro paradigma apresentámos a estratégia que é definida para a construção de um programa, assim como os problemas que a mesma coloca quando a complexidade de um programa aumenta. Se isto acontecer, este paradigma torna-se inadequado, sendo necessário encontrar novas metodologias de desenvolvimento.

A programação orientada por objectos é enunciada como uma solução para os problemas do paradigma anterior, tendo sido apresentados os conceitos básicos associados a este tipo de programação.

Por último, é apresentada a programação por contrato, uma metodologia que potencia a fiabilidade (correcção e robustez) de um programa. Ainda nesta secção, apresentámos os mecanismos de programação por contrato especificados pela linguagem Contract-Java.

No próximo capítulo iremos abordar a programação concorrente, apresentando os seus conceitos básicos, desafios e soluções para os mesmos.

Capítulo 3

Programação Concorrente

Com a programação concorrente surgem novos conceitos e aspectos com os quais é preciso lidar de forma a garantir que os programas fazem o que deles é esperado.

Neste contexto, este capítulo surge para apresentar os conceitos nucleares da Programação Concorrente, os problemas colocados pela integração dos seus mecanismos em linguagens sequenciais e as soluções para esses problemas.

3.1 Conceitos Nucleares

3.1.1 Subprograma

Um programa concorrente é composto por vários subprogramas (*por exemplo, threads ou processos*) de execução autónoma que cooperam para que o programa (como um todo) alcance o(s) seu(s) objectivo(s) pretendido(s). As unidades de processamento são as entidades que executam os subprogramas.

3.1.2 Concorrência Implícita vs Explícita

Um programa pode ser concorrente de forma implícita ou explícita.

A concorrência implícita refere-se aos casos em que os programas ou contêm etapas de processamento independentes que podem ser executadas em paralelo ou despoletam operações de dispositivos que podem ser executadas em paralelo com a execução do programa. A partir desta perspectiva, o comportamento concorrente é da responsabilidade do sistema de compilação e execução.

A concorrência explícita refere-se aos casos em que o comportamento concorrente é especificado pelo programador através do uso de abstracções próprias que lhe conferem esse comportamento. Deste ponto de vista, o sistema de compilação e execução sabe exactamente quais as partes executadas em concorrência.

O nosso trabalho recorre a mecanismos de concorrência explícita.

3.2 Propriedades dos Programas Concorrentes

3.2.1 Não-Determinismo

Num programa sequencial, a ordem de execução de cada uma das instruções é bem definida, nunca se alterando entre execuções diferentes para as mesmas entradas. No entanto, um programa concorrente é caracterizado por uma ordenação parcial, ou seja, as mesmas acções podem ocorrer em diferentes instantes para diferentes execuções. Esta propriedade é denominada de não-determinismo, sendo uma das suas consequências o facto de execuções distintas de um programa concorrente poderem não apresentar a mesma ordem de execução das operações para dados de entrada iguais.

3.2.2 Segurança e Vivacidade

A construção de programas concorrentes requer que dois grupos fundamentais de propriedades sejam assegurados: segurança (de execução) e vivacidade [10].

Segurança

Uma propriedade de segurança estabelece que nada de errado irá acontecer [10]. Os problemas de segurança surgem quando os subprogramas não estão correctamente sincronizados.

O problema de segurança mais comum é o surgimento de uma *race condition*, que faz com que os subprogramas possam aceder a estados instáveis de um recurso, comprometendo a consistência do recurso e, consequentemente, a correcção do programa. No exemplo 3.1, havendo dois consumidores e apenas um produtor, se entre os instantes da verificação das condições e a remoção do elemento da fila, o outro consumidor remove um elemento, é possível que a respectiva condição testada deixe de ser verdade, originando um cenário de *race condition*.

Bloco de Código 3.1: Programação Concorrente - *Race Condition*

```
public class ThreadExample {
   public static void main(String args[]) {
      ArrayedQueue<Integer> q = new ArrayedQueue<Integer>(4);
      Consumer c1 = new Consumer(q);
      Consumer c2 = new Consumer(q);
      Producer p  = new Producer(q);
      . . .
      c1.start();
      c2.start();
      p.start();
   }

   public class Consumer{
      ...
      public void run(){
         if(!q.isEmpty()){
            Thread.sleep(10);
            q.peek();
         }
      }}

public class Producer{
   ...
   public void run(){
      int cont = 1;
      if(!q.isFull()){
         Thread.sleep(10);
         q.in(cont++);
      }
   }}

}
```

Vivacidade

Uma propriedade de vivacidade estabelece que algo de positivo irá ocorrer no programa [10]. Um problema de vivacidade bastante comum é o surgimento de *dealocks* no acesso a recursos partilhados. No exemplo 3.2, os dois subprogramas ficam a aguardar interminavelmente, uma vez que tentam obter os mesmos *locks* mas por ordem contrária.

Bloco de Código 3.2: Programação Concorrente - *Deadlock*

```
public class ThreadExample {
   public static Mutex mtx1 = new Mutex();
   public static Mutex mtx2 = new Mutex();

   public static void main(String args[]) {
      Thread1 t1 = new Thread1();
      Thread2 t2 = new Thread2();
      t1.start();
      t2.start();
   }

   public class Thread1{
      public void run(){
         mtx1.lock();
         Thread.sleep(10);
         System.out.println("Mtx1 Blocked!");
         mtx2.lock();
         System.out.println("Mtx2 Blocked!");
         mtx2.unlock();
         System.out.println("Mtx2 Released!");
         mtx1.unlock();
         System.out.println("Mtx1 Released!");
      }
   }
```

```
    public class Thread2{
        public void run(){
            mtx2.lock();
            Thread.sleep(10);
            System.out.println("Mtx2 Blocked!");
            mtx1.lock();
            System.out.println("Mtx1 Blocked!");
            mt1.unlock();
            System.out.println("Mtx1 Released!");
            mtx2.unlock();
            System.out.println("Mtx2 Released!");
      }
    }
}
```

3.3 Comunicação entre Subprogramas

Em programação concorrente são utilizados essencialmente dois modelos de comunicação entre subprogramas para troca de informação: envio de mensagens (interacção directa) e partilha de memória (interacção indirecta) [11].

3.3.1 Envio de Mensagens

Quando a comunicação é feita através da troca de mensagens, é estabelecido um canal de comunicação que os subprogramas utilizam para comunicar entre si. Consequentemente, é adoptada uma tipologia emissor-receptor que poderá funcionar de forma síncrona ou assíncrona. Na primeira forma, o subprograma emissor só comunica com o subprograma receptor quando este estiver disponível para que tal interacção aconteça. Na segunda forma, assim que a mensagem enviada pelo subprograma emissor for recebida pelo receptor o emissor prossegue a execução do seu programa.

3.3.2 Partilha de Memória

Na interacção por partilha de memória, todos os subprogramas têm acesso a uma entidade partilhada disponível para leitura e/ou escrita.

Neste modelo, para que a comunicação seja bem sucedida, é necessário garantir que não se verificam situações de inconsistência da informação. Em Java existem três conceitos essenciais relacionados com a partilha de informação: atomicidade, visibilidade e ordenação. A atomicidade relaciona-se com a indivisibilidade dos efeitos de uma operação. A visibilidade determina quando é que os restantes subprogramas observam os efeitos de uma operação de

outro subprograma. A ordenação é o aspecto que relaciona as acções de uma operação e o instante em que os efeitos dessas acções são vistos pelos subprogramas.

Desta forma, as leituras posteriores a uma escrita deverão observar o novo valor escrito, mantendo assim a consistência temporal da informação.

3.4 Sincronização

A sincronização entre os subprogramas é absolutamente necessária para que os mecanismos de comunicação funcionem. Neste sentido, os esquemas de sincronismo são utilizados para garantir uma comunicação segura entre subprogramas, assegurando que sempre que um subprograma comunica com outro o resultado de tal interacção é controlado e desejado.

Podemos definir três tipos de sincronização: interna, condicional e externa.

3.4.1 Interna

Sempre que os subprogramas comunicam, partilhando informação, é necessário garantir que a informação partilhada não é corrompida por nenhum dos subprogramas envolvidos.

Assim, a sincronização interna torna-se necessária para assegurar a protecção mínima do estado interno de um recurso partilhado. Consideremos, por exemplo, uma classe do tipo fila composta por dois atributos – uma lista ligada e um atributo que indica o tamanho da fila. O sincronismo interno irá garantir que não existirão leituras externas incorrectas do conjunto dos dois atributos.

Podem ser implementados diferentes esquemas de sincronismo interno, desde os mais restritivos, que impõem um regime de exclusão mútua entre subprogramas (monitores), até aos mais liberais, que permitem o acesso concorrente por diferentes subprogramas à informação partilhada (transacções).

3.4.2 Condicional

A sincronização condicional é necessária quando o acesso a um recurso partilhado depende da verificação de uma condição, cujo valor é dependente do estado desse recurso. A título exemplificativo, pensemos na remoção de um elemento de uma fila partilhada que só deverá ocorrer se a fila não estiver vazia.

Perante uma condição de acesso, uma de três soluções pode ser escolhida [12]: esperar até que a condição seja verdadeira (*Guarded Suspension*), reportar de imediato um "erro"(*Balking*) ou uma combinação destas duas ao esperar durante um determinado período de tempo (*Time-outs*).

Bloco de Código 3.3: Espera Condicional - *Guarded Suspension*

```
public class Example {
   synchronized void guardedMethod() {
      while (!preCondition()) {
         try { wait(); } catch (InterruptedException e) {/*...*/}
      }
      ...        // Implementação da tarefa
   }
   synchronized void alterObjectStateMethod() {
      ...        // Alteração do estado do objecto
      notify(); // Informação os restantes subprogramas em espera
   }
}
```

Bloco de Código 3.4: Espera Condicional - *Balking*

```
public class Example {
   synchronized void method() {
      if (preCondition()) { /*...*/ }
      else{ throw Exception; }
      ...
   }
}
```

Bloco de Código 3.5: Espera Condicional - *Time-outs*

```
public class Example {
   synchronized void guardedMethod() {
      while (!preCondition()) {
         try { wait(TIME_OUT); } catch (InterruptedException e) { /*...*/ }
      }
      ...        // Implementação da tarefa
   }

   synchronized void alterObjectStateMethod() {
      ...        // Alteração do estado do objecto
      notify(); // Informação os restantes subprogramas em espera
   }
}
```

3.4.3 Externa

A sincronização externa é necessária caso se pretenda actuar num recurso partilhado para executar uma sequência de operações, sem que outros subprogramas interfiram. Vejamos o exemplo já apresentado no Bloco de Código 3.1, em que a fila, enquanto recurso partilhado, deverá ser reservada para garantir que entre os instantes da verificação da condição e a

remoção/adição de um elemento mais nenhum subprograma interfere no recurso (linhas 16/18 e 17/19, respectivamente)

3.5 Algumas Linguagens Existentes para Programação OO Concorrente

Ao longo do tempo foram surgindo diferentes linguagens orientadas por objectos concorrentes. Neste sentido, consideramos relevante a análise de algumas das linguagens que mais relevo apresentam na computação orientado por objectos concorrente. Para cada uma das linguagens é exemplificado o problema clássico dos produtores-consumidores.

3.5.1 Eiffel

A primeira versão desta linguagem orientada por objectos pura é de 1986 [8]. Permite o encapsulamento de informação, a existência de relações de herança múltipla e o encaminhamento dinâmico. É uma linguagem que tem mecanismos para excepções. A gestão da memória é feita de forma automática.

Esta linguagem disponibiliza o modelo de concorrência SCOOP (*Simple Concurrent Object Oriented Programming*), desenvolvido em 1990, que permite o desenvolvimento de programas concorrentes nesta linguagem através da utilização da palavra-chave `separate`.

Bloco de Código 3.6: Linguagem Eiffel - Produtores-Consumidores

```
class PRODUCER_CONSUMER create
   make
feature -- Initialization
   b: separate BUFFER
   p: PRODUCER
   c: CONSUMER
   make is -- Creation procedure.
      do
         create b.create_buffer
         create p.create_producer (b)
         create c.create_consumer (b)
      end
end -- class PRODUCER_CONSUMER

separate class PRODUCER create
   make
feature {NONE}
   buffer: separate BUFFER
   create_producer (b: separate BUFFER) is
      do
         buffer := b
         keep_producing
      end
```

```
    keep_producing is
        local i: INTEGER
        do
            from
            until
                False
            loop
                i := (i + 1)
                produce (buffer, i)
            end
        end
    produce (b: BUFFER; i: INTEGER) is
        require b.size <= 2
        do b.put (i) end
end -- class PRODUCER

separate class CONSUMER create
    make
feature {NONE}
    buffer: separate BUFFER
    create_consumer (b: separate BUFFER) is
        do
            buffer := b
            keep_consuming
        end
    keep_consuming is
        do
            from
            until
                False
            loop
                consume (buffer)
            end
        end
    consume (b: separate BUFFER) is
        require b.size > 0
        do b.remove end
end -- class CONSUMER

class BUFFER create
    make
feature
    size: INTEGER is
        do Result := q.size end
    item: INTEGER is
        do Result := q.item end
    put (x: INTEGER) is
        require size <= 3
        do
            q.put (x)
            print ("PUT")
            io.new_line
        ensure
            size = old size + 1
            q.has (x)
        end

    remove is
        require size > 0
        do
            q.remove
            print ("REMOVE")
            io.new_line
        ensure size = old size - 1
        end
```

```
feature {NONE}
   q: QUEUE [INTEGER]
   create_buffer is
      do create {ARRAYED_QUEUE [INTEGER]} q.make (3) end
invariant
   inv: size <= 3
end -- class BUFFER
```

3.5.2 Ada

Esta linguagem surge em 1980 mas é apenas em 1995 que se aproxima do paradigma da programação orientado por objectos [13]. O sistema de tipos é estático. Há a possibilidade de encapsular informação. As relações de herança são simples (na versão de 2005 há a possibilidade de herança múltipla nas interfaces). O encaminhamento pode ser dinâmico. Existe um mecanismo para excepções. O programador é responsável pela gestão de memória.

Na linguagem Ada a concorrência pode ser explicitada através de vários mecanismos, nomeadamente, *tasks* e objectos *protected*. As *tasks* permitem criar subprogramas concorrentes e assíncronos, não sendo necessário que os programadores os controlem explicitamente. No caso dos objectos especificados como *protected*, os mesmos actuam como mecanismos tanto de sincronização como de exclusão mútua.

Bloco de Código 3.7: Linguagem Ada - Produtores-Consumidores

```
with Text_Io;
use  Text_Io;

procedure ProducerConsumer is
   Capacity : constant := 10;
   type buffer is Array (1..Capacity) of integer;

   protected Buffer is
      entry Put    (X: in  integer);
      entry Remove (X: out integer);
   private
      Buffers : buffer;
      head, tail : integer range 1..Capacity := 1;
      size : integer range 0..Capacity := 0;
   end Buffer;

   protected body Buffer is
      entry Put (X: in  integer) when size < Capacity is
      begin
         Buffers (tail) := X;
         Next_in := (tail mod Capacity) + 1;
         size := size + 1;
      end Put;

      entry Remove (X: out integer) when size > 0 is
      begin
         X := Buffers (head);
         head := (head mod Capacity) + 1;
         size := size - 1;
```

```
      end Remove;
   end Buffer;

   task Producer;
   task body Producer is
      item : integer := 0;
   begin
      loop
         item := item + 1;
         Put ("item:" & item'img & "  put in buffer");
         New_line;
         Buffer.Put (item);
      end loop;
   end Producer;

   task Consumer;
   task body Consumer is
      item : integer;
   begin
      loop
         Buffer.Remove (item);
         Put ("item:" & item'img & "  obtained from buffer");
         New_line;
      end loop;
   end Consumer;

begin
   null;
end ProducerConsumer;
```

3.5.3 Java

Linguagem orientada por objectos com suporte para concorrência, cujos programas são baseados em classes, sendo a sua primeira versão de 1995 [14]. O sistema de tipos é estático. Possibilidade de encapsular informação. Herança simples (a partir da versão Java-v8 pode existir herança múltipla nas interfaces). Existe um mecanismo para tratar as excepções. A memória é gerida de forma automática.

A linguagem Java disponibiliza a palavra-chave `synchronized` para proteger algumas partes de código que são executadas por vários subprogramas ao mesmo tempo. Esta palavra assegura que em cada instante apenas um subprograma executa o bloco de código protegido e que um subprograma ao entrar num bloco *synchronized* observa todos os efeitos das modificações anteriores feitas no mesmo *lock*. Se a palavra for aplicada na definição do método, apenas um subprograma o poderá executar em cada instante (os restantes terão de aguardar).

Bloco de Código 3.8: Linguagem Java - Produtores-Consumidores

```
public class ProducerConsumer {
   public static void main(String[] args) {
      Buffer<Integer> buffer = new Buffer<Integer>(10);
      Producer[] producers = new Producer[4];
      for(int i = 0; i < 4; i++){
```

```
            producers[i] = new Producer(buffer);
            producers[i].start();}

        Consumer[] consumers = new Consumer[4];
        for(int i = 0; i < 4; i++){
            consumers[i] = new Consumer(buffer);
            consumers[i].start();}
    }
}

public class Producer extends Thread {
    protected Buffer<Integer> buffer;
    protected int count = 0;
    public Producer(Buffer<Integer> buffer) { this.buffer = buffer; }
    public void run() {
        while(true){
            try {
                buffer.put(++count);
                Sstem.out.println(count + " ADDED");
                Thread.sleep(2000);
            }catch (InterruptedException exception) {} }
    }
}

public class Consumer extends Thread {
    protected Buffer<Integer> buffer;
    public Consumer (Buffer<Integer> buffer) { this.buffer = buffer; }
    public void run() {
        while(true){
            try {
                System.out.println(buffer.remove() + " REMOVED");
                Thread.sleep(3000);
            }catch (InterruptedException exception) {} }
    }
}

public class Buffer<T> {
    private Object[] buffer;
    private int head, tail, size;

    public Buffer(int capacity) {
        buffer = new Object[capacity];
        head = 0, tail = 0, size = 0;
    }

    public synchronized void put(T e) throws InterruptedException {
        while (size == buffer.length){wait();}
        buffer[tail] = e;
        tail++;
        size++;
        if (tail == buffer.length){tail = 0;}
        notifyAll();
    }

    public synchronized T remove() throws InterruptedException {
        while (size == 0){wait();}
        T element = (T) buffer[head];
        head++;
        size--;
        if (head == buffer.length){head = 0;}
        notifyAll();
        return element;
    }
}
```

3.5.4 Go

Esta linguagem, cuja primeira versão é de 2009, foi desenvolvida na Google, sendo baseada na linguagem de programação C [15]. O sistema de tipos estático. Não se verificam propriedades de herança. Não existem excepções. A memória é gerida de forma automática. Suporte (sintaxe da linguagem e bibliotecas) para o desenvolvimento de programas concorrentes.

Nesta linguagem, a estrutura básica de concorrência é uma *goroutine*, semelhante a uma *thread* em Java. A invocação de uma *goroutine* é feita através do prefixo `go`. A comunicação entre *goroutines* é feita, preferencialmente, através de *channels*.

Bloco de Código 3.9: Linguagem Go - Produtores-Consumidores

```
package main

import (
    "fmt"
    "strconv"
    "time"
)

func producer(c chan<- string) {
    i := 1
    for {
        msg := strconv.Itoa(i)
        c <- msg
        fmt.Println("ADDED \t #" + msg)
        i++
    }
}

func consumer(c <-chan string) {
    for {
        time.Sleep(time.Second *4)
        msg := <- c:
        fmt.Println("REMOVED \t #" + msg)
    }
}

func main() {
    buffer := make(chan string, 2)
    go producer(buffer)
    go consumer(buffer)
}
```

3.6 Aproximações OO Concorrentes por Bibliotecas

3.6.1 C++

Esta linguagem é uma extensão da linguagem de programação C, com suporte para mecanismos de programação orientada por objectos, sendo a sua primeira versão de 1983 [16]. O sistema de tipos estático. Existe a possibilidade de encapsulamento de informação e de

relações de herança múltipla (funcionalidade introduzida na versão de 1989). Existe também a possibilidade de encaminhamento dinâmico, que deverá ser explicitado. Apresenta mecanismos para gestão de memória e excepções.

A biblioteca *standard* de C++ suporta a programação concorrente, disponibilizando diversos mecanismos, como por exemplo, monitores, semáforos, referências atómicas ou variáveis de condição.

Bloco de Código 3.10: Linguagem C++ - Produtores-Consumidores

```
struct BoundedBuffer {
   int* buffer;
   int capacity, head, tail, size;
   std::mutex lock;
   std::condition_variable not_full, not_empty;

   BoundedBuffer(int capacity) : capacity(capacity), head(0), tail(0), size(0) {
      buffer = new int[capacity]; }

   void put(int e){
      std::unique_lock<std::mutex> l(lock);
      not_full.wait(l, [this](){return size != capacity; });
      buffer[tail] = e;
      tail = (tail + 1) % capacity;
      ++size;
      not_empty.notify_one();
   }

   int remove(){
      std::unique_lock<std::mutex> l(lock);
      not_empty.wait(l, [this](){return size != 0; });
      int element = buffer[head];
      head = (head + 1) % capacity;
      --size;
      not_full.notify_one();
      return element;
   }
}

void consumer(BoundedBuffer& buffer){
   while(true){
      int value = buffer.remove();
      std::cout << value << " REMOVED" << std::endl;
      std::this_thread::sleep_for(std::chrono::milliseconds(250));
   }
}

void producer(BoundedBuffer& buffer){
   while(true){
      buffer.put(i);
      std::cout << i << " ADDED" << std::endl;
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }
}

int main(){
   BoundedBuffer buffer(200);
   std::thread c1(consumer, std::ref(buffer));
   std::thread p1(producer, std::ref(buffer));
   return 0;
}
```

3.6.2 POSIX Threads

A biblioteca POSIX-Threads acrescenta concorrência à linguagem de programação procedimental C [17]. Permite a criação de fluxos de processos concorrentes, sendo mais eficiente em sistemas com mais do que um processador, ao tirar partido do processamento paralelo ou distribuído. No entanto, os sistemas que apenas têm um processador também podem obter ganhos, ao tirar partido da latência nas operações I/O e noutras funções do sistema que são capazes de abrandar a execução do processo.

Bloco de Código 3.11: POSIX Threads - Produtores-Consumidores

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

#define CAPACITY 100
#define PRODUCERS 5
#define CONSUMERS 5

int size=0;
pthread_mutex_t buffer_mutex;
pthread_cond_t  condition_variable;

void * add(void * arg){
    pthread_mutex_lock(&buffer_mutex);
    while(size+1>CAPACITY){ pthread_cond_wait(&condition_variable,&buffer_mutex); }
    size+=1;
    printf("\nADDED 1 element to buffer");
    pthread_cond_broadcast(&condition_variable);
    pthread_mutex_unlock(&buffer_mutex);
    pthread_exit(0);
}

void * remove(void * arg){
    pthread_mutex_lock(&buffer_mutex);
    while(size-1<0){ pthread_cond_wait(&condition_variable,&buffer_mutex); }
    size-=1;
    printf("\nREMOVED 1 element from buffer");
    pthread_cond_broadcast(&condition_variable);
    pthread_mutex_unlock(&buffer_mutex);
    pthread_exit(0);
}

int main(int argc, char ** argv){
    int i;
    pthread_t producer[PRODUCERS], consumer[CONSUMERS];

    pthread_mutex_init(&buffer_mutex,NULL);
    pthread_cond_init(&condition_variable,NULL);

    for(i=0;i<PRODUCERS;i++){ pthread_create(&producer[i],NULL,add,NULL);      }
    for(i=0;i<CONSUMERS;i++){ pthread_create(&consumer[i],NULL,remove,NULL); }

    pthread_mutex_destroy(&buffer_mutex);
    pthread_cond_destroy(&condition_variable);
    return 0;
}
```

3.7 Síntese do Capítulo

Neste capítulo começámos por enunciar conceitos básicos da programação concorrente, nomeadamente o de concorrência explícita, uma vez que no nosso trabalho o comportamento concorrente dos programas é explicitamente especificado pelo programador.

Abordámos as propriedades principais de uma programa concorrente e daí transitámos para a forma como os subprogramas podem comunicar.

Apresentados os mecanismos de comunicação, identificámos os três aspectos de sincronização necessários para que os mecanismos de comunicação funcionem: interna, condicional e externa.

Nas últimas secções analisámos algumas das linguagens orientados por objectos concorrentes e aquelas cuja componente concorrente existe por meio de bibliotecas, exemplificando a semântica dos mecanismos concorrentes de cada uma das linguagens apresentadas.

O capítulo que se segue fará a apresentação da nossa linguagem, que introduz mecanismos concorrentes de alto nível simples, expressivos e seguros, que facilitam o trabalho do programador no desenvolvimento de programas concorrentes.

Capítulo 4

Concurrent Contract-Java : Linguagem

Neste capítulo pretendemos identificar mecanismos práticos que permitam abordar a concorrência de forma mais simples e segura. Simultaneamente apresentaremos a linguagem que concretiza essas mesmas aproximações.

A linguagem protótipo que desenvolvemos – Concurrent Contract-Java (CCJava) – assenta em três pilares essenciais: programação orientada por objectos, programação por contrato e programação concorrente.

A nossa abordagem utiliza a linguagem Contract-Java [1] como linguagem base, trocando os seus mecanismos concorrentes nativos[1] por outros de alto nível. O objectivo é facilitar, tanto quanto possível, a programação concorrente e garantir estaticamente a segurança dos programas construídos. Para atingir este objectivo, decidimos reutilizar ao máximo os mecanismos da linguagem Contract-Java, adaptando as semânticas desses mecanismos a um contexto concorrente, e adicionar duas novas palavras reservadas ao sistema de tipos da linguagem – `shared` e `remote` –, adaptando a sua semântica para permitir a identificação segura de entidades partilhadas.

O Bloco de Código 4.1 apresenta o problema clássico dos produtores-consumidores desenvolvido na nossa linguagem. O Anexo A contém o código completo, em CCJava, para este problema.

[1] São os mesmos que os da linguagem Java.

Bloco de Código 4.1: CCJava - Problema Produtores-Consumidores

```
public class ProducerConsumerProblem{
   public static void main(String[] args){
      shared ArrayedQueue<Integer> sq = new shared ArrayedQueue<Integer>(25);

      Producer[] producers = new Producer[4];
      for(int i = 0; i < 4; i++)
         producers[i] = new Producer("producer #"+(i+1), sq);

      Consumer[] consumers = new Consumer[4];
      for(int i = 0; i < 4; i++)
         consumers[i] = new Consumer("consumer #"+(i+1), sq);
   }
}

public abstract remote class SQThread{
   public SQThread(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;
      requires sq != null;
   {
      while(true){
         therun(id, sq);
         pause((int)(Math.random()*100.0));
      }
   }

   public abstract void therun(String id, shared ArrayedQueue<Integer> sq);
}

public remote class Producer extends SQThread{
   public Producer(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;
      requires sq != null;
   { super(id, sq); }

   protected int count = 0;
   public void therun(String id, shared ArrayedQueue<Integer> sq){
      sq.in(++count);
      System.out.println("["+id+"]:unique value produced - "+count+"{"+this+"}");
   }
}

public remote class Consumer extends SQThread{
   public Consumer(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;
      requires sq != null;
   { super(id, sq); }

   public void therun(){
      System.out.println("["+id+"]:unique value consumed - " +sq.peekOut()+"{"+this+"}");
   }
}

public class ArrayedQueue<T>{
   invariant (isEmpty() && size() == 0) || (!isEmpty() && size()>0 );
   protected boolean limited;
   protected T[] array = null;
   protected int pout, pin, size;

   public ArrayedQueue(){
      this(100);        // DEFAULT_SIZE = 100
      limited = false;
   }

   public ArrayedQueue(int maxSize)
      requires maxSize >= 0;
```

```
    {
        size = 0; array = (T[]) new Object[maxSize];
        pout = 0; pin = array.length-1;
        limited = true;
    }

    public void in(T e)
        requires !isFull() ;
    {/*CorpoDoMétodo*/}
        ensures  !isEmpty() && peekIn() == e ;

    public void out()
        requires !isEmpty();
    {/*CorpoDoMétodo*/}
        ensures  !isFull();

    public T peek()
        requires !isEmpty();
    {/*CorpoDoMétodo*/}

    public T peekOut()
        requires !isEmpty();
    {/*CorpoDoMétodo*/}
        ensures  !isFull();

    public T peekIn()
        requires !isEmpty();
    {/*CorpoDoMétodo*/}

    public void clear()
    {/*CorpoDoMétodo*/}
        ensures isEmpty();

    public boolean isEmpty()  {/*CorpoDoMétodo*/}

    public boolean isFull()   {/*CorpoDoMétodo*/}

    public int size()         {/*CorpoDoMétodo*/}

    public boolean isLimited(){/*CorpoDoMétodo*/}

    public int maxSize()
        requires isLimited();
    { /*CorpoDoMétodo*/ }
}
```

4.1 Objectos Partilhados

No modelo de comunicação por partilha de memória, os subprogramas comunicam indirectamente através de recursos partilhados. Em linguagens OO puras (como a nossa), os recursos são tipicamente objectos, pelo que este modelo pode ser designado por modelo de comunicação por partilha de objectos.

Em linguagens OO, as propriedades modulares de um objecto estão relacionadas com o facto do objecto ser analisado como uma unidade coesa. Tratar o objecto como uma entidade atómica em contextos de execução concorrente é suficiente para que a sua modularidade,

simplicidade e abstracção se mantenham. O uso concorrente de objectos partilhados atómicos é seguro, uma vez que os objectos só podem ser usados nos seus estados estáveis. A ocorrência de uma falha coloca o objecto num estado instável, do qual poderá recuperar através da cláusula `rescue`.

Assim, a abordagem da nossa linguagem à concorrência é baseada no tratamento dos objectos partilhados como entidades atómicas.

Uma vez que os objectos partilhados requerem código de sincronização próprio, é necessário identificá-los sem ambiguidade. Assim, como identificar de forma inequívoca esses objectos? A nossa solução usa o sistema de tipos estático da linguagem para identificar os objectos partilhados, através da adição da palavra reservada `shared`. Desta forma, o compilador consegue gerar o código necessário para garantir a segurança e eficiência dos programas concorrentes. Sendo um objecto partilhado, a execução dos seus serviços pode ser requerida por mais do que um subprograma em instantes sobrepostos. Todos os objectos que não forem partilhados serão objectos sequenciais.

O programador poderá definir classes partilhadas, ou seja, classes que criam sempre objectos partilhados, ou poderá definir entidades da linguagem que sejam partilhadas (variáveis, atributos, argumentos), situação que apresenta a vantagem de se poder utilizar a mesma classe para construir objectos sequenciais e partilhados – Bloco de Código 4.2.

O sistema de tipos da linguagem assegura que uma entidade partilhada só se refere a objectos partilhados e que uma entidade sequencial só se refere a objectos sequenciais (por exemplo, o resultado de um serviço que retorna um objecto partilhado não pode ser atribuído a um objecto sequencial e vice-versa). Ou seja, é possível garantir a transitividade concorrente da linguagem.

Os objectos partilhados são criados através do uso do operador `new`, aplicado a uma entidade partilhada. A informação que o compilador precisa para criar código concorrente seguro e correcto é apenas esta.

Bloco de Código 4.2: CCJava - Criação de Objectos Partilhados

```
SharedClass sc = new SharedClass();
public shared class SharedClass{...}

shared ArrayedQueue<Integer> queue = new shared ArrayedQueue<Integer>();
```

4.2 Sincronização Automática e Abstracta de Objectos Partilhados

Quando se pretende integrar mecanismos concorrentes em linguagens OO a sincronização suscita alguns problemas [18] [19]. O facto da responsabilidade de identificar os pontos de sincronização dos objectos partilhados pertencer ao programador poderá ser uma das justificações para tais problemas.

Nestes termos, decidimos implementar o sincronismo de forma automática para garantir a segurança estática na sincronização dos objectos. Consequentemente, averiguámos se os esquemas de sincronismo podem ser gerados pelo compilador.

4.2.1 Sincronização Abstracta

Não obstante o facto da sincronização ser automática, deverá ser possível escolher diferentes esquemas de sincronismo. A atomicidade dos objectos partilhados não é necessariamente conseguida apenas com um esquema de sincronismo de exclusão mútua. Podem existir casos em que outros esquemas possam ser aplicados (por exemplo, *lock-free*, exclusão entre leitores-escritor). Assim, restringir a sincronização de um objecto partilhado a uma só escolha pode significar que usos concorrentes seguros desse objecto sejam excluídos.

Neste sentido, admitindo que o programador pode escolher livremente qualquer esquema de sincronismo seguro e executável, cuja correcção não dependa de si, é possível conseguir uma aproximação segura e abstracta a uma linguagem orientada por objectos concorrente.

4.2.2 Sincronização Interna

Quando vários subprogramas acedem ao mesmo objecto torna-se necessário assegurar a sua consistência. Este aspecto de sincronização garante a atomicidade do objecto. O próprio objecto é responsável por garantir que o invariante de classe não é comprometido por usos dessincronizados dos seus serviços.

O esquema de sincronismo mais simples que assegura todos estes requisitos é um esquema de exclusão mútua na utilização dos métodos públicos do objecto. No entanto, tal como explicámos, a atomicidade pode ser conseguida através de vários esquemas, não estando a escolha limitada a uma só possibilidade.

Assim, a sincronização interna da nossa linguagem é abstracta, o que significa que neste aspecto de sincronismo dos objectos partilhados não é imposto um esquema em particular. Ainda assim, o programador pode sugerir um esquema específico ao compilador através do sistema de anotações – Bloco de Código 4.3. Nos casos em que o compilador não pode, por alguma razão, implementar o esquema sugerido, é emitido um aviso e o compilador escolhe um esquema que seja exequível.

Bloco de Código 4.3: CCJava - Sugestão de Esquema para Sincronismo Interno

```
SharedClass sc = @rwex new SharedClass();
shared Queue<Integer> q = @monitor new shared Queue<Integer>();
```

4.3 Contratos e Expressões Concorrentes

A integração de mecanismos concorrentes em linguagens OO com suporte para DbC pode pôr em causa o normal comportamento dos mecanismos da linguagem. Concretamente, referimo-nos aos casos em que os contratos são analisados com um carácter concorrente. Como deverão os mecanismos de DbC funcionar quando as cláusulas são asserções concorrentes?[2] A este propósito, a Figura 4.1 (retirada de [20]) indica os três comportamentos possíveis a executar quando estivermos na presença de uma asserção desse tipo [20].

Assim, a primeira opção, que corresponde a um comportamento sequencial não sincronizado, faz com que o teste da verificação da condição C não seja sincronizado, não sendo por isso uma solução aceitável neste contexto.

Relativamente à segunda opção, o objecto concorrente envolvido na asserção é reservado na sua totalidade. A sua verificação é feita posteriormente e já de uma forma sequencial. No entanto, também este comportamento está incorrecto, pois a reserva exclusiva do objecto não depende da condição da asserção, pelo que essa condição poderá ser verdadeira ou falsa consoante o instante em que a reserva seja efectuada. Desta forma, a asserção deixa de poder ser utilizada como um teste de correcção.

Na última possibilidade, as asserções concorrentes estão associadas a um processo de espera condicional. Anteriormente (3.4.2) identificámos três soluções para implementar este

[2] Uma asserção diz-se concorrente caso a condição que defina seja concorrente. Uma condição é uma expressão booleana que se diz concorrente caso o teste do seu valor dependa de mais do que um subprograma para além do responsável pelo teste.

aspecto de sincronização, sendo que no nosso trabalho optámos pela primeira forma apresentada. Apesar desta escolha poder colocar problemas de *liveness*, é a única que permite construir uma solução modular segura dentro do objecto. As restantes possibilidades requerem que os clientes dos métodos lidem com este tipo de problema. Neste sentido, no nosso trabalho, utilizámos semântica de espera na sincronização condicional. Assim, um subprograma terá sempre que aguardar pela verificação da asserção concorrente em questão. Os contratos concorrentes são, por isso, tratados como uma forma de sincronização condicional.

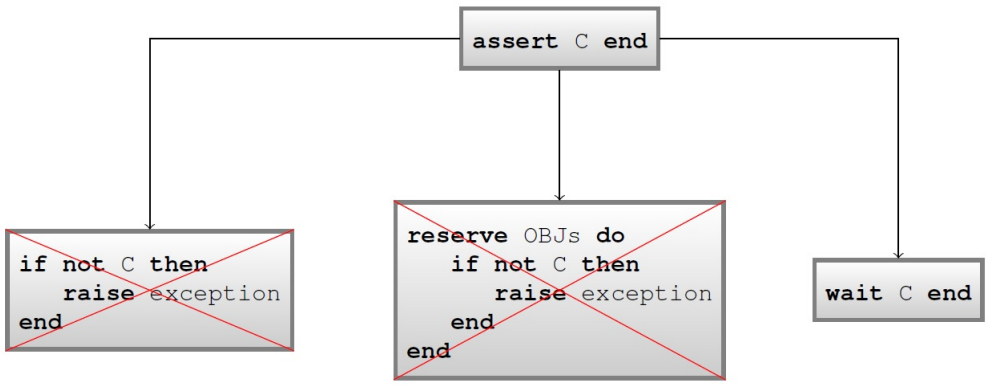

Figura 4.1: Tratamento de Asserções Concorrentes

Num programa concorrente, o valor de uma asserção não depende unicamente do subprograma que a testa. Reforça-se assim a necessidade de adoptar uma estratégia que transforme estas asserções em instruções de espera condicional. No entanto, perante a existência de pré-condições concorrentes, a espera condicional não é suficiente, uma vez que é necessário que tais condições se mantenham entre os instantes das suas verificações e a execução dos corpos dos métodos. Nestes termos, para além da possível espera condicional, faz-se necessária a reserva exclusiva dos objectos partilhados envolvidos nas pré-condições.

4.4 Instruções de Selecção e Expressões Concorrentes

O controlo de fluxo em programas imperativos é determinado por sequências, instruções condicionais, instruções repetitivas e funções. Em programas sequenciais, o significado de cada uma destas abstracções é evidente.

Analisemos com particular atenção as instruções condicionais (`if...do`) e repetitivas

(`do...while; while...do`) que seleccionem algoritmos por condições concorrentes. Num contexto de execução concorrente, se lhes aplicarmos a interpretação sequencial, o código não será seguro, sendo uma *race condition*. Quando a instrução de selecção envolve uma condição concorrente, coloca-se o problema da necessidade de assegurar a condição durante a execução do bloco de código seleccionado. Só desta forma será possível manter a semântica associada a cada uma das instruções elementares desse bloco. É por isso necessário impor a reserva exclusiva dos objectos partilhados envolvidos na condição concorrente.

Nas condições concorrentes deste tipo de instruções, ambos os valores possíveis (verdadeiro ou falso) são importantes para a correção do algoritmo. Estas instruções são de selecção algorítmica e não de correcção. Assim, e ao invés do que acontece nas asserções concorrentes, não fará sentido associar uma acção de espera condicional a estas instruções.

Desta forma, as instruções condicionais e repetitivas são tratadas na nossa linguagem como pontos de sincronização externa. Associando esta semântica a estes tipos de instruções, a reserva exclusiva de objectos concorrentes só será feita quando for estritamente necessária, aumentando desta forma a disponibilidade concorrente dos objectos.

4.5 Criação de Subprogramas

Os mecanismos concorrentes da nossa linguagem protótipo são fortemente baseados naqueles que a linguagem MPEiffel utiliza para abordar concorrência [21]. A linguagem MPEiffel implementa ambos os modelos de comunicação entre subprogramas (secção 3.3), sendo que as entidades de tipo remotas são parte da solução para a implementação do modelo de comunicação por passagem de mensagens.

Presentemente, a linguagem CCJava utiliza entidades de tipo remotas apenas para criar subprogramas. O ciclo de vida do subprograma está associado ao método utilizado para construir o objecto, que se torna o seu método `run`.

Na realidade, o mecanismo utilizado para a criação de um subprograma pode ser interpretado como um mecanismo de passagem de mensagens. A instanciação de um novo objecto remoto, utilizando, quando existe, a informação passada sob a forma de argumentos, e a definição de um algoritmo para o subprograma é a mensagem transmitida – Bloco de Código 4.4.

Bloco de Código 4.4: CCJava - Criação de Subprogramas (1)

```
public abstract remote class SQThread{
   public SQThread(String id, shared Queue<Integer> sq)
      requires id != null && id.length() > 0;    // Pré-condição sequencial
      requires sq != null;                       // Pré-condição sequencial
   { /*Algoritmo do Subprograma*/ }
   ...
}
```

4.6 Gestão de Falhas Concorrentes

Durante a execução de um programa podem ocorrer eventos indesejados, como sejam falhas no sistema onde o programa está a ser executado ou no próprio programa por causa de erros de desenvolvimento.

As excepções são utilizadas como forma de se sinalizar a ocorrência de falhas. Esta sinalização funciona como um mecanismo de comunicação interna e causa a interrupção imediata da execução normal dos programas de modo a que o código especificado para o tratamento das excepções seja executado.

Num programa concorrente, que é geralmente composto por vários subprogramas, quando uma excepção chega ao método de topo de um subprograma, apenas esse subprograma deverá terminar (indicando a ocorrência de uma falha) e não todo o programa. Tal deverá acontecer, uma vez que os restantes subprogramas poderão estar em condições de continuar a execução das suas tarefas.

Assim, num programa desenvolvido num contexto orientado por objectos, uma excepção pode deixar objectos em estados instáveis e levar à terminação da execução de um subprograma.

A nossa linguagem permite a definição de diferentes políticas de terminação de subprogramas – `debug`, `propagate`, `ignore`, e `ignore-debug` (Bloco de Código 4.5). Dependendo da política aplicada, diferentes cenários ocorrem quando um subprograma falha com uma excepcção:

- *Debug*: uma falha causa um *stack dump* de todos os subprogramas, assim como a terminação de todo o programa (política de terminação por omissão).

- *Propagate*: a falha é propagada até ao subprograma onde aquele que falhou foi criado, através da invocação do seu método `interrupt`.

- *Ignore*: a falha é ignorada pelos restantes subprogramas.
- *Ignore-debug*: a *stack* é impressa na consola mas a falha é ignorada em todo o programa.

O Bloco de Código 4.5 ilustra a forma como a política de terminação para um subprograma é feita. De acordo com o apresentado, a ocorrência de uma falha é reportada ao subprograma hierárquico superior.

Bloco de Código 4.5: CCJava - Política de Terminação (1)

```
...
SQThread c = @propagate new Consumer("id", sq);
...
```

4.7 Síntese do Capítulo

Neste capítulo começámos por apresentar o conceito de objectos partilhados no contexto da nossa linguagem e a forma como a linguagem Concurrent Contract-Java os especifica.

As três secções seguintes corresponderam à especificação da linguagem relativamente aos três aspectos de sincronização que são necessários assegurar para que os subprogramas comuniquem. Estes aspectos são especificados através de mecanismos seguros, abstractos e automáticos. São seguros porque a nossa linguagem nunca delega no programador a responsabilidade de garantir a correcção de cada um desses mecanismos. O facto de não ser imposto um esquema específico que assegure o sincronismo interno dos objectos partilhados torna a nossa linguagem abstracta. Sendo possível assegurar a geração correcta dos esquemas de sincronismo pelo compilador, a linguagem será automática relativamente a esse aspecto (é o caso da linguagem CCJava).

Por último, abordámos a forma como são especificadas a criação de um subprograma na nossa linguagem e as políticas de terminação associadas à gestão de falhas nos subprogramas.

No capítulo seguinte iremos explicar como é que os mecanismos da linguagem são (automaticamente) implementados, ou seja, apresentaremos o sistema de compilação da nossa linguagem.

Capítulo 5

Concurrent Contract-Java : Compilador

Depois de apresentada a linguagem CCJava, este capítulo surge com o objectivo de apresentar o sistema de compilação desta linguagem.

A linguagem CCJava é uma extensão da linguagem Java e como tal, de modo a minimizar o trabalho de geração de código pelo compilador, optámos por gerar código Java na compilação de código CCJava, recorrendo a uma biblioteca para concorrência [22].

Encarada a necessidade de construir um compilador, o passo seguinte do nosso trabalho concentrou-se na escolha de um sistema de geração de compiladores, tendo a nossa opção recaído sobre o sistema `ANTLR-v4` (*ANother Tool for Language Recognition*) [23]. As vantagens de uma tal escolha são várias. Desde logo, o facto de ser um sistema de geração de compiladores bastante legível, bem como o facto de ser um sistema de fácil depuração de erros. Adicionalmente, o sistema ANTLR apresenta um suporte embutido para a geração de árvores de sintaxe abstractas. É um sistema desenvolvido em Java, evitando a programação noutra linguagem. Servimo-nos da documentação desta implementação sempre que assim se fez necessário [23].

Uma vez definido o sistema de geração de compiladores a utilizar, escolhemos uma gramática válida para `Java-v7`, na qual foram adicionados os novos mecanismos da linguagem [24].

O compilador contempla três fases:

- Análise Sintática: é verificado para cada ficheiro o cumprimento das regras da gramática;
- Análise Semântica: é garantido para cada ficheiro que o código a produzir faz sentido;
- Geração de Código: é gerado para cada ficheiro o código Java final.

O Anexo B contém o código completo gerado pelo compilador para o problemas dos produtores-consumidores, inicialmente apresentado no Capítulo 4.

5.1 Aspectos Sequenciais da Programação por Contrato

Nas próximas secções iremos apresentar o código gerado pelo compilador para os aspectos sequenciais relacionados com programação por contrato em CCJava.

5.1.1 Contrato de um Método

O tratamento dos contratos dos métodos e dos construtores é em tudo semelhante, excepto no local de verificação dos contratos.

No caso dos construtores, as condições são verificadas dentro do próprio construtor, sem a invocação de funções auxiliares, dado não existirem relações de herança entre construtores – Bloco de Código 5.1.

Bloco de Código 5.1: CCJava - Processamento do Contrato de um Método Construtor

```
public class ArrayedQueue<T>{
   public ArrayedQueue (int maxSize)
      requires maxSize >= 0 ;
      { /*CorpoDoMétodo*/ }
   ...
}
```

```
public class ArrayedQueue<T>{
   public ArrayedQueue (int maxSize){
      if(!(maxSize>=0)) throw new PreConditionFailure();
      /*CorpoDoMétodo*/
   }
   ...
}
```

No caso dos métodos implementados numa classe ("normal"ou abstracta), as pré e pós-condições dão origem a duas novas funções (*void*) responsáveis pelas suas verificações – `methodName_Requires()` e `methodName_Ensures()`, respectivamente. A invocação destas funções é feita dentro do método ao qual o contrato se aplica. Este é o processamento dos métodos não abstractos e que correspondem a comandos – Bloco de Código 5.2.

Bloco de Código 5.2: CCJava - Processamento do Contrato de um Método (Comando)

```
public class ArrayedQueue<T>{
   public void in (T e)
      requires !isFull();
   { /*CorpoDoMétodo*/ }
      ensures  !isEmpty() && peekIn()==e;
}
```

```
public class ArrayedQueue<T>{
   public void in (T e){
      in_Requires();
      /*CorpoDoMétodo*/
      in_Ensures();
   }
   protected void in_Requires(T e){
      if(isFull()) throw new PreConditionFailure();
   }
   protected void in_Ensures(T e){
      if(!(!isEmpty()&&peekIn()==e)) throw new PostConditionFailure();
   }
}
```

Quando o método está implementado numa classe ("normal"ou abstracta) e é uma consulta/consulta pura, os contratos são também processados da forma que já enunciámos. Contudo, de modo a não alterar o código já produzido, este método é incorporado noutro - `methodName_Wrap()`. Desta forma, garante-se que o código original se mantém inalterado – Bloco de Código 5.3.

Bloco de Código 5.3: CCJava - Processamento do Contrato de um Método (Consulta/Consulta pura)

```
public class ArrayedQueue<T>{
   public T peekOut()
      requires !isEmpty();
   { /*CorpoDoMétodo*/ }
      ensures  !isFull();
}
```

```
public class ArrayedQueue<T>{
   private  T peekOut_Wrap()
   { /*CorpoDoMétodo*/ }

   public T peekOut(){
      peekOut_Requires();
      T result = peekOut_Wrap();
      peekOut_Ensures(result);
      return result;
   }
   protected void peekOut_Requires()        { ... }
   protected void peekOut_Ensures(T result){ ... }
}
```

Quando o método (comando, consulta ou consulta pura) é abstracto e o seu contrato está definido, as duas novas funções são criadas na mesma classe onde o método e o seu contrato foram especificados – Bloco de Código 5.4.

Bloco de Código 5.4: CCJava - Processamento do Contrato de um Método Abstracto

```
public abstract class Queue{
   abstract public void in(T e)
      requires !isFull();
      ensures  !isEmpty() && peekIn() == e;

   abstract public T peekOut()
      requires !isEmpty();
      ensures  !isFull();
}
```

```
public abstract class Queue{
   abstract public void in(T e);
   protected void in_Requires(T e){ ... }
   protected void in_Ensures(T e) { ... }

   abstract public T peekOut();
   protected void peekOut_Requires()       { ... }
   protected void peekOut_Ensures(T result){ ... }

}

public class ArrayedQueue<T> extends Queue{
   public void in (T e){
      in_Requires();
      /*CorpoDoMétodo*/
      in_Ensures();
   }

   private  T peekOut_Wrap() { /*CorpoDoMétodo*/ }
   public T peekOut(){
      peekOut_Requires();
      T result = peekOut_Wrap();
      peekOut_Ensures(result);
      return result;
   }
}
```

No caso de métodos definidos em interfaces, as pré-condições e pós-condições não originam, nessa interface, novas funções, uma vez que, por definição, os métodos de uma interface só podem ser especificados e não implementados[1]. As pré-condições e pós-condições dão origem a novas funções nas classes que implementam tais interfaces – Bloco de Código 5.5.

Bloco de Código 5.5: CCJava - Processamento do Contrato de um Método em Interface

```
public interface Queue{
   public void in(T e)
      requires !isFull();
      ensures  !isEmpty() && peekIn() == e;

   public T peekOut()
      requires !isEmpty();
      ensures  !isFull();
}
```

```
public interface Queue{
   public void in(T e);
   public T peekOut();
}
```

[1] Esta afirmação é verdadeira pelo menos até à versão 7 da linguagem Java

```
public class ArrayedQueue<T> implements Queue<T>{
   public void in (T e){
      in_Requires();
      /*CorpoDoMétodo*/
      in_Ensures();
   }
   protected void in_Requires(T e){ ... }
   protected void in_Ensures(T e) { ... }

   private  T peekOut_Wrap()
   { /*CorpoDoMétodo*/ }

   public T peekOut(){
      peekOut_Requires();
      T result = peekOut_Wrap();
      peekOut_Ensures(result);
      return result;
   }
   protected void peekOut_Requires()       { ... }
   protected void peekOut_Ensures(T result){ ... }
}
```

5.1.2 Contrato de uma Classe/Interface

A definição de um invariante corresponde a uma expressão booleana, que de acordo com o seu nível de visibilidade, pode ser verificado numa de três funções distintas – `publicInvariant()`, `protectedInvariant()` ou `privateInvariant()`. A função `invariant()` invoca cada um destes métodos – Bloco de Código 5.6.

Bloco de Código 5.6: CCJava - Processamento do Contrato de uma Classe

```
public class ArrayedQueue<T>{
   public  invariant (isEmpty() && size()==0) || (!isEmpty() && size()>0);
   private invariant this.size >= 0;
}
```

```
public class ArrayedQueue<T>{
   public publicInvaria(){
      if(!((isEmpty()&&size()==0)||(!isEmpty()&&size()>0))) throw InvariantFailure();
   }

   private privateInvariant(){ if(!(this.size>=0)) throw new InvariantFailure(); }

   protected invariant(){
      publicInvarint();
      privateInvariant();
   }
}
```

Quando os invariantes são definidos numa classe abstracta, o processamento é exactamente igual ao apresentado.

Quando os invariantes são definidos numa interface, o compilador assimila essa informação e garante que as funções são criadas na(s) classe(s) que implementa(m) a interface.

5.1.3 Herança de Contratos

Tanto os contratos dos serviços públicos com os invariantes podem ser herdados. Para cada um destes dois cenários existem três relações possíveis de herança: classe-classe, interface-classe ou interface-interface.

No primeiro cenário geral, quando a herança é entre classes, ao criar-se as funções que verificam as pré e pós-condições, evita-se a reescrita das mesmas na classe descendente (basta invocar os métodos definidos na classe ascendente). No entanto, quando um método herda um contrato mas para além desse contrato define novas cláusulas, as novas funções são também criadas na classe descendente e correspondem a uma invocação `super()` seguida da verificação das novas cláusulas. Quando a relação de herança é entre uma classe e uma interface, o processamento origina o mesmo código que o apresentado no Bloco de Código 5.5. Quando a relação de herança é entre interfaces, o sistema de compilação assimila consecutivamente os contratos e quando as classes as implementam, as novas funções são criadas nessas classes.

No segundo caso, quando a herança é entre classes, a classe descendente só herda os invariantes `public` e `protected`. Também neste caso, ao criar-se as funções que verificam esses invariantes, evita-se a reescrita das mesmas na classe descendente e quando um classe herda invariantes mas para além desses invariantes define novas cláusulas, as novas funções são também criadas na classe descendente e correspondem a uma invocação `super()` seguida da verificação das novas cláusulas. Quando a relação de herança é entre uma classe e uma interface, o compilador assimila os invariantes (que deverão ser públicos) e cria as novas funções na classe. Quando a relação de herança é entre interfaces, o sistema de compilação assimila consecutivamente os invariantes e quando as classes as implementam, as novas funções são criadas nessas classes.

5.1.4 Mecanismo Disciplinado de Excepções

Pelos exemplos de código já apresentados, é evidente a existência de um conjunto separado de excepções associadas às falhas de contrato – Figura5.1 – disponível na biblioteca nativa de CCJava (`ccjava.DbC`).

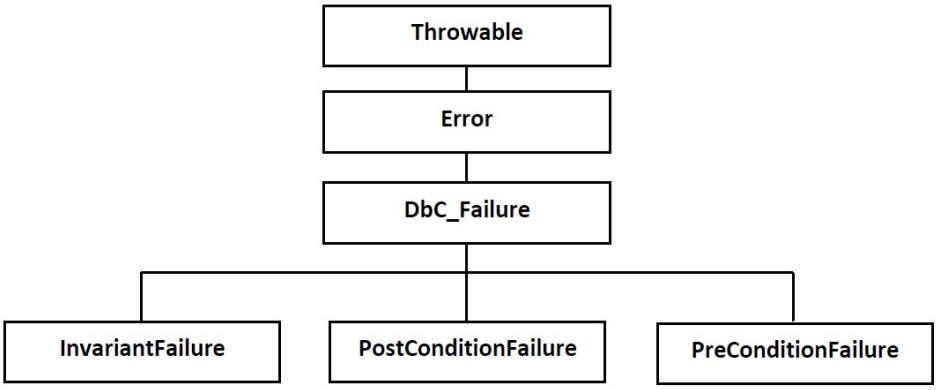

Figura 5.1: CCJava - Conjunto de Excepções DbC_Failure

O mecanismo disciplinado de excepções (secção 2.3), sendo um dos requisitos para o desenvolvimento de uma linguagem com mecanismos de programação por contrato, é especificado pela nossa linguagem mas a versão actual do nosso compilador não o implementa.

Relativamente ao mecanismo `try/catch` da linguagem Java, a análise semântica garante que uma excepção associada a uma falha de contrato nunca é utilizada neste mecanismo. Assim, quando é lançada uma excepção *Throwable* ou *Error* o compilador adiciona um bloco `catch` para apanhar e lançar uma excepção de falha de contrato.

Bloco de Código 5.7: CCJava - Excepções Throwable e Error

```
public void out(){
   try{ ... }
   catch(Throwable excep){ ... }
}
```

```
public void out(){
   try{ ... }
   catch(DbC_Failure dbcFailure){ throw dbcFailure; }
   catch(Throwable excep){ ... }
}
```

5.2 Objectos Partilhados

Na linguagem CCJava podemos definir classes partilhadas (classes que criam sempre objectos partilhadas) e entidades partilhadas (classes que podem criar objectos sequenciais e partilhados) (secção 4.1). No entanto, a versão actual do nosso compilador implementa apenas o segundo mecanismo especificado.

5.3 Sincronização Interna

A sincronização interna de um objecto é abstracta (secção 4.2.2), tem como requisito mínimo a atomicidade na execução dos serviços públicos. Quer isto dizer que a sincronização interna pode ser feita recorrendo a diferentes esquemas de sincronismo. Dos esquemas possíveis, o nosso compilador implementa dois deles: monitores e exclusão entre leitores-escritor.

5.3.1 Monitores

O esquema de Monitores é o esquema mais simples de implementar. Contudo, a sua limitação reside no facto de apenas um subprograma poder aceder em cada instante ao objecto (Figura 5.2 - retirada de [20]). Este esquema requer a imposição de um regime de exclusão mútua em todos os acessos a um objecto, o que se traduz numa menor disponibilidade concorrente do mesmo.

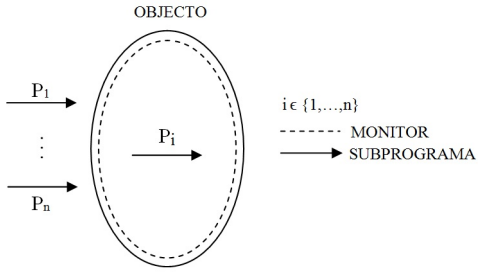

Figura 5.2: Esquema de Sincronismo Monitor

As condições que este esquema impõe sobre os sistemas de compilação de modo a ser exequível são mínimas. Depois de identificado o objecto partilhado, apenas é necessário identificar todos os métodos públicos presentes numa classe, devendo estes ser protegidos de acordo com o código de sincronismo característico de um monitor. Os restantes métodos, não sendo públicos, não são acedidos pelos clientes do objecto, razão pela qual apenas os públicos devem estar protegidos.

Para implementar o esquema de monitores é criada uma nova classe (concorrente) que utiliza a classe original (sequencial). Desta forma, consegue-se preservar a interface da classe sequencial e garantir que não há interferências entre as classes sequenciais e partilhadas. A

nova classe, identificada através do prefixo *Shared* seguido do nome original, contem o código de sincronismo associado a um monitor em todos os métodos públicos. Em cada instante só um subprograma é que tem acesso ao objecto – Bloco de Código 5.8.

Bloco de Código 5.8: CCJava - Sincronização Interna (Monitor)

```
public class SharedArrayedQueue<T>
{
   protected final ArrayedQueue<T> seq_ArrayedQueue;          // Classe Sequencial
   protected final Mutex intra; // Lock de Exclusão Mútua p/ Sincronismo Interno

   public SharedArrayedQueue(){
      seq_ArrayedQueue = new ArrayedQueue();
      intra = new Mutex();
   }

   public void in(T e) {
      intra.lock();
      try {
         ... // Tratar pré-condição (sincronização condicional)
         seq_ArrayedQueue.in(e);
         ... // Notificar alteração do estado do objecto
      } finally { intra.unlock(); }
   }

   public T peek() {
      T result;
      intra.lock();
      try {
         ... // Tratar da pré-condição (sincronização condicional)
         result = seq_ArrayedQueue.peek();
      } finally { intra.unlock(); }
      return result;
   }
   ...
}
```

5.3.2 Exclusão entre Leitores-Escritor

O mecanismo anteriormente apresentado, apesar de ser a forma mais simples para a programação da sincronização de objectos partilhados, impõe uma restrição excessiva (exclusão mútua na execução dos métodos públicos). Alguns subprogramas poderão querer consultar um objecto para extrair informação em paralelo com outras operações, sem qualquer alteração do mesmo, não sendo isso possível com um esquema do tipo monitor. Nestas situações, garantir a exclusão mútua na execução de um serviço que possa modificar o estado do objecto (ou do sistema) é suficiente, permitindo-se o processamento concorrente dos restantes serviços.

Podemos pois denominar este novo esquema de sincronismo por Exclusão entre Leitores-Escritor. Um leitor será um subprograma que executa uma consulta pura. Todos os restantes subprogramas serão escritores. Nesta relação, um escritor exclui todos os subprogramas, mas

vários leitores podem concorrentemente aceder ao objecto (Figura 5.3 - retirada de [20]).

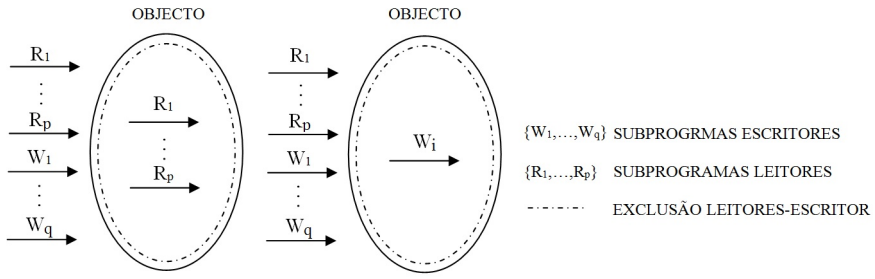

Figura 5.3: Esquema de Sincronismo Exclusão Leitores-Escritor

Assim, este esquema afigura-se igualmente uma opção válida para a programação do sincronismo, existindo um menor número de bloqueios no acesso a objectos concorrentes.

Para que um esquema deste tipo seja automaticamente exequível é necessário, para além da identificação dos métodos públicos, extrair das classes a sincronizar informação que permita ao compilador distinguir comandos e consultas de consultas puras.

A implementação do esquema agora em análise é semelhante ao esquema dos Monitores no sentido em que também é criada uma nova classe que utiliza a classe original. No entanto, só é exequível caso o compilador separe as consultas puras dos restantes métodos (comandos e consultas).

Detecção de Métodos Sem Efeitos Colaterais

Idealmente, para avaliar a pureza de um método, o compilador deverá analisar o algoritmo de cada serviço, assim como o algoritmo de todos os serviços invocados por esses serviços, sejam estes do próprio objecto ou de outros. Um serviço será puro se não fizer atribuição de valores a atributos e se todos os serviços que invoca também forem puros.

Para ser possível identificar os métodos sem efeitos colaterais, seria boa prática que o compilador criasse um grafo de dependências em que os nós corresponderiam a todos os serviços existentes em todo o programa e as ligações corresponderiam às invocações entre serviços [20]. No entanto, a construção desse grafo não foi por nós implementada. A nossa estratégia foi a seguinte: por omissão, o compilador assume que todos os métodos são comandos. No entanto, métodos *non-void* são consultas e algumas podem não ter efeitos colaterais. CCJava define a anotação `@pure` para que o programdor sugira quais os métodos que correspondem a

consultas puras – Bloco de Código 5.9. O compilador usa esta informação para garantir uma implementação segura dos esquemas de sincronismo – Bloco de Código 5.10.

Bloco de Código 5.9: CCJava - Anotação `@pure`

```
public class ArrayedQueue<T>
{
   // Consulta com efeitos colaterais (devolve e remove o primeiro elemento)
   public T peekOut()
      requires !isEmpty();
   {/*CorpoDoMétodo*/}
      ensures  !isFull();

   @pure // Consulta pura (devolve o primeiro elemento)
   public T peek()
      requires !isEmpty();
   {/*CorpoDoMétodo*/}
}
```

Bloco de Código 5.10: CCJava - Sincronização Interna (Exclusão entre Leitores-Escritor)

```
public class SharedArrayedQueue<T> {
   protected final ArrayedQueue<T> seq_ArrayedQueue;        // Classe Sequencial
   protected final RWEx intra;  // Lock Leitores-Escritor p/ Sincronismo Interno

   public SharedArrayedQueue() {
      seq_ArrayedQueue = new ArrayedQueue();
      intra = new RWEx();
   }

   // Operação de Escrita
   protected void in(T e) {
      intra.lockWriter();
      try {
         ... // Tratar pré-condição
         seq_ArrayedQueue.in(e);
         ... // Notificar alteração do estado do objecto
      } finally { intra.unlockWriter(); }
   }

   // Operação de Leitura
   public T peek() {
      T result;
      intra.lockReader();
      try {
         ... //  Tratar pré-condição
         result = seq_ArrayedQueue.peek();
      } finally { intra.unlockReader(); }
      return result;
   }
   ...
}
```

Nos Blocos de Código 5.8 e 5.10 são utilizadas duas classes – `Mutex` e `RWEx` – da biblioteca auxiliar de concorrência, que é incorporada na biblioteca nativa de CCJava (`ccjava.Concurrency`). A primeira classe representa um *lock* de exclusão mútua e os seus serviços principais são `lock` e `unlock`. A operação `lock` é invocada sempre que seja executado um serviço público de um objecto partilhado e deixa os restantes subprogramas a aguardar pela terminação da

execução do serviço e consequente invocação do serviço `unlock`. A segunda classe representa um *lock* leitores-escritor e os seus serviços principais são `lockWriter/lockReader` e `unlockWriter/unlockReader`. As operações `lockWriter/unlockWriter` têm o mesmo efeito que as operações da classe Mutex e são aplicadas em todos os comandos e em todas as consultas. As consultas puras, ao não alterarem o estado de um objecto partilhado, permitem acessos concorrentes por vários subprogramas, pelo que nestes casos são invocadas as operações `lockReader/unlockReader`.

5.4 Sincronização Condicional

O mecanismo de sincronização condicional é utilizado sempre que o acesso a um objecto depende de uma condição de acesso concorrente (secção 4.3).

No modelo de comunicação por partilha de objectos, a implementação do sincronismo condicional pode ser equivalente à utilizada nos monitores, ou seja, pode corresponder à implementação de mecanismos que utilizem variáveis de condição [25]. É possível aplicar sobre estas variáveis três operações distintas: espera (*wait*), que coloca um subprograma, que requer o acesso a um serviço, na fila de espera associada à variável de condição; sinalização (*notify*), que faz com uns dos subprogramas seja retirado da fila de espera, sendo-lhe atribuído o acesso ao monitor e sinalização total (*notifyAll*), que executa a operação anterior, agora para todos os subprogramas e um de cada vez.

Nestes termos, associando uma variável de condição a cada objecto, implementando todas as acções de espera condicional como acções de espera nessa variável e colocando acções de sinalização para todos os subprogramas no término de todas as rotinas públicas do objecto, é possível implementar este esquema de sincronismo de forma segura e automática. Desta forma, quando os subprogramas ganham acesso exclusivo ao objecto verificam se a condição de espera é verdadeira. Sendo a resposta afirmativa, a rotina é executada. Caso contrário, os subprogramas voltam a colocar-se em espera. Este algoritmo pode ser melhorado caso o compilador diferencie comandos, consultas e consultas puras. Se assim for possível, apenas no final da execução de comandos e consultas impuras será necessário notificar a variável de condição, uma vez que só esses poderão alterar o estado do objecto. Esta é a nossa implementação actual deste aspecto de sincronização – Bloco de Código 5.11.

Bloco de Código 5.11: CCJava - Sincronização Condicional

```
public class SharedArrayedQueue<T>
{
   protected final ArrayedQueue<T> seq_ArrayedQueue;          // Classe Sequencial
   protected final Mutex intra;                               // RWEx intra;
   protected GroupMutexComposedCV cv;  // Variável de Condição (Sincr. Condicional)

   public SharedArrayedQueue() {
      seq_ArrayedQueue = new ArrayedQueue();
      intra = new Mutex();                 // new RWEx()
      cv = ...                             // Instaciação da variável de condição
   }

   protected void in(T e) {
      intra.lock()                                        // intra.lockWriter();
      try {
         while (seq_ArrayedQueue.isFull()) { cv.await(); }     // Pré-condição
         seq_ArrayedQueue.in(e);
         cv.broadcast();              // Notificar alteração do estado do objecto
      } finally { intra.unlock();}                        // intra.unlockWriter()
   }

   protected T peek() {
      T result;
      intra.lock()                                      // intra.lockReader();
      try {
         while (seq_ArrayedQueue.isEmpty()) { cv.await(); }  // Pré-condição
         result = seq_ArrayedQueue.peek();
      } finally { intra.unlock();}                     // intra.unlockReader()
      return result;
   }
   ...
}
```

Neste bloco de código identificamos uma nova classe – `GroupMutexComposedCV` –, utilizada para criar uma variável de condição que está associada a todas as operações de espera. Os seus serviços principais são `await()` e `broadcast()` e devem ser utilizados quando um objecto não está no estado desejável e para notificar a alteração do estado do objecto, respectivamente.

5.5 Sincronização Externa

A sincronização externa é utilizada com o propósito de garantir o acesso exclusivo ao objecto para a realização de uma sequência de operações sobre o mesmo (secção 4.4). A realização consecutiva destas operações é a razão pela qual a necessidade da existência deste tipo de sincronismo é externa ao objecto. É pois necessário recorrer à utilização de algoritmos de reserva exclusiva de objectos.

Apesar de ser desejável uma implementação da sincronização externa associada ao módulo da classe, a necessidade de invocação deste tipo de esquema ocorre do lado do cliente. O esquema de sincronismo interno não pode por isso ser utilizado para o mesmo propósito.

Ainda assim, é possível separar de forma automática as sincronizações interna e externa, implementando um esquema misto de sincronismo.

No modelo de partilha de objectos, a implementação automática da sincronização externa é conseguida através de um esquema de exclusão mútua. Uma vez que os esquemas de sincronismo interno e externo devem ser diferentes, uma solução para garantir a integração automática de ambos os esquemas de sincronismo é baseada num esquema misto de sincronismo por exclusão mútua (Figura 5.4 - retirada de [20]).

A implementação prática desta ideia passa pela definição de dois grupos de exclusão mútua de grupos: o sincronismo interno pertence a um grupo e o outro é reservado para o sincronismo externo – Bloco de Código 5.12. Assim se garante que todas as comunicações são seguras.

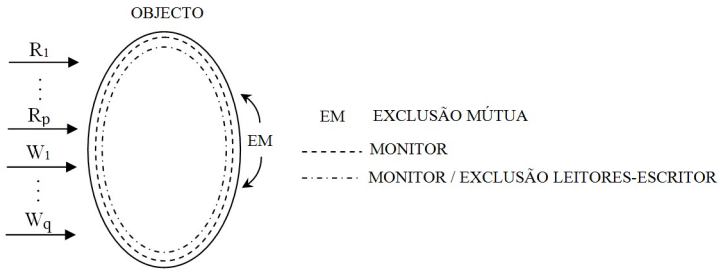

Figura 5.4: Esquema Misto de Sincronismo para Reserva de Objectos

Bloco de Código 5.12: CCJava - Sincronização Externa (1)

```
public class SharedArrayedQueue<T>
{
   protected final GroupMutex gmutex;          // Esq. Exc. Mut. de Grupos
   protected final Mutex intra; //RWEx intra   // Sincronização Interna
   protected final Mutex inter;                // Sincronização Externa
   protected GroupMutexComposedCV cv;          // Sincronização Condicional
   protected final ArrayedQueue<T> seq_ArrayedQueue;  // Classe Sequencial

   public SharedArrayedQueue() {
      seq_ArrayedQueue = new ArrayedQueue();
      intra     = new Mutex();                          // new RWEx()
      inter     = new Mutex();
      gmutex = new GroupMutex(2, GroupMutex.Priority.GROUP_NUMBER_INCREASING);
      Sync[] list = new Sync[] {inter, intra};
      cv = gmutex.newCV(list);
   }

   public boolean isGrabbedByMe() {
      return gmutex.lockIsMine() && gmutex.activeGroup() == 2 && inter.lockIsMine();
   }

   public void grab() {
      if (isGrabbedByMe()) throw new PreConditionFailure();
```

```
        gmutex.lock(2);  // Grupo para sincronização externa
        inter.lock();    // Sincronização Externa
        if (!isGrabbedByMe()) throw new PostConditionFailure();
    }

    public void release() {
        if (!isGrabbedByMe()) throw new PreConditionFailure();
        gmutex.unlock(2);
        inter.unlock();
        if (isGrabbedByMe()) throw new PostConditionFailure();
    }
    ...
}
```

Neste bloco de código identificamos, por último, a utilização de duas novas classes: `GroupMutex` e `Sync`. A primeira classe é utilizada para implementar o esquema de exclusão mútua de grupos (em cada instante, apenas um grupo está activo). No caso apresentado, o primeiro grupo tem a prioridade mais baixa. A segunda classe é utilizada para criar uma lista que contém os esquemas utilizados internamente. A variável de condição é criada através da invocação do serviço `newCV()` da classe `GroupMutex`, que utiliza a lista previamente criada como argumento.

Importa agora apresentar o código gerado que permite o funcionamento independente dos esquemas de sincronismo interno e externo – Bloco de Código 5.13.

Bloco de Código 5.13: CCJava - Sincronização Externa (2)

```
public class SharedArrayedQueue<T>
{
    ... // Atributos

    public SharedArrayedQueue() { ... }

    public void in(T e) {
        boolean precondition = true;
        assert objectInStableState : "Object in unstable state!";
        try {
            if (isGrabbedByMe()) {
                precondition = !seq_ArrayedQueue.isFull();
                seq_ArrayedQueue.in(e);
                cv.broadcast();
            } else { inIntra(e); }
        } catch (DbC_Failure dbc) {if (precondition) objectInStableState = false;}
    }


    protected void inIntra(T e) {
        gmutex.lock(1);
        try {
            intra.lock();       // intra.lockWriter();
            try {
                while (seq_ArrayedQueue.isFull()) {cv.await();}
                seq_ArrayedQueue.in(e);
                cv.broadcast();
            } finally { intra.unlock(); /*intra.unlockWriter();*/ }
        } finally { gmutex.unlock(1); }
    }
```

```
    public T peek() {
        assert objectInStableState : "Object is unstable";
        T result;
        if (isGrabbedByMe()) {result = seq_ArrayedQueue.peek();}
        else { peekIntra(); }
    }

    protected T peekIntra() {
        T result;
        gmutex.lock(1);
        try {
            intra.lock();       // intra.lockReader();
            try {
                while (seq_ArrayedQueue.isEmpty()) { cv.await(); }
                result = seq_ArrayedQueue.peekOut();
            } finally { intra.unlock(); /*intra.unlockReader();*/ }
        } finally { gmutex.unlock(1); }
        return result;
    }
    ...
}
```

No código final gerado pelo compilador relativamente a este aspecto de sincronização identificamos dois padrões de código – `try/catch(DbC_Failure dbc)` e `if(isGrabbedByMe()...)`. O primeiro padrão é gerado para todos os serviços públicos que sejam comandos ou consultas (o lançamento de uma excepção contratual significa, quando a pré-condição for verdadeira ou não existir, que o objecto se encontra num estado instável). As consultas puras apresentam apenas o segundo padrão (que também se aplica nos comandos e consultas), sendo este utilizado para distinguir a invocação de serviços quando o objecto é reservado externamente ou internamente.

Consideremos agora, por exemplo, que no problema dos produtores-consumidores inicialmente apresentado, os produtores introduzem mais do que um elemento na fila por cada execução – Bloco de Código 5.14.

Bloco de Código 5.14: CCJava - Sincronização Externa (3)

```
public void therun(String id, shared ArrayedQueue<Integer> sq){
    int n = (int)(Math.random()*10);
    for(i = 0; !sq.isFull() && i < n; i++){
        sq.in(++count);
        System.out.println("["+id+"]:value"+count+" produced {"+this+"}");
    }
}
```

Analisando este bloco de código, podemos identificar a presença de uma instrução repetitiva, o que significa que essa instrução é um ponto de sincronização externa. De acordo com a forma como os pontos de sincronização externa devem ser tratados, o Bloco de Código 5.15 representa o código que o compilador deverá gerar do lado do cliente.

O compilador é capaz de separar, de forma não ambígua, as condições sequenciais das que possivelmente poderão ser concorrentes, utilizando apenas o sistema de tipos da linguagem. Uma expressão que utilize uma entidade partilhada tem todo o potencial para se tornar numa expressão concorrente. No entanto, é preciso ter em conta o contexto de execução da expressão. No exemplo 5.15, a pré-condição do método `sq.in()` comporta-se como um contrato sequencial, uma vez que o teste da condição `sq.isFull()` corresponde a um ponto de sincronização externa.

Bloco de Código 5.15: CCJava - Sincronização Externa (4)

```
public void therun(String id, SharedArrayedQueue<Integer> sq){
   sq.grab();
   ...
   sq.release();
   finished = ( (int) (Math.random()*50) == 0);
}
```

O nosso compilador implementa este sincronismo tal como o descrevemos. No entanto, a construção dos blocos de sincronização externa do lado do cliente não é feita pelo nosso compilador.

5.6 Criação de Subprogramas

Na nossa linguagem, o ciclo de vida de um subprograma deixa de estar associado ao método `run` e passa a ser explicitado no construtor do objecto (secção 5.6). Em vez da classe nativa `Thread` do Java, utilizámos a classe `CThread` da biblioteca para concorrência (nesta classe o método `run` é substituído pelo método `arun` – Bloco de Código 5.16.

Bloco de Código 5.16: CCJava - Criação de Subprogramas (2)

```
public abstract remote class SQThread{
   public SQThread(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;    // Pré-condição sequencial
      requires sq != null;                       // Pré-condição sequencial
   {
      while(true){
         therun(id, sq);
         pause((int)(Math.random()*maxPause));
      }
   }
   ...
}
```

```
public abstract class SQThread extends CThread
{
   protected String id;                          // Automaticamente Gerado
   protected SharedArrayedQueue<Integer> sq;     // Automaticamente Gerado
```

```
    public SQThread(String id, SharedArrayedQueue<Integer> sq){
        // Automaticamente Gerado
        if (!(id != null && id.length() > 0)) throw new PreConditionFailure();
        if (!(sq != null))                    throw new PreConditionFailure();
        this.id = id;
        this.sq = sq;
    }

    // Automaticamente Gerado
    public void arun() {
        while (true) {
            therun(id, sq);
            pause((int) (Math.random() * 100.0));
        }
    }
    ...
}
```

5.7 Gestão de Falhas Concorrentes

O programador pode especificar diferentes políticas de terminação que actuam de forma distinta no programa quando é lançada uma excepção. A nossa linguagem utiliza o sistema de anotações para as representar (secção 4.6) e o sistema de compilação é capaz de gerar automaticamente o código responsável pela definição de tal política (Bloco de Código 5.17). Se não for definida nenhuma política de terminação, a própria classe `CThread` estabelece uma por omissão.

Bloco de Código 5.17: CCJava - Política de Terminação (2)

```
Producer[] producers = @propagate    new Producer[numProducers]
Consumer[] consumers = @igone-debug new Consumer[numConsumers];
```

```
public class Producer extends SQThread{
    public Producer(String id, SharedArrayedQueue<Integer> sq){
        super(id, sq);
        start(TerminationPolicy.PROPAGATE);
    }
}

public class Consumer extends SQThread{
    public Consumer(String id, SharedArrayedQueue<Integer> sq){
        super(id, sq);
        start(TerminationPolicy.IGNORE_DEBUG);
    }
}
```

5.8 Utilização do Protótipo

A utilização do nosso compilador é feita da seguinte forma:

```
$ java -jar CCJava.jar <LIST_OF_CCJAVA_FILES>.
```

A extensão do ficheiro, que deverá estar organizado de forma hierárquica, tem de ser *.txt* e cada linha terá de conter o caminho de apenas um ficheiro para o compilador analisar.

O código fonte do nosso compilador está disponível em `https://code.ua.pt/projects/concurrent_contract-java`. No mesmo repositório é disponibilizada a biblioteca nativa de CCJava, que contém as sub-bibliotecas `ccjava.DbC` e `ccjava.Concurrency` e o ficheiro *jar* executável.

5.9 Síntese do Capitulo

Neste capítulo descrevemos o sistema de compilação da nossa linguagem, que é capaz de gerar de forma automática o código necessário para garantir a correcta e segura sincronização de objectos partilhados, tornando mais fácil e apelativo o desenvolvimento de programas concorrentes com aspectos de programação por contrato em linguagens orientadas por objectos.

Dos três aspectos de sincronização apresentados, merecem destaque as sincronizações interna e externa. Na sincronização interna, indicámos os esquemas que o nosso compilador é capaz de implementar, apresentando os desafios impostos por cada um desses esquemas e respectivas soluções. Na sincronização externa, destacámos o facto do sincronismo externo ser implementado apenas do lado do objecto, o que faz com que a geração de código não seja totalmente automática, uma vez que é preciso lidar com o lado do cliente.

Apresentados estes aspectos, ilustrámos o código gerado quando é criado um subprograma e quando é definida uma política de terminação de um subprograma para lidar com falhas concorrentes.

Por fim descrevemos o modo de utilização do nosso protótipo.

Capítulo 6

Conclusão

Neste trabalho quisemos especificar uma linguagem orientada por objectos concorrente. Começámos por sintetizar alguns paradigmas da programação sequencial e apresentámos também a linguagem Contract-Java, que especifica mecanismos de programação por contrato (Capítulo 2). O objectivo foi sempre integrar a programação concorrente em linguagens orientadas por objectos e não o contrário, sendo que o Capítulo 3 surge para se apresentar, com detalhe, este tipo de programação e os desafios por ela colocados, bem como a solução para os mesmos. No Capítulo 4 apresentámos a nossa linguagem protótipo – Concurrent Contract-Java – que especifica mecanismos de concorrência de alto nível simples e expressivos, capazes de permitir o desenvolvimento seguro de programas concorrentes. Por fim, no Capítulo 5, descrevemos o funcionamento do sistema de compilação da nossa linguagem, dando conta das soluções tomadas para os problemas emergentes. Quanto ao resultado final, esta linguagem destaca-se pela segurança estática, expressividade e abstracção dos mecanismos que propõe.

6.1 Resultados

A nossa linguagem apresenta as seguintes contribuições:

- Integração de mecanismos de concorrência e de DbC em linguagens orientadas por objectos (secção 2.3.3 e capítulo 4);
- Identificação inequívoca de objectos partilhados (secção 4.1);
- Sincronização abstracta de objectos partilhados (secção 4.2.1);
- Comportamento seguro dos contratos concorrentes (secção 4.3);

- Comportamento seguro das instruções de selecção concorrentes (secção 4.4);
- Desenvolvimento de um compilador para o nossa linguagem (5);
- Sincronização automática de objectos partilhados (secções 5.3, 5.4, 5.5);
- Simplificação do mecanismo para criação de subprogramas (secção 5.6).

6.2 Trabalho Futuro

Resta-nos perspectivar o trabalho futuro a elaborar de forma a tornar o nosso compilador ainda mais completo. De imediato, destacamos a necessidade do nosso compilador efectuar uma análise semântica mais aprofundada. Impõe-se também a criação de um sistema de detecção de serviços sem efeitos colaterais mais eficiente. O compilador deverá ainda implementar outras características da linguagem, como sejam a integração de um mecanismo disciplinado para excepções DbC e a ideia apresentada em [26], que propõe que o compilador deverá, de forma automática, gerar tanta informação para depuramento de erros quanto possível quando um contrato falha. Por último, o compilador deverá ser capaz de gerar os códigos relacionados com a sincronização externa do lado do cliente e a definição de classes partilhadas.

Podemos mesmo assim afirmar que as características do compilador que desenvolvemos são suficientes para que o mesmo sirva como prova de conceito de mecanismos de linguagens concorrentes de objectos partilhados, tais como os que descrevemos, de modo a desenvolver programas concorrentes seguros.

Apêndice A

Código Fonte

Bloco de Código A.1: Produtores-Consumidores : `ProducerConsumerProblem.ccjava`

```
public class ProducerConsumerProblem
{
   static int numProducers = 4;
   static int numConsumers = 4;

   public static void main(String[] args){
      shared ArrayedQueue<Integer> sq = @rwex new shared ArrayedQueue<Integer>(25);

      Producer[] producers = new Producer[numProducers];
      for(int i = 0; i < numProducers; i++)
        producers[i] = new Producer("producer #"+(i+1), sq);

      Consumer[] consumers = new Consumer[numConsumers];
      for(int i = 0; i < numConsumers; i++)
         consumers[i] = new Consumer("consumer #"+(i+1), sq);
   }
}
```

Bloco de Código A.2: Produtores-Consumidores : `SQThread.ccjava`

```
public abstract remote class SQThread
{
   public SQThread(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;
      requires sq != null;
   {
      while(true){
         therun(id, sq);
         sleep((int)(Math.random()*100.0));
      }
   }
   public abstract void therun(String id, shared ArrayedQueue<Integer> sq);
}
```

Bloco de Código A.3: Produtores-Consumidores : `Producer.ccjava` e `Consumer.ccjava`

```
public remote class Producer extends SQThread
{
   public Producer(String id, shared ArrayedQueue<Integer> sq)
      requires id != null && id.length() > 0;
      requires sq != null;
   { super(id, sq); }

   protected int count = 0;
```

```
    public void therun(String id, shared ArrayedQueue<Integer> sq){
        sq.in(++count);
        System.out.println("["+id+"]:unique value produced - "+count+"{"+this+"}");
    }
}

public remote class Consumer extends SQThread
{
    public Consumer(String id, shared ArrayedQueue<Integer> sq)
        requires id != null && id.length() > 0;
        requires sq != null;
    { super(id, sq); }

    public void therun(){
        System.out.println("["+id+"]:unique value consumed - "+sq.peekOut()+"{"+this+"}");
    }
}
```

Bloco de Código A.4: Produtores-Consumidores : `ArrayedQueue.ccjava`

```
public class ArrayedQueue<T>
{
    public invariant (isEmpty()&&size()== 0) || (!isEmpty()&&size()>0);
    protected static final int DEFAULT_SIZE = 100;
    protected boolean limited;
    protected T[] array = null;
    protected int pout; pin; size;

    public ArrayedQueue(){
        this(DEFAULT_SIZE);
        limited = false;
    }

    public ArrayedQueue(int maxSize)
        requires maxSize >= 0;
    {
        size = 0;
        array = (T[]) new Object[maxSize];
        pout = 0;
        pin = array.length-1;
        limited = true;
    }

    public void in(T e)
        requires !isFull();
    {
        if (!limited && size == array.length){
            T[] a = array;
            int pout = this.pout;
            array = (T[]) new Object[array.length + 1 + (array.length/10)];
            this.size = 0;
            this.pout = 0;
            this.pin = array.length-1;
            for(int i = 0; i < a.length; i++){
                in(a[pout]);
                pout = (pout+1)% a.length;
            }
        }
        pin = (pin+1)% array.length;
        array[pin] = e;
        size++;
    }
        ensures  !isEmpty() && peekIn() == e ;
```

```
    public void out()
        requires !isEmpty();
    {
        array[pout] = null;
        pout = (pout+1) % array.length;
        size--;
    }
        ensures  !isFull();

    @pure
    public T peek()
        requires !isEmpty();
    { return array[(pout) % array.length]; }

    public T peekOut()
        requires !isEmpty();
    {
        T result = array[(pout) % array.length];
        array[pout] = null;
        pout = (pout + 1) % array.length;
        size--;
        return result;
    }
        ensures  !isFull();

    @pure
    public T peekIn()
        requires !isEmpty();
    { return array[(pout+(size-1)) % array.length]; }

    public void clear(){
        int p = pout;
        for(int i = 0; i < size; i++){
            array[p] = null;
            p = (p + 1) % array.length;
        }
        size = 0;
    }
        ensures isEmpty();

    @pure
    public boolean isEmpty(){ return size == 0; }

    @pure
    public boolean isFull(){ return limited && (size == array.length); }

    @pure
    public int size(){ return size; }

    @pure
    public boolean isLimited(){ return limited; }

    @pure
    public int maxSize()
        requires isLimited();
    { return array.length; }
}
```

Apêndice B

Código Gerado

Bloco de Código B.1: Produtores-Consumidores : `ProducerConsumerProblem.java`

```
public class ProducerConsumerProblem{
   static int numProducers = 4;
   static int numConsumers = 4;

   public static void main(String[] args){
      SharedArrayedQueue<Integer> sq = new SharedArrayedQueue<Integer>(25);

      Producer[] producers = new Producer[numProducers];
      for(int i = 0; i < numProducers; i++)
         producers[i] = new Producer("producer #"+(i+1), sq);

      Consumer[] consumers = new Consumer[numConsumers];
      for(int i = 0; i < numConsumers; i++)
         consumers[i] = new Consumer("consumer #"+(i+1), sq);
   }
}
```

Bloco de Código B.2: Produtores-Consumidores : `SQThread.java`

```
import ccjava.DbC.*; import ccjava.Concurrency.*;

public abstract class SQThread extends CThread {
   protected String id;
   protected SharedArrayedQueue<Integer> sq;

   public SQThread(String id, SharedArrayedQueue<Integer> sq){
      if (!(id != null && id.length() > 0)) throw new PreConditionFailure();
      if (!(sq != null)) throw new PreConditionFailure();
      this.id = id;
      this.sq = sq;
   }

   public void arun() {
      while (true) {
         therun(id, sq);
         pause((int) (Math.random() * maxPause));
      }
   }
   public abstract void therun(String id, SharedArrayedQueue<Integer> sq);
}
```

Bloco de Código B.3: Produtores-Consumidores : `Producer.java` e `Consumer.java`

```
public class Producer extends SQThread {
   public Producer(String id, SharedArrayedQueue<Integer> sq){
      super(id, sq);
      start();
   }

   protected int count = 0;
   public void therun(String id, SharedArrayedQueue<Integer> sq) {
      ++count;
      sq.in(count);
      System.out.println("["+id+"]: unique value produced - "+count+" {"+this+"}");
   }
}

public class Consumer extends SQThread {
   public Consumer(String id, SharedArrayedQueue<Integer> sq){
      super(id, sq);
      start();
   }

   public void therun(String id, SharedArrayedQueue<Integer> sq) {
      System.out.println("["+id+"]: unique value consumed - "+sq.peekOut()+" {"+this+"}");
   }
}
```

Bloco de Código B.4: Produtores-Consumidores : `ArrayedQueue.java`

```
import ccjava.DbC.*;

public class ArrayedQueue<T> {
   protected boolean limited;
   protected T[] array = null;
   protected int pout;
   protected int pin;
   protected int size;
   protected static final int DEFAULT_SIZE = 100;

   public ArrayedQueue() {
      this(DEFAULT_SIZE);
      limited = false;
      invariant();
   }

   public ArrayedQueue(int maxSize){
      if (!(maxSize >= 0)) throw new PreConditionFailure();
      size = 0;
      array = (T[]) new Object[maxSize];
      pout = 0;
      pin = array.length - 1;
      limited = true;
      invariant();
   }

   public void in(T e){
      in_Requires(e);
      invariant();
      if (!limited && size == array.length) {
         T[] a = array;
         int pout = this.pout;
         array = (T[]) new Object[array.length + 1 + (array.length / 10)];
         this.size = 0;
         this.pout = 0;
         this.pin = array.length - 1;
         for (int i = 0; i < a.length; i++) {
            in(a[pout]);
```

```
            pout = (pout + 1) % a.length;
        }
    }
    pin = (pin + 1) % array.length;
    array[pin] = e;
    size++;
    invariant();
    in_Ensures(e);
}

protected void in_Requires(T e) {
    if (!(!isFull())) throw new PreConditionFailure();
}

protected void in_Ensures(T e) {
    if (!(!isEmpty() && peekIn() == e)) throw new PostConditionFailure();
}

public void out(){
    out_Requires();
    invariant();
    array[pout] = null;
    pout = (pout + 1) % array.length;
    size--;
    invariant();
    out_Ensures();
}

protected void out_Requires() {
    if (!(!isEmpty())) throw new PreConditionFailure();
}

protected void out_Ensures() {
    if (!(!isFull())) throw new PostConditionFailure();
}

private T peek_Wrap(){
    return array[(pout) % array.length];
}

public T peek() {
    peek_Requires();
    T result = peek_Wrap();
    return result;
}

protected void peek_Requires() {
    if (!(!isEmpty())) throw new PreConditionFailure();
}

private T peekOut_Wrap(){
    T result = array[(pout) % array.length];
    array[pout] = null;
    pout = (pout + 1) % array.length;
    size--;
    return result;
}

public T peekOut() {
    peekOut_Requires();
    invariant();
    T result = peekOut_Wrap();
    invariant();
    peekOut_Ensures(result);
    return result;
}
```

```
protected void peekOut_Requires() {
   if (!(!isEmpty())) throw new PreConditionFailure();
}

protected void peekOut_Ensures(T result) {
   if (!(!isFull())) throw new PostConditionFailure();
}

private T peekIn_Wrap(){
   return array[(pout + (size - 1)) % array.length];
}

public T peekIn() {
   peekIn_Requires();
   T result = peekIn_Wrap();
   return result;
}

protected void peekIn_Requires() {
   if (!(!isEmpty())) throw new PreConditionFailure();
}

public void clear() {
   invariant();
   int p = pout;
   for (int i = 0; i < size; i++) {
      array[p] = null;
      p = (p + 1) % array.length;
   }
   size = 0;
   invariant();
   clear_Ensures();
}

protected void clear_Ensures() {
   if (!(isEmpty())) throw new PostConditionFailure();
}

public boolean isEmpty() {
   return size == 0;
}

public boolean isFull() {
   return limited && (size == array.length);
}

public int size() {
   return size;
}

public boolean isLimited() {
   return limited;
}

private int maxSize_Wrap(){
   return array.length;
}

public int maxSize() {
   maxSize_Requires();
   int result = maxSize_Wrap();
   return result;
}

protected void maxSize_Requires() {
```

```
        if (!(isLimited())) throw new PreConditionFailure();
    }

    public void invariant() {
        publicInvariant();
    }

    public void publicInvariant() {
    if (!((isEmpty() && size() == 0) || (!isEmpty() && size() > 0)))
        throw new InvariantFailure();
    }
}
```

Bloco de Código B.5: Produtores-Consumidores : `SharedArrayedQueue.java`

```
import ccjava.Concurrency.*;
import ccjava.DbC.*;

public class SharedArrayedQueue<T>{
    protected final GroupMutex gmutex;
    protected final Mutex extMutex;
    protected final RWEx intra;
    protected GroupMutexComposedCV cv ;
    protected final ArrayedQueue<T> seq_ArrayedQueue;
    protected boolean objectInStableState = true;

    public SharedArrayedQueue(){
        gmutex = new GroupMutex (2 , GroupMutex.Priority.GROUP_NUMBER_INCREASING);
        extMutex = new Mutex();
        intra = new RWEx();
        Sync[] list = new Sync[]{extMutex,intra};
        cv = gmutex.newCV(list);
        seq_ArrayedQueue = new ArrayedQueue<T>();
    }

    public SharedArrayedQueue(int maxSize){
        if(!(maxSize>=0)) throw new PreConditionFailure();
        gmutex = new GroupMutex (2 , GroupMutex.Priority.GROUP_NUMBER_INCREASING);
        extMutex = new Mutex();
        intra = new RWEx();
        Sync[] list = new Sync[]{extMutex,intra};
        cv = gmutex.newCV(list);
        seq_ArrayedQueue = new ArrayedQueue<T>(maxSize);
    }

    public void in(T e){
        boolean precondition = true;
        assert objectInStableState : "Object in unstable state!";
        try{
            if (isGrabbedByMe()){
                precondition = !seq_ArrayedQueue.isFull();
                seq_ArrayedQueue.in(e);
                cv.broadcast();
            }else{inIntra(e);}
        }catch(DbC_Failure dbc){if(precondition) objectInStableState = false;}
    }

    protected void inIntra(T e){
        gmutex.lock(1);
        try{
            intra.lockWriter();
            try{
                while(!(!seq_ArrayedQueue.isFull())){cv.await();}
                seq_ArrayedQueue.in(e);
                cv.broadcast();
            }finally{intra.unlockWriter();}
```

```
    }finally{gmutex.unlock(1);}
}

public void out(){
    boolean precondition = true;
    assert objectInStableState : "Object in unstable state!";
    try{
        if (isGrabbedByMe()){
            precondition = !seq_ArrayedQueue.isEmpty();
            seq_ArrayedQueue.out();
            cv.broadcast();
        }else{outIntra();}
   }catch(DbC_Failure dbc){if(precondition) objectInStableState = false;}
}


protected void outIntra(){
    gmutex.lock(1);
    try{
        intra.lockWriter();
        try{
            while(!(!seq_ArrayedQueue.isEmpty())){cv.await();}
            seq_ArrayedQueue.out();
            cv.broadcast();
        }finally{intra.unlockWriter();}
    }finally{gmutex.unlock(1);}
}

public T peek(){
    assert objectInStableState : "Object in unstable state!";
    T result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.peek();}
    else{ result = peekIntra();}
    return result;
}

protected T peekIntra(){
    T result;
    gmutex.lock(1);
    try{
        intra.lockReader();
        try{
            while(!(!seq_ArrayedQueue.isEmpty())){cv.await();}
            result = seq_ArrayedQueue.peek();
        }finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public T peekOut(){
    boolean precondition = true;
    assert objectInStableState : "Object in unstable state!";
    T result = null;
    try{
        if (isGrabbedByMe()){
            precondition = !seq_ArrayedQueue.isEmpty();
            result = seq_ArrayedQueue.peekOut();
            cv.broadcast();
        }else{ result = peekOutIntra();}
    }catch(DbC_Failure dbc){if(precondition) objectInStableState = false;}
    return result;
}

protected T peekOutIntra(){
    T result = null;
    gmutex.lock(1);
```

```
    try{
        intra.lockWriter();
        try{
            while(!(!seq_ArrayedQueue.isEmpty())){cv.await();}
            result = seq_ArrayedQueue.peekOut();
            cv.broadcast();
        }finally{intra.unlockWriter();}
    }finally{gmutex.unlock(1);}
    return result;
}

public T peekIn(){
    assert objectInStableState : "Object in unstable state!";
    T result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.peekIn();}
    else{ result = peekInIntra();}
    return result;
}

protected T peekInIntra(){
    T result;
    gmutex.lock(1);
    try{
        intra.lockReader();
        try{
            while(!(!seq_ArrayedQueue.isEmpty())){cv.await();}
            result = seq_ArrayedQueue.peekIn();
        }finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public void clear(){
    assert objectInStableState : "Object in unstable state!";
    try{
        if (isGrabbedByMe()){
            seq_ArrayedQueue.clear();
            cv.broadcast();
        }else{clearIntra();}
    }catch(DbC_Failure dbc){objectInStableState = false;}
}

protected void clearIntra(){
    gmutex.lock(1);
    try{
        intra.lockWriter();
        try{
            seq_ArrayedQueue.clear();
            cv.broadcast();
        }finally{intra.unlockWriter();}
    }finally{gmutex.unlock(1);}
}

public boolean isEmpty(){
    assert objectInStableState : "Object in unstable state!";
    boolean result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.isEmpty();}
    else{ result = isEmptyIntra();}
    return result;
}

protected boolean isEmptyIntra(){
    boolean result;
    gmutex.lock(1);
    try{
        intra.lockReader();
```

```
        try{result = seq_ArrayedQueue.isEmpty();}
        finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public boolean isFull(){
    assert objectInStableState : "Object in unstable state!";
    boolean result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.isFull();}
    else{ result = isFullIntra();}
    return result;
}

protected boolean isFullIntra(){
    boolean result;
    gmutex.lock(1);
    try{
        intra.lockReader();
        try{result = seq_ArrayedQueue.isFull();}
        finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public int size(){
    assert objectInStableState : "Object in unstable state!";
    int result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.size();}
    else{ result = sizeIntra();}
    return result;
}

protected int sizeIntra(){
    int result;
    gmutex.lock(1);
    try{
        intra.lockReader();
        try{result = seq_ArrayedQueue.size();}
        finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public boolean isLimited(){
    assert objectInStableState : "Object in unstable state!";
    boolean result;
    if (isGrabbedByMe()){result = seq_ArrayedQueue.isLimited();}
    else{ result = isLimitedIntra();}
    return result;
}

protected boolean isLimitedIntra(){
    boolean result;
    gmutex.lock(1);
    try{
        intra.lockReader();
        try{result = seq_ArrayedQueue.isLimited();}
        finally{intra.unlockReader();}
    }finally{gmutex.unlock(1);}
    return result;
}

public int maxSize(){
    assert objectInStableState : "Object in unstable state!";
    int result;
```

```
        if (isGrabbedByMe()){result = seq_ArrayedQueue.maxSize();}
        else{ result = maxSizeIntra();}
        return result;
    }

    protected int maxSizeIntra(){
        int result;
        gmutex.lock(1);
        try{
            intra.lockReader();
            try{
                while(!(seq_ArrayedQueue.isLimited())){cv.await();}
                result = seq_ArrayedQueue.maxSize();
            }finally{intra.unlockReader();}
        }finally{gmutex.unlock(1);}
        return result;
    }

    public boolean isGrabbedByMe(){
        return gmutex.lockIsMine() && gmutex.activeGroup()== 2 && extMutex.lockIsMine();
    }

    public void grab(){
        if (isGrabbedByMe()) throw new PreConditionFailure();
        gmutex.lock(2);
        extMutex.lock();
        if (!isGrabbedByMe()) throw new PostConditionFailure();
    }

    public void release(){
        if (!isGrabbedByMe()) throw new PreConditionFailure();
        gmutex.unlock(2);
        extMutex.unlock();
        if (isGrabbedByMe()) throw new PostConditionFailure();
    }
}
```

Apêndice C

Nova Regras da Linguagem

Bloco de Código C.1: Novas Regras da Linguagem

```
//VARIÁVEIS & ARGUMENTOS
modifier
:   classOrInterfaceModifier
|   ('native' | 'synchronized' | 'transient' | 'volatile' | 'shared');

variableModifier : 'final' | 'shared' | annotation;

//CLASSES
typeDeclaration
    : classModifier* classDeclaration  | classOrInterfaceModifier* enumDeclaration
    | classOrInterfaceModifier* interfaceDeclaration
    | classOrInterfaceModifier* annotationTypeDeclaration
    |   ';' ;

classModifier
    : annotation
    |   ('public'   | 'protected' | 'private'  | 'static'
        |'abstract' | 'final'     | 'strictfp' | 'shared' | 'remote');


classDeclaration: classDeclarationSignature classBody;

classDeclarationSignature
    :'class' Identifier typeParameters?
    ('extends' type)?
    ('implements' typeList)?;

classBodyDeclaration
    :   ';'
    |   'static'? block
    |   modifier* memberDeclaration
    |   invariantClause;

//INTERFACES
interfaceBodyDeclaration
    :   modifier* interfaceMemberDeclaration
    |   invariantClause | ';' ;

interfaceMethodDeclaration
    : (type|'void') Identifier formalParameters ('[' ']')*
      ('throws' qualifiedNameList)? (interfaceMethodBody | ';');

interfaceMethodBody : requiresClause* ensuresClause*;
```

```
//DBC
constructorDeclaration: Identifier formalParameters ('throws' qualifiedNameList)?
                        requiresClause* constructorBody ensuresClause* ;

methodBody            : requiresClause* block? ensuresClause* (rescueBlock)?;

requiresClause        : 'requires' assertionClause;

ensuresClause         : 'ensures' assertionClause;

rescueBlock           : 'rescue' (excpDecl)? '{' blockStatement '}';

excpDecl              : '(' excpTypeList Identifier ')';

excpTypeList          : excpType ('|' excpType )*;

excpType              : Identifier ('.' Identifier)*;

invariantClause       :  ('public' | 'protected' | 'private')? 'invariant' assertionClause;

assertionClause       : conditionalExpression (':' expression)? ';';

conditionalExpression :   expression;

//OBJECTOS PARTILHADOS & REMOTOS
expression : . . . | sharedObjectCreator | remoteObjectCreator;

sharedObjectCreator   : (schemeAnnotation)? 'new' 'shared' creator
                      | (schemeAnnotation)? 'new' creator;

schemeAnnotation      : '@' Identifier;

remoteObjectCreator   : (policyAnnotation)? 'new' creator;

policyAnnotation      : '@' Identifier;

//COMENTÁRIOS & ESPAÇOS BRANCOS
WS:   [ \t\r\n\u000C]+ -> channel(1);

COMMENT:    '/*' .*? '*/' -> channel(1);

LINE_COMMENT:    '//' ~[\r\n]* -> channel(1);

// LEXER (palavras-chave)
ENSURES : 'ensures';
INVARIANT : 'invariant';
REQUIRES : 'requires';
SHARED : 'shared';
REMOTE : 'remote';
RETRY : 'retry';
RESCUE : 'rescue';
```

Bibliografia

[1] Pedro G. Francisco. Contract-Java: Programação por Contrato em Java, Universidade de Aveiro, 2012. Disponível em `http://hdl.handle.net/10773/11035`.

[2] António Adrego da Rocha e Osvaldo Rocha Pacheco. *Introdução à Programação em Java*. Second edition.

[3] Miguel Oliveira e Silva. Metodologias e Mecanismos para Linguagens de Programação Concorrente Orientadas por Objectos, Universidade de Aveiro, 2007. Disponível em `http://sweet.ua.pt/mos/pubs/phd-thesis.pdf`.

[4] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Second edition, 1997.

[5] Robert W. Floyd. Assigning Meaning to Programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–32, 1967.

[6] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 1969.

[7] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[9] Bertrand Meyer. Disciplined exceptions. *Report TR-EI-13/EX, Interactive Software Engineering*, 1988.

[10] Leslie Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[11] G. R. Andrews e F. B. Scheneider. Concepts and Notations for Concurrent Programming. *ACM Computing Systems*, 15(1):3–43, 1983.

[12] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, Second edition, 2000.

[13] Ada. *Ada95 Reference Manual (Language and Standard Libraries)*. United States of America Government, 1995.

[14] J. Gosling e B. Joy e G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] Alan A. A. Donovan e Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2009.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Fourth edition.

[17] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley.

[18] D. Holmes e J. Noble e J. Potter. Toward Reusable Synchronisation for Object-Oriented Languages. *Workshop on Object-Oriented Technology, ECOOP98*, page 439, 1998.

[19] J. P. Briot e R. Guerraoui e K. P. Lohr. Concurrency and Distribution in Object-oriented programming. *ACM Computing Surveys (CSUR)*, 30(3):291–329, 1998.

[20] Miguel Oliveira e Silva. Automatic Realizations of Statically Safe Intra-Object Synchronization schemes in mp-eiffel. *Proceedings of the First Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 91–118, 2006.

[21] Miguel Oliveira e Silva. Concurrent Object-Oriented Programming: The MP-Eiffel Approach. *Journal of Object Technology*, 3(4):97–124, 2004.

[22] Biblioteca de Concorrência para a Linguagem de Programação Java. Disponível em `http://sweet.ua.pt/mos/pt.ua.concurrent/index.xhtml`.

[23] Terence Parr. *The Definite ANTLR4 Reference*. The Pragmatic Programmers.

[24] Gramática Java-v7 para ANTLR-v4. Disponível em `https://github.com/antlr/grammars-v4/blob/master/java/Java.g4`.

[25] C. A. R. Hoare. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[26] Pedro G. Francisco e Miguel Oliveira e Silva. Contract-Java: Design by Contract in Java with Safe Error Handling. *3rd Symposium on Languages, Applications and Technologies, SLATE'14*, pages 127–142, 2014.