Universidade Federal de Pernambuco
Centro de Ciências Exatas e da Natureza
Departamento de Informática

Graduação em Ciência da Computação

Desenvolvimento Sistemático de Programas Concorrentes Orientados a Objetos

por

Sérgio Castelo Branco Soares

Trabalho de Graduação

Recife, 30 de Julho de 1999

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA

Sérgio Castelo Branco Soares

Desenvolvimento Sistemático de Programas Concorrentes Orientados a Objetos

Este trabalho foi apresentado à Graduação em Ciência da Computação do Centro de Ciências Exatas e da Natureza da Universidade Federal de Pernambuco como requisito para a disciplina Trabalho de Graduação em Engenharia de Software.

ORIENTADOR:
Prof. Paulo Henrique Monteiro Borba

Agradecimentos

Inicialmente a meu orientador, que me deu oportunidade de crescer, desde os tempos das monitorias de LP3 e da especialização, até a iniciação científica, e graças a Deus, no mestrado. Por sua paciência e sobretudo competência, me orientando acadêmica e pessoalmente, como amigo, que eu considero muito.

A minha equipe de desenvolvimento do CESAR, Faro, Denise, Cuca, d'Amorim, Gibeon e Valença, e a todos os colaboradores, que além de terem me aturado, contribuiram para aumentar o meu senso de trabalho em equipe, fazendo-me sentir muito bem nos 12 meses em que trabalhei no CESAR.

A todos os meus amigos do DI que de alguma maneira me ajudaram nestes logos anos de luta na graduação. Aos professores Fernando Fonseca e Ana Carolina, que também me deram oportunidades de ter experiências acadêmicas em iniciação científica e monitorias.

Por fim a minha família, que sempre esteve do meu lado, me dando forças, sempre pensando no meu futuro, A meu pai, meus irmãos e minha esposa, em especial a minha mãe, e a minha filha, que é a mais nova razão para enfrentar as tantas outras lutas que estão por vir.

Resumo

O uso cada vez maior de ambientes de programação concorrentes torna necessário a formalização de leis (regras) que garantam um desenvolvimento seguro e eficiente de programas concorrentes.

Durante o desenvolvimento deste tipo de programa, deve-se ter cuidado com problemas ocasionados pela natureza do ambiente concorrente, como, por exemplo, acessos indevidos a variáveis compartilhadas.

Ao mesmo tempo, o uso de mecanismos que implementam acesso exclusivo a regiões críticas deve ser feito com cuidado, uma vez que tais mecanismos podem adicionar *deadlocks* durante a execução concorrente do programa. Na linguagem Java [9], o qualificador de métodos `synchronized` é um dos meios de garantir acesso exclusivo a trechos de código, pois quando um método qualificado como `synchronized` é executado, todos os demais métodos qualificados como `synchronized` deste objeto ficam boqueados pelo `thread` que o está executando.

Este trabalho propõem uma lei que guie a utilização deste mecanismo de sincronização com segurança, mostrando exemplos e contra-exemplos do seu uso, bem como um estudo de caso onde a lei proposta é aplicada a um sistema não trivial.

Índice

Lista de Figuras

1 Introdução

Este trabalho de graduação tem como objetivo propor uma lei de refinamento [15] que guie a sincronização de métodos de classes Java [9], de forma que classes projetadas para executar em ambientes sequenciais, tornem-se aptas a executar com segurança em ambientes concorrentes. A sincronização em Java é feita com o uso do qualificador de métodos `synchronized` [9]. Todavia, o qualificador `synchronized` deve ser utilizado com cuidado, caso contrário podem ser inseridos erros sutis suficientes para comprometer a corretude do programa. Daí a necessidade de uma lei de refinamento formal que guie a utilização deste mecanismo, indicando precisamente as condições necessárias à sincronização com segurança

Coube a este trabalho o estudo dos aspectos de concorrência em Java [12], a definição de um subconjunto da linguagem Java, para o qual foi definida a lei para sincronizar métodos, bem como a definição de predicados que tem um alto grau de automação. Estes predicados são responsáveis pela verificação de que métodos são livres de *livelock* e *deadlock*, e são utilizados para especificar as condições para a utilização da lei.

Aqui são apresentados também exemplos e contra-exemplos que mostram a necessidade e a utilidade da lei proposta, incluindo suas condições. Além disso, descrevemos um estudo de caso que mostra a aplicação da lei a métodos de classes de um sistema real e não trivial, validando assim as definições e de mostrando o elevado grau de automação das mesmas.

Este trabalho de graduação faz parte de um projeto de pesquisa [3] que visa investigar e desenvolver técnicas e ferramentas que sejam úteis para suportar o desenvolvimento sistemático de programas concorrentes de alta qualidade. Este projeto de pesquisa é realizado no Departamento de Informática da Universidade Federal de Pernambuco, sendo coordenado pelo Professor Paulo Borba, que também orienta esta trabalho. Tal projeto vem resultando em artigos [4, 5, 16] que mostram, respectivamente, a necessidade de regras formais para programação orientada a objetos, padrões de projetos para sistemas concorrentes distribuídos, e a lei proposta neste trabalho, que aprofunda as seções sobre concorrência existentes nos dois primeiros artigos.

Este trabalho está organizado da seguinte maneira. Primeiro discutimos a arquitetura em camadas utilizada como referência na elaboração da lei de sincronização. Depois apresentamos exemplos onde o uso do qualificador `synchronized` introduz erros. Em seguida definimos uma formalização para representar definições de classes, um subconjunto simplificado da linguagem Java, algumas funções sobre o mesmo e a lei de sincronização seguida pela definição dos predicados que especificam as condições para que a mesma possa ser aplicada. Por fim apresentamos um estudo de caso, onde a lei definida é aplicada em classes do sistema SGC [6], e em seguida uma conclusão sobre este trabalho de graduação.

1.1 Contexto

Esta seção corresponde a introdução do projeto de pesquisa [3] no qual este trabalho de graduação é baseado, podendo também ser vista como o contexto e a motivação para o trabalho.

Programas concorrentes e distribuídos são extremamente importantes para a sociedade. Finanças, telecomunicações, aviação e simulação são alguns domínios de aplicação importantes onde concorrência é normalmente um fator central no desenvolvimento. Concorrência tem sido especialmente útil na implementação de aplicações que têm que responder imediatamente à interação com os usuários, independente de que processamento esteja sendo feito, como deveriam ser as interfaces gráficas dos sistema operacionais. Atualmente, estamos observando um impacto ainda maior de programas concorrentes na sociedade com o advento de sistemas de informação baseados na Internet e na *World Wide Web* (WWW).

Entretanto, é extremamente difícil desenvolver programas concorrentes e distribuídos. Conseqüências disto são freqüentemente relatadas pela imprensa quando programas não funcionam corretamente devido a erros sutis de implementação. A dificuldade é notória tanto quando concorrência é inerente à aplicação — vários agentes competindo pelos mesmos recursos ao mesmo tempo — quanto quando concorrência é implementada simplesmente para aumentar a eficiência dos programas.

Interferência, que é inerente aos sistemas concorrentes, é a causa principal destas dificuldades. A execução de um programa concorrente pode ser interferida pela execução de outros programas concorrentes compartilhando e acessando os mesmos recursos ao mesmo tempo. De fato, erros sutis podem resultar destas interferências, tornando-se difícil entender e prever o comportamento dos sistemas concorrentes. As técnicas de teste e depuração, que já não são tão efetivas para garantir a corretude de programas seqüenciais, são ainda menos efetivas para programas concorrentes, já que estes são normalmente não-determinísticos devido às várias possíveis interferências que podem ocorrer durante a execução.

Este cenário indica que nós precisamos de meios mais adequados para o desenvolvimento de programas concorrentes e distribuídos. Assim, este trabalho de graduação, propõe uma lei que é útil para suportar o desenvolvimento sistemático de programas concorrentes de alta qualidade. A abordagem utilizada visa a integração de orientação a objetos com métodos formais. Certamente, esta integração tem sido muito útil para o desenvolvimento de programas seqüenciais [10, 14, 17, 13]; e também deve ser para o desenvolvimento de programas concorrentes.

De fato, técnicas de análise, projeto e programação orientadas a objetos são extremamente eficientes para reuso, manutenção e desenvolvimento incremental de software, como tem sido constatado com o desenvolvimento de aplicações industriais [17, 1]. Mais importante, programação orientada a objetos tem todo o potencial para suportar meios adequados para o desenvolvimento de programas concorrentes: objetos são naturalmente distribuídos, e os conceitos de orientação a objetos são poderosas ferramentas para controlar interferência [11, 2].

Por outro lado, a importância de métodos formais para desenvolvimento de software é significativamente reconhecida [7], sendo justificada pelo alto nível de confiabilidade dos sistemas desenvolvidos com linguagens de especificação oferecendo um alto nível de abstração, tendo uma semântica matemática bem definida, e permitindo provas formais de que projetos e implementações satisfazem (refinam) especificações [15]. Em particular, métodos formais podem ter um impacto considerável sobre o desenvolvimento de programas concorrentes, onde a eficácia das técnicas tradicionais de teste e depuração é substancialmente reduzida.

O fator mais importante da integração de concorrência com programação orientada a objetos e métodos formais é o suporte natural a um método de desenvolvimento que possibilita um programador começar com uma implementação (ou especificação) seqüencial do sistema e progressivamente ir considerando concorrência. Certamente, a abstração inicial dos aspectos sutis de concorrência é muito importante para dominar a complexidade dos sistemas atuais, mas isso só não é suficiente: também precisamos de justificativas precisas para a introdução progressiva de concorrência. E métodos formais certamente podem dar estas justificativas através do uso de regras de transformação e refinamento que preservam significado e garantem a introdução de concorrência sem causar problemas. Só desta maneira programadores podem evitar a introdução de detalhes que comprometem a corretude dos programas. Em particular, a lei aqui apresentada indica em que situações é possível introduzir o qualificador de métodos `synchronized`.

2 Arquitetura de Camadas

Nesta seção introduzimos a arquitetura de software utilizada como referência na elaboração da lei de sincronização proposta por este trabalho. De fato, a regra proposta se aplica a sistemas orientados a objetos que tenham uma arquitetura de camadas representada pela Figura 1.[1]

onde

- as Classes Básicas de Negócio representam os objetos (básicos) do sistema, como por exemplo clientes, endereços e telefones.
- as Classes Coleção de Dados são responsáveis pelo armazenamento e recuperação de instâncias das classes básicas de negócio, como `RepositorioClienteArray`.
- as Classes Coleção de Negócio são também responsáveis pelo armazenamento e recuperação das instâncias das classes básicas de negócio, mas além disso estas classes encapsulam as verificações e validações segundo as regras e políticas da aplicação em questão. Tais classes utilizam os serviços das classes de coleção de dados.
- a Classe Fachada de Negócio [8] provê uma interface unificada para todos os serviços do sistema, através da centralização de todas as instâncias das classes de coleção de negócio. Representa os serviços oferecidos pelo sistema.

Além do ilustrado na figura da arquitetura, as seguintes restrições também devem ser satisfeitas:

- as Classes Básicas de Negócio não chamam métodos das Classes Coleção de Dados, Coleção de Negócio e Fachada de Negócio.
- as Classes Coleção de Dados não chamam métodos das Classes Coleção de Negócio e Fachada de Negócio;
- as Classes Coleção de Negócio não chamam métodos da Classe Fachada de Negócio.

Em resumo, métodos de classes de um nível n não chamam métodos de uma classe de nível maior que n. Além desta, temos uma restrição adicional:

- não podem haver métodos mutuamente recursivos, como ilustrado a seguir, seja em uma mesma classe ou entre duas ou mais classes:

```
public class C {
   public void m() { n(); ... }
   public void n() { m(); ... }
   ⋮
}
```

Tais restrições são necessárias para facilitar a definição da lei de sincronização, Seção 5, e dos predicados, Seção 5.1. Uma vez que eliminam a possibilidade dos predicados entrarem em *loop*.

[1] Em uma arquitetura real utiliza-se também o conceito de interfaces, classes abstratas e herança, mas por simplicidade utilizamos a arquitetura descrita na Figura 1.

3 Exemplos

Nesta seção apresentamos contra-exemplos que ilustram casos em que a utilização do qualificador `synchronized` pode inserir erros em um programa. Primeiro introduzimos uma notação que representa a execução concorrente de dois comandos, em seguida apresentamos um exemplo em que a sincronização de um método que não termina pode introduzir um *deadlock* no programa, depois mostramos um exemplo em que a sincronização de um método que entra em *deadlock* pode introduzir novos *deadlocks* no programa, e por fim apresentamos um exemplo em que a sincronização de um método de um programa que não segue a arquitetura proposta na seção 2 também pode introduzir um *deadlock* no mesmo.

3.1 Composição paralela de comandos

Introduzimos aqui uma notação para a composição paralela de dois comandos em Java, correspondendo a execução paralela dos mesmos. Basicamente, dois comandos separados por duas barras horizontais paralelas representam a composição paralela dos mesmos. Por exemplo, o trecho de código

```
C c = new C();
c.m() || c.m()
```

representa a execução concorrente de duas chamadas do método `m` de uma classe `C` qualquer. Os comandos da composição paralela são executados por dois `threads` diferentes. De fato, esta notação é uma abreviação para

```
public class T extends Thread {
   private C c;
   public T(C c) {
      this.c = c;
   }
   public void run() {
      c.m();
   }
}
```

onde `T` é um *thread* Java que chama o método `m` de um objeto `C`, juntamente com o seguinte trecho de código:

```
C c = new C();
T t1 = new T(c);
T t2 = new T(c);
t1.start();
t2.start();
t1.join();
t2.join();
```

onde são criados e executados dois *threads*, que fazem referência para um mesmo objeto do tipo `C`. No trecho acima, o método `start` inicia a execução de um *thread*, e o método `join` espera ate que o *thread* acabe a sua execução.

Da mesma forma, o código

```
C c = new C();
D d = new D();
c.m() || (c.n();d.m())
```

corresponde a execução concorrente de "`c.m()`" e "`c.n(); d.m()`" podendo haver entrelaçamento entre a execução dos dois comandos. Por exemplo, é possível que "`c.n()`" seja executado primeiro, depois "`c.m()`", e por fim "`d.m()`". Na verdade parte da execução de "`c.m()`" pode ser entrelaçada com parte da execução de "`d.m()`".

Nas subseções seguintes, esta notação é utilizada com o intuito de simplificar o entendimento de situações onde comandos executados concorrentemente podem levar um sistema a um estado inconsistente, ou a executar uma operação de maneira errada. Sempre que nos referirmos ao primeiro *thread* de uma composição paralela, estamos nos referindo ao *thread* que executa o comando a esquerda das barras paralelas.

3.2 Métodos que não terminam

Nesta seção mostra-se um exemplo onde a sincronização de um método que não termina pode introduzir *deadlock*. Considere a classe `File`, definida a seguir, que possui um método que imprime um atributo, e em seguida começa a execução de um laço infinito:

```
public class File {
   private String text;
   public File(String text) {
      this.text = text;
   }
   public void print(String message) {
      System.out.println(text);
      while(true) {
         System.out.print(message);
      }
   }
}
```

Utilizando a notação definida anteriormente, o código

```
File file = new File("Test");
file.print("1") || file.print("2")
```

representa a execução concorrente de duas chamadas do método `print` de uma mesma instância da classe `File`. Esta execução tem como um dos possíveis resultados a saída:

```
Test
111Test
21212211121212122211121 ...
```

caso o primeiro *thread* inicie a executar o método `print` primeiro, e

```
Test
2Test
11212211121212122211121 ...
```

se o segundo *thread* iniciar a executar o método `print` antes do primeiro *thread*. Estes resultados da execução mostram que os dois *threads* executaram o corpo do método `print`, onde o comando "`System.out.print(message)`" é executado indefinidamente.

Sincronizando o método `print`, a execução do mesmo trecho de código, só pode ter como resultado:

```
Test
2222222222222222222222 ...
```

no caso do segundo *thread* executar o método `print` antes do primeiro *thread* e

```
Test
1111111111111111111111 ...
```

no caso do primeiro *thread* executar o método `print` primeiro. Nos dois casso, apenas um dos *threads* consegue executar o método `print` já que a sincronização do método `print` impossibilita a execução de duas chamadas do método em um mesmo objeto.

A execução do mesmo trecho de código com o método `print` sincronizado e não sincronizado geraram resultados diferentes. Quando o método não está sincronizado, os dois *threads* executam o comando que imprime o atributo, e entram em *loop* imprimindo indefinidamente parâmetro `message`. Quando o método `print` está sincronizado, apenas o *thread* que primeiro executa a chamada ao método consegue executá-lo. O outro *thread* ao tentar executar a chamada ao método entra em *deadlock*, pois o método `print` já está sendo executado no mesmo objeto por um outro *thread*, que em função do *loop* infinito, está em *livelock* e não liberará a execução de métodos sincronizados do objeto.

Este exemplo mostra que sincronizar um método que não termina não é um refinamento válido, uma vez que a sincronização pode introduzir um *deadlock*, que não existia, na execução do programa.

3.3 Métodos que levam a *deadlock*

Esta seção mostra um outro exemplo onde a qualificação de um método como `synchronized` pode introduzir mais *deadlocks* do que existiam antes da qualificação. Isto ocorre quando sincroniza-se métodos cuja execução pode levar um *thread* a entrar em *deadlock*

Considere uma classe com apenas um método para incrementar um contador:

```
public class Counter {
   private int counter;
   public Counter() {
      counter = 0;
   }
   public void add(String message) {
      counter = counter + 1;
      System.out.println("The counter is "+counter+" by thread "+
                         message);
      this.deadLock();
   }
}
```

mas que na verdade, incrementa o contador, imprime a mensagem e chama um método que leva a um *deadlock* ao executar o comando "`this.deadLock`". Assim a execução do trecho de código:

```
Counter co = new Counter();
co.add("1") || co.add("2")
```

pode gerar como resultado

```
The counter is 1 by thread 1
The counter is 2 by thread 2
```

no caso do primeiro *thread* inicie e termine a execução da chamada do método `add` antes que o segundo *thread* comece a executar, ou

```
The counter is 1 by thread 2
The counter is 2 by thread 1
```

caso contrário.

Agora, sincronizando o método `add`, a execução do mesmo trecho de código só pode gerar como resultado:

```
The counter is 1 by thread 1
```

ou

```
The counter is 1 by thread 2
```

mostrando que apenas um dos *threads* consegue executar a chamada do método `add`, já que quando um dos *threads* executa a chamada ao método `add`, não permite que o outro execute a mesma chamada que está sendo executada no mesmo objeto por um outro *thread*, que em função do *deadlock* não liberará a execução de métodos sincronizados deste objeto.

Como no exemplo da seção anterior, a execução de um mesmo trecho de código com o método `add` sincronizado e não sincronizado foram diferentes uma vez que a sincronização do método `add` pode inserir um *deadlock* na execução concorrente do mesmo.

Este exemplo mostra que sincronizar um método que não é livre de *deadlock* não é um refinamento válido, uma vez que a sincronização pode introduzir um novo *deadlock*, na execução do programa.

3.4 Métodos Recursivos

Esta seção mostra mais um exemplo onde a inserção do qualificador `synchronized` em métodos pode introduzir *deadlock*. Dessa vez o problema é com métodos recursivos, quando a chamada que caracteriza a recurção é feita para um outro objeto da mesma classe do método.

Para ilustrar melhor a situação, considere a definição da classe `File` que possui um atributo do mesmo tipo representando o restante do arquivo, e um atributo `text` que representa o conteúdo desta parte do arquivo.

```
public class File {
   private File otherPart;
   private String text;
   public File(String text) {
      this.text = text;
   }
   public void setFile(File otherPart) {
      this.otherPart = otherPart;
   }
```

A classe `File` tem ainda um método que imprime o texto inicial do arquivo, e outro método que imprime o texto inicial do arquivo e o texto das demais partes do arquivo.

```
   public void print(String message) {
      System.out.println(text+" by thread "+message);
   }
   public void printAll(String message) {
      this.print(message);
      otherPart.printAll(message);
   }
}
```

Como prova de que sincronizar esta classe não é uma tarefa simples, considere a execução do trecho de código

```
File f1 = new File("FileOne");
File f2 = new File("FileTwo");
f1.setFile(f2);
f2.setFile(f1);
f1.printAll("1") || f2.printAll("2")
```

onde as duas instâncias de `File` referenciam uma a outra. Assim a execução do trecho de código pode ter como resultado

```
FileOne by thread 1
FileTwo by thread 2
FileTwo by thread 1
FileOne by thread 2
⋮
```

ou

```
FileTwo by thread 2
FileOne by thread 1
FileOne by thread 2
FileTwo by thread 1
⋮
```

dependendo de qual *thread* inicie a execução da chamada de método primeiro.

Executando o mesmo trecho de código, mas agora com o método `printAll` sincronizado, pode-se ter como resultado

```
FileOne by thread 1
FileTwo by thread 2
```

ou

```
FileTwo by thread 2
FileOne by thread 1
```

já que cada *thread* ao executar a chamada do método `printAll` do seu atributo imprime o atributo `text` e chama o método `printAll` do atributo `otherPart`. Observe que neste momento da execução, cada *thread* bloqueou os métodos sincronizados do seu atributo, e tenta executar o método `printAll` do outro, o que nunca acontecerá, uma vez que os *threads* não conseguem executar a chamada, pois nehum deles liberará a execução de métodos sincronizados dos seus atributos.

Podemos então afirmar que os dois *threads* entraram em *deadlock*, pois um está esperando que o outro libere o boqueio do seu atributo.

Concluimos então que não é um refinamento válido sincronizar um método recursivo na forma aqui apresentada. Por isso na definição da arquitetura utilizada, veja seção 2, são impostas algumas restrições que também deixam de foram este tipo de classe.

4 Formalização

Aqui vamos apresentar uma álgebra para definição de classes, uma definição de um subconjunto da linguagem Java, seguida da definição de algumas funções que são utilizados pela lei proposta neste trabalho

4.1 Representando definições de classes

Seguindo [4], nesta seção apresentamos uma espécie de álgebra para definição de classes, através do uso de operadores que constrõem classes a partir de suas partes. O operador "$\langle \ \bullet \ \ \bullet \ \rangle$" constroi classes como por exemplo

$$\langle A \ \bullet \ M \ \bullet \ \imath \rangle$$

que denota uma definição de classe que é formada pelos atributos privados em A, os métodos em M, e os inicializadores (construtores) em I. Já

$$\langle A \otimes a \ \bullet \ M \oplus sync(m) \ \bullet \ \imath \rangle$$

denota a classe formada pelos atributos privados em A mais o atributo privado a, pelos métodos em M mais o método m, e os inicializadores em I. A função *sync* qualifica um método como `synchronized`.

Podemos formalizar refinamento de classes como

$$\langle A \ \bullet \ M \oplus m \ \bullet \ \imath \rangle \sqsubseteq \langle A \ \bullet \ M \oplus m' \ \bullet \ \imath \rangle$$

Se $m \sqsubseteq m'$, tal representação indica que refinamento de método implica refinamento de classe.

4.2 O MiniJava

Esta seção define um subconjunto da linguagem Java na intenção de simplificar as definições dos predicados e da lei proposta. Este subconjunto tem declarações de classe, de variáveis e de construtores, além dos seguintes comandos: atribuições, chamadas de métodos, levantamento de exceções, retorno de variáveis, `if` − `else` e `while`. Nós chamamos este subconjunto de MiniJava.

A definição de um programa P em MiniJava é dada por

$$P = ClassDecs \bullet Command$$

onde *ClassDecs* são declarações de classe e *Command* é um comando que determina a execução do programa.

Uma declaração de classe é denotada por

$$\langle A \ \bullet \ M \oplus m \ \bullet \ \imath \rangle$$

onde o corpo de m é definido como

Body:
    DecVars; Command;
    DecVars; Command; `return` *var_or_cte;*

onde *var_or_cte* denota uma variável ou constante, declarações de variáveis tem a seguinte forma

DecVars:
  Type var
  DecVars; Type var

e comandos tem a seguinte forma

Command:
  $var = primitive\_exp$
  $var = var'.m(\overline{var\_or\_cte})$
  $var =$ `new` *Type()*
  $var.m(\overline{var\_or\_cte})$
  `throw` *var*
  Command; Command
  `if`(*exp_bool*){*Command*}`else`{*Command*}
  `while`(*var*> `0`){*Command*}

onde *var* é uma variável, *primitive_exp* é uma expressão formada pr tipos primitivos, *Type* é o nome de uma classe, e *exp_bool* é uma expressão booleana.

4.3 Definições auxiliares

Agora vamos definir algumas funções sobre programas em MiniJava, no sentido de auxiliar a definição dos predicados que verificam se um método é livre de *livelock* e de *deadlock*, os quais serão usados para especificar quando a lei definida neste trabalho pode ser aplicada.

Primeiro vamos definir a função *decs* que recebe como parâmetro declarações de variáveis e retorna um mapeamento dos identificadores das variáveis nos nomes dos tipos das mesmas

$decs(\ [\ ]\ ) = \emptyset$
$decs(\ (Type\ var){:}ds\ ) = \{var \mapsto Type\} \cup decs(ds)$

Além da função *body* que recebe um método e retorna o seu corpo (*Body*), vamos utilizar uma outra função *body* que é útil na definição dos predicados *livelockfree* e *deadlockfree*. Esta outra função recebe o nome de uma classe C, o nome de um método m, um programa P onde a classe e o método estão definidos, e retorna o corpo do método m definido em C. Por simplicidade não estamos considerando a possibilidade de *overloading*, já que uma classe pode ter métodos com um mesmo nome, mas com assinaturas diferentes [9].

Considerando um programa P dado por

$ClassDecs \cup \{\ L{:}\langle A\ \bullet\ M \oplus m\ \bullet\ \imath\rangle\ \} \bullet Command$

a função *body* é definida por

$body(L,\ name(m),\ P) = body(m)$

onde *name* retorna o nome de um método.

Por fim, considere o comando `while` padrão na forma

```
while(i>0) {
    ⋮
}
```

sobre o qual definimos o predicado $terminates($`while`$(var > 0)\{Command\})$ que é válido caso o comando `while` recebido termine. Segundo [15], uma interação termina sse:

- *var* é uma variável inteira, chamada de variante do *loop*;
- o valor de *var* nunca é menor que 0;
- *var* é estritamente decrescente a cada execução do corpo do *loop*.

5 Introduzindo Sincronização em Métodos

O primeiro passo para tornar um programa seguro para ser executado em um ambiente concorrente é sincronizar métodos, evitando interferências indesejadas. Assim, considerando a definição de classe

$$\langle A \bullet M \oplus sync(m) \bullet \imath \rangle$$

onde $sync(m)$ denota o método m qualificado como `synchronized`, para um conjunto de definições de classes P, temos a seguinte regra de refinamento

$$P \cup \{\langle A \bullet M \oplus m \bullet \imath \rangle\} \sqsubseteq P \cup \{\langle A \bullet M \oplus sync(m) \bullet \imath \rangle\}$$

que é válida se

- a execução do corpo do método m é livre de *livelock*, quando executado no contexto de $P \cup \{\langle A \bullet M \oplus m \bullet \imath \rangle\}$; e
- a execução o corpo do método m é livre de *deadlock*, quando executado no contexto de $P \cup \{\langle A \bullet M \oplus m \bullet \imath \rangle\}$.

Em outras palavras, a regra indica que podemos introduzir um `synchronized` em um método desde que o mesmo seja livre de *livelock* e de *deadlock*. A formalização de predicados que indicam se métodos são livres de *livelock* e de *deadlock* é dada nas seções 5.1 e 5.2, respectivamente.

Por exemplo, podemos utilizar esta regra para tornar segura uma classe que armazena objetos do tipo `Telephone` e possui um método para inserí-los:

```
public class Telephones {
   private Telephone[] telephones;
   public void add(Telephone phone) {
      telephones[index] = phone;
      index = index + 1;
   }
   ⋮
}
```

Note que o método `add` pode levar um objeto do tipo `Telephones` a um estado inconsistente. Isto pode acontecer, por exemplo, quando dois *threads*, *t1* e *t2*, chamam o método `add` de uma mesma instância de `Telephones` concorrentemente. Neste caso, suponha que *t1* execute apenas o comando de atribuição, e que em seguida *t2* inicie a sua execução. Quando *t2* executar a primeira atribuição, a referência antes atribuída por *t1* será sobrescrita, uma vez que a variável `index` não foi atualizada por *t1*.

Como o corpo do método `add` realiza apenas duas atribuições e uma soma de uma variável de tipo primitivo com uma constante, podemos dizer que o mesmo é livre de *livelock* e *deadlock*. Podemos então aplicar a regra definida nesta seção sincronizando o método `add`. Fazendo isso, a sequência de execução problemática do exemplo acima não é mais possível, uma vez que os dois *threads* não podem executar o método `add` concorrentemente, eliminando assim, interferências indesejadas na sua execução.

Nas próximas seções definimos predicados que auxiliam a verificação de que o corpo de um método é livre de *livelock* e de *deadlock*.

5.1 O Predicado *livelockfree*

Nesta seção definimos o predicado *livelockfree*, onde, sendo *body* a função que retorna o corpo de um método m, $lfB^P(body(m))$ verifica se a execução do corpo de m não entra em *livelock* quando executado no contexto de um programa P.

Utilizando apenas MiniJava para implementar o corpo de um método, o predicado lfB pode ser definido por

$$\begin{array}{ll} lfB^P(DecVars;\ Command;) & = lfC^{P,decs(VarsAttrbs)}(Command) \\ lfB^P(DecVars;\ Command;\ \texttt{return}\ var\_or\_cte;) & = lfC^{P,decs(VarsAttrbs)}(Command) \end{array}$$

onde $DecVars$ são declarações de variáveis, $VarsAttrbs$ é a união de $DecVars$ com $Attribs$ que são declarações de atributos da classe mais e a declaração de uma variável `this` do tipo da classe, $decs$ é uma função que recebe um conjunto de declarações de variáveis e retorna um mapeamento das variáveis nos tipos das mesmas, var_or_cte é uma variável ou constante, e $Command$ é um comando MiniJava.

O predicado $lfC^{P,V}(c)$ verifica se o comando c não entra em *livelock* quando executado no contexto do programa P, onde V é um mapeamento das variáveis usadas no comando, dos atributos e da variável `this`, nos tipos das mesmas.

Utilizando comandos de MiniJava, o predicado lfC pode ser definido pelas equações a seguir:

$$lfC^{P,V}(var = primitive\_exp) \quad = true$$

onde $primitive\_exp$ são expressões de tipo primitivo, ou de tipos já definidos na linuagem Java [9].

$$\begin{array}{ll} lfC^{P,V}(var = var') & = true \\ lfC^{P,V}(var = var'.m(\overline{var\_or\_cte})) & = lfC^{P,V}(var'.m(\overline{var\_or\_cte})) \\ lfC^{P,V}(var = \texttt{new}\ Class()) & = true \\ lfC^{P,V}(var.m(\overline{var\_or\_cte})) & = lfB^P(body(V(var),\ m,\ P)) \end{array}$$

onde $V(var)$ retorna o tipo de var, $body$ é a função que recebe um nome de classe $V(var)$, o nome de um método m, e um programa P, e retorna o corpo do método m definido em $V(var)$.

$$\begin{array}{ll} lfC^{P,V}(\texttt{this}.m(\overline{var\_or\_cte})) & = lfB^P(body(V(\texttt{this}),\ m,\ P)) \\ lfC^{P,V}(\texttt{throw}\ var) & = true \\ lfC^{P,V}(Command_1;\ Command_2) & = lfC^{P,V}(Command_1) \wedge \\ & \quad lfC^{P,V}(Command_2) \\ lfC^{P,V}(\texttt{if}(bool\_var)\{Command_1\}\ \texttt{else}\{Command_2\}) & = lfC^{P,V}(Command_1) \wedge \\ & \quad lfC^{P,V}(Command_2) \end{array}$$

Já a definição de lfC para o comando `while` deve verificar se o comando do corpo do mesmo é livre de *livelock* e se o *loop* termina. Para isso usamos o predicado $terminates(\texttt{while}(var > 0)\{Command\})$ que recebe um comando `while` e verifica se o mesmo termina. Assim, o predicado lfC para o comando `while` pode ser definido da seguinte maneira:

$$\begin{array}{l} lfC^{P,V}(\texttt{while}(var > 0)\ \{Command\}) = lfC^{P,V}(Command) \wedge \\ \qquad\qquad terminates(\texttt{while}(var > 0)\{Command\}) \end{array}$$

Na Seção 4.3 é mostrada a definição do predicado *terminates*, bem como das demais funções auxiliares utilizadas acima.

5.2 O Predicado *deadlockfree*

Semelhante ao predicado definido na seção anterior, o predicado $dfB^P(body(m))$ verifica se a execução do corpo de um método m não entra em *deadlock* quando executado no contexto de um programa P, e o predicado $dfC^{P,V}(c)$ verifica se o comando c não entra em *deadlock* quando executado no contexto do programa P.

Da mesma maneira que na seção anterior, onde o predicado *lfB* é definido em função de *lfC*, o predicado *dfB* é definido em função do predicado *dfC*. Por exemplo, a definição de *dfB* pode ser obtida substituindo *lfB* por *dfB* e *lfC* por *dfC*, nas equações da seção anterior, assim teremos

$$\begin{aligned} dfB^P(DecVars;\ Command;) &= dfC^{P,decs(VarsAttrbs)}(Command) \\ dfB^P(DecVars;\ Command;\ \texttt{return}\ var\_or\_cte;) &= dfC^{P,decs(VarsAttrbs)}(Command) \end{aligned}$$

Da mesma maneira definimos *dfC*, o qual difere de *lfC* apenas em dois casos, mostrados a seguir. No caso da chamada de um método, o predicado *dfC* deve verificar se todos os métodos da classe da variável utilizada para a chamada são livres de *deadlock*, caso contrário a chamada pode entrar em *deadlock*. Logo, sendo M o conjunto de métodos de $V(var)$, temos

$$dfC^{P,V}(var.m(\overline{var\_or\_cte})) = \forall\ i \in M \bullet\ dfB^P(body(V(var),\ i,\ P))$$

Note que esta equação não definiria o predicado caso haja métodos recursivos conforme ilustrado na seção 3.4 e proibidos pela arquitetura definida na seção 2. Já a definição de *dfC* para o comando `while` deve verificar apenas se o comando do corpo do `while` é livre de *deadlock*, ficando então definida como

$$dfC^{P,V}(\texttt{while}(var > 0)\ \{Command\}) = dfC^{P,V}(Command).$$

Devemos salientar que assumimos que as operações dos tipos definidos pela API de Java, bem como acesso a posições de *arrays* são livres de *livelock* e de *deadlock*.

6 Estudo de Caso

Nesta seção, mostramos um exemplo prático onde a regra definida na Seção 5, e os predicados *livelockfree* e *deadlockfree* são utilmente aplicados a um subconjunto simplificado do sistema SGC (Sistema de Gerência de Clientes) [6], mostrando que a regra e a definição dos predicados podem ser úteis para o desenvolvimento de outros sistemas reais, certamente não útil em geral, mas para uma classe considerável de sistemas. O SGC tem como funcionalidade principal fornecer serviços e informações a clientes sobre os seus telefones celulares via *browsers* Web. Iniciamos sincronizando uma Coleção de Dados, em seguida uma Coleção de Negócio e por fim a classe Fachada do sistema SGC.

6.1 Sincronizando uma Coleção de Dados

Considere a classe `RepositorioTelefonesArray`, uma Coleção de Dados, onde objetos do tipo `Telefone` são armazenados em um array de tamanho definido no momento da criação de um repositório[2]:

```
public class RepositorioTelefonesArray {
   private Telefone[] telefones;
   private int indice;
   private int proximo;
   public RepositorioTelefonesArray(int tamanho) {
      telefones = new Telefone[tamanho];
      indice = 0;
   }
```

onde `indice` e `proximo` implementam, respectivamente, ponteiros para a próxima posição livre do array e para a próxima posição do array durante uma varredura do mesmo. Telefones são armazenados no repositório através do método `insere`[3]

```
   public void insere(Telefone tel) {
      telefones[indice] = tel;
      indice = indice + 1;
   }
```

A classe `RepositorioTelefonesArray` que recebe um telefone e o armazena no array, também tem um método que verifica se existe um telefone com um dado número no repositório:

```
   public boolean consulta(Numero num) {
      boolean resposta = false;
      int i = 0;
      int y = indice;
      while(y > 0) {
         Numero numAux = telefones[i].getNumero();
```

[2]A versão final do sistema SGC, utiliza banco de dados relacional, o código apresentado foi usado apenas para prototipação.

[3]Por simplicidade, os códigos apresentados não são tão robustos quanto deveriam, neste caso, por exemplo, teríamos que verificar se o parâmetro passado é diferentes de `null`.

```
            String numeroAux = numAux.getNumero();
            String numero = num.getNumero();
            boolean achou = numero.equals(numeroAux);
            if (achou) {
               resposta = true;
            }
            i = i + 1;
            y = y - 1;
         }
         return resposta;
      }
```

onde o método `equals` de `String` compara-o a um `String` passado. Similarmente ao `consultar` o método `seleciona` recebe um número e retorna o telefone ou levanta uma exceção caso não exista um telefone com este número:

```
      public Telefone seleciona(Numero num)
         throws TelefoneNaoEncontradoException
      {
         TelefoneInvalidoException exception;
         int i = 0;
         int y = indice;
         while(y > 0) {
            Numero numAux = telefones[i].getNumero();
            String numeroAux = numAux.getNumero();
            String numero = num.getNumero();
            boolean achou = numero.equals(numeroAux);
            if (achou) {
               return telefones[i];
            }
            i = i + 1;
            y = y - 1;
         }
         exception = new TelefoneNaoEncontradoException();
         throw exception;
         }
      }
      ⋮
   }
```

Para sincronizar o método `seleciona`, devemos primeiro aplicar os predicados *livelockfree* e *deadlockfree*, segundo a regra definida na Seção 5, ao corpo do método para verificar se o mesmo pode ser sincronizado. Pela definição de *livelockfree* e *deadlockfree*, declarações e atribuições de variáveis, incremento de variáveis inteiras e levantamento de exceções são livres de *livelock* e de *deadlock*. O corpo de `seleciona` sem tais comandos se resume a:

```
   while(y > 0) {
```

```
    Numero numAux = telefones[i].getNumero();
    String numeroAux = numAux.getNumero();
    String numero = num.getNumero();
    boolean achou = numero.equals(numeroAux);
    if (achou) {
       return telefones[i];
    }
    i = i + 1;
    y = y - 1;
 }
```

Note que o primeiro comando do `while` é

```
Numero numeroAux = telefones[i].getNumero();
```

que é um comando mal formado segundo a BNF da linguagem MiniJava, mas é semanticamente equivalente a:

```
Numero numeroAux;
numeroAux = telefones[i].getNumero();
```

que é bem formado, considerando que o comando "`telefones[i]`" pode ser visto como uma variável do tipo `Telefone`. Logo pela definição de *livelockfree*, este primeiro comando é livre de *livelock* se o comando "`telefones[i].getNumero()`" também for.

Devemos então aplicar a o predicado *lfB* ao corpo do método `getNumero` da classe `Telefone`, a qual tem como atributos o número do telefone, e o propietário do mesmo:

```
public class Telefone  {
   private Numero numero;
   private Cliente cliente;
   public Telefone(Cliente cliente, Numero numero) {
      this.numero= numero;
      this.cliente = cliente;
   }
```

Esta classe também possui métodos para retornar e modificar o número e o propietário de um telefone

```
   public Numero getNumero() {
      return numero;
   }
   public Cliente getCliente() {
      return cliente;
   }
   public void setNumero(Numero numero) {
      this.numero = numero;
   }
   public void setCliente(Cliente cliente) {
      cliente = cliente;
```

```
    }
    ⋮
}
```

Como o método `getNumero` tem apenas um comando, "`return numero`", o qual pela definição de *lfC* é livre de *livelock*, podemos concluir que o método `getNumero` é livre de *livelock*.

Continuando a aplicar o predicado *lfB* ao comando `while` do método `seleciona` o próximo comando a considerar é:

```
String numeroAux = numAux.getNumero();
```

onde semelhante ao comando anterior, deveremos aplicar o predicado *lfB* ao corpo do método `getNumero` da classe `Numero`, que é o tipo do atributo `numAux`.

O passo seguinte é aplicar ao predicado *lfB* ao corpo do método `getNumero` da classe `Numero`, que possui como atributos o prefixo e o sufixo de um telefone, além de um método que retorna o número formado pelos mesmos:

```
public class Numero {
    private String prefixo;
    private String sufixo;
    public Numero(String pref, String suf) {
        prefixo = pref;
        sufixo = suf;
    }
    public String getNumero() {
        String numero = prefixo + sufixo;
        return numero;
    }
    ⋮
}
```

Pela definição de *lfB* o corpo do método `getNumero` é livre de *livelock*. O próximo comando do `while`, é a chamada ao método `equals`, mas como assumimos que os métodos de tipos definidos na API de Java são livres de *livelock* e de *deadlock*, a chamada é livre de *livelock*, uma vez que o mesmo não foi redefinido. O comando `if` também é livre de *livelock*, pois executa apenas o comando "`resposta = true`", assim como os comandos que incrementam as variáveis `i` e `y`. Temos ainda de verificar se o comando `while` termina, aplicando o predicado *terminates*, o qual é válido, uma vez que as condições impostas pelo predicado se verificam:

- A variável inteira, chamada de variante do *loop*, é `y`;
- o valor de `y` nunca é menor que 0, pois o mesmo é inicializado com o valor de indice, que é maior ou igual a zero, e a condição de parada do *loop* não permite que o valor de `y` seja menor que zero;
- `y` é estritamente decrescente a cada execução do corpo do *loop*, basta notar que o único comando que altera o valor de `y` decrementa o mesmo.

Tendo verificado que o método `selecionar` é livre de *livelock* o próximo passo, segundo a Seção 5, é verificar se o método é livre de *deadlock*.

Pela definição de *deadlockfree*, um método é livre de *deadlock* se o mesmo possui as mesmas restrições impostas pelo predicado *livelockfree*, com exceção em duas delas. No caso do comando *while*, a restrição para o mesmo ser livre de *deadlock* é mais fraca, pois não é necessário que o mesmo termine, logo tal condição já está verificada, a outra diferença na restrição é para chamadas de métodos da forma "$var.m(\overline{var\_or\_cte})$" onde deve-se verificar se cada método do tipo de *var* é livre de *deadlock*, note que esta verificação não contempla casos de redefinição do método em subclasses do tipo de *var*, veja mais detalhes na conclusão.

Assim, resta apenas verificar se o corpo dos métodos de `Telefone` e `Numero` são livres de *deadlock*. Pelas definições de `Telefone` e `Numero` mostradas nesta Seção, e pela definição do predicado *deadlockfree*, os métodos das duas classes são livres de *deadlock* já que os mesmos realizam apenas atribuições, incremento e retorno de variáveis.

Podemos então qualificar o método `seleciona` como `synchronized`, passando o mesmo a não ser mais executado em um objeto ao mesmo tempo por mais de um *thread*.

6.2 Sincronizando uma Coleção de Negócio

Nesta Seção sincronizamos um método de uma Coleção de Negócio do sistema SGC. Considere a classe `CadastroTelefones`, uma Coleção de Negócio, responsável por cadastrar os telefones do sistema SGC. Um objeto `CadastroTelefones` cadastra telefones em uma Colecção de Dados, `RepositorioTelefonesArray`, o qual é inicializado no momento da criação do cadastro.

```
public class CadastroTelefones {
   private RepositorioTelefonesArray telefones;
   public CadastroTelefones(int tamanho) {
      telefones = new RepositorioTelefonesArray(tamanho);
   }
```

Os objetos do tipo `CadastroTelefones` também dispõem de métodos para incluir e retornar um telefone cadastrado:

```
   public void incluirTelefone(Telefone telefone)
      throws PreCondicaoTelefoneException
   {
      PreCondicaoTelefoneException exception;
      Numero numero = telefone.getNumero();
      boolean existe = telefones.consulta(numero);
      if(existe) {
         exception = new PreCondicaoTelefoneException();
         throw exception;
      }
      telefones.insere(telefone);
   }
   public Telefone buscarTelefone(Numero numero)
```

```
        throws TelefoneInvalidoException
    {
        return telefones.seleciona(numero);
    }
    ⋮
}
```

onde o método `incluirTelefone` levanta uma exceção se existir um telefone com o mesmo número do telefone a ser cadastrado.

Para sincronizar o método `incluirTelefone` de `CadastroTelefones`, devemos primeiro aplicar os predicados *livelockfree* e *deadlockfree* ao corpo do método para verificar se o mesmo pode ser sincronizado. Desconsiderando as declarações, atribuições e incremento de variáveis, pois na Seção anterior mostramos que tais comandos são livres de *livelock* e de *livelock*, o primeiro comando do corpo de `incluirTelefone` é

```
telefone.getNumero();
```

Logo pela definição de *livelockfree*, devemos aplicar o predicado *lfB* ao corpo do método `getNumero` da classe `Telefone`, o qual é livre de *livelock* e de *deadlock*, conforme mostrado na Seção anterior.

Continuando a aplicar o predicado *lfB* ao método `incluirTelefone` o próximo comando, excluindo as declarações, atribuições e incremento de variáveis, é:

```
telefones.consulta(numero);
```

De forma semelhante ao comando anterior, devemos aplicar o predicado *lfB* ao corpo do método `consulta` de `RepositorioTelefonesArray`, que é o tipo do atributo `telefones`.

Excluindo do método `consulta` as declarações, atribuições e incremento de variáveis, o mesmo se resume aos comandos:

```
while(y > 0) {
    Numero numAux = telefones[i].getNumero();
    String numeroAux = numAux.getNumero();
    String numero = num.getNumero();
    boolean achou = numero.equals(numeroAux);
    if (achou) {
        resposta = true;
    }
    i = i + 1;
    y = y - 1;
}
```

os quais são livres de *livelock* e de *deadlock*, conforme mostrado na Seção anterior, quando verificamos se o método `seleciona` é livre de *livelock* e de *deadlock*.

De volta ao método `incluirTelefone` de `CadastroTelefones`, o próximo comando é um `if` que tem como comandos do seu corpo uma declaração de variável, a criação de um objeto e o lançamento de uma exceção pelo comando `throw`, os quais pela definição de *lfC* são livres de *livelock*. O último comando do corpo do método `incluirTelefone` é "`telefones.insere(telefone)`" o qual nos leva a verificar se o corpo do método `insere` de `RepositorioTelefonesArray` é livre de *livelock*.

O método `insere` tem como comandos uma atribuição e um incremento de variável, o que também é trivialmente livre de *livelock* segundo a definição de *lfC*.

Depois de verificarmos que `incluirTelefone` é livre de *livelock*, resta apena verificar se o mesmo também é livre de *deadlock*, para que possamos sincronizá-lo.

Pela definição de *deadlockfree*, devemos verificar se o corpo dos métodos de `RepositorioTelefonesArray`, `Telefone` e `Numero` são livres de *deadlock*, o que já foi mostrado na Seção anterior.

Logo, podemos concluir que o método `incluirTelefone` de `CadastroTelefones` pode ser sincronizado.

6.3 Sincronizando a Fachada

Esta Seção mostra a sincronização de um método da classe Fachada do sistema SGC. Esta classe tem como atributos Coleções de Negócio que cadastram os clientes e os telefones do sistema:

```
public class SGC {
   private CadastroClientes clientes;
   private CadastroTelefones telefones;
```

Assumindo que estas coleções são inicializadas pelo construtor da Fachada, o método para cadastrar um telefone seria definido da seguinte maneira:

```
   public void cadastraTelefone(Telefone telefone)
      throws PreCondicaoTelefoneException
   {
      telefones.incluirTelefone(telefone);
   }
```

Além destes métodos, a classe Fachada oferece serviços simplesmente invocando o serviço correspondente da Coleção de Negócio responsável, como, por exemplo, alterar número, localidade e serviços de um telefone.

```
   public void setLocalidadeNumeroTelefone(Numero numero,
      Localidade local, Numero novoNumero)
      throws TelefoneInvalidoException
   {
      Telefone fone;
      fone = telefones.buscarTelefone(numero);
      fone.setLocalidade(local);
      fone.setNumero(novoNumero);
   }
```

onde `Localidade` representa a área na qual o telefone está habilitado.

```
   public void solicitaServicosSuplementares(Numero numero,
      RepositorioServicos servicos) throws TelefoneInvalidoException,
```

```
        PreCondicaoServicoException
    {
        Telefone fone;
        Servico servico;
        fone = telefones.buscarTelefone(numero);
        servicos.iniciaBusca();
        servico = servicos.proximoObjeto();
        while (servico != null) {
            fone.incluiServico(servico);
            servico = servicos.proximoObjeto();
        }
    }
    ⋮
}
```

Observe que o método `cadastraTelefone` faz apenas uma chamada ao método `incluirTelefone` de `CadastroTelefones`, o qual verificamos, na Seção anterior, ser livre de *livelock* e de *deadlock*. isto garante que `cadastraTelefone` é livre de *livelock*, entretando, pela definição de *deadlockfree*, o método `cadastraTelefone` só é livre de *deadlock* se os demais métodos de `CadastroTelefones` também o forem. Portanto, devemos verificar se o método `buscarTelefone` de `CadastroTelefone` também é livre de *deadlock*.

O método `buscarTelefone` faz apenas uma chamada ao método `seleciona` de `RepositorioTelefonesArray` o qual é livre de *livelock* e de *deadlock*, como mostrado na Seção 6.1. Novamente, pela definição de *deadlockfree*, todos os métodos de `RepositorioTelefonesArray` tem que ser livre de *deadlock* para que o método `buscarTelefone` também o seja. Na Seção 6.2 foi feita esta verificação para sincronizar um método de `CadastroTelefones`.

Concluímos então que podemo sincronizar o método `incluirTelefone` tornando-o seguro para ser executado em um ambiente concorrente.

7 Conclusão

Como resultado deste trabalho, temos uma lei de refinamento formal que guia a sincronização de métodos, garantindo a segurança na execução concorrente dos mesmos, exemplos que mostram a necessidade deste tipo de lei, predicados que especificam as condições para a utilização da lei direcionados a facilitar a verificação e a automatização, e um estudo de caso onde a lei é aplicada a um sistema real, o que vem a ser uma contribuição na direção de obter um método para desenvolvimento sistemático de programas concorrentes orientados a objetos. Vale salientar que coube a este trabalho a definição de uma lei de sincronização, não fazendo parte do escopo do mesmo a verificação formal da lei aqui proposta.

Os predicados *livelockfree* e *deadlockfree* propostos neste trabalho possuem restrições fortes demais, como por exemplo as do comando condicional:

$$\begin{aligned} \mathit{lfC}^{P,V}(\mathtt{if}(\mathit{bool\_var})\{\mathit{Command_1}\}\ \mathtt{else}\{\mathit{Command_2}\}) \quad &= \mathit{lfC}^{P,V}(\mathit{Command_1}) \wedge \\ &\quad\ \mathit{lfC}^{P,V}(\mathit{Command_2}) \end{aligned}$$

que exigem que os dois comandos do condicional sejam livres de *livelock* e de *deadlock*, quando bastaria que apenas um dos comandos, o que fosse executado, fosse livre de *livelock* e de *deadlock*. Além disso, por outro lado, os predicados *livelockfree* e *deadlockfree* tem restrições fracas demais, pois, por exemplo, não se preocupam em garantir que todas as redefinições de um método via herança são livres de *livelock* e de *deadlock*. A intenção destes predicados não é definir *livelock* e *deadlock*, mas prover formas tão fáceis e computáveis quanto possível para verificar estes aspectos.

Apesar destas "limitações", os predicados *livelockfree* e *deadlockfree* podem ser bastante úteis, o que é mostrado na Seção 6, onde os mesmos são aplicados a um sistema real e não-trivial, o SGC, sincronizando alguns de seus métodos, facilitando a verificação das condições para a aplicação da lei proposta.

Podemos propor como trabalhos futuros refinamentos para as definições dos predicados, de modo a considerar redefinição de métodos. Tais atividades serão executadas em uma tese de mestrado dando continuidade a este trabalho, juntamente com a criação de regras para retirar o qualificador `synchronized` de métodos, em casos que os mesmos não são necessários, o que aumentará a eficiência do código, bem como a validação e possíveis refinamentos das mesmas.

A Prova Formal

Esta seção aplica formalmente a lei proposta neste trabalho ao sistema SGC, em contrapartida à Seção 6 onde são mostradas apenas justificativas informais. As classes e os métodos utilizados nesta seção são os mesmos da Seção 6, os quais também estão disponíveis no Apêncice B

Classes Básicas

Inicialmente verificamos se os predicados *deadlockfree* e *livelockfree* são válidos para os métodos das classes Básicas de Negócio `Telefone` e `Numero`. Onde $CSGC$ denota o conjunto de classes do sistema SGC (Classes SGC), logo

1

$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getNumero}\ , CSGC)\ )$
$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getCliente}\ , CSGC)\ )$
$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setNumero}\ , CSGC)\ )$
$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setCliente}\ , CSGC)\ )$

1.1

$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getNumero}\ , CSGC)\ )$
$= lfC^{CSGC,\ \emptyset}(\ \texttt{return numero}\ )$
$= true$

1.2

$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getCliente}\ , CSGC)\ )$
$= lfC^{CSGC,\ \emptyset}(\ \texttt{return cliente}\ )$
$= true$

1.3

$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setNumero}\ , CSGC)\ )$
$= lfC^{CSGC,\ \emptyset}(\ \texttt{this.numero = numero}\ )$
$= true$

1.4

$lfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setCliente}\ , CSGC)\ )$
$= lfC^{CSGC,\ \emptyset}(\ \texttt{this.cliente = cliente}\ )$
$= true$

2

$dfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getNumero}\ , CSGC)\ )$
$dfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getCliente}\ , CSGC)\ )$
$dfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setNumero}\ , CSGC)\ )$
$dfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{setCliente}\ , CSGC)\ )$

2.1

$dfB^{CSGC}(\ body(\texttt{Telefone}\ ,\ \texttt{getNumero}\ , CSGC)\ )$

$= dfC^{CSGC,\ \emptyset}$ (`return numero`)
$= true$

2.2
dfB^{CSGC}($body$(`Telefone` , `getCliente` ,$CSGC$))
$= dfC^{CSGC,\ \emptyset}$ (`return cliente`)
$= true$

2.3
dfB^{CSGC}($body$(`Telefone` , `setNumero` ,$CSGC$))
$= dfC^{CSGC,\ \emptyset}$ (`this.numero = numero`)
$= true$

2.4
dfB^{CSGC}($body$(`Telefone` , `setCliente` ,$CSGC$))
$= dfC^{CSGC,\ \emptyset}$ (`this.cliente = cliente`)
$= true$

3
lfB^{CSGC}($body$(`Numero` , `getNumero` ,$CSGC$))

3.1
lfB^{CSGC}($body$(`Numero` , `getNumero` ,$CSGC$))
$= lfC^{CSGC,\ decsMap1}$ (`numero = prefixo + sufixo; return numero`)
$= lfC^{CSGC,\ decsMap1}$ (`numero = prefixo + sufixo`)
$\wedge$
$lfC^{CSGC,\ decsMap1}$ (`return numero`)
$= true$
$\wedge$
$true$

onde $decsMap1$ é o resultado da função $decs$ aplicada as declarações de variáveis do método `getNumero` da classe `Numero` mais declarações de atributos da classe `Numero` e a declaração de uma variável `this` do tipo `Numero`, que neste caso seria {`numero` $\mapsto$ `String`, `prefixo` $\mapsto$ `String`, `sufixo` $\mapsto$ `String`, `this` $\mapsto$ `Numero` }

4
dfB^{CSGC}($body$(`Numero` , `getNumero` ,$CSGC$))

4.1
dfB^{CSGC}($body$(`Numero` , `getNumero` ,$CSGC$))
$= dfC^{CSGC,\ decsMap1}$ (`numero = prefixo + sufixo; return numero`)
$= dfC^{CSGC,\ decsMap1}$ (`numero = prefixo + sufixo`)
$\wedge$
$dfC^{CSGC,\ decsMap1}$ (`return numero`)
$= true$
$\wedge$

$true$

Como assumimos que as operações dos tipos definidos pela API de Java são livres de *livelock* e de *deadlock*, logo, podemos concluir que todos os métodos das classes `Telefone` e `Numero` são livres de *livelock* e de *deadlock*.

Coleção de Dados

Agora verificamos os predicados *deadlockfree* e *livelockfree* para os métodos da Classe Coleção de Dados `RepositorioTelefonesArray`. Como dito na seção 5.2, assumimos que acessos a posições de um *arrays* são livres de *livelock* e de *deadlock*.

5

$lfB^{CSGC}($ $body($`RepositorioTelefonesArray` , `consulta` $,CSGC)$ $)$
$lfB^{CSGC}($ $body($`RepositorioTelefonesArray` , `insere` $,CSGC)$ $)$
$lfB^{CSGC}($ $body($`RepositorioTelefonesArray` , `seleciona` $,CSGC)$ $)$

5.1

$= lfB^{CSGC}($ $body($`RepositorioTelefonesArray` , `consulta` $,CSGC)$ $)$
$= lfC^{CSGC,\ decsMap2}($ `resposta = false; i = 0; y = indice;`
`while(y > 0) { wbody1 }; return false` $)$
$= lfC^{CSGC,\ decsMap2}($ `resposta = false` $)$
$\wedge$
$lfC^{CSGC,\ decsMap2}($ `i = 0` $)$
$\wedge$
$lfC^{CSGC,\ decsMap2}($ `y = indice` $)$
$\wedge$
$lfC^{CSGC,\ decsMap2}($ `while(y > 0) { wbody1 }` $)$
$\wedge$
$lfC^{CSGC,\ decsMap2}($ `return false` $)$

onde *decsMap2* é o resultado da função *decs* aplicada as declarações de variáveis do método `consulta` da classe `RepositorioTelefonesArray` mais declarações de atributos da classe `RepositorioTelefoneArray` e a declaração de uma variável `this` do tipo `RepositorioTelefonesArray`. O corpo do `while` denotado por `wbody1` é

```
Numero numAux = telefones[i].getNumero();
String numeroAux = numAux.getNumero();
String numero = num.getNumero();
boolean achou = numero.equals(numeroAux);
if (achou) {
   resposta = true;
}
i = i + 1;
y = y - 1;
```

5.1.1

$lfC^{CSGC,\ decsMap2}($ `resposta = false` $) = true$

5.1.2

$lfC^{CSGC,\ decsMap2}($ `i = 0` $) = true$

5.1.3

$lfC^{CSGC,\ decsMap2}($ `y = indice` $) = true$

5.1.4

$lfC^{CSGC,\ decsMap2}($ `while(y > 0) { wbody1 }` $)$

$= lfC^{CSGC,\ decsMap2}($ `telefones[i].getNumero()` $)$

$\wedge$

$lfC^{CSGC,\ decsMap2}($ `numeroAux.getNumero()` $)$

$\wedge$

$lfC^{CSGC,\ decsMap2}($ `numero.equals(numero2)` $)$

$\wedge$

$lfC^{CSGC,\ decsMap2}($ `if (achou) { return true; }` $)$

$\wedge$

$lfC^{CSGC,\ decsMap2}($ `i = i + 1` $)$

$\wedge$

$lfC^{CSGC,\ decsMap2}($ `y = y - 1` $)$

$\wedge$

$terminates($ `while(y > 0) { wbody1 }` $)$

5.1.4.1

$lfC^{CSGC,\ decsMap2}($ `telefones[i].getNumero()` $)$

$= lfB^{CSGC}(\ body($`Telefone` , `getNumero` $, CSGC)\ )$

$= true$ (por 1.1)

considerando que o acesso a uma posição do array pode ser considerada como uma variável do tipo do mesmo, neste caso de `Telefone`.

5.1.4.2

$lfC^{CSGC,\ decsMap2}($ `numeroAux.getNumero()` $)$

$= lfB^{CSGC}(\ body($`Numero` , `getNumero` $, CSGC)\ )$

$= true$ (por 3.1)

5.1.4.3

$lfC^{CSGC,\ decsMap2}($ `numero.equals(numero2)` $)$

$= lfB^{CSGC}(\ body($`String` , `equals` $, CSGC)\ )$

$= true$

novamente considerando que as operações dos tipos definidos pela API de Java são livres de *livelock* e de *deadlock*

5.1.4.4

$lfC^{CSGC,\ decsMap2}($ `if (achou) { return true; }` $)$

$= lfC^{CSGC,\ decsMap2}($ `return true` $)$

$= true$

5.1.4.5

$lfC^{CSGC,\ decsMap2}($ `i = i + 1` $)$
$= true$

5.1.4.6

$lfC^{CSGC,\ decsMap2}($ `y = y - 1` $)$
$= true$

5.1.4.7

$terminates($ `while(y > 0) { wbody1 }` $)$
$= true$

pois

- A variável inteira, chamada de variante do *loop* é `y`;
- o valor de `y` nunca é menor que 0;
- `y` é estritamente decrescente a cada execução do corpo do *loop*;

5.1.5

$lfC^{CSGC,\ decsMap2}($ `return false` $)$
$= true$

5.2

$lfB^{CSGC}(\ body($`RepositorioTelefonesArray` , `insere` $,CSGC)\ )$
$= lfC^{CSGC,\ decsMap3}($ `telefones[indice] = tel; indice = indice + 1` $)$
$= lfC^{CSGC,\ decsMap3}($ `telefones[indice] = tel` $)$
$\wedge$
$lfC^{CSGC,\ decsMap3}($ `indice = indice + 1` $)$
$= true$
$\wedge$
$true$

onde *decsMap3* é o resultado da função *decs* aplicada as declarações de variáveis do método `insere` da classe `RepositorioTelefonesArray` mais declarações de atributos da classe `RepositorioTelefoneArray` e a declaração de uma variável `this` do tipo `RepositorioTelefonesArray`.

5.3

$lfB^{CSGC}(\ body($`RepositorioTelefonesArray` , `seleciona` $,CSGC)\ )$
$= lfC^{CSGC,\ decsMap4}($ `i = 0` $)$
$\wedge$
$lfC^{CSGC,\ decsMap4}($ `y = indice` $)$
$\wedge$
$lfC^{CSGC,\ decsMap4}($ `while(y > 0) { wbody2 }` $)$
$\wedge$
$lfC^{CSGC,\ decsMap4}($ `return false` $)\ )$

onde *decsMap4* é o resultado da função *decs* aplicada as declarações de variáveis do método `seleciona` da classe `RepositorioTelefonesArray` mais declarações de atributos da classe `RepositorioTelefoneArray` e a declaração de uma variável `this` do tipo `RepositorioTelefonesArray`. O corpo do `while` denotado por `wbody2` é

```
Numero numAux = telefones[i].getNumero();
String numeroAux = numAux.getNumero();
String numero = num.getNumero();
boolean achou = numero.equals(numeroAux);
if (achou) {
   return telefones[i];
}
i = i + 1;
y = y - 1;
```

5.3.1

$lfC^{CSGC,\ decsMap4}($ `i = 0` $)$

$= true$

5.3.2

$lfC^{CSGC,\ decsMap4}($ `y = indice` $)$

$= true$

5.3.3

$lfC^{CSGC,\ decsMap4}($ `while(y > 0) { wbody2 }` $)$

$= lfC^{CSGC,\ decsMap4}($ `telefones[i].getNumero()` $)$

$\wedge$

$lfC^{CSGC,\ decsMap4}($ `numeroAux.getNumero()` $)$

$\wedge$

$lfC^{CSGC,\ decsMap4}($ `numero.equals(numero2)` $)$

$\wedge$

$lfC^{CSGC,\ decsMap4}($ `if (achou) { return telefones[i]; }` $)$

$\wedge$

$lfC^{CSGC,\ decsMap4}($ `i = i + 1` $)$

$\wedge$

$lfC^{CSGC,\ decsMap4}($ `y = y - 1` $)$

$\wedge$

terminates(`while(y > 0) { wbody2 }` *)*

5.3.3.1

$lfC^{CSGC,\ decsMap4}($ `telefones[i].getNumero()` $)$

$= lfB^{CSGC}($ *body(*`Telefone` , `getNumero` ,*CSGC))*

$= true$ (por 1.1)

5.3.3.2

$lfC^{CSGC,\ decsMap4}($ `numeroAux.getNumero()` $)$

$= lfB^{CSGC}($ *body(*`Numero` , `getNumero` *,CSGC))*
$= true$ `(por 3.1)`

`5.3.3.3`
$lfC^{CSGC,\ decsMap4}($ `numero.equals(numero2)` $)$
$= true$ `(por 5.1.4.3)`

`5.3.3.4`
$lfC^{CSGC,\ decsMap4}($ `if (achou) { return telefones[i]; }` $)$
$= true$

`5.3.3.5`
$lfC^{CSGC,\ decsMap4}($ `i = i + 1` $)$
$= true$

`5.3.3.6`
$lfC^{CSGC,\ decsMap4}($ `y = y - 1` $)$
$= true$

`5.3.3.7`
terminates(`while(y > 0) { wbody2 }` *)*
$= true$

pois

- A variável inteira, chamada de variante do *loop* é `y`;
- o valor de `y` nunca é menor que 0;
- `y` é estritamente decrescente a cada execução do corpo do *loop*;

`5.3.4`
$lfC^{CSGC,\ decsMap4}($ `return false` $)$
$= true$

`6`
$dfB^{CSGC}($ *body(*`RepositorioTelefonesArray` , `consulta` *,CSGC))*
$dfB^{CSGC}($ *body(*`RepositorioTelefonesArray` , `insere` *,CSGC))*
$dfB^{CSGC}($ *body(*`RepositorioTelefonesArray` , `seleciona` *,CSGC))*

`6.1`
$dfB^{CSGC}($ *body(*`RepositorioTelefonesArray` , `consulta` *,CSGC))*
$= dfC^{CSGC,\ decsMap2}($ `resposta = false; i = 0; y = indice;`
`while(y > 0) { wbody1 }; return false` $)$
$= dfC^{CSGC,\ decsMap2}($ `resposta = false` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `i = 0` $)$

$\wedge$
$dfC^{CSGC,\ decsMap2}($ `y = indice` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `while(y > 0) { wbody1 }` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `return false` $)$

6.1.1
$dfC^{CSGC,\ decsMap2}($ `resposta = false` $) = true$

6.1.2
$dfC^{CSGC,\ decsMap2}($ `i = 0` $) = true$

6.1.3
$dfC^{CSGC,\ decsMap2}($ `y = indice` $) = true$

6.1.4
$dfC^{CSGC,\ decsMap2}($ `while(y > 0) wbody1 ... }` $)$
$= dfC^{CSGC,\ decsMap2}($ `telefones[i].getNumero()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `numeroAux.getNumero()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `numero.equals(numero2)` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `if (achou) { return true; }` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `i = i + 1` $)$
$\wedge$
$dfC^{CSGC,\ decsMap2}($ `y = y - 1` $)$

6.1.4.1
$dfC^{CSGC,\ decsMap2}($ `telefones[i].getNumero()` $)$
$= dfB^{CSGC}(\ body($`Telefone` , `getNumero` $, CSGC)\ )$
$= true$ (por 2.1)

6.1.4.2
$dfC^{CSGC,\ decsMap2}($ `numeroAux.getNumero()` $)$
$= dfB^{CSGC}(\ body($`Numero` , `getNumero` $, CSGC)\ )$
$= true$ (por 4.1)

6.1.4.3
$dfC^{CSGC,\ decsMap2}($ `numero.equals(numero2)` $)$
$= dfB^{CSGC}(\ body($`String` , `equals` $, CSGC)\ )$
$= true$ (por 5.1.4.3)

6.1.4.4

$dfC^{CSGC,\ decsMap2}($ `if (achou) { return true; }` $)$
$= dfC^{CSGC,\ decsMap2}($ `return true` $)$
$= true$

6.1.4.5
$dfC^{CSGC,\ decsMap2}($ `i = i + 1` $)$
$= true$

6.1.4.6
$dfC^{CSGC,\ decsMap2}($ `y = y - 1` $)$
$= true$

6.1.5
$dfC^{CSGC,\ decsMap2}($ `return false` $)$
$= true$

6.2
$dfB^{CSGC}(\ body($`RepositorioTelefonesArray` , `insere` $,CSGC)\ )$
$= dfC^{CSGC,\ decsMap3}($ `telefones[indice] = tel; indice = indice + 1` $)$
$= dfC^{CSGC,\ decsMap3}($ `telefones[indice] = tel` $)$
$\wedge$
$dfC^{CSGC,\ decsMap3}($ `indice = indice + 1` $)$
$= true$
$\wedge$
$true$

6.3
$dfB^{CSGC}(\ body($`RepositorioTelefonesArray` , `seleciona` $,CSGC)\ )$
$= dfC^{CSGC,\ decsMap4}($ `i = 0` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `y = indice` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `while(y > 0) wbody2 ... }` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `return false` $)\ )$

6.3.1
$dfC^{CSGC,\ decsMap4}($ `i = 0` $)$
$= true$

6.3.2
$dfC^{CSGC,\ decsMap4}($ `y = indice` $)$
$= true$

6.3.3
$dfC^{CSGC,\ decsMap4}($ `while(y > 0) { wbody2 }` $)$

$= dfC^{CSGC,\ decsMap4}($ `telefones[i].getNumero()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `numeroAux.getNumero()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `numero.equals(numero2)` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `if (achou) { return telefones[i]; }` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `i = i + 1` $)$
$\wedge$
$dfC^{CSGC,\ decsMap4}($ `y = y - 1` $)$

`6.3.3.1`
$dfC^{CSGC,\ decsMap4}($ `telefones[i].getNumero()` $)$
$= dfB^{CSGC}($ $body($`Telefone` , `getNumero` $, CSGC))$
$= true$ `(por 2.1)`

`6.3.3.2`
$dfC^{CSGC,\ decsMap4}($ `numeroAux.getNumero()` $)$
$= dfB^{CSGC}($ $body($`Numero` , `getNumero` $, CSGC))$
$= true$ `(por 4.1)`

`6.3.3.3`
$dfC^{CSGC,\ decsMap4}($ `numero.equals(numero2)` $)$
$= true$ `(por 5.1.4.3)`

`6.3.3.4`
$dfC^{CSGC,\ decsMap4}($ `if (achou) { return telefones[i]; }` $)$
$= true$

`6.3.3.5`
$dfC^{CSGC,\ decsMap4}($ `i = i + 1` $)$
$= true$

`6.3.3.6`
$dfC^{CSGC,\ decsMap4}($ `y = y - 1` $)$
$= true$

`6.3.4`
$dfC^{CSGC,\ decsMap4}($ `return false` $)$
$= true$

Portanto, podemos concluir que os métodos `RepositorioTelefonesArray` são livres de *livelock* e de *deadlock*.

Coleção de Negócio

Agora aplicaremos os predicados *livelockfree* e *deadlockfree* aos métodos da classe Coleção de Negócio `CadastroTelefones`.

7

$lfB^{CSGC}($ $body($`CadastroTelefones` , `incluirTelefone` $,CSGC)$ $)$
$lfB^{CSGC}($ $body($`CadastroTelefones` , `buscarTelefone` $,CSGC)$ $)$

7.1

$lfB^{CSGC}($ $body($`CadastroTelefones` , `incluirTelefone` $,CSGC)$ $)$
$= lfC^{CSGC,\ decsMap5}($ `telefone.getNumero()` $)$
$\wedge$
$lfC^{CSGC,\ decsMap5}($ `telefones.consulta(numero)` $)$
$\wedge$
$lfC^{CSGC,\ decsMap5}($ `if(existe) { ... }` $)$
$\wedge$
$lfC^{CSGC,\ decsMap5}($ `telefones.insere(telefone)` $)$

onde *decsMap5* é o resultado da função *decs* aplicada as declarações de variáveis do método `incluirTelefone` da classe `CadastroTelefones` mais declarações de atributos da classe `CadastroTelefones` e a declaração de uma variável `this` do tipo `CadastroTelefones`.

7.1.1

$lfC^{CSGC,\ decsMap5}($ `telefone.getNumero()` $)$
$= true$ (por 1.1)

7.1.2

$lfC^{CSGC,\ decsMap5}($ `telefones.consulta(numero)` $)$
$= true$ (por 5.1)

7.1.3

$lfC^{CSGC,\ decsMap5}($ `if(existe) { ... }` $)$
$= lfC^{CSGC,\ decsMap5}($ `exception = new PreCondicaoTelefoneException()` $)$
$\wedge$
$lfC^{CSGC,\ decsMap5}($ `throw exception` $)$
$= true$
$\wedge$
$true$

7.1.4

$lfC^{CSGC,\ decsMap5}($ `telefones.insere(telefone)` $)$
$= true$ (por 5.2)

7.2

$lfB^{CSGC}($ $body($`CadastroTelefones` , `buscarTelefone` $,CSGC)$ $)$
$= lfC^{CSGC,\ decsMap6}($ `telefones.seleciona(numero)` $)$
$= true$ (por 5.3)

onde *decsMap5* é o resultado da função *decs* aplicada as declarações de variáveis do método `burcarTelefone` da classe `CadastroTelefones` mais declarações de atributos da classe `CadastroTelefones` e a declaração de uma variável `this` do tipo `CadastroTelefones`.

8

$dfB^{CSGC}($ *body(*`CadastroTelefones` , `incluirTelefone` ,$CSGC)$ $)$
$dfB^{CSGC}($ *body(*`CadastroTelefones` , `buscarTelefone` ,$CSGC)$ $)$

8.1

$dfB^{CSGC}($ *body(*`CadastroTelefones` , `incluirTelefone` ,$CSGC)$ $)$
$= dfC^{CSGC,\ decsMap5}($ `telefone.getNumero()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap5}($ `telefones.consulta(numero)` $)$
$\wedge$
$dfC^{CSGC,\ decsMap5}($ `if(existe) { ... }` $)$
$\wedge$
$dfC^{CSGC,\ decsMap5}($ `telefones.insere(telefone)` $)$

8.1.1

$dfC^{CSGC,\ decsMap5}($ `telefone.getNumero()` $)$
$= true$ `(por 2.1)`

8.1.2

$dfC^{CSGC,\ decsMap5}($ `telefones.consulta(numero)` $)$
$= true$ `(por 6.1)`

8.1.3

$dfC^{CSGC,\ decsMap5}($ `if(existe) { ... }` $)$
$= dfC^{CSGC,\ decsMap5}($ `exception = new PreCondicaoTelefoneException()` $)$
$\wedge$
$dfC^{CSGC,\ decsMap5}($ `throw exception` $)$
$= true$
$\wedge$
$true$

8.1.4

$dfC^{CSGC,\ decsMap5}($ `telefones.insere(telefone)` $)$
$= true$ `(por 6.2)`

8.2

$dfB^{CSGC}($ *body(*`CadastroTelefones` , `buscarTelefone` ,$CSGC)$ $)$
$= dfC^{CSGC,\ decsMap6}($ `telefones.seleciona(numero)` $)$
$= true$ `(por 6.3)`

Logo, concluímos que a classe `CadastroTelefones` é livre de *livelock* e de *deadlock*.

Fachada

Aplicaremos a lei de sincronização na classe Fachada `SGC`. Podemos sincronizar o método `cadastraTelefone` de `SGC` se:

9

$$lfB^{CSGC}(\ body(\texttt{SGC},\ \texttt{cadastraTelefone}, CSGC)\ )$$
$$\wedge$$
$$dfB^{CSGC}(\ body(\texttt{SGC},\ \texttt{cadastraTelefone}, CSGC)\ )$$

9.1

$$\begin{aligned} & lfB^{CSGC}(\ body(\texttt{SGC},\ \texttt{cadastraTelefone}, CSGC)\ ) \\ & \quad = lfC^{CSGC,\ decsMap7}(\ \texttt{telefones.incluirTelefone(telefone)}\ ) \\ & \quad = lfB^{CSGC}(\ body(\texttt{CadastroTelefones},\ \texttt{incluirTelefone}, CSGC)\ ) \\ & \quad = true\ \texttt{(por 7.1)} \end{aligned}$$

onde *decsMap7* é o resultado da função *decs* aplicada as declarações de variáveis do método `cadastraTelefone` da classe `SGC` mais declarações de atributos da classe `SGC` e a declaração de uma variável `this` do tipo `SGC`.

9.2

$$\begin{aligned} & dfB^{CSGC}(\ body(\texttt{SGC},\ \texttt{cadastraTelefone}, CSGC)\ ) \\ & \quad = dfC^{CSGC,\ decsMap7}(\ \texttt{telefones.incluirTelefone(telefone)}\ ) \\ & \quad = dfB^{CSGC}(\ body(\texttt{CadastroTelefones},\ \texttt{incluirTelefone}, CSGC)\ ) \\ & \quad = true\ \texttt{(por 8.1)} \end{aligned}$$

Logo, podemos concuir que o método `cadastraTelefone` de `SGC` é livre de *livelock* e de *deadlock*, e portanto pode ser qualificado como `synchronized` estando assim apto a executar comsegurança em um ambiente concorrente.

B Classes do SGC

Este apêndice listas as implementações das classes utilizadas neste trabalho. Devemos mais uma vez salientar que as mesmas sofreram simplificações para simplificar a definição da lei de sincronização e dos predicados.

SGC

```
public class SGC {
   private CadastroClientes clientes;
   private CadastroTelefones telefones;
   public void cadastraTelefone(Telefone telefone)
      throws PreCondicaoTelefoneException
   {
      telefones.incluirTelefone(telefone);
   }
   public void setLocalidadeNumeroTelefone(Numero numero,
      Localidade local, Numero novoNumero)
      throws TelefoneInvalidoException
   {
      Telefone fone;
      fone = telefones.buscarTelefone(numero);
      fone.setLocalidade(local);
      fone.setNumero(novoNumero);
   }
   public void solicitaServicosSuplementares(Numero numero,
      RepositorioServicos servicos) throws TelefoneInvalidoException,
      PreCondicaoServicoException
   {
      Telefone fone;
      Servico servico;
      fone = telefones.buscarTelefone(numero);
      servicos.iniciaBusca();
      servico = servicos.proximoObjeto();
      while (servico != null) {
         fone.incluiServico(servico);
         servico = servicos.proximoObjeto();
      }
   }
}
```

CadastroTelefones

```
public class CadastroTelefones {
   private RepositorioTelefonesArray telefones;
   public CadastroTelefones(int tamanho) {
      telefones = new RepositorioTelefonesArray(tamanho);
```

```
    }
    public void incluirTelefone(Telefone telefone)
       throws PreCondicaoTelefoneException
    {
       PreCondicaoTelefoneException exception;
       Numero numero = telefone.getNumero();
       boolean existe = telefones.consulta(numero);
       if(existe) {
          exception = new PreCondicaoTelefoneException();
          throw exception;
       }
       telefones.insere(telefone);
    }
    public Telefone buscarTelefone(Numero numero)
       throws TelefoneInvalidoException
    {
       return telefones.seleciona(numero);
    }
 }
```

RepositorioTelefonesArray

```
 public class RepositorioTelefonesArray {
    private Telefone[] telefones;
    private int indice;
    private int proximo;
    public RepositorioTelefonesArray(int tamanho) {
       telefones = new Telefone[tamanho];
       indice = 0;
    }
    public void insere(Telefone tel) {
       telefones[indice] = tel;
       indice = indice + 1;
    }
    public boolean consulta(Numero num) {
       boolean resposta = false;
       int i = 0;
       int y = indice;
       while(y > 0) {
          Numero numAux = telefones[i].getNumero();
          String numeroAux = numAux.getNumero();
          String numero = num.getNumero();
          boolean achou = numero.equals(numeroAux);
          if (achou) {
             resposta = true;
          }
          i = i + 1;
```

```
            y = y - 1;
         }
         return resposta;
      }
      public Telefone seleciona(Numero num)
         throws TelefoneNaoEncontradoException
      {
         TelefoneInvalidoException exception;
         int i = 0;
         int y = indice;
         while(y > 0) {
            Numero numAux = telefones[i].getNumero();
            String numeroAux = numAux.getNumero();
            String numero = num.getNumero();
            boolean achou = numero.equals(numeroAux);
            if (achou) {
               return telefones[i];
            }
            i = i + 1;
            y = y - 1;
         }
         exception = new TelefoneNaoEncontradoException();
         throw exception;
         }
      }
   }
```

Telefone

```
   public class Telefone  {
      private Numero numero;
      private Cliente cliente;
      public Telefone(Cliente cliente, Numero numero) {
         this.numero= numero;
         this.cliente = cliente;
      }
      public Numero getNumero() {
         return numero;
      }
      public Cliente getCliente() {
         return cliente;
      }
      public void setNumero(Numero numero) {
         this.numero = numero;
      }
      public void setCliente(Cliente cliente) {
         cliente = cliente;
```

```
    }
}
```

Numero

```java
public class Numero {
    private String prefixo;
    private String sufixo;
    public Numero(String pref, String suf) {
        prefixo = pref;
        sufixo = suf;
    }
    public String getNumero() {
        String numero = prefixo + sufixo;
        return numero;
    }
}
```

Referências

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

[2] Paulo Borba. Systematic development of concurrent object-oriented programs. In *US–Brazil Joint Workshops on the Formal Foundations of Software Systems*, volume 14 of *Electronic Notes in Theoretical Computer Science.* Rio de Janeiro, Brazil, 5-9th May, 1997, and New Orleans, USA, 13-16th November, 1997.

[3] Paulo Borba. Desenvolvimento sistemático de programas concorrentes orientados a objetos, 1997.

[4] Paulo Borba. Where are the laws of object-oriented programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[5] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed java applications. In *Engineering Distributed Objects*, pages 40–47, Los Angeles, California, EUA, 17th–18th May 1999.

[6] Paulo Borba, André Santos, Denise Neves, Marconi Lima, Marcelo Faro e Sérgio Soares. *Sistema de Gerência de Clientes.* Primeiro Concurso Nacional de Software de Telecomunicações, promovido pela ADETEC e Softex, Londrina, Brasil, Março 1998.

[7] George Cleland and Donald MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *International Workshop on Industrial Strength Formal Specification Techniques.* IEEE Computer Society Press, Boca Raton, Florida, April 1995.

[8] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[10] Cliff Jones. Does the O-O community need formal methods? In *TOOLS4: Technology of O-O Languages and Systems*, pages 15–22. Prentice-Hall, 1991.

[11] Cliff Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[12] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[13] Silvio Meira and Ana Cavalcanti. Modular object-oriented Z specifications. In Prof. C.J. van Rijsbergen, editor, *Workshop on Computing Series.* Springer-Verlag, Oxford, December 1990.

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

[15] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[16] Sérgio Soares e Paulo Borba. *Leis de Sincronização.* In *II Brazilian Workshop on Formal Methods*, Florianópolis, Santa Catarina, Brasil, Outubro 1999. Submetido.

[17] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architectures—Analysis and Design of Reliable Systems.* Prentice-Hall, 1995.

Figura 1: Arquitetura em Camadas