UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

VERIFICANDO REFATORAÇÕES AUTOMATICAMENTE

TRABALHO DE GRADUAÇÃO

Aluno: José Olino de Campos Lima Junior (joclj@cin.ufpe.br)

Orientador: Alexandre Cabral Mota (acm@cin.ufpe.br)

Setembro de 2006

aos meus pais

Agradecimentos

Agradeço aos meus pais, por terem me proporcionado a oportunidade de estudar o que gosto e na cidade que escolhi.

A Clarice, minha namorada, pela paciência e compreensão.

Ao meu orientador, o professor Alexandre Mota, pelos preciosos ensinamentos.

A Joabe, pelas discussões enriquecedoras.

À professora Edna Barros e ao professor Paulo Borba que, além do excelente trabalho como coordenadores dos cursos de Computação, me orientaram em momentos difíceis que passei no início do curso.

Aos meus amigos, especialmente Bruno, Diego, Turah, Adson, Diogo, Victor, André, Francielle, Patrícia, Tiago, César, Evilásio, Márcio e Luiz Fernando, pelo companheirismo e apoio prestado.

E a todas as pessoas contribuíram com este trabalho, meus sinceros agradecimentos.

Resumo

A refatoração de códigos é uma técnica muito difundida e utilizada em muitas metodologias de desenvolvimento de software. Ela consiste na reestruturação de um código (ou trecho de código) sem alterar sua funcionalidade, para atingir um determinado objetivo, que pode ser um ganho na legibilidade, extensibilidade, modularidade, dentre outros. Porém, esta transformação de código pode introduzir erros, por ser baseada geralmente em processos informais.

Com o objetivo de tentar garantir a preservação da funcionalidade do sistema após a aplicação de um refatoramento, faz-se uso intensivo de testes de software. O problema desta abordagem é que não há uma suíte de testes que garantam a preservação da funcionalidade, exceto em casos muito simples, além de que determinar a suíte de testes suficiente é tarefa não trivial.

Este trabalho propõe uma nova abordagem para verificar a refatoração de códigos: capturar a funcionalidade (comportamento) do programa antes e depois de aplicar um refatoramento através de um modelo descrito numa linguagem formal. Então, podemos verificar se a funcionalidade foi preservada usando uma técnica denominada de verificação de refinamentos. A eficiência desta nova abordagem pode ser obtida com a automatização da criação do modelo formal e com a verificação de refinamentos referente ao modelo, a qual é completamente automática. As duas principais vantagens desta abordagem são: o programador não precisa se preocupar em criar uma suíte de testes e a verificação de refinamentos garante a preservação da funcionalidade, ao invés de simplesmente constatar que em alguns casos isto se dá, como no caso de testes de software.

Como forma de ilustrar nossa abordagem, consideramos um estudo de caso onde refatoramentos são aplicados em um programa inicial e a verificação

de refinamentos é usada para constatar que houve ou não preservação da funcionalidade.

Índice

Índice de figuras

1. Introdução

Nesta seção apresentaremos o contexto no qual este trabalho está inserido, assim como o problema atacado. Então, apresentaremos os objetivos do nosso trabalho e, logo após, uma visão geral deste documento.

1.1 Contexto

As plataformas de hardware apresentaram um enorme aumento no poder computacional nas últimas décadas. Os avanços da engenharia eletrônica (no sentido de obter uma maior miniaturização dos componentes) foram grandes e proporcionaram a viabilização da comercialização de poderosas plataformas de computador.

Com o surgimento dos poderosos computadores de terceira geração [13], soluções de software de porte até então inimagináveis puderam ser concebidas. Estas soluções passaram a representar uma proporção cada vez maior dos custos finais de sistemas baseados em computador. Pudemos observar um enorme crescimento no porte e complexidade destas soluções. Os prazos para a conclusão dos projetos foram estendidos e as equipes também ficaram maiores.

A observação do fracasso de um grande número de projetos ao longo da história levou os especialistas da época a questionar a informalidade dos processos de desenvolvimento de software. É grande o número de casos em que sistemas críticos[1] de grande porte apresentaram falhas com conseqüências desastrosas:

- Em 1991 [4], o sistema de telefonia da Califórnia e a região ao longo da costa leste dos Estados Unidos saiu do ar pouco depois de uma pequena

[1] Sistemas onde vidas humanas ou grandes quantidades de dinheiro estão envolvidos.

modificação no software do sistema. A mudança de três linhas de código, em meio aos muitos milhões de linhas existentes, deixou a população da região sem comunicação telefônica. O impacto da modificação foi subestimado devido ao pequeno número de linhas modificadas e o sistema, então, não foi devidamente testado.

- Em 1992 [4], um detento escapou de um presídio de New Jersey devido a uma falha na monitoração, que era baseada em computadores. Ele conseguiu se livrar das algemas eletrônicas e saiu do prédio. Um dos computadores detectou a fuga, mas quando foi contatar um outro computador recebeu um sinal de ocupado e não respondeu mais. O resultado foi catastrófico: o detento cometeu um assassinato após a fuga.

Ainda atualmente podemos observar que grande parte dos projetos estoura custo ou prazo e muitos outros são cancelados. Além do crescimento da complexidade já citado, são apontados alguns outros fatores causadores da chamada "crise do software" [4].

Um deles é a mudança nos requisitos do sistema, que se tornam um problema crítico em projetos com prazos estendidos. A entrada de um novo requisito pode implicar em grandes mudanças no sistema e exigir que o projeto retorne a fases anteriores, causando atrasos no cronograma e aumento dos custos no desenvolvimento do sistema. Os novos requisitos passam por todos as fases do projeto, mas não podem ser analisados isoladamente. Pode ser que todo o projeto precise ser reavaliado.

A partir deste cenário fez-se necessária a criação de um novo campo do conhecimento: a Engenharia de Software [1], que visa obter uma maior sistematização do processo de produção de programas de software.

1.1.1 Engenharia de Software

Ao contrário dos processos de desenvolvimento de software, nas engenharias tradicionais, como engenharia civil ou engenharia eletrônica, o

processo de desenvolvimento já está bastante consolidado e sistematizado. Como exemplo de resultado dessa discrepância de amadurecimento, podemos observar que atualmente os maiores entraves para o desenvolvimento tecnológico se deve a falhas de software, e não de hardware [5]. A tecnologia de projeto de componentes de hardware atingiu um alto grau de sofisticação e conseguiu reduzir drasticamente a probabilidade de falha nestes dispositivos.

O objetivo da Engenharia de Software é se espelhar nas engenharias tradicionais e obter uma maior sistematização do desenvolvimento de software, buscando uma qualidade do produto final aceitável, proporcional ao investimento. Ela fornece um conjunto de técnicas, métodos e práticas para que possamos lidar melhor com os problemas que aparecem.

Dentre estas técnicas, podemos destacar a técnica de refatoração de códigos.

Refatoração de Código

A técnica de refatoração de código consiste na reestruturação interna de um código (ou trecho de código) sem alterar seu comportamento externo. Em outras palavras, a funcionalidade daquele trecho de código é mantida após a transformação. A reestruturação pode ser feita tendo em vista objetivos diversos, tais como: melhorar a eficiência do código, aumentar sua extensibilidade, evitar replicação, melhorar a legibilidade ou conseguir mais facilidade na manutenção do código.

Poderíamos citar como exemplos de refatoração de código o encapsulamento dos atributos de um objeto, a substituição de códigos de erro por exceções ou até mesmo a simples mudança de nome de uma variável.

Em uma análise extremamente superficial, poderíamos deduzir que com o uso de refatorações realizamos um esforço maior do que o mínimo para se obter determinada funcionalidade, e que, portanto, geramos um overhead desnecessário no cronograma do projeto. O erro deste pensamento é desprezar os ganhos conseguidos com a refatoração. Ao realizarmos a

refatoração para aumentar a extensibilidade de um código, por exemplo, o esforço realizado é mais do que compensado pela redução do esforço na adição de novas funcionalidades. Além disso, não se realizam refatorações de forma descriteriosa. Sugere-se [14] que elas sejam aplicadas quando o sistema "solicita". Quando um código é duplicado na adição de uma nova funcionalidade, por exemplo, ele pode estar precisando de uma refatoração.

Convém explicitar que refatoração é um termo muito mais genérico, que pode ser aplicado em outros contextos. Poderíamos fazer uma refatoração num banco de dados [2], num projeto de software[1] ou até mesmo em outras áreas fora do contexto da programação. O trabalho relatado em [3], por exemplo, utiliza refatorações em trechos de código genético de uma bactéria. O objetivo é tentar realizar as transformações no DNA preservando o comportamento externo, visando obter um melhor entendimento do processo.

Em metodologias ágeis, como o *Extreme Programming* [14,15], a refatoração não só é muito utilizada, como também faz parte do ciclo de desenvolvimento do software. Apesar das vantagens oferecidas, essa transformação do código pode inserir erros e o código resultante pode não se comportar externamente da mesma forma do código original. Para que a transformação seja realmente benéfica, é preciso realizar a verificação do código transformado, para que os erros sejam identificados e corrigidos.

Porém, a verificação de software é uma tarefa difícil. O custo de verificação de um projeto de software pode chegar a representar 80% do custo total do projeto [32]. A abordagem de verificação adotada precisa ser eficiente, já que a correção de um erro de um programa é mais barata quando o erro é identificado mais cedo. A abordagem também precisa ser eficaz, para que nenhum erro grave e comprometedor seja inserido no código e passado para as demais fases do projeto.

[1] A refatoração aplicada no projeto de software é análoga à aplicada no código, porém é realizada em nível de projeto. Por exemplo, poderíamos já na fase de projeto modificar a estrutura de um Projeto Orientado a Objetos introduzindo herança em duas classes com muitas características em comum, evitando replicação de código.

1.2 A Abordagem Tradicional

Em metodologias ágeis, onde refatorações são amplamente utilizadas e fazem parte do ciclo de desenvolvimento do software, o desenvolvimento é dirigido a testes: primeiro se escreve um caso de teste[1] e somente depois se escreve o código para passar naquele teste. Isso significa que desenvolvimento dirigido a testes é um método de desenvolvimento de software e de testes de software ao mesmo tempo.

A seqüência de passos do método [15] é descrita abaixo:

- *Adicione um teste* – o primeiro passo do desenvolvimento é sempre a criação de um teste. O desenvolvedor deve entender claramente a especificação e os requisitos do sistema.

- *Rode todos os testes e verifique se o novo teste falhou* – este passo visa detalhar mais os testes. O sistema não deve passar no teste se nenhum código novo for inserido. Com este passo, podemos encontrar casos de teste mal elaborados.

- *Escreva o código* – neste passo o desenvolvedor escreverá o código que deve passar nos testes. Vale salientar que o código aqui não precisa ser perfeito: ele pode satisfazer aos testes de uma maneira deselegante, se a maneira elegante não for simples de ser concebida ou tomar muito tempo. Isto pode ser corrigido em passos futuros. Aqui o principal objetivo é fazer o código passar nos testes.

- *Rode os testes automatizados* – o código é testado através de execuções dos testes automatizados, que foram escritos anteriormente. Caso ele passe nos testes, assume-se que ele

[1] Um caso de teste é um conjunto de condições ou variáveis sobre os quais os testes vão determinar se uma aplicação satisfaz parcialmente ou integralmente um requisito.

atende aos requisitos testados. Caso reprove em algum teste, o novo código deve ser revisado.

- *Refatore o código* – O objetivo desta fase é remover a "deselegância" do código permitida no passo 3. Aqui busca-se código mais elegante, legível, extensível e principalmente com menos duplicações. Este processo de refatoração é informalmente chamado de "limpeza de código". A cada nova transformação, os testes são executados novamente para verificar que o novo código não apresenta falhas e está de acordo com os requisitos.

Portanto, para verificar as refatorações segundo esta abordagem, a mesma suíte de testes (T) que foi aplicada no programa original (P1) é aplicada no programa refatorado (P2). Se este programa passar nos testes, é assumido que nenhum erro foi inserido. A Figura 1 ilustra o processo:

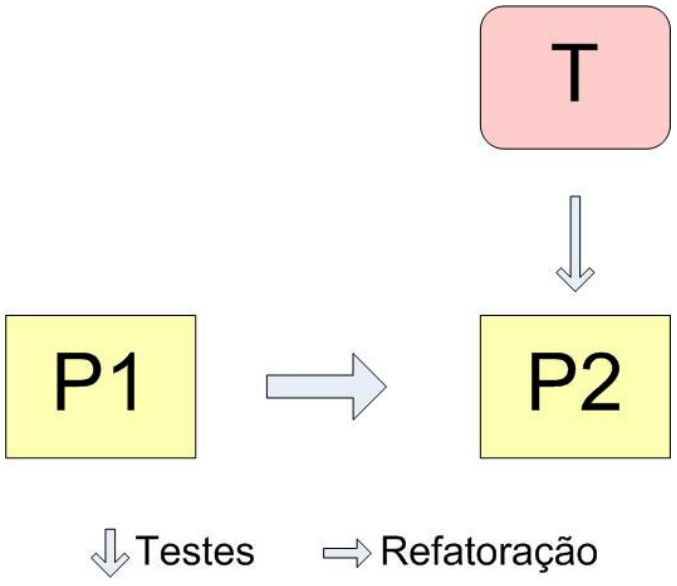


Figura 1 – Abordagem tradicional de verificação de refatorações.

O problema deste procedimento é que mesmo um conjunto de testes muito grande é insuficiente para garantir que a corretude foi mantida. A corretude só é garantida para aquele conjunto de casos de teste, que pode não

ser suficiente para garantir a qualidade do produto. Geralmente escolher o conjunto de casos de teste adequado é uma tarefa difícil e trabalhosa.

No caso em que uma refatoração substitui um trecho de código seqüencial por um paralelo, para aumentar a eficiência, por exemplo, o número de estados do código resultante passa a crescer exponencialmente com o número de componentes, ao invés de linearmente como no caso seqüencial. A complexidade e o tamanho do conjunto de testes necessário em casos como este podem contribuir para que erros passem despercebidos.

1.3 Objetivos

O Objetivo deste trabalho é propor uma maneira alternativa de realizar verificação de software neste contexto de refatorações. A proposta consiste no mapeamento automático de uma linguagem de implementação numa linguagem de especificação formal. A verificação, então, poderá ser feita através da técnica de verificação de refinamentos, a qual também é automática. A Figura 2 ilustra o processo de obtenção dos modelos:

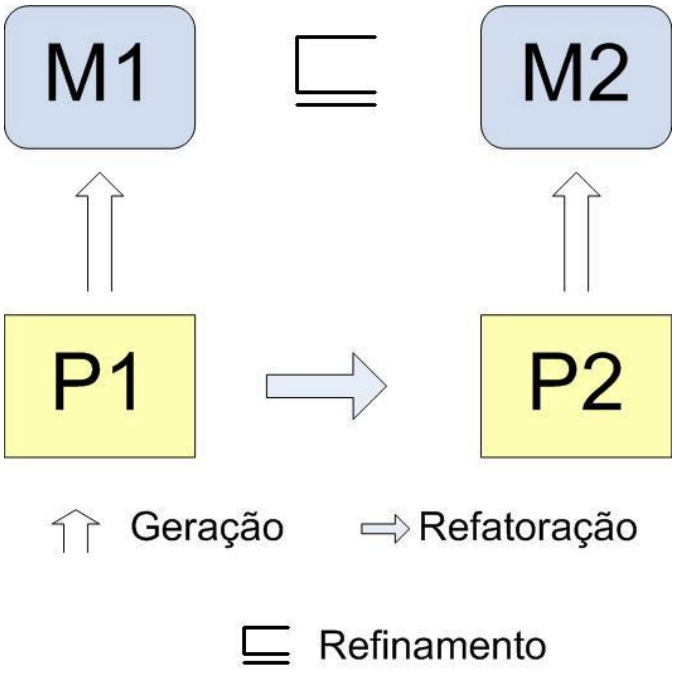


Figura 2 – Ilustração da abordagem de verificação automática

P1 representa o programa original, antes da refatoração. É gerado, de maneira automática, um modelo M1 a partir de P1. A refatoração aplicada em P1 gera um novo programa P2. Também é criado automaticamente um modelo M2 a partir de P2. Então, utilizamos uma ferramenta para provar a M2 preserva o comportamento de M1 (M1 ⊑ M2). Uma vez que os modelos são construídos de tal forma a capturar a funcionalidade externa de seus respectivos programas, estaremos provando também que o programa refatorado preserva as funcionalidades do programa original.

Dentre as diversas linguagens de especificação formal de sistemas, destacamos duas que são muito utilizadas na indústria: Z[9,10] e CSP[6,7,8]. Ambas possuem uma teoria de refinamentos, essencial na abordagem de verificação. Porém, esta última se destaca por possuir um tratamento adequado a sistemas concorrentes e será escolhida para o papel de linguagem formal em nosso estudo. A linguagem de implementação focada será Java [16]. Mais especificamente será considerado um subconjunto biblioteca JCSP [17], pois ela está mais próxima de CSP do que Java puro.

Tanto o código Java original quanto o código obtido após a refatoração serão mapeados em modelos descritos em CSP. É neste nível que faremos a verificação: provando, com o uso do verificador de refinamentos FDR [33], que o modelo obtido preserva as funcionalidades externas do modelo original. Desta forma, a probabilidade de erros serem introduzidos na refatoração pode ser reduzida. Além disso, como o processo de verificação passa a ser completamente automático, o programador não precisa criar uma suíte de testes, o que seria uma tarefa trabalhosa e problemática.

É importante salientar que este trabalho é apenas um passo inicial na criação desta nova abordagem. Aqui consideraremos apenas um subconjunto de JCSP. Trabalhos futuros podem considerar um conjunto maior da linguagem, ou até mesmo considerar outras linguagens formais e de implementação no mapeamento.

Para o mapeamento de um programa JCSP em um modelo CSP foram tomadas utilizadas as regras de mapeamento definidas nos trabalhos [11,12]. As regras foram investigadas e observamos algumas dificuldades na implementação de algumas regras e falhas na definição de algumas outras regras. De maneira geral, o trabalho se mostrou bastante criativo e serviu como base para a criação de um protótipo que faz o mapeamento automaticamente.

1.4 Visão Geral

Este trabalho está estruturado da seguinte forma:

Capítulo 2: Define os conceitos básicos e apresenta uma visão geral da linguagem formal CSP, utilizada neste trabalho.

Capítulo 3: Introduz a linguagem de implementação Java, que será utilizada neste trabalho. Em especial, serão introduzidos elementos da biblioteca JCSP.

Capítulo 4: Descreve a nova abordagem e fornece uma investigação inicial para a concepção da abordagem. Apresenta algumas regras de mapeamento de JCSP para CSP que apresentaram problemas ou dificuldades na implementação.

Capítulo 5: Demonstra o estudo de caso realizado por meio da aplicação da nova abordagem de verificação em uma refatoração.

Capítulo 6: São apresentadas as conclusões deste trabalho, trabalhos relacionados e uma lista com pontos a serem desenvolvidos no futuro.

2. CSP

CSP surgiu na década de 1970 através do trabalho de Sir com C. A. Hoare [6]. Ele introduziu a idéia de processos com variáveis locais que interagem apenas através de trocas de mensagens. Em sua essência, aquela versão era apenas uma linguagem para programação concorrente e não tinha uma semântica matematicamente definida.

Nos anos posteriores a idéia evoluiu muito e Hoare apresentou a versão teórica de CSP [7]. Desde então, a teoria de CSP apresentou pequenas mudanças. A motivação para a criação dessas mudanças é a concepção de ferramentas de análise e verificação automáticas. O texto definitivo para a teoria de CSP é apresentado no trabalho de A. W. Roscoe [8] e é nesta última versão que este trabalho se baseará.

CSP é uma álgebra de processos, nome dado às abordagens relacionadas à análise e modelagem formal de sistemas concorrentes. Ela provê meios para descrever de maneira abstrata e analisar as interações, comunicações e sincronizações entre uma coleção de processos independentes. Ela também fornece leis algébricas para a manipulação e análise de descrições de processos e permite a realização de provas formais de refinamentos entre processos.

A concorrência considerada em CSP ocorre no nível da comunicação. Ou seja, ela é realizada via troca de mensagens. Este tipo de concorrência é muito popular em sistemas distribuídos, onde cada processo é executado em uma máquina diferente, cada um com sua memória[1].

[1] A maneira alternativa de realizar concorrências é através de compartilhamento de memória. A memória utilizada pelo processo não é distinguida da memória utilizada para realizar a comunicação. Esta abordagem possui um desempenho superior, mas o tratamento da concorrência é mais trabalhoso.

A próxima seção apresentará elementos básicos de CSP. A notação utilizada é o CSP_M, ou CSP *machine readable*, que foi criada para encorajar a criação de novas ferramentas para CSP.

2.1 Conceitos Básicos

Processos – são as entidades básicas que capturam um comportamento. Os nomes de processos, por convenção, são escritos utilizando-se letras maiúsculas. Portanto, para descrever um relógio em CSP poderíamos definir um processo RELOGIO, por exemplo.

Por questões de modularidade, podemos descrever um comportamento através de um conjunto de processos. Em um banco, poderíamos descrever processos ATENDIMENTO e CAIXA, para separar as funcionalidades de atendimento ao cliente das funcionalidades referentes às transações de um caixa.

Além de denotar módulos de um sistema, os nomes de processos podem denotar o estado de um processo.

Processos primitivos – representam comportamentos fundamentais. Aqui iremos abordar dois processos primitivos:

- STOP é o processo que não comunica nada e é usado para descrever a quebra de um sistema bem com uma situação de deadlock;
- SKIP é o processo que indica que a execução foi encerrada com sucesso.

Ambiente – os processos podem interagir com o ambiente. Na maioria dos contextos, o ambiente pode ser interpretado como o usuário do sistema.

Comunicação – a comunicação em CSP significa três coisas: interação, pois dois ou mais processos interagem através da comunicação; observação, pois só podemos observar o comportamento dos processos através da sua

comunicação; e sincronização, pois dois processos que estão executando paralelamente sincronizam suas execuções através da comunicação. Por convenção, o nome da comunicação é escrito utilizando-se letras minúsculas.

Uma comunicação pode ser:

- **Um evento** – os eventos descrevem as interações mais simples entre os processos. Para um processo `RELOGIO`, poderíamos definir os eventos `tic` e `tac`, que indicam a passagem do tempo.

- **Um canal** – os canais diferem-se dos eventos devido ao fato de transmitirem dados. Os tipos dos dados transmitidos devem estar de acordo com o tipo do canal. Em um sistema que especifica um banco, um processo `CAIXA` poderia informar o valor debitado da conta de um cliente ao processo `SERVIDOR` através de um canal debitar, por exemplo. Além de informar a ação do débito, a comunicação informa o valor do débito.

Alfabeto – o alfabeto de uma especificação é a união de todas as comunicações presentes nas definições de todos os processos.

2.2 Operadores

Prefixo (->) – Este operador combina um evento e um processo para produzir um novo processo.

Por exemplo,

```
a -> P
```

é o processo que tenta comunicar o evento a. Depois de comunicar a ele se comporta como P.

Igualdade (=) – Os comportamentos dos processos podem ser definidos através de uma equação. A descrição de processos através de equações é essencial na definição de processos recursivos. Se quisermos definir um

processo RELOGIO que tenta comunicar alternadamente os eventos tic e tac indefinidamente, poderíamos definir este processo através da equação:

```
RELOGIO = tic -> tac -> RELOGIO
```

Também é possível definir um processo através de um número maior de equações mutuamente recursivas. O processo RELOGIO' definido por:

```
RELOGIO' = tic -> RELOGIO2

RELOGIO2 = tac -> RELOGIO'
```

se comporta da mesma maneira que o processo anterior.

Escolha Externa ([]) – Este operador define uma escolha entre dois processos componentes. Essa escolha é resolvida pelo ambiente, que aceita um dos eventos iniciais dos processos componentes para decidir.

Por exemplo:

```
( a -> P [] b -> Q)
```

O processo tenta comunicar os eventos iniciais a e b. Caso o ambiente aceite comunicar a, o processo passa a se comportar como P. Caso o evento b seja aceito pelo ambiente, o processo se comporta como Q.

Devido ao fato de o ambiente resolver a escolha, este operador também é conhecido como escolha determinística.

Escolha Interna (|~|) – Este operador é semelhante ao operador anterior, sendo que a escolha é decidida internamente. O ambiente não tem o poder de interferir na escolha neste caso.

O processo

```
( a -> P |~| b -> Q)
```

tenta comunicar inicialmente os eventos a ou b, mas isso não pode ser determinado previamente. Ele tenta comunicar um dos dois eventos e, caso o ambiente não aceite a comunicação, ele entrará em deadlock.

Para entender melhor a diferença entre os dois operadores de escolha, considere uma descrição simplificada de uma máquina de refrigerantes. Numa máquina comum, o usuário da máquina (o ambiente) coloca o dinheiro e pode escolher o sabor do refrigerante. A escolha do sabor seria especificada usando-se uma escolha externa. A máquina, porém, pode apresentar falhas e não aceitar a moeda, por exemplo. O usuário (ambiente) não pode influenciar na aceitação da moeda. Portanto, o reconhecimento da moeda seria especificado utilizando-se uma escolha interna. O processo M definido através das duas equações seguintes ilustra esse exemplo:

```
M = dinheiro -> ( sabor1 -> M2 [] sabor2 -> M2)

M2 = ( aceitar -> M) |~| (recusar -> M )
```

Entrelaçamento (|||) – Este é um operador de paralelismo entrelaçado. Este operador representa atividade concorrente totalmente independente. Os processos

```
P ||| Q
```

executam em paralelo, mas não realizam nenhuma sincronização nas suas execuções. Os eventos desses processos são arbitrariamente intercalados no tempo.

Paralelismo Generalizado ([| X |]) – Este é um operador de paralelismo generalizado. O processo

```
P [| X |] Q,
```

onde P e Q são processos e X é um conjunto de eventos, só permite que P e Q comuniquem eventos pertencentes ao conjunto X se ambos estiverem prontos para comunicar o evento. Em outras palavras, eles devem sincronizar nos

eventos do conjunto X. Entretanto, caso ocorra um evento Então, por exemplo, o processo

```
(a -> P) [| { a } |] (a -> Q)
```

pode comunicar o evento a e se tornar o processo:

```
P [| { a } |] Q
```

Por outro lado, o processo

```
(a -> P) [| { a } |] (b -> Q)
```

não pode comunicar a, pois os dois processos componentes devem sincronizar sobre o evento `a` e somente o componente esquerdo está pronto para comunicá-lo. Então, o processo pode somente comunicar b e se tornar o processo

```
(a -> P) [| { a } |] Q
```

Composição Seqüencial (;) – Este operador é semelhante ao operador de composição seqüencial das linguagens de implementação, exceto pelo fato de que ele não propaga estados. O processo

```
P ; Q
```

se comporta como P até ocorrer uma terminação com sucesso (processo primitivo SKIP). Depois disso, ele passa a se comportar como Q.

Se os processos P e Q forem definidos como

```
P = a -> SKIP
```

```
Q = b -> STOP
```

O processo definido por P;Q se comporta de forma indêntica ao processo R definido por:

```
R = a -> b -> STOP
```

Existem outros operadores importantes de CSP, como *hiding* e *renaming*[8], mas que não serão utilizados neste trabalho.

2.3 Modelos Semânticos de CSP

CSP pode ser visto sob 3 estilos semânticos diferentes: denotacional, operacional e algébrico. A escolha do estilo está intimamente relacionada ao propósito de seu uso, pois cada um possui um tratamento matemático diferente. A noção de refinamento é concebida com uso do estilo semântico denotacional, por isso este será focado neste trabalho. A semântica denotacional é definida em termos de modelos de traces, falhas e divergências.

2.3.1 Modelo de Traces

Um *trace* é uma seqüência de comunicações que o processo pode realizar com o ambiente. Um processo pode realizar comunicações indefinidamente e nunca parar de executar, gerando um trace infinito. O trace gerado é finito quando o processo chega em um ponto em que ele não consegue mais comunicar com o ambiente (STOP) ou a execução do processo termina com sucesso (SKIP).

Formalmente, o conjunto de traces de um processo P (traces(P)) é um conjunto de seqüências, onde cada seqüência é um subconjunto de $\Sigma^{\surd}$ (alfabeto completo).

Para qualquer processo P, o conjunto traces(P) deve obedecer às seguintes propriedades:

- traces(P) sempre contém as seqüência vazia (<>).
- Se s^t pertence a traces(P), então s também pertence.

Para ilustrar este modelo, considere o processo S definido por

```
S = a -> b -> c -> STOP
```

ele é representado no modelo de traces como

traces(S) = { <> , < a > , < a , b > , < a, b, c> }

Os traces de um processo são obtidos a partir de um conjunto de regras, que definem como obter os traces de processos simples (como SKIP e STOP) diretamente e traces de processos compostos (como P [] Q) em termos dos traces de seus componentes (P e Q). A seguir são apresentadas algumas destas regras:

1. traces(STOP) = { < > }

2. traces(a -> P) = { < > } U { < a > ^ s | s ∈ traces(P) }

3. traces(P |~| Q) = traces(P) U traces(Q)

4. traces(P [] Q) = traces(P) U traces(Q)

Vale explicitar que o modelo não serve para verificar determinismo de um processo. Ele apenas serve para explicar o que o processo pode fazer. Por isso, as regras para o tratamento dos operadores de escolha interna (|~|) e de escolha externa ([]) produzem a mesma saída.

2.3.2 Modelo de Falhas

Este modelo é mais poderoso que o anterior, em termos de investigação de propriedades: além de indicar o que o processo pode fazer, ele indica onde os processos falham.

A representação de um processo P no modelo de falhas consiste na tupla (s,X), onde s é um seqüência que pertence a traces(P) e X é o conjunto das comunicações que P não aceita após ter comunicado a seqüência s. Chamamos de failures(P) o conjunto das falhas de P.

As falhas de um processo também são obtidas a partir de um conjunto de regras, de forma análoga à obtenção dos traces. Algumas regras são ilustradas abaixo:

1. failures(STOP) = { (< >, X) | X ⊆ $\Sigma^{\surd}$ }

2. failures(SKIP) = { (< >, X) | X ⊆ Σ } U { (< >, X) | X ⊆ $\Sigma^{\surd}$ }

3. failures (P |~| Q) = failures (P) U failures (Q)

2.3.3 Modelo de Falhas e Divergências

Este modelo permite a investigação mais completa dos comportamentos de um processo. Além de também permitir investigar o que um processo pode fazer e identificar as falhas do processo, ele permite investigar as divergências do processo. Um processo diverge quando ele está realizando eventos invisíveis ao ambiente. O ambiente não consegue diferenciar se o processo parou ou divergiu, pois não consegue enxergar os eventos.

Quando um processo diverge, é assumido que ele pode recusar qualquer evento, rejeitar qualquer evento e sempre divergir mesmo após voltar a se comunicar com o ambiente. Então o conjunto divergences(P) contém os traces s nos quais P pode divergir e as suas extensões (s^t). Este conjunto também pode ser obtido através de regras. Algumas delas são:

1. divergences(STOP) = { }

2. divergences(SKIP) = { }

3. divergences (P |~| Q) = traces(P) U traces(Q)

4. divergences (P [] Q) = traces(P) U traces(Q)

O conjunto de falhas é extendido para capturar a idéia de um processo poder falhar após ter divergido. O conjunto extendido é definido por:

$$failures_{\perp}(P) = failures(P) \cup \{ (s, X) \mid s \in divergences(P)\}$$

A representação de um processo no modelo de falhas e divergências consiste na tupla:

$$(failures_{\perp}(P) , divergences(P))$$

Este modelo é o único que descreve completamente o comportamento de um processo.

2.4 Refinamentos

Para dois modelos P e Q, dizemos que Q (o modelo refinado) refina P (o modelo original) quando Q satisfaz, no mínimo, a propriedades de P. Quando isto ocorre, o modelo refinado pode substituir o modelo original em seu contexto sem causar impactos negativos no resto do sistema.

O conceito de refinamento é muito utilizado em linguagens de especificações formais. A idéia é partir de um sistema formal mais abstrato e ir adicionando detalhes de implementação aos modelos refinados. Assim, temos um modelo mais próximo da implementação que satisfaz as propriedades do modelo original.

A definição da relação de refinamento em CSP depende do modelo semântico utilizado. As formas relevantes de refinamento de CSP correspondem aos três modelos semânticos:

Refinamentos por traces – um processo P refina um outro processo Q no modelo de traces ($P \sqsubseteq_T Q$) se e somente se $traces(Q) \subseteq traces(P)$. Isto significa que o processo Q (o refinado) não pode realizar seqüências de comunicações que o processo P não realiza. Ele só pode realizar um subconjunto daquelas comunicações.

Partindo desta definição, podemos observar que o processo STOP refina qualquer outro processo no modelo de traces (traces(STOP) = {<>}). De fato, este nível de refinamento não investiga falhas, divergências ou não-determinismos.

Refinamentos por falhas – um processo P refina um outro processo Q no modelo de falhas ($P \sqsubseteq_F Q$) se e somente se traces(Q) ⊆ traces(P) ^ failures (Q) ⊆ failures (P). Isto significa que o processo refinado Q não introduz nenhuma falha quando substituído pelo processo P no seu contexto, além de não produzir nenhuma seqüência nova de comunicações. Divergência não é verificada nesse refinamento.

Refinamentos por falhas e divergências – este é o refinamento mais importante de CSP. Somente ele é capaz de investigar se um processo refinado não introduz mais divergências. A definição de refinamento neste modelo é dada por:

$P \sqsubseteq_{FD} Q \equiv failures_{\perp}(Q) \subseteq failures_{\perp}(P)$ ^ divergences(Q) ⊆ divergences(P)

A escolha do modelo correto depende de quais propriedades deseja-se investigar. Apesar do modelo de falhas e divergências ser o mais completo, o refinamento neste modelo é o mais complexo de ser atingido. As provas deste refinamento também são complexas e costumam levar mais tempo do que para os outros dois modelos mais simples. Portanto, é preciso conhecer as limitações de cada modelo e decidir as propriedades que desejamos investigar para depois escolher o modelo mais adequado.

2.5 Ferramentas

Dois tipos de ferramentas merecem destaque: as animadoras (*animators*) e as provadoras de refinamento (*refinement checkers*).

Com as ferramentas animadoras, o usuário pode fazer o papel de ambiente e escolher um dos eventos que o processo está tentando comunicar a cada passo. Na verdade, ela permite inclusive que o usuário resolva as escolhas internas. Uma das mais utilizadas em especificações CSP é o ProBE. A figura abaixo ilustra um exemplo de animação da ferramenta.

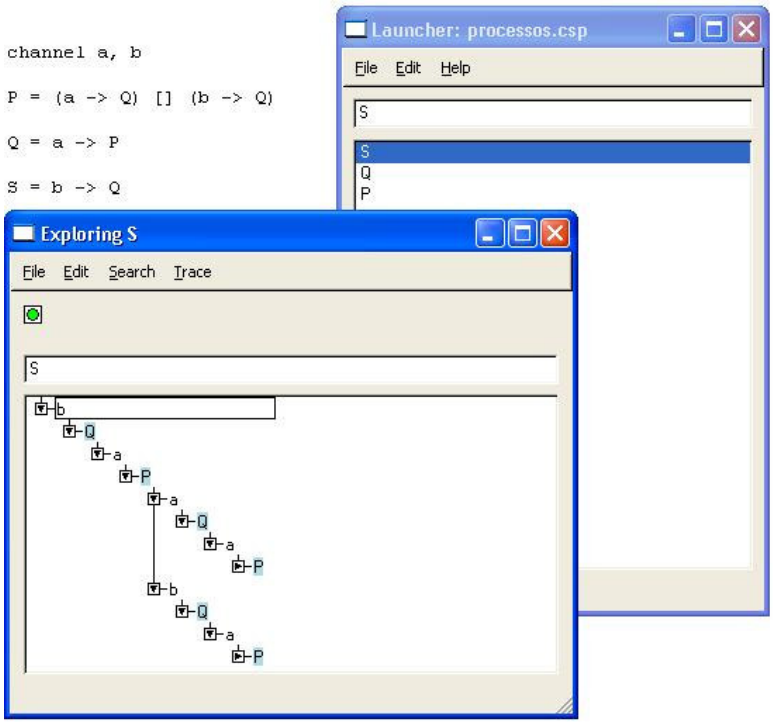


Figura 3 – Ferramenta ProBE

As ferramentas provadoras de refinamento possuem um outro propósito. Com elas o usuário pode testar afirmações sobre processos, como equivalências entre processos e existência de deadlocks, livelocks ou não-determinismo. A ferramenta FDR (Failures-Divergences Refinement) trabalha com modelos descritos em CSP. Com ela podemos provar refinamentos de modelos, além de insvestigar as propriedades de deadlocks, livelocks e determinismo.

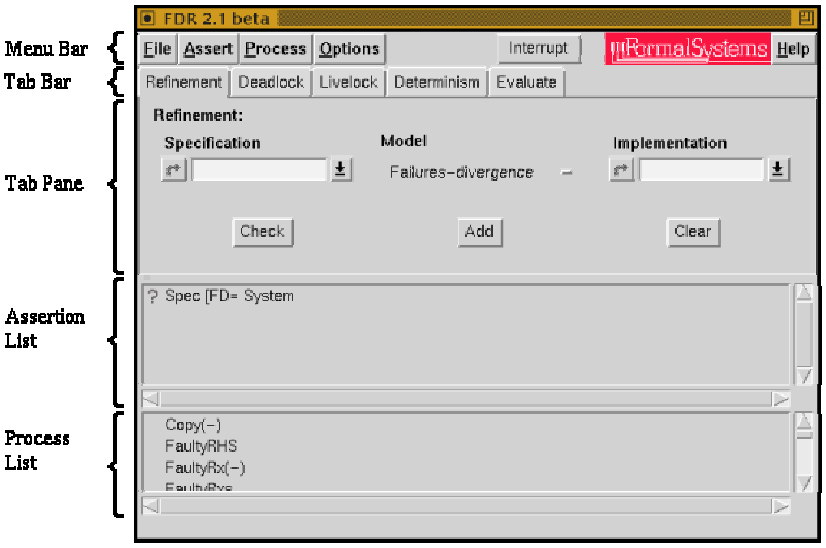


Figura 4 – Ferramenta FDR

3. Java

Java é um conjunto de produtos de software e especificações da *Sun Microsystems* que provêem um sistema para desenvolvimento de aplicações multi-plataforma. Seu desenvolvimento tem início em 1991, num projeto de pesquisa com o codinome *Green*.

Este projeto resultou em uma linguagem chamada *Oak*, nome de uma árvore de carvalho. Mais tarde, por questões de licença, a linguagem recebeu o nome *Java*. A linguagem é baseada em C++ [18] e visa melhorar características relativas a segurança, portabilidade e facilitar o tratamento de concorrência, distribuição e gerenciamento de memória. O foco inicial do projeto era o mercado de sistemas eletrônicos inteligentes voltados para o consumo popular. Como a arquitetura destes sistemas variava muito, a portabilidade também era um dos principais objetivos.

O mercado de dispositivos inteligentes não apresentou o crescimento esperado pela *Sun* e o projeto passou por dificuldades. Chegou a correr o risco de ser cancelado. Os principais responsáveis pelo projeto se reuniram e identificaram uma oportunidade: mudar o foco para a *World Wide Web*, que apresentava um enorme crescimento na época. A idéia era revolucionar aproveitando o potencial de Java para adicionar conteúdo dinâmico, como interatividade e animações, às páginas eletrônicas. A mudança de foco chamou a atenção do mercado, devido ao enorme interesse na *World Wide Web*.

Atualmente, podemos encontrar a tecnologia Java sendo empregada em plataformas de computadores de natureza variada, que vai desde dispositivos embarcados até servidores e supercomputadores.

Java apresentou melhorias em diversos aspectos, em relação às linguagens existentes na época. Podemos notar que o tratamento de

concorrência em Java é muito mais simples e elegante do que linguagens de programação como C++. O programador pode especificar aplicativos que contêm *threads* de execução, onde cada *thread* representa uma parte do programa que vai executar de forma concorrente com outras *threads*. Este conceito é chamado *multi-threading* e não era disponível no núcleo das linguagens C/C++[15].

Porém, o modelo de *monitor-thread* oferecido por Java ainda possui diretivas (*synchronize*, *wait*, *notify*, *notifyAll*) com um nível de abstração muito baixo e que são difíceis de serem usadas corretamente na prática. Os maus funcionamentos das aplicações, que podem resultar em *deadlocks*[1], *livelocks*[2] ou *resource starvation*[3], podem ter causas difíceis de serem identificadas e podem ocorrer em intervalos de tempo arbitrários. Um programa pode funcionar bem durante muito tempo, antes de ocorrer uma falha desastrosa.

Outro problema está na verificação dos sistemas concorrentes em Java. Por não ser uma linguagem formal, a forma natural de verificação é através de testes. Porém, o cenário de objetos concorrentes cria um espaço de estados que pode crescer exponencialmente com o número de componentes do programa. Isto dificulta a escolha de um conjunto de testes que garanta uma boa confiabilidade no sistema e aumenta a probabilidade de que algum erro passe despercebido.

A fim de resolver problemas como estes, foram criadas bibliotecas que fornecem um maior nível de abstração no tratamento da distribuição e da concorrência. Podemos destacar JCSP[17] (*Java Communicating Sequential Processes*) e CTJ[26] (*Communicating Thread for Java*). Ambas visam prover um meio de implementar sistemas concorrentes e distribuídos de forma aproximada a CSP.

[1] Um deadlock ocorre quando todos os processos/threads estão parados, aguardando um recurso que não poderão alocar.
[2] Ocorre quando os processos/threads estão em execução, mas nenhuma tarefa útil é realizada.
[3] Ocorre quando parte dos processos/threads não conseguem alocar um determinado recurso.

Este trabalho foca na API JCSP, por possuir uma rica quantidade de extensões CSP e uma documentação mais detalhada.

3.1 A API JCSP

A biblioteca JCSP foi criada com o objetivo de prover uma maneira real e prática de tratar a distribuição e a concorrência de maneira mais alto nível. Assim como CSP, JCSP tem como um de seus elementos básicos um processo.

Um Processo em JCSP consiste em uma classe que implementa a Interface CSProcess. O exemplo seguinte define um processo que imprime o resultado do fatorial de um número:

```
public class Fatorial implements CSProcess {

      private int x;

      public Fatorial(int n){
            this.x = n;
      }

      public void run(){
            System.out.println(x);
            int total = x;
            While(x > 1){
                  total = total * x;
                  x = x – 1;
            }
            System.out.println(total);
      }
}
```

Esta classe implementa a interface CSProcess, a qual define um único método `public abstract void run()`. O atributo `x` armazena o número cujo fatorial será calculado e a variável local total do método `run`() armazena o resultado do fatorial. Note que este é um processo simples, que não realiza

nenhuma comunicação e possui um comportamento finito (o laço while acaba quando x > 1).

3.1.1 Canais

Em JCSP existem dois conjuntos de canais: os que transportam inteiros e os que transportam quaisquer tipos de objetos (elementos do tipo Object). Os canais do primeiro conjunto devem implementar as interfaces ChannelInt, ChannelInputInt ou ChannelOutputInt. Já os do segundo, devem implementar os canais Channel, ChannelInput ou ChannelOutput.

Os processos que fazem somente a leitura de um canal, devem implementar o método `read`(), definido nas interfaces ChannelIput e ChannelInputInt. Analogamente, o método `write`() é utilizado para enviar dados num canal. As interfaces para um canal de saída são ChannelOutput e ChannelOutputInt. As interfaces Channel e ChannelInt são utilizadas por canais que podem ser entrada ou saída de um processo.

Existem diversas classes que implementam essas interfaces. A mais simples delas é a classe One2OneChannel, onde o canal serve para realizar a comunicação entre um único processo leitor e um único processo escritor. Esta classe também tem uma classe correspondente para trabalhar com inteiros (One2OneChannelInt). As classes Any2OneChannel, One2AnyChannel e Any2AnyChannel lidam com leitores e/ou escritores múltiplos.

Uma diferença entre JCSP e CSP reside no fato de JCSP não suportar multi-sincronização. A biblioteca não suporta mais dois processos tentando sincronizar num mesmo canal. Quando existem dois ou mais processos tentando ler ou escrever no canal, eles competem pelo uso do canal.

Para capturar este comportamento, o mapeamento deveria envolver uma estrutura de contextos para tratar as diferenças. O mapeamento investigado neste trabalho, que foi introduzido no [12], ainda não provê tal estrutura.

Podemos ilustrar o uso de canais através do processo Eco:

```
public class Eco implements CSProcess{

      private ChannelInputInt entrada;
      private ChannelOutputInt saida;

      public Eco(ChannelInputInt entrada, ChannelOutputInt saida){
            this.entrada = entrada;
            this.saida = saida;
      }

      public void run(){
            int i = 0;
            while(i < 5){
                  int dado = entrada.read();
                  saida.write(dado);
                  i = i + 1;
            }
      }
}
```

O processo Eco possui como atributos um canal de entrada e um canal de saída que trabalham com valores inteiros. Seu comportamento é inferido a partir da observação do corpo de `run`() : ele simplesmente lê um valor do canal de entrada e o escreve no canal de saída e realiza esta tarefa 5 vezes. Este processo difere do anterior no fato de poder interagir com outros processos através dos canais.

3.1.2 Processos

Ao definir um processo JCSP, alguns critérios devem ser obedecidos, além de ter que implementar a interface CSProcess:

- Os canais de um Processo devem ser declarados em campos privados, assim como os demais atributos da classe.

- Os métodos de modificação e acesso (gets e sets) devem ser chamados enquanto o processo ainda não está rodando.

- Durante a execução, a alteração do estado de um processo pode ser requisitada através de canais. O processo, porém, tem a liberdade de rejeitar a solicitação de modificação de seu estado.

- A mesma instância de um processo só pode ser executada mais de uma vez de forma seqüencial. A execução paralela se dá através de instâncias diferentes do processo.

JCSP aproxima muito CSP na descrição de processos. Podemos descrever em JCSP processos formados pela composição paralela de outros processos, processos formados pela composição seqüencial e processos básicos, como SKIP e STOP.

Processos básicos SKIP e STOP - possuem tipos próprios em JCSP (Skip e Stop, respectivamente). Eles implementam a interface CSProcess e podem ser executados através da chamada do método run() :

```
Skip skip = new Skip();
skip.run();
Stop stop = new Stop();
stop.run();
```

Composição seqüencial de processos - é obtida em JCSP através da classe Sequence. Um processo Sequence é inicializado com um array de processos. O primeiro processo é executado e cada processo posterior só é executado se o processo anterior tiver encerrado com sucesso. Em outras palavras, um processo posterior não é executado quando o processo em execução nunca termina (um laço infinito no método `run`(), por exemplo) ou caso ocorra a execução de um processo Stop.

O exemplo abaixo ilustra o uso deste construtor:

```
class EcoSequencial{

      public static void main(String argv[])[
            One2OneChannelInt entrada = new One2OneChannelInt();
            One2OneChannelInt saida = new One2OneChannelInt();

            One2OneChannelInt entrada2 = new One2OneChannelInt();
            One2OneChannelInt saida2 = new One2OneChannelInt();

            Eco eco = new Eco(entrada, saida);
            Eco eco2 = new Eco(entrada2, saida2);

            CSProcess processos = new CSProcess() {eco, eco2};
            Sequence seq = new Sequence(processos);

            seq.run();
      }
}
```

Neste processo temos a execução de duas instâncias da classe Eco em seqüência. A segunda instância só é executada quando a primeira termina com sucesso.

É possível adicionar ou remover processos a uma composição seqüencial durante a sua execução. Porém, as mudanças só terão efeito em uma execução posterior.

Procesos Paralelos – O paralelismo é concebido em CSP através da classe Parallel. O construtor da classe recebe um *array* de processos que serão executados de forma paralela. A execução é encerrada quando o último dos processos pára.

Além de não permitir multi-sincronização, JCSP apresenta uma outra diferença com relação a CSP no operador de paralelismo: por ser baseado na versão de CSP de Hoare, os processos sincronizam de forma diferente da versão mais moderna de CSP. Naquela versão os processos sincronizam obrigatoriamente em todos os eventos em comum. Na versão mais atual

devemos dizer explicitamente em quais canais queremos que um processo sincronize. Inclusive, temos ainda a opção de não realizar nenhuma sincronização com uso do operador de Interleave (|||). Isto só ocorre em JCSP quando os processos que estão executando em paralelo não possuem eventos em comum.

Para ilustrar esse operador, definimos a classe EcoParalelo abaixo:

```
class EcoParalelo{

      public static void main(String argv[])[
            One2OneChannelInt entrada = new One2OneChannelInt();
            One2OneChannelInt elo = new One2OneChannelInt();
            One2OneChannelInt saida = new One2OneChannelInt();

            Eco eco = new Eco(entrada, elo);
            Eco eco2 = new Eco(elo, saida);

            CSProcess processos = new CSProcess() {entrada, saida};
            Parallel par = new Parallel(processos);

            par.run();
      }
}
```

Primeiro o processo relacionado à instância `eco` realiza uma comunicação com o ambiente através do canal entrada. Depois os processos sincronizam no canal `elo`, e o processo elo comunica o valor recebido anteriormente para o processo `eco2`. Este processo então comunica o valor ao ambiente através do canal de saída.

Escolhas – O operador de escolhas de JCSP é implementado de maneira diferente em relação a CSP. Neste, o operador é um construtor de processos. O operador de escolhas de JCSP, que é implementado pela Classe Alternative, não possui processos como componentes. Ele recebe um *array* de canais no

momento em que é instanciado e seleciona o canal que será lido. Se mais de um canal estiver pronto para ser lido, ele escolhe aleatoriamente um canal.

A escolha do canal pode ser feita a partir da invocação de três métodos: `select`(), que escolhe aleatoriamente um dos canais que estão prontos para serem lidos; `priSelect`(), que escolhe o canal de índice mais baixo no *array* de canais para ser lido; e `fairSelect`(), que realiza a leitura de todos canais antes de realizar uma segunda leitura de um canal.

O exemplo abaixo ilustra um exemplo de processo que realiza escolha:

```
class Escolha implements CSProcess{

      private AltingChannelInputInt entrada1;
      private AltingChannelInputInt entrada2;
      private ChannelOutputInt saida1;
      private ChannelOutputInt saida2;

      public Escolha(AltingChannelInputInt e1,
                     AltingChannelInputInt e2,
                     ChannelOutputInt s1,
                     ChannelOutputInt s2){

            this.entrada1 = e1;
            this.entrada2 = e2;
            this.saida1= s1;
            this.saida2= s2;
      }

      public void run(){
            Guard guarda[] = new Guard[] { entrada1, entrada2 };
            Alternative alt = new Alternative(guarda);

            While(true){
                  int i = alt.select();
                  switch(i){
                        case 0:
                              int valor = entrada1.read();
                              saida1.write(valor);
                        default:
```

```
                        int valor = entrada2.read();
                        saida2.write(valor);
                }
            }
        }
}
```

Este processo possui dois canais de entrada e dois canais de saída que trabalham com valores inteiros. Um *array* de guardas do canal é criado (do tipo Guard) e inicializado com os canais de entrada. O processo fica num laço infinito escolhendo um dos canais aleatoriamente: se o canal `entrada1` for escolhido, o valor é lido e escrito no canal `saida1`; se o canal `entrada2` for escolhido, o valor é escrito no canal `saida2`.

4. Verificando Refatorações Automaticamente

Este capítulo descreve a nova abordagem de verificação automática de códigos no contexto de refatoração. Primeiramente descreveremos na seção 4.1 os detalhes da nova abordagem. Em seguida, na seção 4.2 introduziremos a implementação de uma ferramenta de abstração automática de programas. Na seção 4.3 descreveremos algumas dificuldades e problemas encontrados na implementação das regras de geração de código. Por fim, apresentaremos as considerações finais do capítulo na seção 4.4.

4.1 A nova abordagem

Este trabalho propõe uma nova abordagem que consiste no mapeamento automático de uma linguagem de implementação numa linguagem de especificação formal. A verificação, então, poderá ser feita através de provas, também realizadas de maneira automática usando uma ferramenta chamada de verificador de refinamentos.

A verificação de refinamentos entre modelos CSP pode ser facilmente realizada com o uso da ferramenta FDR, introduzida no capítulo 2, ferramenta já consolidada e utilizada na indústria européia por uma empresa chamada *QinetQ*.

Desde que o mapeamento de um programa Java (JCSP) para um modelo CSP preserve o comportamento externo, como é o nosso caso, pois utilizamos o trabalho relatado em [12] para realizar o mapeamento, a prova de equivalência entre os modelos fornece uma maior confiabilidade na investigação de uma refatoração de programas.

Portanto, como a prova de equivalência entre os modelos já pode ser feita de forma automática (através da ferramenta FDR), para possibilitar a

utilização da abordagem é preciso conceber somente a ferramenta que gera o modelo CSP de um programa JCSP.

4.2 Geração automática dos modelos

Uma ferramenta de tradução recebe um texto expresso em uma linguagem (a linguagem fonte) e gera um texto (semanticamente equivalente) expresso em uma outra linguagem (a linguagem destino). Em compiladores tradicionais, a linguagem fonte geralmente é uma linguagem de programação de alto nível e a linguagem destino é um código objeto.

Uma ferramenta que abstrai um código JCSP para uma especificação em CSP pode utilizar algumas idéias que são usadas num processo de tradução tradicional, pois também se trata de uma ferramenta processadora de linguagens.

Um processo de tradução deve ter as seguintes fases:

- **Análise sintática** – neste passo ocorrem: o *scanning*, onde os caracteres do texto da linguagem fonte são agrupados para formar os *tokens*, que podem ser identificadores, literais, operadores e palavras-chave; e o *parsing*, onde os *tokens* são analisados para ser criada uma estrutura abstrata que represente o programa. Comumente a estrutura utilizada é uma árvore de sintaxe abstrata.

- **Análise contextual** – neste passo é realizada uma análise estática da estrutura do programa obtida no passo anterior. A identificação, a verificação regras de escopo e a checagem estática de tipos são realizadas nesta fase.

- **Geração de código** - dentre as tarefas realizadas nesta fase destacamos a seleção de código, onde se decide quais trechos de código da linguagem destino serão gerados a partir de um

trecho de código da linguagem fonte. Para isso, regras de mapeamento são definidas.

A utilização de ferramentas como o *SableCC* [22] ou o *JavaCC* [23], por exemplo, podem automatizar o processo de geração de *parsers*. De fato, estas ferramentas já possuem uma gramática para Java disponível e podem ser livremente utilizadas. Desta forma, uma vez familiarizado com uma dessas ferramentas, o desenvolvedor não tem dificuldades em realizar a análise sintática.

Para o caso de uma linguagem como Java, que já possui um compilador implementado, tarefas como verificação estática de tipos e verificação da satisfação às regras de escopo podem ser realizas simplesmente compilando-se a linguagem em um compilador existente. Os erros identificados serão reportados e o usuário poderá refazer o código.

Portanto, no processo de abstração de um programa, os esforços maiores estão na geração de código. Antes de conceber um gerador, devemos especificar como a tradução deve ser feita de maneira indutiva, especificando a tradução de elementos individuais da linguagem fonte para a linguagem de especificação através de regras.

Esse trabalho de definição das regras já foi iniciado em [11, 12]. A seção seguinte discute os problemas e dificuldades de se implementar as regras definidas no trabalho.

4.3 Investigação das regras

Esta seção apresentará algumas regras definidas em [12], procurando demonstrar algumas dificuldades e os problemas que encontramos ao implementa-las. Porém, antes de introduzirmos as regras, é necessário explicitar as restrições que são feitas aos programas JCSP que servem de entrada para o mapeamento. Algumas das restrições assumidas são:

- Diretivas import e package são descartadas;

- O modificador static é descartado;
- As primitivas de concorrência de Java (wait, notify, notifyAll) não são capturadas. A concorrência é tratada num nível de abstração maior;
- Herança ainda não é capturada nesta versão das regras;
- Um programa JCSP é composto de um conjunto de classes que implementam a interface CSProcess e uma classe que implementa o método main. Esta última classe, que chamaremos de agora em diante de classe Main, é ponto de entrada do programa JCSP, funciona como processo principal que inicializa os outros processos que compõem o sistema.

O primeiro problema encontrado durante a implementação das regras foi o mapeamento da declaração de canais. A seguinte regra ilustra um exemplo de como é mapeada a declaração de um único canal:

Regra 10 (varDeclT). *Channel Transformation*
Modifier* Type Identifier [= new Type()];
↪ **channel** Identifier : typeT ⟦Type⟧

Condição: Type *é um tipo de canal JCSP*.

A função typeT ⟦.⟧ retorna o tipo CSP referente ao tipo do canal declarado.

O primeiro problema investigado é que definir a saída da função typeT ⟦.⟧ é uma tarefa difícil. Poríamos pensar que classes que implementam a interface `ChannelInt` teriam como resultado da função o tipo *Int* (inteiro de CSP). Porém, este mapeamento não funciona, pois *Int* é um tipo infinito em CSP. As ferramentas anunciam uma mensagem de alerta quando processa um script com a leitura de um canal do tipo *Int*.

Como em Java o tipo *int* possui 32 bits, poderíamos tentar definir o tipo como { -2147483648 .. 2147483647} , que seria o conjunto { -2^{31} .. 2^{31}-1}. Porém, a definição de canais deste tipo e a realização de leituras nesses canais resultam em um espaço de estados muito grande. As ferramentas levariam muito tempo para realizar a verificação.

O ideal seria definir no tipo dos canais somente os valores que estão sendo realmente usados. Trabalhos futuros podem considerar o uso de anotações de código (invariantes) no próprio programa, como especificações, para definir o tipo dos canais.

Como solução *ad hoc*, enquanto o mapeamento deste tipo não está sendo definido, consideramos um conjunto menor de valores inteiros: { -100 .. 100 }. Como exemplo de funcionamento da regra, considere a seguinte declaração:

```
One2OneChannelInt meuCanal = new One2OneChannelInt();
```

Aplicando a regra 10, obtemos:

```
channel meuCanal : { -100 .. 100}
```

Também é difícil definir a saída da função typeT ⟦.⟧ para canais que implementam ou estendem a interface `Channel`. Tais canais trabalham com elementos do tipo `Object` e que, portanto, podem ser `String`, `Integer`, `Double` ou qualquer outro tipo não-primitivo.

A estratégia de omitir a definição de um tipo no canal declarado não funciona, pois em CSP_M, quando declaramos um canal sem explicitarmos o tipo de dado que ele transporta, na realidade estamos declarando um evento. Em outras palavras, estamos assumindo que a comunicação ocorrerá sem transporte de dados. Se quisermos escrever algum valor no canal, devemos incluir na declaração do canal o tipo dos valores que serão transportados.

Devido a esta dificuldade, consideramos neste trabalho que canais do tipo *Channel* somente serão utilizados como eventos. Para simular eventos em JCSP, fazemos leituras e escritas sem transporte de dados relevantes, comunicando sempre somente o valor `null`. Como o valor `null` em JCSP é mapeado para 0 (zero) em CSP (expT ⟦null⟧ = 0), a função typeT ⟦.⟧ retorna { 0 } para estes canais. Por exemplo:

```
One2OneChannel meuCanal2 = new One2OneChannel();
```

Aplicando a regra 10, obtemos:

```
channel meuCanal2 : {0}
```

Com a restrição de valores escritos ao valor `null`, as operações de escrita serão mapeadas corretamente:

```
meuCanal2.write(null) ↪   meuCanal2 ! 0 -> SKIP
```

Um outro problema encontrado está na definição de funções auxiliares. Duas dessas funções são utilizadas em manipulações de arrays nas seguintes regras:

Regra 23 (expT). *Read Element Array Transformation*

Identifier [index] ↪ arrayGet(Identifier, index)

Portanto, a expressão a[i] é transformada em arrayGet(a, i)

Regra 24 (expT). *Write Element Array Transformation*

Identifier[index] = Expression; Statement

↪ **let** Identifier' = arraySet(Identifier, index, expT ⟦Expression⟧)

within comT ⟦Statement⟧ / [Identifier' / Identifier]

Condição: *Identifier não é um tipo de canal JCSP*.

Assim, a atribuição *a[i] = valor* se torna *a' = arraySet(a, i, valor)*

Estas funções auxiliares devem ser anexadas na especificação CSP. Suas definições são:

```
arrayGet(<h>^t, p) = if p==0 then h else arrayGet(t,p-1)
arrayGet(<>, p) = p

arraySet(<h>^t, p, e) = if p==0 then <e>^t
                                    else <h>^arraySet(t,p-1,e)
arraySet(<>, p, e) = <>
```

A partir dessas definições, verificamos que existem casos em que o mapeamento pode resultar numa especificação que não preserva a semântica do programa. Para exemplificar um caso destes, considere uma variável "inteiros", em JCSP, que é um *array* de 3 elementos inteiros: 0, 1 e 2.

A expressão

```
inteiros[2]
```

resulta em

```
arrayGet(inteiros, 2)
```

que retorna corretamente o valor 2. Mas, a expressão

```
inteiros[5]
```

resulta em uma exceção do tipo ArrayIndexOutOfBoundsException em JCSP. O seu mapeamento em CSP é

```
arrayGet(inteiros, 5)
```

que retorna o valor 2 e, portanto, a semântica não foi preservada.

Podemos notar que a função `arraySet` também pode apresentar problemas quando há um "estouro" de *array*. Ela retorna sempre uma

seqüência vazia nesses casos, quando também deveria produzir um resultado semanticamente equivalente à exceção *ArrayIndexOutOfBoundsException*.

Para que a semântica seja preservada em tais casos, é preciso que haja uma captura do tratamento de exceções de Java. Como as exceções de Java ainda não foram capturadas, o problema ainda não foi resolvido.

A regra que captura operações de escrita em um elemento de um *array* de canais, que também faz uso dessas funções auxiliares, é definida por:

Regra 35 (comT). *Write Array Transformation*
c[i].write(exp) ↪ arraySet(c, i, expT ⟦exp⟧) -> SKIP

Condição: *c representa um array de canais de um tipo JCSP.*

O mapeamento acima está incorreto, pois nenhum valor está sendo escrito no canal. A chamada da função arraySet(.) está simplesmente retornando uma nova seqüência que é uma cópia de c com o i-ésimo elemento sendo substituído por expT ⟦exp⟧ . Ou seja, ao invés de escrever um valor no i-ésimo canal do array, o mapeamento está substituindo o canal pelo valor fornecido.

A dificuldade de se definir corretamente a regra reside no fato de existirem dois tipos de resultados de mapeamento do array de canais. O primeiro, resultante da Regra 7 definida abaixo, é um canal CSP que realiza transferência múltipla de dados numa única comunicação. O primeiro dado se refere ao índice do elemento no *array* e o segundo, ao valor comunicado no canal:

Regra 7 (varDeclT). *Array Channel Transformation*
Modifier Type[] Identifier = new Type[length];
↪ **channel** Identifier:{0..length – 1}.typeT ⟦Type⟧

Condição: *c representa um array de canais de um tipo JCSP.*

O acesso a um elemento do array `c[i]` deve ser mapeado para `c.i` neste caso. A próxima regra também captura uma declaração de array de canais, mas esta possui uma inicialização:

Regra 9 (varDeclT). *Array Channel Initialization Transformation*

Modifier Type[] Identifier = [new Type[]] { Expression };

↪ Identifier = expT ⟦{Expression};⟧

Condição: *c representa um array de canais de um tipo JCSP.*

onde Expression é uma lista de expressões constituída pelos elementos do array de canais. O resultado de expT ⟦{Expression};⟧ é uma seqüência, tipo básico de CSP, onde os elementos são os canais indicados em { Expression }.

O acesso a um elemento do array `c[i]` de um canal declarado desta maneira se dá através de `arrayGet(c,i)`, pois o canal é mapeado para uma sequência CSP.

Portanto, temos duas regras tratando declarações de arrays de canais e que resultam em elementos diferentes: a primeira resulta em um canal que realiza transporte múltiplo de dados e a segunda resulta em uma seqüência de canais. Devido a isto, o acesso a um elemento do array é feito de forma diferente para cada caso.

Se um array de canais fosse capturado pela Regra 7, a escrita em um elemento deste array, definida em JCSP como

c[i].write(exp)

deveria resultar em:

c . i ! expT ⟦exp⟧

O mapeamento da mesma operação sobre um elemento de um array de canais declarado de acordo com o padrão de entrada da Regra 9, deveria resultar em

arrayGet(c, i) ! expT ⟦exp⟧

Portanto, uma dificuldade reside no fato de o acesso a um elemento de o array depender da forma que o array foi declarado. Além disso, a Regra 7 se precipita em mapear um *array* de canais que ainda não foi inicializado. A regra não funciona quando os elementos do *array* são inicializados com canais já declarados, como em:

```
One2OneChannel a = new One2OneChannel();
One2OneChannel b = new One2OneChannel();
One2OneChannel canais[] = new One2OneChannel[2];
canais[0] = a;
canais[1] = b;
```

As atribuições realizadas são mapeadas para

```
...
let
     canais' = arraySet(canais, 0, a)
     canais'' = arraySet(canais, 1, b)
within
...
```

Porém, como a variável `canais` não foi mapeada para uma seqüência CSP de elementos, pois sua declaração é tratada pela Regra 7, a função arraySet(.) não funciona. Esta função só trabalha com elementos do tipo seqüência CSP.

Devido a esta dificuldade, neste trabalho estamos considerando somente a captura de arrays de canais cujo mapeamento resulta em uma seqüência de canais. Portanto, um canal declarado sem inicialização é declarado como uma seqüência do tipo < 0, ..., 0 >, onde cada 0 (zero) representa um elemento

nulo. O tamanho da seqüência é dado pelo tamanho do array informado na inicialização:

new Type[length] ↪ < 0, …, 0>

Condições:

- *O tamanho da seqüência gerada é igual a length.*
- *O array deve ser preenchido posteriormente com canais já declarados*

Para isto funcionar, o array deve ser preenchido com canais já declarados anteriormente. Isto foi considerado porque seqüências de arrays de canais são muito úteis nas operações de escolha.

Portanto a escrita seria realizada na forma:

c[i].write(exp) ↪ arrayGet(c, i) ! expT ⟦Expression⟧

A regra que trata a operação de escolha foi outra regra que apresentou problemas na implementação.

Regra 41 (expT). *Alternative Transformation*
Alternative Identifier = new Alternative(Expression); Statement
↪ [ ] Identifier : set(listExpT ⟦Expression⟧) @

comT ⟦Statement[indx(Identifier, listExpT ⟦Expression⟧)/Identifier.select()] ⟧

onde: indx(e, s) = if e == head(s) then 0 else 1 + indx(e,tail(s))
indx(e, < >) = 0

Condição: *O bloco de comandos* Statement *deve conter o comando Identifier.select( ) como seu primeiro comando.*

A regra assume um padrão de entrada pouco genérico: a resolução de uma escolha (através da chamada do método `select`()) deve ser sempre realizada após a instanciação de um objeto do tipo Alternative.

Além disso, a função auxiliar indx(.) retorna sempre o índice da primeira instância do canal procurado. Na prática, Expression pode ser uma lista onde a instância de um canal pode aparecer mais de uma vez.

Quando a instância de um canal que está em mais de uma posição no array é selecionada, a escolha em JCSP é não-determinístico. Por exemplo, considere o seguinte Processo:

```
public class Escolha extendes CSProcess{

      private AltingChannelInput canal1;
      private AltingChannelInput canal2;

      public Escolha(AltingChannelInput c1, AltingChannelInput c2){
            canal1 = c1;
            canal2 = c2;
      }

      public void run(){
            Guard guarda = new Guard[]{canal1, canal2};
            Alternative escolha = new Alternative(guarda);
            int i = escolha.select;
            guarda[i].read();
      }
}
```

Caso a inicialização deste processo seja realizada com uma mesma instância de um canal servindo de argumento para os dois parâmetros do construtor, a escolha entre o canal escolhido para leitura seria não determinística em JCSP. Abaixo segue uma ilustração da inicialização do processo com uma instância de um canal:

```
...
One2OneChannel canalUnico = new One2OneChannel();
Escolha e = new Escolha(canalUnico, canalUnico);
...
```

O mapeamento definido pela Regra 41 não está considerando este cenário, já que a função indx(.) retorna sempre somente o primeiro índice de ocorrência do canal na expressão Expression. Para preservar a semântica neste caso específico, a regra deveria resultar em uma escolha entre os índices de cada ocorrência da instância do canal. A solução para o problema ainda está sendo investigada.

Um dos operadores mais importantes de JCSP é o operador de paralelismo. O mapeamento do operador de paralelismo de JCSP para o operador de CSP não é direto, já que a biblioteca JCSP é baseada na versão de Hoare de CSP. O operador de paralelismo de Hoare é diferente do operador da versão moderna, como já foi explicado no capítulo 3 deste trabalho.

Regra 40 (expT). *Parallel Transformation*
new Parallel(Expression)
↪ ||j:{0..#listExpT ⟦Expression⟧ - 1} @ [α exp[j]] exp[j]

onde: exp[j] = arrayGet(listExpT ⟦Expression⟧ , j)

Condição: *O operador* α *aplicado ao nome do processo retorna o alfabeto do mesmo.*

Podemos ver que o paralelismo de JCSP é mapeado no paralelismo alfabetizado de CSP. Na regra, todos os processos devem sincronizar em seus alfabetos.

A dificuldade de implementar esta regra reside no fato de o operador α, que retorna o alfabeto de um processo, não existir na versão moderna de CSP. Ao contrário da versão de CSP de Hoare, na versão moderna o alfabeto está associado a toda a especificação. Não existe o conceito de alfabeto de um processo.

A estratégia de colocar os processos em paralelo sincronizando em todo o alfabeto da especificação não funciona, pois se um dos processos

realizar uma comunicação de um evento que só pertence ao seu alfabeto, todo o sistema entrará em *deadlock*. Os processos devem sincronizar somente nos eventos em comum aos alfabetos, para preservar a semântica de JCSP.

Porém, obter o alfabeto de um processo a partir do código do programa JCSP é uma tarefa difícil. Primeiramente, vemos que não adianta olhar para a classe que define o processo. No mapeamento desta classe os canais viram variáveis de estado daquele processo. Para descobrirmos o verdadeiro alfabeto de um processo, devemos considerar a sua inicialização na classe Main.

Portanto, para o processo `Eco` definido na seção 3.1, quando ocorrer a seguinte a inicialização

```
One2OneChannel in = new One2OneChannel();
One2OneChannel out = new One2OneChannel();
Eco eco = new Eco(in, out);
```

O alfabeto do processo será {`in`, `out`}, independentemente do nome dos atributos dos canais na classe Eco. Ou seja, o alfabeto está associado ao objeto e não à classe que define o processo. O problema é que há casos em que não é possível definir o alfabeto do objeto em tempo de compilação, como no exemplo:

```
One2OneChannel in[] = One2OneChannel.create(5);
One2OneChannel out[] = One2OneChannel.create(5);
Eco eco[] = new Eco[5];

int  i = 0;
while( i < 5) {
      eco[i] = new Eco(in[i], out[i]);
      i++;
}
```

Note que, a partir do formato da Regra 40, os alfabetos dos processos devem estar previamente resolvidos e dispostos de maneira que seja possível

acessar o alfabeto a partir de uma dada instância de um processo. No exemplo acima, o valor da variável `i` só é conhecido em tempo de execução, o que impede a definição do alfabeto de cada processo previamente, em tempo de compilação.

Além disso, se um processo possui como atributo um canal `procCanal` do tipo `ChannelInt`, que trabalha com valores inteiros, mas o processo só escreve o valor 5 no canal, por exemplo, o alfabeto do processo não contém todos os eventos associados ao canal. Somente o evento `procCanal`.5 pertenceria ao alfabeto do processo. Então, também não é suficiente olhar somente para a inicialização do processo na classe Main. É preciso saber com quais valores dos canais o processo está trabalhando para definir seu alfabeto corretamente. Isto cria mais uma dificuldade na definição do seu alfabeto.

Neste trabalho consideramos as seguintes restrições:

- Os processos em paralelo devem sincronizar em todo o alfabeto da especificação; e
- Apenas dois processos podem compor o sistema paralelo.

A primeira restrição foi imposta devido à dificuldade de obtermos o alfabeto de um determinado processo. A segunda é devido ao fato de JCSP não permitir multi-sincronização. Como os objetos sincronizam em todos os eventos, o mapeamento só preserva a semântica se considerarmos apenas 2 processos.

Portanto, o mapeamento obtido é dado por:

new Parallel(Expression)

↪ ||j:{0..#listExpT ⟦Expression⟧ - 1} @ [**Events**] exp[j]

onde: exp[j] = arrayGet(listExpT ⟦Expression⟧ , j)

Em CSP_M, Events referencia todos os eventos da especificação.

A chamada de métodos como comando foi capturada a partir da seguinte regra:

Regra 26 (comT). *Method Call Transformation*

Identifier(Expression) ↪ bodyOf(Identifier(Expression))

Condição: *No corpo do método* Identifier *não deve ocorrer o comando* return.

A função bodyOf(.) retorna o corpo do método já devidamente transformado pela aplicação de comT ⟦.⟧ aos comandos do corpo deste método. A restrição de não admitir comandos return deve-se a esta estratégia.

Um dos problemas desta regra é que a estratégia de simplesmente expandir o corpo do método não trata adequadamente a questão de variáveis locais de um método. O escopo destas variáveis deveria estar restrito ao bloco em que foram declaradas. Para exemplificar um cenário onde isto traz problemas, considere uma chamada a um método que somente declara uma variável:

```
...
      int i = 1;
      this.declararI();
...

public void declararI(){
      int i = 2;
}
...
```

Após o mapeamento do trecho de código, teremos:

```
...
      let
            i = 1
      within
```

```
      let
            i = 2
      within
...
```

A segunda declaração está sobrescrevendo a primeira. O valor da variável `i` após a chamada do método será 2, e não 1 como deveria.

Além deste problema, fica difícil fazer a ligação entre os argumentos da chamada do método e os parâmetros formais da definição do corpo do método. Em Java, a passagem de parâmetros se dá por valor para tipos primitivos e mudanças nessas variáveis dentro do método não deveriam influenciar as variáveis do argumento Expression. Em caso de passagem de objetos como parâmetros, como um array, por exemplo, as mudanças deveriam persistir.

Devido a estas dificuldades, este trabalho só está considerando métodos sem parâmetros. Além disso, os nomes das variáveis locais devem ser únicos naquela classe.

Para mapear uma programa JCSP completo, é preciso uma regra que capture a chamada do método `run`() da instância de um processo. O tratamento desta chamada não é trivial, já que a mesma instância de um processo pode ser executada seqüencialmente mais de uma vez e o estado final do processo deve ser preservado de uma chamada para outra. Neste trabalho, consideramos uma única chamada de processos que possuem execução infinita. Ou seja, os processos executados possuem um laço infinito no corpo do método run(). O mapeamento é definido por:

Identifier.run() ↪ Identifier

Condição: Identifier *referencia uma instância de um processo JCSP*.

A estratégia de realizar uma chamada ao processo impede que os comandos que estejam após a chamada sejam considerados. Isto não é um

problema quando os processos possuem uma execução infinita, pois os próximos comandos nunca serão alcançados.

As próximas três dificuldades encontradas estão relacionadas à propagação de estados em composições seqüenciais de comandos. A primeira delas define como se dá a propagação:

Regra 28 (expT). *Sequential Composition Transformation*
Statement1 ; Statement2

↪ comT ⟦Statement1⟧ *[* Update(F') / SKIP *]*

Update(F) = comT ⟦Statement2⟧

Condições:

- F *representa o conjunto de variáveis modificadas por* Statement1 *e as utilizadas por* Statement2*.*
- *O nome do processo* Update() *deve ser livre no contexto*.
- F' *contém as variáveis de F no seu estado atual*.
- *O processo* Update() *deve ser criado no nível global da especificação, exceto quando haja a necessidade dele pertencer a uma declaração local.*

No trabalho, porém, a quarta condição não está bem definida. Não está claro quando há a necessidade do processo Update() ser criado localmente. Além disso, a composição seqüencial de comandos mostrou dificuldades quando combinada com outras duas regras: a primeira define o mapeamento de um comando do tipo loop *while*, que possui um outro bloco de comandos internamente; a segunda define o mapeamento comando do tipo desvio *if*.

Regra 27 (expT). *While Transformation*
while(Expression) { Statement1 } Statement2

↪ **let**

while(false, F) = comT ⟦Statement2⟧

while(true, F) = comT ⟦Statement1⟧ *[]* while(expT ⟦Expression'⟧ , F') / SKIP *[]*

within while(expT ⟦Expression⟧ , F)

onde: exp[j] = arrayGet(listExpT ⟦Expression⟧ , j)

Condições:

- F *representa o conjunto de variáveis usadas por* Expression *e* Statement1 *e as utilizadas por* Statement2*.*
- F *não contém variáveis declaradas em* Statement1 *e variáveis que apenas recebem leitura de um canal*.

Note que o comando *while* deve propagar as variáveis que foram modificadas e as utilizadas por Statement2. Portanto, todos os comandos do bloco do while (pertencentes a Statement1) devem propagar essas variáveis. Em Essas variáveis devem ser agregadas ao conjunto F da Regra 28 para cada comando de Statement1.

Além disso, é difícil a integração das regras 27 e 28 com a regra que define o mapeamento do comando *if.*

Regra 30 (expT). *If Transformation*
If (Expression) { Statement1 } else { Statement2 }

↪ **if** expT ⟦Expression⟧ **then** comT ⟦Statement1⟧

else comT ⟦Statement2⟧

O problema encontrado é que o *template* do código de entrada da regra não funciona na integração com a regra 28. Por exemplo, se tivermos um comando if seguido de um bloco de comandos:

if (Expression) { Statement1 } else { Statement2 } Statement3

teríamos com a aplicação da regra 28

comT ⟦if (Expression) { Statement1 } else { Statement2 }⟧

[Update(F') / SKIP *]*

Update(F) = comT ⟦Statement3⟧

Acontece que Statement1 pode realizar atualizações em um conjunto de variáveis e Statement2 em outro. Assim, após Statement1 teríamos um conjunto F1 de variáveis modificadas e após Statement2 teríamos um conjunto F2. Assim, na realidade deveríamos definir um processo Update(F1) = comT ⟦Statement3⟧ , para propagar F1 e um outro processo Update2(F2) = comT ⟦Statement3⟧ , para propagar F2.

Note que F1 e F2 podem distinguir pelo conjunto de variáveis, e não somente pelo estado das variáveis. O conjunto F de variáveis que devem ser propagadas dentro do comando *while* deve englobar as variáveis de F1 e as variáveis de F2. Portanto, se tivermos um comando *if* dentro de um comando *while*:

while(Expression){

if(Expression2) { Statement1 } else { Statement2}

}

Statement3

Teríamos após a aplicação da Regra 27:

let

while(false, F) = comT ⟦Statement3⟧

while(true, F) = comT ⟦if(Expression2) { Statement1 } else {

Statement2}⟧ *[* while(expT ⟦Expression'⟧ , F') / SKIP *]*

within while(expT ⟦Expression⟧ , F)

Que, após aplicação da Regra 30, resulta em:

let

while(false, F) = comT ⟦Statement3⟧

while(true, F) = (**if** expT ⟦Expression2⟧ comT ⟦Statement1⟧

else comT ⟦Statement2⟧)

[while(expT ⟦Expression'⟧ , F') / SKIP *]*

within while(expT ⟦Expression⟧ , F)

A mesma substituição *[*while(expT ⟦Expression'⟧ , F') / SKIP *]* para Statement1 e Statement2 não funciona, já que as variáveis do conjunto F podem estar em estados diferentes no fim de comT ⟦Statement1⟧ e comT ⟦Statement2⟧ .

Como tivemos dificuldades em implementar a propagação de estados dessas três regras, o trabalho atual não considera a atualização de variáveis dentro de blocos do comando *if* e do comando *while*.

Uma outra difculdade encontrada é que as regras foram criadas para resultar em um único arquivo CSP, para cada sistema JCSP fornecido. Porém, para adequação ao nosso propósito, precisamos realizar uma adaptação para que possamos carregar na ferramenta FDR todos os processos dos dois sistemas (o original e o refatorado).

A solução que encontramos foi acrescentar uma diretiva (include) no início do arquivo do sistema refatorado para incluir o arquivo CSP do modelo original. Para não haver conflitos de nomes de processos (redefinições), os processos no script CSP refatorado recebem o prefixo "_Refatorado" (automaticamente). Além disso, as funções auxiliares (`arrayGet`, `ArraySet` e `indx`) e os canais só são declarados no arquivo CSP do sistema refatorado. Por isso, quando o usuário está gerando o modelo do script refatorado, ele deve informar na ferramenta a localização do arquivo CSP do modelo original. O manual do protótipo fornece os passos de utilização no ANEXO deste trabalho.

5. Estudo de Caso

Este capítulo ilustra o funcionamento da nova abordagem de verificação através de dois exemplos: primeiramente aplicamos uma refatoração que não conserva o comportamento externo do programa e constatamos que a nossa abordagem aponta a falha. Depois de identificado o erro, a correção é feita e contatamos que a nossa abordagem verifica que a transformação foi correta.

O exemplo utilizado neste estudo de caso descreve um modelo fictício de uma máquina de refrigerantes. Seguindo a especificação:

- É assumido que a máquina receberá apenas moedas de $50 e de $100;
- Cada refrigerante custa $100;
- A máquina não dá troco.
- Depois de inserir o valor do refrigerante, o usuário pode escolher o sabor ou pedir devolução do dinheiro. Como a máquina não dá troco, o usuário só pode pedir devolução se não tiver escolhido um refrigerante.

Suponhamos que tal máquina deva obedecer ao seguinte diagrama de transições:

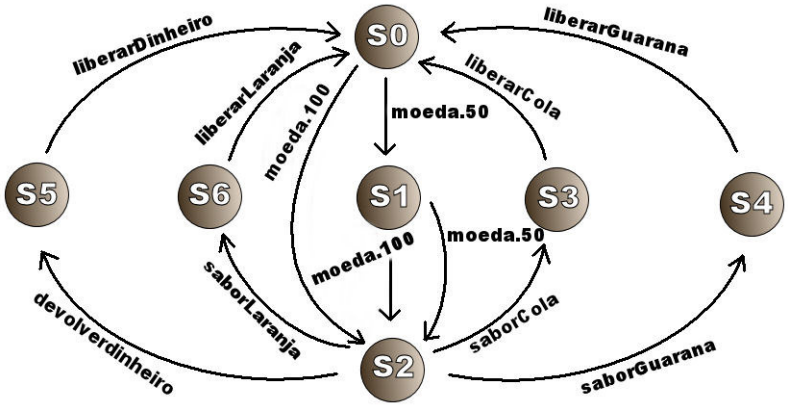


Figura 5 – Diagrama de transições da máquina de refrigerantes

Podemos notar que o modelo apresenta alguns problemas. Um deles, por exemplo, é que a máquina deveria oferecer a opção de devolução do dinheiro mesmo se o usuário não tivesse atingido o valor do refrigerante. De fato, este modelo simplificado será utilizado apenas para ilustrar a aplicação de algumas refatorações. Em um caso real a máquina deveria ser modelada de outra forma.

Na figura, Si representa o i-ésimo estado. Os eventos moeda.50 e moeda.100 representam a maquina recebendo moedas de $50 e de $100 respectivamente. Quando a maquina recebe uma moeda de $50, o valor do refrigerante ainda não foi atingido. Então, a máquina espera uma outra moeda antes de oferecer as opções dos sabores de refrigerantes ou devolução do dinheiro. Depois de inserir uma nova moeda, o usuário pode escolher um sabor de refrigerante ou requisitar o dinheiro de volta.

Os eventos saborLaranja, saborCola, saborGuarana e devolverDinheiro representam a escolha do usuário. Já os eventos liberarDinheiro, liberarCola, liberarGuarana e liberarLaranja representam a máquina executando o pedido do usuário.

A codificação em JCSP da máquina seria representada por uma classe que implementa a interface CSProcess :

```
public class Maquina implements CSProcess {

      private ChannelInputInt moeda;
      private AltingChannelInput saborCola;
      private AltingChannelInput saborLaranja;
      private AltingChannelInput saborGuarana;
      private AltingChannelInput devolverDinheiro;
      private ChannelOutput liberarCola;
      private ChannelOutput liberarLaranja;
      private ChannelOutput liberarGuarana;
      private ChannelOutput liberarDinheiro;

      public Maquina(ChannelInputInt m, AltingChannelInput sc,
                  AltingChannelInput sl, AltingChannelInput sg,
                  AltingChannelInput dd, ChannelOutput lc,
                      ChannelOutput ll, ChannelOutput lg,
                      ChannelOutput ld ){
            this.moeda = m;
            this.saborCola = sc;
            this.saborLaranja = sl;
            this.saborGuarana = sg;
            this.devolverDinheiro = dd;
            this.liberarCola = lc;
            this.liberarLaranja = ll;
            this.liberarGuarana = lg;
            this.liberarDinheiro = ld;
      }
```

A máquina possui um canal moeda que trabalha com valores inteiros. Através deste canal a máquina receberá o valor que representa a moeda inserida. Os canais saborCola, saborLaranja, saborGuarana e devolverDinheiro também são de entrada, e representarão a escolha do usuário. Analogamente, os canais liberarCola, liberarLaranja, liberarGuarana e liberarDinheiro são canais de saída e representam a resposta da máquina ao pedido do usuário.

```
      public void run(){
            AltingChannelInput[] guarda = new AltingChannelInput[4];
            guarda[0] = saborCola;
            guarda[1] = saborLaranja;
            guarda[2] = saborGuarana;
            guarda[3] = devolverDinheiro;
```

```
        while(true){
              int valor = moeda.read();
              if(valor >= 100 ){
                    Alternative alt = new Alternative(guarda);
                    int indice = alt.select();
                    guarda[indice].read();
                    if(indice == 0){
                          liberarCola.write(null);
                    } else if(indice == 1){
                          liberarLaranja.write(null);
                    } else if(indice == 2){
                          liberarGuarana.write(null);
                    } else {
                          liberarDinheiro.write(null);
                    }
              }
```

O método `run()` define o comportamento da máquina. Inicialmente é criada uma guarda como variável local do método, para ser utilizada na escolha do usuário. A guarda é inicializada com todos os canais de entrada.

O laço infinito define o comportamento da máquina após a inicialização. Primeiramente, um valor de moeda é lido. Caso o valor tenha atingido o preço do refrigerante, é criado um objeto do tipo `Alternative`, que vai oferecer as opções de canais de entrada que estão no *array* `guarda`. Depois da opção do usuário ser lida, a máquina comunica através do canal de saída correspondente que o pedido foi atendido. Como os canais de saída não estão num array, o índice da posição da guarda referente ao pedido do usuário precisa ser testado para achar o canal correto.

```
              else {
                    moeda.read();
                    Alternative alt = new Alternative(guarda);
                    int indice = alt.select();
                    guarda[indice].read();
                    if(indice == 0){
                          liberarCola.write(null);
                    } else if(indice == 1){
                          liberarLaranja.write(null);
                    } else if(indice == 2){
                          liberarGuarana.write(null);
                    } else {
                          liberarDinheiro.write(null);
                    }
              }
        }
  }
```

```
}
```

Caso o usuário tenha inserido uma moeda de $50, o usuário deve colocar uma outra moeda. Apesar de o canal moeda ser do tipo inteiro, assumimos na especificação que a máquina só receberá valores $50 ou $100. Portanto, quando o usuário inserir uma segunda moeda, o valor total inserido obrigatoriamente atingirá o valor do refrigerante. A máquina não precisa guardar o valor total, pois não fornece troco.

Depois de receber o valor da moeda, a máquina pode oferecer as opções de escolha ao usuário e atender o pedido correspondente.

Para interagir com este processo, podemos definir um processo Consumidor que realiza pedidos e aguarda o atendimento desses pedidos:

```
public class Consumidor implements CSProcess {

      private ChannelOutputInt moeda;
      private ChannelOutput escolherCola;
      private ChannelOutput pedirDevolucao;
      private ChannelInput receberCola;
      private ChannelInput receberDevolucao;

      public Consumidor(ChannelOutputInt m, ChannelOutput ec,
                        ChannelOutput pd, ChannelInput rc,
                        ChannelInput rd) {

            this.moeda = m;
            this.escolherCola = ec;
            this.pedirDevolucao = pd;
            this.receberCola = rc;
            this.receberDevolucao = rd;
      }
```

As funções dos canais deste processo são análogas às funções dos canais do processo anterior. Os canais de saída deste processo correspondem a canais de entrada do processo máquina e vice-versa. Este processo tem um número menor de canais, pois este consumidor realiza um número menor de tipos de pedidos, como podemos ver no método run( ) :

```
      public void run(){
            while(true){
```

```
                moeda.write(50);
                moeda.write(50);
                escolherCola.write(null);
                receberCola.read();
                moeda.write(100);
                pedirDevolucao.write(null);
                receberDevolucao.read();
            }
        }
}
```

O consumidor realiza sempre a mesma seqüência de ações: insere duas moedas de $50, escolhe um refrigerante de cola, recebe o refrigerante, insere uma moeda de $100, pede devolução do dinheiro e recebe a devolução.

O sistema define os canais e inicializa uma instância de cada um desses processos interagindo em paralelo:

```
public class Sistema {

  public static void main(String argv[]){

      One2OneChannelInt moeda = new One2OneChannelInt();

      One2OneChannel escolhaCola = new One2OneChannel();
      One2OneChannel escolhaLaranja = new One2OneChannel();
      One2OneChannel escolhaGuarana = new One2OneChannel();
      One2OneChannel escolhaDevolucao = new One2OneChannel();
      One2OneChannel entregaCola = new One2OneChannel();
      One2OneChannel entregaLaranja = new One2OneChannel();
      One2OneChannel entregaGuarana = new One2OneChannel();
      One2OneChannel entregaDinheiro = new One2OneChannel();

      MaquinaRefatorada m =
                new MaquinaRefatorada(moeda, escolhaCola, escolhaLaranja,
                                      escolhaGuarana, escolhaDevolucao,
                                      entregaCola, entregaLaranja,
                                      entregaGuarana, entregaDinheiro);

      Consumidor c = new Consumidor(moeda, escolhaCola, escolhaDevolucao,
                                    entregaCola, entregaDinheiro);

      CSProcess processos[] = new CSProcess[2];
      processos[0] = m;
      processos[1] = c;

      Parallel sistema = new Parallel(processos);
      sistema.run();

  }
}
```

5.1 Verificando a refatoração

Suponhamos que, após a codificação apresentada, o desenvolvedor detectou uma oportunidade de refatoração no método run( ) do processo Maquina. Verificou que o método poderia ser reescrito como:

```
public void run(){
      while(true){
            int valor = moeda.read();
            if(valor >= 100 ){
                  this.atenderPedido();
            } else {
                  moeda.read();
                  this.atenderPedido();
            }
      }

}
```

onde `atenderPedido( )` é o novo método da classe, criado para lidar com a duplicação de código:

```
public void atenderPedido(){
      AltingChannelInput[] guarda = new AltingChannelInput[4];
      guarda[0] = saborCola;
      guarda[1] = saborLaranja;
      guarda[2] = saborGuarana;
      guarda[3] = devolverDinheiro;

      Alternative alt = new Alternative(guarda);
      int indice = alt.select();
      guarda[indice].read();
      if(indice == 0){
            liberarDinheiro.write(null);
      } else if(indice == 1){
            liberarCola.write(null);
      } else if(indice == 2){
            liberarLaranja.write(null);
      } else {
            liberarGuarana.write(null);
      }
}
```

A refatoração realizada eliminou a replicação de código[1]. Este tipo de refatoração, conhecido como Extract Method [21], é utilizado não somente para evitar replicação, como também para obter maior modularização e facilitar a leitura do código.

O desenvolvedor, para verificar a tranformação, gera o modelo dos sistemas a partir do protótipo[2]. As figuras abaixo mostram a geração dos modelos do sistema original e do sistema refatorado:

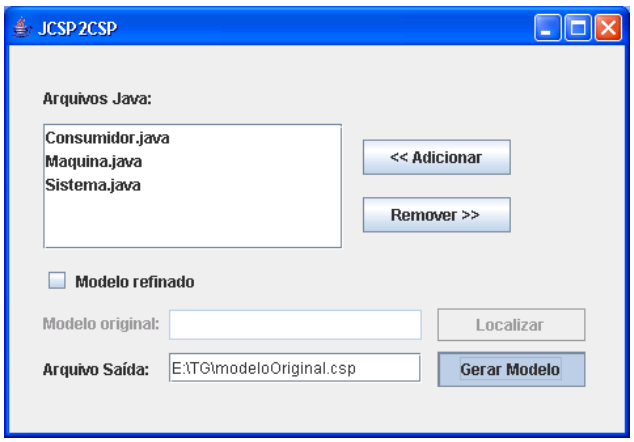


Figura 6 – Geração do modelo do sistema original

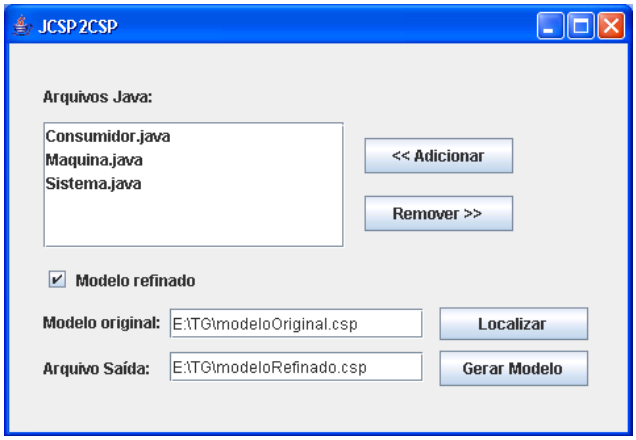


Figura 7 – Geração do modelo do sistema Refatorado

[1] Idealmente, o array de guarda deveria ser criado no início método run( ), fora do bloco while( ), e ser um parâmetro do método atenderPedido( ). Assim, seria evitado que uma nova instância do objeto fosse criada a cada execução do laço. Em versões futuras do processo de abstração, quando as passagens de parâmetros em chamadas de métodos forem tratadas corretamente, o problema poderá ser contornado.

[2] Os códigos dos scripts CSP encontram-se no Apêndice A. O protótipo se encontra disponível em http://www.cin.ufpe.br/~joclj/jcsp2csp.zip

São gerados dois arquivos CSP: o primeiro contém a mapeamento do sistema original e o segundo contém o mapeamento do sistema refatorado. O desenvolvedor deve carregar no FDR o arquivo CSP do modelo refatora, que contém uma diretiva (`include`) para incluir o arquivo CSP do modelo orginal.

Então, ele verifica o refinamento no FDR, que apresenta o resultado: os dois sistemas não são equivalentes. A figura abaixo ilustra uma captura da tela do FDR:

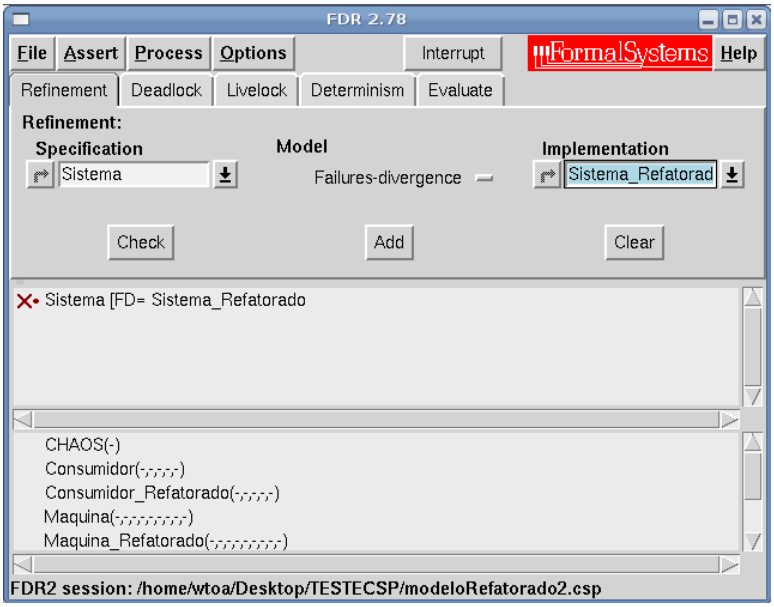


Figura 8 – Refatoração incorreta

O desenvolvedor revisa o código e descobre o erro: os canais de saída não estão colocados corretamente nos blocos do comando `if`. O canal de entrada representa o pedido do usuário e o canal de saída a execução do pedido. No método atenderPedido(), as execuções dos pedidos não estão correspondendo com as requisições. O desenvolvedor então reescreve o método como:

```
public void atenderPedido(){
      AltingChannelInput[] guarda = new AltingChannelInput[4];
      guarda[0] = saborCola;
      guarda[1] = saborLaranja;
      guarda[2] = saborGuarana;
      guarda[3] = devolverDinheiro;
```

```
        Alternative alt = new Alternative(guarda);
        int indice = alt.select();
        guarda[indice].read();
        if(indice == 0){
              liberarCola.write(null);
        } else if(indice == 1){
              liberarLaranja.write(null);
        } else if(indice == 2){
              liberarGuarana.write(null);
        } else {
              liberarDinheiro.write(null);
        }
}
```

O desenvolvedor gera os modelos dos sistemas no protótipo novamente e, a partir do script CSP gerado, faz uma nova verificação no FDR. A ferramenta verifica o refinamento e o desenvolvedor constata que a verificação está correta. A figura abaixo mostra a captura da tela do FDR na nova verificação:

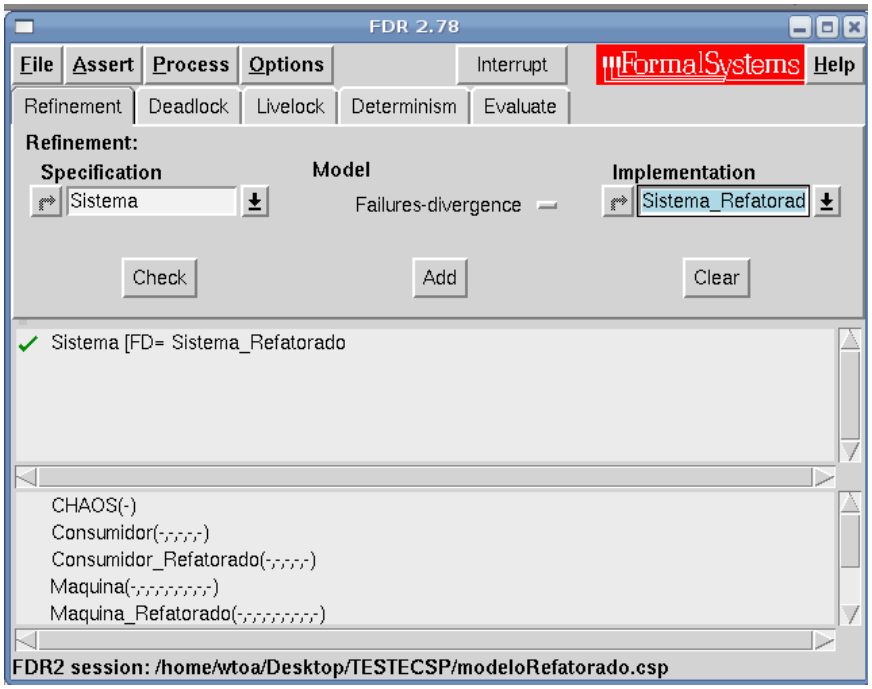


Figura 9 – Refatoração correta

6. Conclusão

Neste trabalho propusemos uma nova abordagem para realizar a verificação de refatorações de código Java, focando inicialmente em um subconjunto da biblioteca JCSP. Esta abordagem é baseada na verificação automática de refinamentos entre modelos. Dessa forma, para aplicar a abordagem como forma de verificar refatoramentos de programas Java (JCSP), o primeiro passo que fizemos foi mostrar como capturar programas Java, antes e após o refatoramento, como modelos descritos em CSP. Esta tradução é automática baseada em uma ferramenta que elaboramos.

Esta abordagem é vantajosa em relação ao uso de testes automatizados na verificação em dois aspectos: o programador não precisa escrever uma suíte de testes, que é uma tarefa difícil e trabalhosa; e uma confiabilidade maior é obtida, visto que o uso de testes só garante que a funcionalidade do sistema está preservada para os casos testados.

Apresentamos uma introdução aos principais conceitos de CSP utilizados no trabalho com exemplos práticos para ilustrá-los. Também apresentamos elementos da linguagem Java, focando na API JCSP, que é uma biblioteca que trata concorrência de maneira bastante próxima de CSP.

Mostramos as tarefas básicas que uma ferramenta de geração automática de modelos a partir de códigos de programas deve realizar. Além da explicação das etapas que devemos realizar no processamento do código (análise sintática, análise contextual e geração de código), apresentamos também ferramentas que auxiliam algumas dessas etapas.

Indicamos o trabalho [13], que fornece as regras que poderiam ser utilizados na geração de código. Realizamos uma investigação dessas regras e encontramos alguns problemas nas regras. Um deles, por exemplo, era a

declaração de Canais. Segundo a regra, Canais JCSP do tipo ChannelInt deveriam ser mapeados em canais CSP que transportam valores Int. Porém, o tipo Int de CSP é infinito e, como as ferramentas trabalham com modelos finitos, a definição de um canal desta maneira inviabializa a verificação do modelo. A ferramenta FDR exibe uma mensagem de alerta e não realiza nenhuma verificação. Como solução *ad hoc* para o problema, consideramos o mapeamento do tipo para um conjunto de inteiros num intervalo finito reduzido.

Além dos problemas das regras em si, encontramos dificuldades na implementação das regras. Estas dificuldades implicaram em restrições na ferramenta desenvolvido. Algumas das restrições são:

- Devido à dificuldade de se encontrar o alfabeto de um processo, na regra 40 (*Parallel Transformation*), consideramos que os processos em paralelo devem sincronizar em todo o alfabeto da especificação. Isto funciona quando os processos comunicam apenas eventos que são pertencentes aos alfabetos de ambos os processos. Além disso, como JCSP não permite multi-sincronização e os processos sincronizam em todos os eventos, o mapeamento só funciona quando consideramos apenas dois processos em paralelo.

- Devido à dificuldade encontrada na propagação de estados em combinações das Regras 28 (*Sequential Composition Transformation*), 27 (*While Transformation*) e 30 (*If Transformation*), as atribuições em variáveis dentro de blocos while não preservam o estado de uma iteração para outra. Além disso, não são consideradas atribuições de variáveis em blocos de comando *if*.

Porém, o mais importante não é as limitações do protótipo desenvolvido, e sim a ilustração da idéia de que refatoramento via verificação

de refinamentos funciona na prática, uma vez que o mapeamento esteja implementado corretamente.

Vale explicitar que uma das limitações inerentes da abordagem é referente à explosão do espaço de estados do modelo. As ferramentas de verificação, como o FDR, carregam todo o espaço de estados do programa na memória e, por isso, levam um tempo muito grande na verificação.

FDR tem evoluído no sentido de lidar melhor com o problema. Ela realiza uma compressão no espaço de estados gerado de maneira que o número de estados que podem ser carregados na memória é muito maior. Além disso, a eficiência foi aumentada. Mesmo assim, o tempo de verificação ainda é muito longo, por isso a nossa abordagem se torna desinteressante se o código a ser verificado for muito grande.

6.1 Trabalhos relacionados

O trabalho [31] é o mais próximo de nossa abordagem. Ele também busca um embasamento formal na realização de refatorações. Porém, o trabalho difere do nosso em diversos pontos.

Primeiramente, esta outra abordagem não é baseada na verificação do código. O trabalho apresenta leis que definem para cada transformação *Ta* realizada no modelo uma *Regra a*, que aponta uma seqüência de transformações *Tx, ..., Ty* a serem realizadas no programa. Conseqüentemente, cada transformação realizada no modelo acarreta numa seqüência de transformações realizadas no programa.

Portanto, a refatoração é aplicada nos dois níveis, para gerar um modelo e um novo programa refatorados, sendo que a refatoração no programa pode ser realizada automaticamente, após a refatoração no modelo.

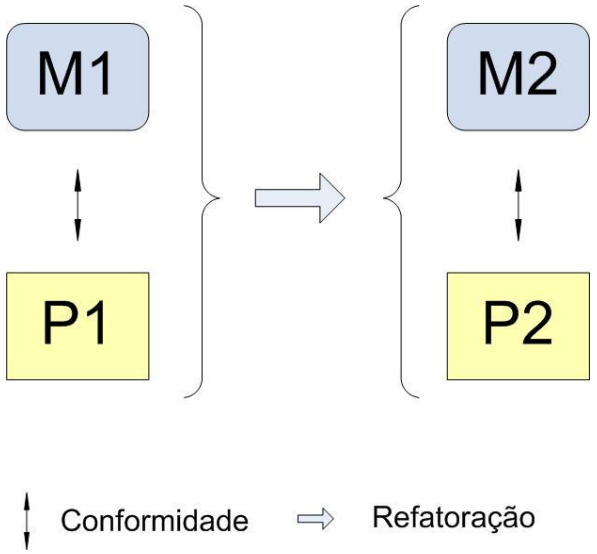


Figura 10 – Uma outra abordagem de verificação

Para assumir a equivalência semântica entre o modelo e o programa, ele introduz o conceito de *conformidade estrutural*[25]. Além da conformidade estrutural, o programa também deve manter as invariantes do modelo sempre válidas, para que estas invariantes possam ser utilizadas para obter a estrutura da *heap* do programa que, uma vez conhecida, possibilita a modificação automática no código do programa.

As linguagens consideradas neste trabalho são diferentes das linguagens consideradas no nosso: a linguagem formal utilizada é Alloy [27], uma linguagem formal que permite um tratamento mais adequado a orientação a objetos (na realidade, está sendo considerado um subconjunto de Alloy); e a linguagem de programação é ROOL [24], uma linguagem semelhante a Java, mas que não possui mecanismos de referência e concorrência.

Os refatoramentos de Alloy e ROOL são provados formalmente que preservam comportamento com relação a uma semântica formal. Por isso, esta abordagem tem a vantagem de não ter que realizar uma verificação a cada refatoração.

Uma vantagem de nossa abordagem em relação a essa é que ela permite que o programador trabalhe no nível de código. O programador não precisa dominar os formalismos e as notações de uma linguagem de especificação formal para desenvolver o sistema. Desta maneira, a equipe de desenvolvimento pode se adaptar mais facilmente à nossa abordagem.

O trabalho [30] também busca uma maior confiabilidade na realização de refatorações. Porém, as refatorações são consideradas no nível de diagramas de classe UML [28]. É introduzida a noção de equivalência classes de diagramas de uma maneira flexível, comparando a semântica dos diagramas de classe somente para elementos (atributos, classes e associações) considerados relevantes. Se estes elementos possuem a mesma interpretação nos dois diagramas de classe, os diagramas são assumidos como equivalentes.

Em seguida, o trabalho define leis de transformação que preservam a semântica do diagrama UML e uma translação semântica de diagramas de classe UML para Alloy. A translação é empregada para provar a corretude das leis de transformação de diagramas de classe UML.

Este trabalho é diferente do nosso, pois não verifica as refatorações realizadas pelo desenvolvedor. Ele define um conjunto de leis de transformações que preservam a semântica para o desenvolvedor utilizar na refatoração. Além disso, a transformação considerada é em diagramas de classe UML, que não são muito utilizadas em metodologias ágeis, onde refatorações costumam ser utilizadas de maneira extensiva e fazem parte do ciclo de desenvolvimento do projeto.

6.2 Trabalhos Futuros

Os trabalhos futuros para concepção da abordagem devem realizar esforços em duas frentes: uma deve prosseguir na definição de regras de mapeamento e a outra na concepção da ferramenta.

A primeira frente teria como objetivo capturar mais elementos de orientação a objetos de Java, como atributos estáticos e herança. Além disso, outras bibliotecas de Java, como a biblioteca de concorrência com primitivas de sincronização baseada em monitores (`wait`, `synchronize`, `notify`, ...). Também poderiam ser definidas regras para melhorar a estrutura do script CSP gerado para melhorar a legibilidade. O script gerado atualmente ainda é difícil de ler.

A segunda frente poderia realizar a implementação de uma ferramenta de maneira mais eficiente e amigável. A ferramenta poderia ser implementada através de um *plugin* de alguma ferramenta de ambiente de desenvolvimento integrado (IDE). Dessa forma, o processo de verificação dos modelos seria mais ágil, visto que o desenvolvedor não precisaria iniciar uma nova ferramenta realizar a verificação.

Seria interessante que as duas frentes fossem conduzidas em paralelo, já que a segunda ajuda a verificar a primeira. A consistência das regras definidas não foi provada (de fato, a prova seria bastante complexa e trabalhosa) e a intuição advém de experimentos práticos realizados. A implementação das regras ajuda a consolidar a idéia da consistência das regras ou apontar erros e falta de generalidade.

O mapeamento também poderia considerar no mapeamento outras linguagens de programação, como C++ [18]. Uma outra possibilidade é considerar o mapeamento para uma linguagem formal que possua um tratamento mais adequado a orientação a objetos, como *OhCircus*[29], que, além de processos, trata naturalmente elementos como classes, herança e ligação dinâmica.

Apêndice A – Scripts CSP do Estudo de Caso

Primeiramente, temos o script CSP gerado no estudo de caso a partir do sistema original, antes das modificações:

```
-- Processo Consumidor
Consumidor( m, ec, pd, rc, rd) = let
moeda = m
escolherCola = ec
pedirDevolucao = pd
receberCola = rc
receberDevolucao = rd
within
let
while(false) = SKIP
while(true) = moeda!50 -> moeda!50 -> escolherCola!0 ->
receberCola?x0'' -> moeda!100 -> pedirDevolucao!0 ->
receberDevolucao?x1'' -> while(true)
within
while(true)

-- Processo Maquina
Maquina( m, sc, sl, sg, dd, lc, ll, lg, ld) = let
moeda = m
saborCola = sc
saborLaranja = sl
saborGuarana = sg
devolverDinheiro = dd
liberarCola = lc
liberarLaranja = ll
liberarGuarana = lg
liberarDinheiro = ld
within
let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
```

```
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
let
while(false) = SKIP
while(true) = moeda?valor -> if (valor >= 100) then
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarCola!0 -> while(true)
else
if (indice == 1) then
liberarLaranja!0 -> while(true)
else
if (indice == 2) then
liberarGuarana!0 -> while(true)
else
liberarDinheiro!0 -> while(true)
else
moeda?x1'' -> [] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x2'' -> if (indice == 0) then
liberarCola!0 -> while(true)
else
if (indice == 1) then
liberarLaranja!0 -> while(true)
else
if (indice == 2) then
liberarGuarana!0 -> while(true)
else
liberarDinheiro!0 -> while(true)
within
while(true)

-- Processo Sistema
```

```
Sistema = let
m = Maquina(moeda, escolhaCola, escolhaLaranja, escolhaGuarana,
escolhaDevolucao, entregaCola, entregaLaranja, entregaGuarana,
entregaDinheiro)
within
let
c = Consumidor(moeda, escolhaCola, escolhaDevolucao, entregaCola,
entregaDinheiro)
within
let
processos = < 0, 0>
within
let
processos_2' = arraySet(processos, 0, m)
within
let
processos_3' = arraySet(processos_2', 1, c)
within
let
sistema = || j : { 0 .. #processos_3' - 1 } @ [ Events ]
arrayGet(processos_3', j)
within
sistema
```

Depois, temos o script da modificação incorreta. Note que os nomes dos processos foram modificados e este script define os canais e funções auxiliares:

```
--Especificacao do modelo original
include "E:\IC\original.csp"

-- Funcoes Auxiliares
arrayGet(<h>^t, p) = if p==0 then h else arrayGet(t,p-1)
arrayGet(<>, p) = p

arraySet(<h>^t, p, e) = if p==0 then <e>^t
                        else <h>^arraySet(t,p-1,e)
arraySet(<>, p, e) = <>

indx(e,s) = if e == head(s) then 0 else 1 + indx(e,tail(s))
indx(e,<>) = 0
```

```
-- Processo Consumidor_Refatorado
Consumidor_Refatorado( m, ec, pd, rc, rd) = let
moeda = m
escolherCola = ec
pedirDevolucao = pd
receberCola = rc
receberDevolucao = rd
within
let
while(false) = SKIP
while(true) = moeda!50 -> moeda!50 -> escolherCola!0 ->
receberCola?x0'' -> moeda!100 -> pedirDevolucao!0 ->
receberDevolucao?x1'' -> while(true)
within
while(true)

-- Processo Maquina_Refatorado
Maquina_Refatorado( m, sc, sl, sg, dd, lc, ll, lg, ld) = let
atenderPedido = let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarDinheiro!0 -> SKIP
else
if (indice == 1) then
liberarCola!0 -> SKIP
else
```

```
if (indice == 2) then
liberarLaranja!0 -> SKIP
else
liberarGuarana!0 -> SKIP
within
let
moeda = m
saborCola = sc
saborLaranja = sl
saborGuarana = sg
devolverDinheiro = dd
liberarCola = lc
liberarLaranja = ll
liberarGuarana = lg
liberarDinheiro = ld
within
let
while(false) = SKIP
while(true) = moeda?valor -> if (valor >= 100) then
let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarDinheiro!0 -> while(true)
else
if (indice == 1) then
liberarCola!0 -> while(true)
```

```
else
if (indice == 2) then
liberarLaranja!0 -> while(true)
else
liberarGuarana!0 -> while(true)
else
moeda?x1'' -> let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarDinheiro!0 -> while(true)
else
if (indice == 1) then
liberarCola!0 -> while(true)
else
if (indice == 2) then
liberarLaranja!0 -> while(true)
else
liberarGuarana!0 -> while(true)
within
while(true)

-- Declaracoes de canais
channel moeda : { -100 .. 100}
channel escolhaCola : { 0 }
channel escolhaLaranja : { 0 }
channel escolhaGuarana : { 0 }
```

```
channel escolhaDevolucao : { 0 }
channel entregaCola : { 0 }
channel entregaLaranja : { 0 }
channel entregaGuarana : { 0 }
channel entregaDinheiro : { 0 }

-- Processo Sistema_Refatorado
Sistema_Refatorado = let
m = Maquina_Refatorado(moeda, escolhaCola, escolhaLaranja,
escolhaGuarana, escolhaDevolucao, entregaCola, entregaLaranja,
entregaGuarana, entregaDinheiro)
within
let
c = Consumidor_Refatorado(moeda, escolhaCola, escolhaDevolucao,
entregaCola, entregaDinheiro)
within
let
processos = < 0, 0>
within
let
processos_2' = arraySet(processos, 0, m)
within
let
processos_3' = arraySet(processos_2', 1, c)
within
let
sistema = || j : { 0 .. #processos_3' - 1 } @ [ Events ]
arrayGet(processos_3', j)
within
sistema
```

Após corrigir a refatoração, o script gerado foi:

```
--Especificacao do modelo original
include "E:\IC\original.csp"

-- Funcoes Auxiliares
arrayGet(<h>^t, p) = if p==0 then h else arrayGet(t,p-1)
arrayGet(<>, p) = p

arraySet(<h>^t, p, e) = if p==0 then <e>^t
                        else <h>^arraySet(t,p-1,e)
```

```
arraySet(<>, p, e) = <>

indx(e,s) = if e == head(s) then 0 else 1 + indx(e,tail(s))
indx(e,<>) = 0

-- Processo Consumidor_Refatorado
Consumidor_Refatorado( m, ec, pd, rc, rd) = let
moeda = m
escolherCola = ec
pedirDevolucao = pd
receberCola = rc
receberDevolucao = rd
within
let
while(false) = SKIP
while(true) = moeda!50 -> moeda!50 -> escolherCola!0 ->
receberCola?x0'' -> moeda!100 -> pedirDevolucao!0 ->
receberDevolucao?x1'' -> while(true)
within
while(true)

-- Processo Maquina_Refatorado
Maquina_Refatorado( m, sc, sl, sg, dd, lc, ll, lg, ld) = let
atenderPedido = let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
```

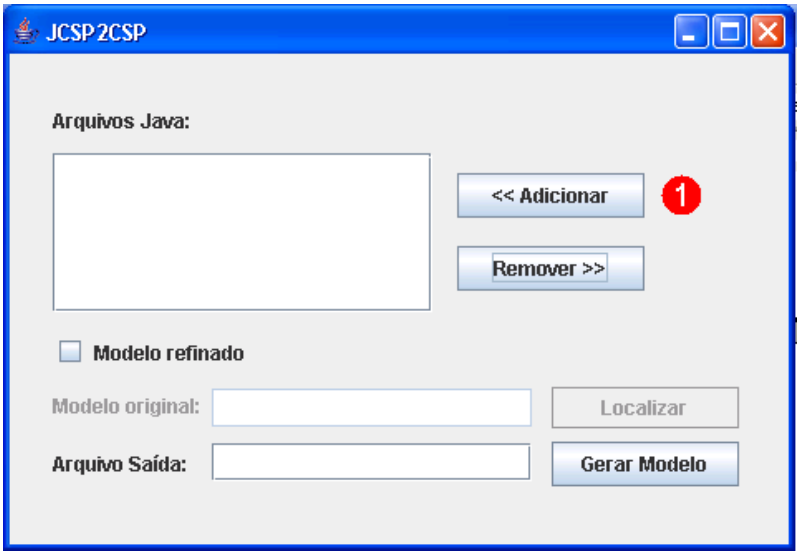
```
liberarCola!0 -> SKIP
else
if (indice == 1) then
liberarLaranja!0 -> SKIP
else
if (indice == 2) then
liberarGuarana!0 -> SKIP
else
liberarDinheiro!0 -> SKIP
within
let
moeda = m
saborCola = sc
saborLaranja = sl
saborGuarana = sg
devolverDinheiro = dd
liberarCola = lc
liberarLaranja = ll
liberarGuarana = lg
liberarDinheiro = ld
within
let
while(false) = SKIP
while(true) = moeda?valor -> if (valor >= 100) then
let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
```

```
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarCola!0 -> while(true)
else
if (indice == 1) then
liberarLaranja!0 -> while(true)
else
if (indice == 2) then
liberarGuarana!0 -> while(true)
else
liberarDinheiro!0 -> while(true)
else
moeda?x1'' -> let
guarda = < 0, 0, 0, 0>
within
let
guarda_2' = arraySet(guarda, 0, saborCola)
within
let
guarda_3' = arraySet(guarda_2', 1, saborLaranja)
within
let
guarda_4' = arraySet(guarda_3', 2, saborGuarana)
within
let
guarda_5' = arraySet(guarda_4', 3, devolverDinheiro)
within
[] alt: set(guarda_5') @ let
indice = indx(alt, guarda_5')
within
arrayGet(guarda_5', indice)?x0'' -> if (indice == 0) then
liberarCola!0 -> while(true)
else
if (indice == 1) then
liberarLaranja!0 -> while(true)
else
if (indice == 2) then
liberarGuarana!0 -> while(true)
else
liberarDinheiro!0 -> while(true)
within
while(true)
```

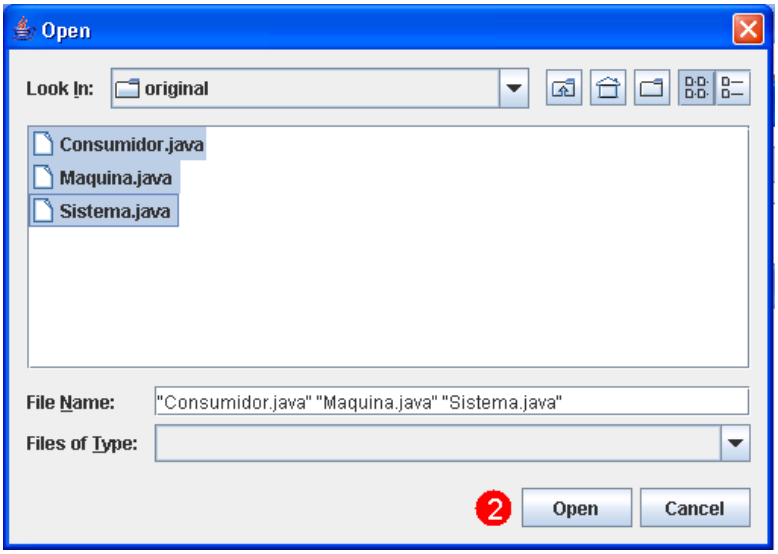
```
-- Declaracoes de canais
channel moeda : { -100 .. 100}
channel escolhaCola : { 0 }
channel escolhaLaranja : { 0 }
channel escolhaGuarana : { 0 }
channel escolhaDevolucao : { 0 }
channel entregaCola : { 0 }
channel entregaLaranja : { 0 }
channel entregaGuarana : { 0 }
channel entregaDinheiro : { 0 }

-- Processo Sistema_Refatorado
Sistema_Refatorado = let
m = Maquina_Refatorado(moeda, escolhaCola, escolhaLaranja,
escolhaGuarana, escolhaDevolucao, entregaCola, entregaLaranja,
entregaGuarana, entregaDinheiro)
within
let
c = Consumidor_Refatorado(moeda, escolhaCola, escolhaDevolucao,
entregaCola, entregaDinheiro)
within
let
processos = < 0, 0>
within
let
processos_2' = arraySet(processos, 0, m)
within
let
processos_3' = arraySet(processos_2', 1, c)
within
let
sistema = || j : { 0 .. #processos_3' - 1 } @ [ Events ]
arrayGet(processos_3', j)
within
sistema
```

Anexo – Manual do Protótipo

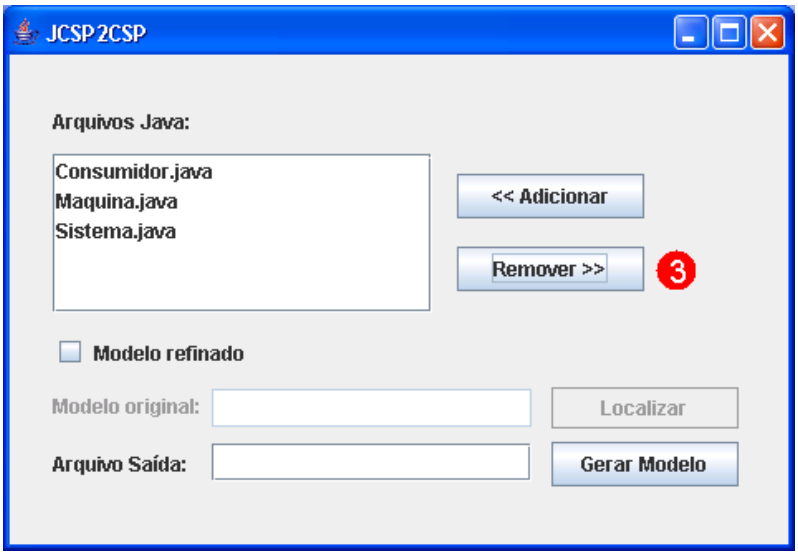
O protótipo desenvolvido gera automaticamente modelos CSP a partir de programas JCSP. Ele apresenta uma interface gráfica simples e amigável. A seguir descrevemos os passos de utilização, ilustrando com figuras:



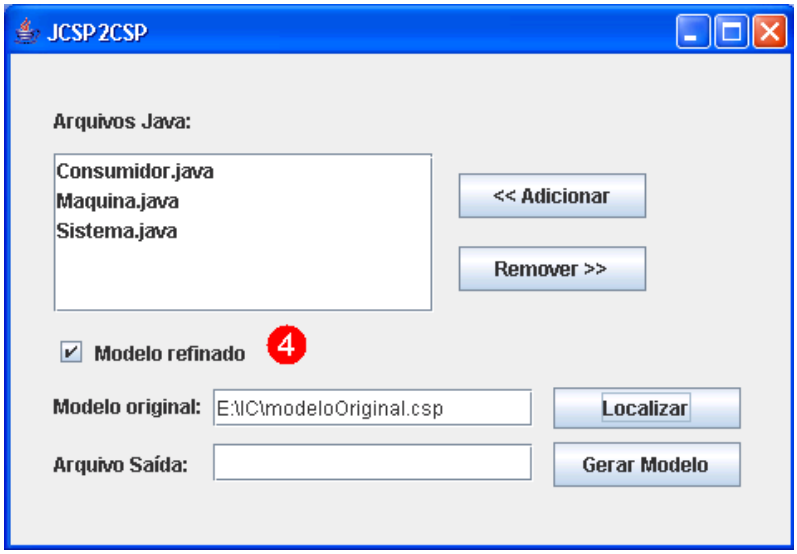
Passo 1 – Primeiramente o usuário deve clicar no botão "Adicionar", para abrir o *browser* dos arquivos.



Passo 2 – O usário deve localizar os arquivos e clicar no botão "Open" para adicionar os arquivos à lista

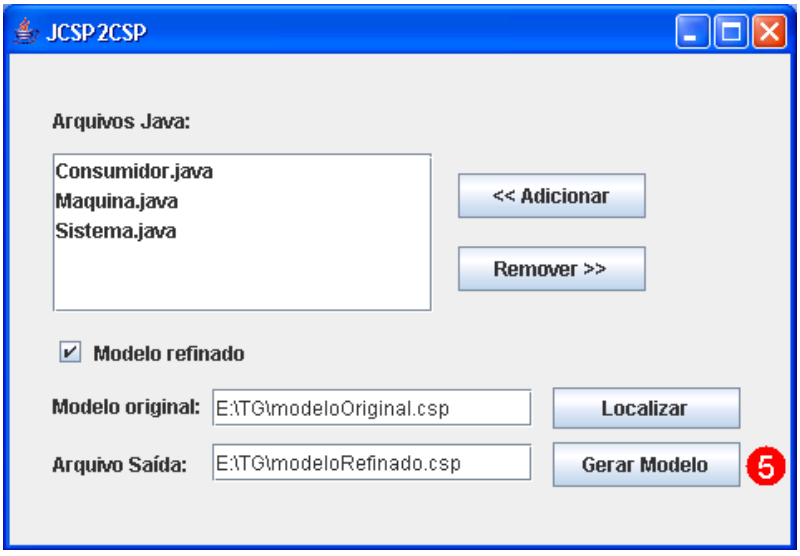


Passo 3 – Caso o usuário tenha inserido algum arquivo indesejado, ele pode retira-lo da lista selecionando o arquivo e clicando no botão "Remover".



Passo 4 – Caso o sistema cujo modelo será gerado seja o sistema original, deve-se pular para o passo 5. Caso contrário o usuário deve marcar o *checkbox* "Modelo refinado" e fornecer a localização do arquivo CSP do modelo original (que deve ter sido gerado previamente). Este passo tem o fim de adicionar no arquivo CSP do sistema refatorado a linha "`include arquivoOriginal.csp`", onde "`arquivoOriginal.csp`" é o nome do arquivo do

modelo do sistema original. Assim todos os processos podem ser carregados no FDR e a ferramenta pode verificar o refinamento.



Passo 5 – O usuário deve especificar o nome do arquivo de saída do modelo. Feito isso, ele deve clicar no botão "Gerar modelo". Ao fim deste passo um novo arquivo CSP será criado com o nome fornecido.

O modelo do sistema refatorado é o que deve ser carregado no FDR, pois é ele que contém um comando `include` que adiciona o script CSP do modelo original.

Referências

[1] Sommerville, Ian "Engenharia de Software" – 6ª Edição, Prentice-Hall, 2003.

[2] Ambler, Scott W. / Pramodkumar Sadalage "Refactoring Databases: Evolutionary Database Design", Addison Wesley, 2006.

[3] Chan, Leon Y/ Sriram, Kosuri / Endy, Drew "Refactoring Bacteriophage T7" Molecular Systems Biology, artigo número 2005.0018, 2005

[4] Gibbs, W. "Software`s Chronic Crisis". Scientific American (edição internacional), 1994

[5] Pressman, Roger S. "Engenharia de Software", Makron Books, 1995

[6] Hoare, C. A. "Communicating Sequential Processes", Communications of the ACM, 1978.

[7] Hoare, C. A. "Communicating Sequential Processes", Prentice-Hall, 1985.

[8] Roscoe, A. W. "The Theory and Practice of Concurrency", Journal of the ACM, 1997.

[9] Woodcock, J. / Davis, J. "Using Z – Specification, Refinement and Proof", Prentice-Hall, 1996.

[10] Moura, A. V. "Especificações em Z: uma introdução", editora Unicamp, 2001.

[11] Mota, Alexandre / Nascimento, Carla "Model Checking JCSP Programs", Relatório Técnico, 2006.

[12] Mota, Alexandre / Nascimento, Carla "Verificação de Modelos para programas JCSP" Dissertação de Mestrado do Cin, UFPE, 2006

[13] Tanembaum, Andrew S. "Sistemas Operacionais Modernos", 2nd edition, editora LTC, 2000.

[14] Beck, Kent "Extreme Programming Explained", Pearson, 2000.

[15] Beck, K. Test-Driven Development by Example, Addison Wesley, 2003

[16] Deitel, H. M. / Deitel, P. J. "Java como programar", 6a edição, Pearson Prentice-Hall, 2005.

[17] Java Communicating Sequential Process (JCSP)
http://http://www.cs.kent.ac.uk/projects/ofa/jcsp/
visitado em 13/05/2006 às 17h 32 min.

[18] Stroustrup, B. "The C++ Programming Language" Addison-Wesley, 3rd Edition, 2000.

[19] A. Farias / Mota, A. / Sampaio, A. C. A. "Efficient Analysis of Infinite CSP-Z Specifications" Workshop de Métodos Formais, pags 113-128, Gramado, 2002.

[20] Extreme Programming: A gentle introduction
http://www.extremeprogramming.org
visitado em 24/07/2006 às 13h 24 min.

[21] Refactoring: Extract Method
http://www.refactoring.com/catalog/extractMethod.html
visitado em 15/06/2006 às 17h 56min.

[22] SableCC, Java Parser Generador
http://sablecc.org/
visitado em 15/09/2006 às 22h 17min

[23] JavaCC: Project Home
https://javacc.dev.java.net/
visitado em 15/09/2006 às 22h 20min

[24] Borba, Paulo / Sampaio, Augusto / Cavalcanti, Ana / Cornélio, Márcio "Algebraic Reasoning for Object-Oriented Programming", Informatics Center, Federal University of Pernambuco, Recife 50740-540, Brazil e Computing Laboratory, University of Kent, Canterbury CT2 7NF, England, Elsevier Science, 2004.

[25] Massoni, Tiago / Gheyi, Rohit / Borba, Paulo "A Formal Framework for Establishing Conformance between Object Models and Object-Oriented Programs" Informatics Center Federal University of Pernambuco, Recife, PE.

[26] Communicating Threads for Java^{tm} (CTJ)
http://www.ce.utwente.nl/javapp/information/CTJ/main.html
visitado em 05/10/2005 às 4h 8 min.

[27] Jackson, D. "Software Abstractions: Logic, Language and Analysis" MIT press, 2006.

[28] Booch, G. et al. "The Unified Modeling Language User Guide" Object Technology. Addison Wesley, 1999.

[29] Cavalcanti, A. / Sampaio, A. / Woodcock., J. "A Unified Language of Classes and Processes" In St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering, Satellite Workshop at FM'03, unknown 2003.

[30] Massoni, Tiago / Gheyi, Rohit / Borba, Paulo "Formal Refactoring for UML Class Diagrams" XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia, MG, 2005.

[31] Massoni, Tiago / Gheyi, Rohit / Borba, Paulo "A Model-driven Approach to Formal Refactoring " OOPSLA'05, ACM 1-59593-193-7/05/0010. San Diego, California, USA, 2005.

[32] Testing Object-Oriented Software - Preface
http://www.computer.org/portal/site/store/menuitem.41cf17dc879177c86ee948ce8bcd45f3/index.jsp?&pName=store_level1&path=store/catalog/BP08520&file=preface.xml&xsl=generic.xsl&
visitado em 03/10/2006 às 22h 25min

[33] Formal Systems (Europe) Ltd: Software
http://www.fsel.com/software.html
visitado em 03/10/2006 às 22h 37min

Assinaturas

Recife, 03 de Julho de 2005.

_____

Alexandre Cabral Mota (Orientador)

_____

José Olino de C. Lima Junior (Proponente)