ALEXANDRE RADEMAKER

Uma Ferramenta Formal para Especificação e Análise de Arquiteturas de Software

NITERÓI

2005

ALEXANDRE RADEMAKER

Uma Ferramenta Formal para Especificação e Análise de Arquiteturas de Software

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Distribuído e Paralelo / Métodos Formais.

Orientador:
Christiano de Oliveira Braga

Co-orientador:
Alexandre Sztajnberg

Universidade Federal Fluminense

NITERÓI

2005

Uma Ferramenta Formal para Especificação e Análise de Arquiteturas de Software

Alexandre Rademaker

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. Christiano de Oliveira Braga / IC-UFF (Presidente)

Prof. Alexandre Sztajnberg / IME-UERJ

Prof. Edward Hermann Haeusler / PUC-RJ

Prof. Orlando Loques / IC-UFF

Prof. Paulo Borba / CI-UFPE

Niterói, 30 de Maio de 2005

À minha família: meus pais, André e Silvia; minhas irmãs, Andréa e Christianne; e minha esposa, Camila. A meu avô Augusto Hammann Rademaker Grünewald (in memoriam), um exemplo e inspiração.

Agradecimentos

Ao meu amigo e orientador Christiano Braga, pelas oportunidades que me proporcionou, por sua dedicação, confiança e ensinamentos. No professor Christiano encontrei inspiração para a carreira acadêmica, exemplo de profissionalismo e disciplina e suporte irrestrito para meu aprendizado. Do amigo Christiano recebi estímulo e as palavras certas, nas horas certas.

Ao meu amigo e co-orientador Alexandre Sztajnberg, por sua paciência, disponibilidade e didática nas explicações sobre CBabel. Trabalhar com o Alexandre é um prazer, sobretudo pelo processo de aprendizado colaborativo que ele sempre consegue estabelecer.

À minha querida Camila, pelo estímulo nos momentos mais difíceis e sugestões tão valiosas. Eu não seria capaz de descrever a importância da Camila em minha vida.

À toda minha família, pelo incentivo, dedicação e amor. À minha sogra Stella Maris e a meu sogro Vinícius, pelas encorajadouras conversas e criativas idéias.

Ao Fabrício Chalub, grande companheiro durante estes dois últimos anos em todas as etapas do mestrado. Fabrício tem uma mente brilhante, e sempre está disposto em compartilhar seus conhecimentos e discutir idéias. Ao Fabrício agradeço especialmente por sua amizade fiel.

A Peter Csaba Ølveczky, por seus comentários em nosso artigo e ensinamentos sobre Maude. A Edward Hermann Haeusler, por suas explicações, opiniões e sugestões. Aos demais membros da banca e ao professor Peter Mosses, pelos excelente e enriquecedores comentários que ajudaram a melhorar este trabalho.

À Escola de Pós-Graduação em Economia da Fundação Getulio Vargas, pela oportunidade que me proporcionou de realizar meu mestrado. Em especial, ao professor Clovis de Faro, por sua confiança e incentivo.

Aos professores e alunos do Instituto de Computação da UFF, pelo agradável ambiente que proporcionaram e companheirismo durante estes dois anos de convívio.

Aos amigos Guilherme Hoffmann, Sérgio Martello e Guilherme Fonseca.

Resumo

Sistemas computacionais complexos podem ser estruturados em *componentes* que executam de forma *concorrente* e, possivelmente, *distribuída*. A modelagem de tais sistemas deve então, invariavelmente, especificar os aspectos de coordenação que compreendem os estilos de interação entre os componentes ("inter-component") e os aspectos de concorrência e sincronização dentro dos componentes ("intra-component"). No "framework" CR–RIO, que integra técnicas de metaprogramação e arquitetura de "software" por configuração, os aspectos dea coordenação podem ser tratados na arquitetura do "software", usando, para isso, a linguagem de descrição de arquiteturas (ADL) CBabel.

CBabel é uma ADL que, além das usuais primitivas arquiteturais como componentes e portas, oferece contratos como construções básicas da linguagem. Com isso, os aspectos de coordenação são, em CBabel, descritos por contratos. Os contratos, por sua vez, são encapsulados em *conectores*, que intermedeiam todas as interações entre os componentes funcionais da arquitetura. Com esta abordagem, os aspectos de coordenação são separados dos aspectos funcionais. A utilização de uma ADL para especificação de um sistema permite que o sistema seja modelado em um nível de abstração apropriado à realização de análises e verificações de propriedades do sistema nas fases iniciais do projeto. Mas para a realização de análises sobre propriedades de uma arquitetura é necessário que, tanto a ADL quando as propriedades a serem verificadas, tenham um modelo semântico formal que dê significado preciso e não-ambíguo a estas. Lógica de reescrita é uma lógica e um formalismo semântico para a qual diversos modelos computacionais, lógicas e linguagens de especificação foram mapeados, dada sua visão unificada de computação e dedução.

Nesta dissertação, apresentamos uma semântica formal de CBabel em lógica de reescrita. Também apresentamos a implementação desta semântica, a ferramenta Maude CBabel tool, um protótipo de ambiente executável para CBabel. Maude CBabel tool permite que descrições arquiteturais em CBabel sejam executadas e analisadas no sistema Maude, uma implementação de lógica de reescrita com suporte à metaprogramação e que dispõe de uma boa variedade de ferramentas de análise embutidas. Desta forma, durante a modelagem de sistemas complexos, podemos formalmente analisar suas descrições arquiteturais, identificando possíveis problemas e sugerindo soluções ainda na fase inicial do ciclo de desenvolvimento do sistema.

Abstract

Complex computational systems can be organized as components, that execute in a *concurrent* and possibly in a *distributed* way. The modeling of such systems has to consider coordination requirements comprising inter-component interaction styles, and intra-component concurrency and synchronization aspects. In the CR–RIO framework, which makes use of meta-level and architecture configuration techniques, the coordination aspects can be treated at the software architecture level using the CBabel architecture description language (ADL).

CBabel is an ADL that, besides the usual architectural primitives such as components and ports, provides contracts as first class constructions. In that way, coordination aspects can be described with CBabel contracts. Coordination aspects are encapsulated in connectors that mediate all interactions among functional modules. With this approach, one separates coordination aspects concerns from functional aspects, which do not need to be included in the design or implementation of functional modules.

The use of a ADL for the specification of a system allows the system to be described in an appropriate level of abstraction allowing the analysis and verifications of architecture level properties in the initial phases of the project. But for the accomplishment of analysis of an architecture it is necessary that both the ADL and the properties to be verified have a formal semantics that gives precise and not-ambiguous meaning for them. Rewriting logic is a logic and semantic framework to which several models of computation, logics and specification languages have been mapped to, due to its unified view of computation and logic.

In this dissertation, we present a formal semantics of CBabel in rewriting logic. We also present the implementation of this semantics, the tool Maude CBabel tool, a prototype executable environment for CBabel. Maude CBabel tool is implemented on top of the Maude system, a fast realization of rewriting logic with support to reflection and with a good variety of analysis tools. With Maude CBabel tool during the modeling of complex systems, we can formally analyze CBabel architectural descriptions, identifying possible problems and suggesting solutions still in the initial phase of its life cycle.

Palavras-chave

1. linguagens de descrição de arquitetura de “software”
2. ADL
3. lógica de reescrita
4. Maude
5. contratos

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Sistemas computacionais complexos podem ser estruturados em *componentes* que executam de forma *concorrente* e, possivelmente, *distribuída* [1, 2, 3]. A modelagem de tais sistemas deve então, invariavelmente, especificar os aspectos de coordenação que compreendem: os estilos de interação entre os componentes ("inter-component") e os aspectos de concorrência e sincronização dentro dos componentes ("intra-component"). Por exemplo, componentes podem interagir usando invocações de métodos síncronas ou assíncronas. Da mesma forma, a invocação de dois ou mais métodos por um componente deve considerar as restrições de concorrência e sincronização. Os aspectos de coordenação são usualmente tratados na fase da programação do sistema, com a utilização de construções lingüísticas [4, 5] ou de serviços oferecidos por bibliotecas [6, 7] providas pelos sistemas operacionais. Esta abordagem *ad hoc*, onde à codificação dos requisitos de coordenação é misturada a codificação dos requisitos funcionais, resulta em módulos menos reutilizáveis e mais sujeitos a erros [8, 9]. Mesmo com a utilização de técnicas, como padrões de *design* [10], é tarefa do projetista utilizar o padrão adequado considerando a separação dos conceitos e modularidade [11].

No "framework" CR–RIO [2, 12], que integra técnicas de metaprogramação e arquitetura de "software" por configuração, os aspectos de coordenação podem ser tratados na arquitetura do "software", usando, para isso, a linguagem de descrição de arquiteturas (ADL), CBabel. Uma ADL permite que a descrição de como componentes distribuídos são *conectados* seja separada da descrição do comportamento interno de tais componentes. A separação de interesses provida por descrições arquiteturais oferece propriedades convenientes como: modularidade da descrição arquitetural, reutilização de componentes em diferentes arquiteturas e capacidade de reconfiguração dinâmica. A separação de interesses também auxilia o projetista a entender o sistema como um todo, facilitando sua configuração e adequação a questões não-funcionais específicas que podem surgir.

CBabel é uma ADL que, além das usuais primitivas arquiteturais [1] como componentes e portas, oferece contratos [13, 14, 15] como construções básicas da linguagem. Com

isso, os aspectos de coordenação são, em CBabel, descritos por contratos. Os contratos, por sua vez, são encapsulados em *conectores*, que intermedeiam todas as interações entre os componentes funcionais da arquitetura. Com esta abordagem, os aspectos de coordenação são separados dos aspectos funcionais, não sendo mais necessário sua inclusão na especificação ou implementação dos componentes funcionais. Na realidade, diferentes instâncias de um mesmo componente podem ser submetidas a especificações distintas de coordenação.

Como apontado em [16], a utilização de uma ADL tem como benefício permitir que o projetista modele o sistema em um nível de abstração apropriado à realização de análises e verificações de propriedades do sistema nas fases iniciais do projeto. Mas para isso, uma ADL deve dispor de um conjunto rico de ferramentas como editores, simuladores e ferramentas de análise. Em especial, para a realização de análises sobre propriedades de uma arquitetura, é necessário que tanto a ADL quando as propriedades a serem verificadas tenham um modelo semântico formal, que dê significado preciso e não-ambíguo a estas [17, 1, 16]. Neste contexto, a presente dissertação contribuiu com:

- uma formalização das construções de CBabel [2], isto é, sua semântica, em lógica de reescrita [18];
- um ambiente executável para CBabel desenvolvido em Maude [19], uma implementação de alto desempenho para lógica de reescrita. Maude CBabel tool é uma implementação direta da semântica de CBabel em lógica de reescrita que permite a execução e análises de descrições CBabel;
- um estudo de caso sobre o uso de lógica de reescrita como formalismo semântico para uma ADL e o uso de Maude como metalinguagem. Também apresentamos alguns comentários em relação às ferramentas de análise de Maude.

Resultados preliminares sobre os dois primeiros itens acima foram publicados em [20, 21].

Lógica de reescrita é uma lógica e um formalismo semântico para a qual diversos modelos computacionais, lógicas e linguagens de especificação foram mapeados [22], dada sua visão unificada de computação e dedução. Maude é uma implementação de lógica de reescrita com suporte à metaprogramação e uma boa variedade de ferramentas de análise embutidas no sistema, como o verificador de modelos em LTL [23], ou desenvolvidas na própria linguagem Maude, como um provador indutivo de teoremas [24]. Maude é

enriquecido com o ambiente Full Maude [25], uma álgebra extensível de módulos que estende a linguagem Maude com uma sintaxe orientada a objetos.

Maude CBabel tool é implementada precisamente como uma extensão conservativa de Full Maude, seguindo a interpretação natural dos conceitos de CBabel em conceitos relacionados à orientação a objetos. A semântica de reescrita dada à linguagem CBabel usa a notação para orientação a objetos de lógica de reescrita e é implementada como uma função de transformação em Maude, usando os recursos de metaprogramação do sistema. Esta função de transformação é a essência da ferramenta Maude CBabel tool.

Esta dissertação foi organizada da seguinte forma. No Capítulo 2, apresentamos a linguagem CBabel, lógica de reescrita, o sistema e a linguagem Maude e as principais ferramentas de análise do sistema Maude. No Capítulo 3, apresentamos a semântica de CBabel em lógica de reescrita e sua implementação em Maude, a ferramenta Maude CBabel tool. No Capítulo 4, apresentamos alguns estudos de caso de especificações e análises de arquiteturas CBabel em Maude CBabel tool. No Capítulo 5, avaliamos a ferramenta Maude CBabel tool, apresentamos alguns trabalhos relacionados a nossa pesquisa encontrados na literatura e comparamos nossa abordagem com estes trabalhos. Para avaliarmos Maude CBabel tool, apresentaremos algumas de nossas experiências com a ferramenta durante a realização dos estudos de caso. No Capítulo 6 são apresentados os comentários finais, os resultados alcançados e alguns possíveis trabalhos futuros. Finalmente, no Apêndice A, apresentamos o código completo da ferramenta Maude CBabel tool e no Apêndice B, os módulos orientados a objetos produzidos por Maude CBabel tool para os componentes das arquiteturas CBabel dos estudos de caso apresentados no Capítulo 4.

2 CBabel, lógica de reescrita e Maude

Neste capítulo apresentamos os conceitos que serão utilizados no decorrer desta dissertação. Este capítulo está estruturado da seguinte forma. Na Seção 2.1, apresentamos a linguagem CBabel. Na Seção 2.2, apresentamos lógica de reescrita, a lógica e formalismo semântico que será utilizada no Capítulo 3 para darmos uma semântica formal à linguagem CBabel. Na Seção 2.3, apresentamos a linguagem e o sistema Maude, uma particular implementação de lógica de reescrita, escolhida para a implementação da ferramenta Maude CBabel tool, uma implementação da semântica de CBabel em lógica de reescrita que será vista também no Capítulo 3. Finalmente, na Seção 2.4, apresentaremos as ferramentas e métodos de análise que o sistema Maude oferece. Tais ferramentas serão usadas nos Capítulos 4 e 5, quando mostraremos como arquiteturas descritas em CBabel, depois de *carregadas* pela ferramenta Maude CBabel tool no sistema Maude, podem ser analisadas.

2.1 A linguagem CBabel

Nesta seção, apresentamos a linguagem de descrição de arquiteturas (ADL) CBabel ("Building Applications by Evolution with Connectors"), componente do "middleware" reflexivo CR–RIO. Antes de apresentarmos CBabel, vamos inicialmente descrever brevemente o "framework" CR–RIO.

Arquiteturas de "software" (SA, da sigla em inglês) podem ser descritas através de ADLs. Com uma ADL, um projetista de sistemas pode especificar a composição funcional de um sistema através da seleção de módulos e associar determinados estilos de interação entre estes módulos, através de portas e conectores. Esta atividade é denominada programação por configuração (CP, da sigla em inglês).

A programação de metanível (M-LP, da sigla em inglês) permite que um "software" seja descrito em níveis diferentes de interesse. A utilização de reflexão torna possível o

tratamento dos aspectos não-funcionais do sistema em um nível *meta*, sem misturá-los assim aos aspectos funcionais do sistema.

O "framework" CR–RIO ("Concurrent Reflective Reconfigurable Interconnectable Objects") [2, 12] combina os conceitos de SA/CP e M-LP para descrição e execução de aplicações, sendo constituído pelos seguintes elementos:

- um modelo de componentes baseado nos conceitos de SA/CP. Os *módulos* são os componentes da arquitetura que encapsulam os aspectos funcionais da aplicação. Os *conectores* são os componentes que definem as relações de interação entre os módulos. As *portas* são os pontos de acesso dos componentes, através das quais são requisitados ou oferecidos serviços;

- um modelo de gerência de configuração, que permite a criação, ligação, terminação e reconfiguração dos componentes;

- a ADL CBabel é, como outras ADLs, uma linguagem declarativa. CBabel permite descrever: os componentes de uma aplicação (módulos e conectores) e suas ligações; contratos que especificam aspectos não-funcionais comuns em diferentes domínios de aplicação como padrões de interação, coordenação, distribuição e qualidade de serviço;

- um "middleware" reflexivo que permite a execução de uma arquitetura de "software", sua reconfiguração dinâmica e serviços de gerência;

- uma metodologia para a configuração de arquiteturas de "software" que estimula a separação dos aspectos funcionais nos módulos e os aspectos não-funcionais nos conectores.

Para apresentarmos a sintaxe de CBabel, vamos utilizar duas arquiteturas para o problema dos produtores e consumidores. O problema dos produtores e consumidores é comumente utilizado na literatura, por exemplo em [26], para descrição de aspectos de coordenação entre processos concorrentes que disputam acesso a um recurso comum. O problema pode assim ser descrito. Um módulo produtor produz interativamente um item e envia uma requisição para o "buffer", solicitando-lhe o armazenamento do mesmo. Um módulo consumidor interativamente envia uma requisição para o "buffer", solicitando a retirada de um item armazenado. O "buffer" é implementado para receber requisições de um produtor para armazenar um item e requisições de um consumidor para retirar

um item. Para acessar o "buffer" os módulos produtor e consumidor devem obedecer a uma disciplina de exclusão mútua, isto é, o "buffer" não pode receber, simultaneamente, requisição do produtor e do consumidor.

Uma primeira arquitetura CBabel para esta aplicação é mostrada na Figura 1. A arquitetura desta aplicação é composta de 3 módulos: `PRODUCER`, `CONSUMER` e `BUFFER`. Por decisão do projetista, a comunicação entre os módulos é síncrona (estilo pedido/resposta). Cada módulo tem um conjunto de portas definido de acordo com sua função. Por exemplo, o módulo `BUFFER` tem duas portas de entrada para receber requisições de colocação (`put`) e retirada (`get`) de itens do produtor e consumidor, respectivamente. A topologia da aplicação é descrita no módulo `PRODUCER-CONSUMER`. Neste módulo, uma instância de cada tipo de módulo é criada e as instâncias conectadas.

```
module PRODUCER {
  out port int (int item) put;
}

module CONSUMER {
  out port int (void) get;
}

module BUFFER {
  in port int (int item) put;
  in port int (void) get;
}
```

```
application PRODUCER-CONSUMER {
  instantiate BUFFER as buff;
  instantiate PRODUCER as prod;
  instantiate CONSUMER as cons;

  link prod.put to buff.put;
  link cons.get to buff.get;
}
```

Figura 1: Produtores e consumidores em CBabel

Deve-se observar que no módulo `PRODUCER-CONSUMER` as instâncias dos módulos produtor e consumidor são ligadas a instância do "buffer" por um conector padrão, isto é, nenhum conector é explicitamente declarado. O conector padrão simplesmente realiza um curto-circuito das portas de saída com as portas de entrada.

A descrição arquitetural da Figura 1 define apenas um conjunto de componentes e sua topologia de interação. A exclusão mútua nos acessos ao "buffer" ainda não é garantida nesta arquitetura. Para atendermos o requisito de exclusão mútua, um contrato de coordenação será adicionado à descrição arquitetural. Todas as interações com o "buffer" serão então mediadas por um único conector que declara um contrato de coordenação de exclusão mútua.

A Figura 2 apresenta a descrição do conector `MUTEX`. Requisições concorrentes recebidas pelas portas de entrada do conector (`put-in` e `get-in`) são enfileiradas antes de serem encaminhadas para as respectivas portas de saída (`put-out` e `get-out`). Desta forma, nenhum acesso concorrente ao "buffer" irá ocorrer. A cláusula `exclusive` define quais portas de saída irão trabalhar sob a disciplina de exclusão mútua.

```
connector MUTEX {
  exclusive {
    out port put_out;
    out port get_out;
  }
  in port put_in;
  in port get_in;
} mutx
```

```
application PRODUCER-CONSUMER-MUTEX {
  instantiate BUFFER as buff;
  instantiate PRODUCER as prod;
  instantiate CONSUMER as cons;

  link prod.put to buff.put by mutx;
  link cons.get to buff.get by mutx;
}
```

Figura 2: Exclusão mútua de produtores e consumidores

Ainda na Figura 2 o módulo `PRODUCER-CONSUMER-MUTEX` define uma nova topologia para a aplicação. Inicialmente são criadas as instâncias dos módulos e, em seguida, estas instâncias são ligadas. As ligações às instâncias `prod` e `cons` à instância `buff` são agora mediadas pela instância `mutx` do conector `MUTEX` (a partir do casamento das portas com assinaturas compatíveis). A instância `mutx` é implicitamente criada pelo "middleware" CR–RIO, não sendo explicitamente criada na descrição da topologia.

2.2 Lógica de reescrita

Lógica de reescrita [18] é uma lógica e um formalismo semântico capaz de representar de maneira bastante intuitiva diversos modelos computacionais, lógicas e linguagens [22]. O texto desta seção é adaptado de [18, 22, 27, 28, 29, 30, 19]. Podemos adotar o ponto de vista lógico ou computacional para interpretação dos conceitos de lógica de reescrita, por existir uma bijeção entre estes. Por exemplo, os *estados* de um sistema correspondem a *fórmulas* (módulo qualquer axioma "estrutural" que estas fórmulas satisfaçam, por exemplo, comutatividade ou associatividade) e as *computações concorrentes* do sistema correspondem às *provas* possíveis na lógica. A partir desta equivalência entre computação e lógica, um axioma em lógica de reescrita na forma $t \rightarrow t'$ tem duas leituras. Do ponto de vista computacional, especifica que um fragmento do estado do sistema que seja instância do padrão descrito por t pode *mudar* para uma instância de t', concorrentemente com qualquer outra mudança no estado do sistema. Ou seja, computacionalmente significa uma *transição concorrente localizada*. Do ponto de vista lógico, especifica uma regra de inferência, ou seja, que podemos derivar a fórmula t' a partir de t.

Em lógica de reescrita, a estrutura de t é completamente flexível, sendo totalmente descrita pelo usuário por um tipo de dados algebricamente definido que satisfaça determinados axiomas (equacionalmente especificados). Desta forma, as reescritas ou deduções são executadas *módulo* tais axiomas.

Mais formalmente, uma teoria de reescrita $\mathcal{R}$ é uma tupla (Σ, E, L, R) onde Σ é um

alfabeto indexado de símbolos funcionais, E é um conjunto de Σ-equações, L é um conjunto de rótulos e R é um conjunto de pares $R \subseteq L \times T_{\Sigma,E}(X)$ onde o primeiro componente é um rótulo e o segundo é um par de termos na mesma classe de E-equivalência, sendo $X = \{x_1, \ldots, x_n, \ldots\}$ um conjunto contável infinito de variáveis. Os elementos de R são chamados *regras de reescrita*. [1] A regra $(l, [t], [t'])$ é dita uma conseqüência rotulada e é expressa pela notação $l : [t] \rightarrow [t']$. Para indicar que $\{x_1, \ldots, x_n\}$ é um conjunto de variáveis em t e t', escrevemos $l : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)]$ ou $l : (\forall X)[t] \rightarrow [t']$ ou podemos ainda utilizar a notação abreviada $l : [t(\bar{x})] \rightarrow [t'(\bar{x})]$.

Dizemos que (Σ, E) é a teoria equacional [2] que define a *assinatura* da teoria de reescrita $\mathcal{R}$ e que as regras de reescrita são *sentenças* em lógica de reescrita.

Dada uma teoria de reescrita $\mathcal{R}$, dizemos que uma sentença $t \rightarrow t'$ é dedutível em $\mathcal{R}$ ou que $t \rightarrow t'$ é uma $\mathcal{R}$-reescrita (concorrente) e escrevemos $\mathcal{R} \vdash t \rightarrow t'$ se e somente se $t \rightarrow t'$ pode ser obtida pela aplicação de um número finito de vezes das regras de inferência abaixo. Nas regras abaixo, a versão de lógica de reescrita com operadores "frozen" não está sendo considerada. [3]

- **Reflexão**. Para cada termo t na álgebra inicial de Σ com variáveis $T_\Sigma(X)$,

$$\frac{}{(\forall X)t \longrightarrow t}$$

- **Igualdade**.

$$\frac{(\forall X)u \longrightarrow v \quad E \vdash (\forall X)u = u' \quad E \vdash (\forall X)v = v'}{(\forall X)u' \longrightarrow v'}$$

- **Congruência**. Para cada $f : k_1 \ldots\ k_n \rightarrow k$ em Σ, com $t_i, t_i' \in T_\Sigma(X)_{k_i}, 1 \leq i \leq n$,

[1] Para simplificar a apresentação da lógica, apresentamos o caso onde as regras são não-condicionais. No entanto, em [18], os conceitos aqui apresentados são generalizados em regras de reescrita condicionais na forma

$$l : [t] \rightarrow [t'] \ \textit{if}\ [u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k]$$

que aumenta consideravelmente a expressividade de lógica de reescrita. Pode-se ainda considerar regras que incluam equações em suas condições.

[2] A teoria equacional que parametriza lógica de reescrita pode ser não-sortida, multi-sortida, ordenada-sortida [31] ou de pertinência [32]. No decorrer deste texto, utilizamos a apresentação ordenada-sortida, por ser esta a lógica equacional explorada no Capítulo 3 que apresenta a semântica de CBabel em lógica de reescrita e sua implementação em Maude.

[3] Em [33] a noção de operadores "frozen" foi adicionada às teorias de reescrita, generalizando ainda mais a lógica.

$$\frac{(\forall X)t_1 \longrightarrow t'_1 \ldots \ (\forall X)t_m \longrightarrow t'_m}{(\forall X)f(t_1, \ldots \ t_m) \longrightarrow (\forall X)f(t'_1, \ldots \ t'_m)}$$

- **Substituição**. Para cada finita substituição $\theta : X \to T_\Sigma(Y)$, e para cada regra na forma $l : (\forall X)t \longrightarrow t'$ sendo $t = f(t_1, \ldots, t_n)$ e $t' = f(t'_1, \ldots, t'_m)$

$$\frac{(\forall Y)\theta(t_1) \longrightarrow \theta(t'_1) \ldots \ (\forall Y)\theta(t_m) \longrightarrow \theta(t'_m)}{(\forall Y)\theta(t) \longrightarrow \theta(t')}$$

- **Transitividade**.

$$\frac{(\forall X)t_1 \longrightarrow t_2 \ldots \ (\forall X)t_2 \longrightarrow t_3}{(\forall X)t_1 \longrightarrow t_3}$$

Lógica de reescrita é uma lógica computacional para especificação de sistemas concorrentes que tem *estados* e evolui através da *transição* entre estes estados. A assinatura da teoria de reescrita descreve uma particular estrutura para os estados do sistema (i.e., multiset, árvore binária, "string") e, desta forma, seus estados podem ser *distribuídos*, de acordo com esta estrutura. As regras de reescrita da teoria descrevem as *transições elementares localizadas* que podem ocorrer em um estado distribuído do sistema, pela aplicação concorrente de transformações locais. Com as regras de inferência supra, podemos descrever todas as possíveis computações finitas concorrentes de um sistema, especificado por uma teoria de reescrita $\mathcal{R}$ como:

- *Reflexão* é a possibilidade de termos uma transição *neutra*;
- *Igualdade* significa que dois estados são iguais *módulo* E;
- *Congruência* é uma forma geral de *paralelismo real*;
- *Substituição* combina uma *transição atômica* mais externa, usando uma regra, com *concorrência aninhada* na substituição.
- *Transitividade* é a composição seqüencial.

O ponto de vista lógico também pode ser adotado considerando as regras de reescrita como meta-regras para dedução de sistemas lógicos. Neste caso, cada passo de reescrita é uma dedução no sistema. Os pontos de vista computacionais e lógicos em que lógica de reescrita pode ser interpretada podem ser sumarizados pela Tabela 1.

Estado	$\leftrightarrow$ Termo	$\leftrightarrow$ Proposição
Transição	$\leftrightarrow$ Reescrita	$\leftrightarrow$ Dedução
Estrutura distribuída	$\leftrightarrow$ Estrutura Algébrica	$\leftrightarrow$ Estrutura proposicional

Tabela 1: Interpretações para lógica de reescrita

A última linha expressa uma condição fundamental de que um estado só pode ser transformado de forma concorrente se não for atômico. Isto é, apenas se o estado do sistema for composto por componentes menores, estes podem ser independentemente alterados. Em lógica de reescrita esta composição de estados concorrentes é formalizada por operadores da assinatura Σ da teoria de reescrita $\mathcal{R}$ que especifica o sistema. Do ponto de vista lógico, tais operadores podem ser vistos como conectivos proposicionais definidos pelo usuário para descreverem a particular estrutura do sistema. Sobre a última linha, deve-se ainda dizer que a estrutura algébrica do sistema envolve equações que descrevem o estado global do sistema como uma estrutura de dados concorrente.

Lógica de reescrita é reflexiva [34], isto é, existe uma teoria $\mathcal{U}$ apresentada de forma finita que é *universal*, na qual podemos representar qualquer teoria de reescrita $\mathcal{R}$ apresentada de forma finita (incluindo $\mathcal{U}$) como um termo $\overline{\mathcal{R}}$ (a meta-representação de $\mathcal{R}$), qualquer termo t, t' em $\mathcal{R}$ como termos $\overline{t}$ e $\overline{t'}$ e qualquer par $(\mathcal{R}, t)$ como um termo $\langle\overline{\mathcal{R}}, \overline{t}\rangle$. A partir desta definição, temos a seguinte equivalência

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \to \langle\overline{\mathcal{R}}, \overline{t'}\rangle. \tag{2.1}$$

Mais ainda, como $\mathcal{U}$ pode ser representada nela mesma, podemos ter a seguinte "torre de reflexão", com um número arbitrário de níveis de reflexão:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \to \langle\overline{\mathcal{R}}, \overline{t'}\rangle \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t}\rangle}\rangle \to \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t'}\rangle} \ldots$$

Nesta cadeia de equivalências, dizemos que a reescrita mais à esquerda ocorre no nível 0, a seguinte no nível 1 e assim por diante. Na implementação destes conceitos, cada nível nesta torre de reflexão é conseguido com um custo computacional maior, isto porque, simular um passo de reescrita em um nível de reflexão pode envolver a execução de várias reescritas no nível de reflexão acima. Sendo assim, é importante que existam formas sistemáticas para tornar o mais baixo possível o nível de reflexão de uma reescrita, deixando para um nível mais alto de reflexão apenas as reescritas que realmente precisam ocorrer em um nível superior de reflexão.

2.3 O sistema Maude

O sistema Maude [19] é uma implementação de alto desempenho da linguagem Maude, linguagem esta com suporte à metaprogramação e especificações em lógica de reescrita e lógica equacional de pertinência [32]. Esta seção apresenta uma introdução ao sistema e a linguagem Maude, sendo baseada em [19, 27, 28, 29].

O sistema Maude, em sua versão 2.1.1, é organizado em dois componentes principais: Core Maude e Full Maude. A linguagem Core Maude fornece as construções básicas da linguagem Maude para especificação de teorias de reescrita e construções para aplicações em meta-nível. Em Core Maude é feita a implementação da máquina de reescrita. Em resumo, na linguagem Core Maude temos:

- Módulos funcionais e módulos de sistema que implementam, respectivamente, teorias equacionais e teorias de reescrita. Ambos são suportados ao nível objeto (nível-base) ou meta-nível.
- Funções para construção de aplicações em meta-nível que permitem a movimentação entre níveis de reflexão.
- Um mecanismo extensível de "loop" do tipo leitura-avaliação-escrita que fornece ao sistema Maude uma interface de linha de comando para aplicações em meta-nível.

Full Maude é uma meta-aplicação escrita na linguagem Core Maude que estende Core Maude com uma álgebra de módulos extensível. A álgebra de módulos de Full Maude apresenta as noções de teorias, módulos e teorias orientados a objetos, módulos e teorias parametrizados e visões. A extensibilidade de Full Maude é um importante aspecto que foi explorado no desenvolvimento da ferramenta Maude CBabel tool, apresentada no Capítulo 3.

Esta seção está organizada da seguinte forma. Nas seções 2.3.1 e 2.3.2, apresentamos os módulos funcionais e módulos de sistema, respectivamente. Na Seção 2.3.3, apresentamos as noções de hierarquia entre módulos em Core Maude. Na Seção 2.3.4, mostramos como reflexão é implementada em Maude. Na Seção 2.3.5, mostramos como é criado um ambiente executável para uma linguagem em Maude. Finalmente, na Seção 2.3.6, apresentamos os módulos orientados a objetos de Full Maude.

2.3.1 Módulos funcionais

Um módulo funcional em Core Maude é uma teoria equacional. Módulos funcionais definem tipos de dados e funções sobre estes tipos. Assumisse que os módulos funcionais de Maude têm a agradável propriedade de suas equações, consideradas como regras de simplificação usadas somente da esquerda para a direita, serem *Church-Rosser* e *com terminação* [18]. Como conseqüência, a aplicação repetida das equações como regras de simplificação eventualmente alcança um termo no qual nenhuma equação poderá mais ser aplicada, chamado de *forma canônica*. Além disso, este termo é o mesmo, independente da ordem de aplicação das equações. Desta forma, cada classe de equivalência tem um representante natural, sua forma canônica, que pode ser computada por simplificações equacionais. Ou seja, o modelo matemático dos tipos de dados e funções é fornecido pela álgebra inicial da teoria, onde os elementos são os termos canônicos representando classes de equivalência dos termos sem variáveis ("ground") módulo as equações. Isto é, dois termos sem variáveis representam o mesmo elemento se, e somente se, pertencem a mesma classe de equivalência determinada pelas equações.

A lógica equacional suportada nos módulos funcionais de Maude é uma extensão da lógica equacional ordenada sortida [31] chamada lógica equacional de pertinência [32]. Desta forma, módulos funcionais suportam a especificação de tipos de dados, relação de inclusão entre tipos através da definição de subtipos, sobrecarga de operadores e assertivas de pertinência em um tipo. Estas últimas definem que um termo é de um tipo se uma condição (que pode consistir de um conjunto de equações e testes de pertinência não condicionados) for verdadeira.

Como exemplo de um módulo funcional, apresentamos uma variação simplificada da definição de hierarquia de conjuntos apresentada em [27].

```
fmod SET-HIERARCHY is
 protecting NAT .

 sorts Set Elt Magma .
 subsorts Set < Elt < Magma .
 subsorts Nat < Elt .

 *** empty set
 op mt : -> Set [ctor] .
 op _,_ : Magma Magma -> Magma [assoc comm ctor] .

 *** set constructor
 op {_} : Magma -> Set [ctor] .

 *** union and intersection
```

```
 ops _U_ _I_ : Set Set -> Set [assoc comm] .

 *** membership
 op _in_ : Elt Set -> Bool .

 vars L M : Magma .
 vars E F : Elt .
 vars S T : Set .

 *** equation to eliminate duplicate elements
 eq { L , L , M } = { L , M } .
 eq { L , L } = { L } .

 *** membership
 eq E in mt = false .
 ceq E in { F } = true if (E == F) .
 ceq E in { F } = false [owise] .
 eq E in { F , L } = if E == F then true else E in { L } fi .

 *** set union
 eq S U mt = S .
 eq { L } U { M } = { L , M } .

 *** set intersection
 eq mt I S = mt .
 eq { E } I S = if E in S then { E } else mt fi .
 eq { E , L } I S = ( { E } I S ) U ({ L } I S ) .
endfm
```

A palavra reservada `fmod`, seguida de um nome, declara um módulo funcional em Maude. A primeira declaração no corpo do módulo é de uma importação do módulo `NAT` (Seção 2.3.3), um módulo predefinido no sistema Maude definido no arquivo `prelude.maude` que o sistema lê quando é iniciado. As definições de tipos e relações de inclusão entre eles são definidas pelas palavras reservadas `sort` e `subsort`. Em seguida, algumas operações são definidas usando-se a palavra reservada `op` seguida de um símbolo que define o nome da operação. Após `:` temos os tipos que correspondem aos seus argumentos e após `->` o tipo de retorno da operação. As variáveis são declaradas usando-se a palavra reservada `var` seguida de um símbolo que será o nome da variável e, após `:`, seu tipo. Por fim, as equações são definidas através das palavras reservadas `eq` e `ceq`.

No módulo `SET-HIERARCHY` um conjunto finito é representado usando-se a notação { `E1` , ... , `En` } como uma coleção (associativa comutativa) de elementos. Tais elementos são construídos pela aplicação da operação construtora {_} sobre elementos da forma `E1`, ... , `En` do tipo `Magma`. Como `Set` é um subtipo de `Magma`, conjuntos podem conter outros conjuntos como elementos, logo, temos uma hierarquia de conjuntos. As operações construtoras são identificadas pelo atributo `ctor`.

Em Maude, toda linha iniciada com `***` ou `---` é considerada um comentário.

A declaração das operações de união e interseção como associativas (`assoc`) e comutativas (`comm`) faz com que a máquina de reescrita de Maude, nas operações de reescrita, não considere os aninhamentos de parênteses ou a ordem dos parâmetros no casamento de padrões. Em geral, Maude pode realizar reescritas módulo diferentes combinações dos axiomas de associativade, comutatividade e identidade, declarados nos diferentes operadores de uma especificação. Isto resulta no fato das reescritas ocorrerem sobre as classes de equivalência (módulo os axiomas) dos termos. Os atributos são ditos equacionais, pois, semanticamente, a declaração de um conjunto de atributos em uma operação é equivalente à especificação de equações correspondentes para os operadores. Operacionalmente, no entanto, a declaração de tais atributos evita problemas de terminação que a definição equacional (por exemplo, de identidade) poderia provocar [19], e permite melhor desempenho nas reescritas de termos que contenham estes operadores.

Maude utiliza as equações como regras de simplificação da esquerda para a direita, casando os padrões de termos módulo os axiomas definidos como atributos das operações. Para terminar esta seção, apresentamos alguns exemplos de reduções de expressões sobre conjuntos onde `U` representa a união e `I` a operação de interseção de conjuntos.

```
Maude> red {1,2,3} U {4,5} .
reduce in SET-HIERARCHY : {1,2,3} U {4,5} .
rewrites: 1 in 1ms cpu (1ms real) (1000 rewrites/second)
result Set: {1,2,3,4,5}

Maude> red {1,2,3} I {4,5} .
reduce in SET-HIERARCHY : {1,2,3} I {4,5} .
rewrites: 22 in 0ms cpu (0ms real) (~ rewrites/second)
result Set: mt

Maude> red {1,2,3,{4,5}} U {4,5} .
reduce in SET-HIERARCHY : {1,2,3,{4,5}} U {4,5} .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Set: {1,2,3,4,5,{4,5}}
```

2.3.2 Módulos de sistema

Um sistema computacional dinâmico é quase sempre reativo, interagindo com o ambiente e reagindo a entradas do ambiente com a mudança de seu estado ou com a produção de alguma saída. Sistemas reativos são quase sempre não determinísticos e sem terminação, e suas propriedades de interesse são seu estado corrente e as suas possíveis respostas a estímulos do ambiente.

O problema de utilizar equações para modelar sistemas dinâmicos é que a igualdade que uma equação especifica é simétrica, ou seja, uma mudança é reversível. Em lógica de reescrita o comportamento dinâmico é, ao invés disso, modelado por regras de reescrita. Reescritas em um módulo funcional terminam sempre com um valor como resultado. Nestes módulos, cada passo de reescrita representa uma substituição simétrica de igual por igual até que uma forma totalmente reduzida seja obtida e nenhuma reescrita adicional seja possível. Em módulos de sistema o conjunto de regras de reescrita não precisa ter terminação ou ser Church-Rosser. Ou seja, não apenas é possível termos uma cadeia infinita de aplicações de regras sobre um termo como também podemos ter diferentes *caminhos* ou *seqüências* de reescrita, possivelmente divergentes, para um dado termo.

Em lógica de reescrita um termo t não é interpretado como uma expressão funcional, mas como um estado de um sistema; uma regra de reescrita $t \rightarrow t'$ não é mais uma igualdade, mas uma transição de estado localizada, isto é, significa que uma parte do sistema cujo padrão é descrito por t, mudou para uma instância de t'. Transições como esta podem ocorrer de forma independente e, assim sendo concorrentemente, com qualquer outra, desde que não sobre o mesmo *pedaço* do sistema. As expressões "termo" e "estado" poderão ser usadas de forma intercambiável, dado que um termo representa o estado de um sistema.

Reescritas em um módulo de sistema ocorrem módulo qualquer axioma estrutural, equacionalmente definido, que o estado do sistema satisfaça. Assim sendo, podemos representar o estado de um sistema distribuído como um *multiset*, empregando um construtor de estado associativo e comutativo.

Um módulo de sistema de Maude especifica o modelo inicial $\mathcal{T}_{\mathcal{R}}$ de uma teoria de reescrita $\mathcal{R}$. O modelo inicial é um sistema de transição no qual seus estados são as classes de equivalência $[t]$ dos termos sem variáveis, módulo as equações E em $\mathcal{R}$ e onde as transições são provas $\alpha : [t] \rightarrow [t']$ em lógica de reescrita. Em Maude, para que um módulo de sistema seja admissível e tenha assim condições de ser executado, algumas condições são necessárias. Por exemplo, as regras em R devem satisfazer a condição de serem coerentes [19] em relação às equações E (módulo os axiomas). Isto quer dizer que podemos alternar entre reescritas com regras e reescritas com equações sem perder computação de reescritas pela falha na aplicação de uma regra, que poderia ser executada, antes que um passo de dedução equacional fosse dado. Maude utiliza como método geral para reescritas de termos, a cada passo de reescrita, primeiro reduzir o termo a sua forma canônica com as equações E (módulo os axiomas) e depois executar um passo de reescrita

com as regras em R. Mais detalhes sobre a construção de $\mathcal{T}_{\mathcal{R}}$ podem ser obtidos em [35].

Para exemplificar, apresentamos a seguir o módulo `RIVER-CROSSING`, que implementa o clássico problema da travessia do rio, utilizado em [36] para exemplificar as aplicações de Maude. O exemplo pode assim ser descrito. Um pastor precisa transportar para o outro lado do rio um lobo, uma cabra e uma couve. Ele tem apenas um barco com espaço para ele e mais um item. O problema é que na ausência do pastor, o lobo come a cabra e a cabra come a couve.

```
mod RIVER-CROSSING is
  sorts Side Group Obj .
  subsort Obj < Group .

  ops left right : -> Side .
  op change : Side -> Side .

  ops s w g c : Side -> Obj [ctor] .
  op __ : Group Group -> Group [assoc comm] .
  op initial : -> Group .

  vars S S' : Side .

  eq change(left) = right .
  eq change(right) = left .

  ceq w(S) g(S) s(S') = w(S) s(S') if S =/= S' .
  ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S =/= S' .

  eq initial = s(left) w(left) g(left) c(left) .

  rl [shepherd-alone] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

Módulos de sistema são declarados no sistema Maude pela sintaxe

mod `<Name>` **is** `<Declarações>` **endm**

onde as declarações são todas que também podem ser feitas nos módulos funcionais: importações de outros módulos, declarações de tipos (`sort`) e relação de tipos (`subsort`), declarações de operadores (`op`), declarações de variáveis (`var`), declarações de equações (`eq` ou `ceq`) e assertivas de pertinência (`md` ou `cmb`); e ainda declarações de regras de reescrita (`rl` ou `crl`). As regras de reescrita podem ainda ser rotuladas. No exemplo supra, as regras de reescrita foram rotuladas com o nome do item sendo transportado.

Representamos com as constantes `right` e `left` os dois lados do rio. O pastor e os demais itens foram representados como objetos, [4] com um atributo indicando o lado do rio onde este está localizado. O estado global do sistema é representado pelo conjunto de objetos, um elemento do tipo `Obj`, construído pela operação `_ _`, associativa-comutativa. A constante `initial` denota a situação inicial onde assumimos que todos os itens estão localizados na margem esquerda do rio. As regras representam as maneiras permitidas de atravessar o rio, considerando a capacidade do barco. O operador auxiliar `change` é usado para modificar o atributo dos objetos a cada travessia do rio.

Uma interessante decisão na especificação acima refere-se à opção de utilizar equações para representar o fato do lobo comer a cabra quando estão sozinhos em uma margem do rio, e da cabra comer a couve. Deve-se observar que a descrição do problema está incompleta. Não está claro o que exatamente ocorre quando o lobo, a cabra e a couve são deixados sozinhos em uma margem do rio. Na especificação supra, foi definido que a cabra não é rápida o bastante, sendo comida pelo lobo antes de conseguir dar uma mordida na couve. Se utilizássemos regras ao invés de equações para especificar as transições onde algum item é comido, teríamos uma especificação incorreta. Isto porque permitiríamos caminhos de execução onde, se o pastor deixasse, por exemplo, a cabra e a couve sozinhas, poderia retornar e encontrar a couve intacta. Como as regras de reescrita são aplicadas módulo as equações, nossa especificação está correta. Deve-se notar também que a aplicação das regras módulo as equações permite que a operação `change` seja executada de forma atômica, não representando no sistema uma transição entre estados.

Destacamos ainda que em Maude, por padrão, todos os módulos importam o módulo predefinido `BOOL`. Por isso, o símbolo funcional `_=/=_` (in-fixado) está disponível para ser usado na condição das equações.

Após carregar no sistema Maude o módulo `RIVER-CROSSING`, podemos exibir uma particular seqüência de reescrita a partir do estado inicial com o comando `rewrite`. Existem certas estratégias na aplicação das regras de reescrita. Tais estratégias definem uma "ordem de operação" para aplicação das regras de reescrita. O comando `rewrite` implementa a estratégia padrão, "fair top-down" [19], para reescrita de um termo de um módulo de sistema. Na invocação do comando `rewrite`, podemos passar um número natural entre colchetes, representando o número máximo de regras de reescritas que podem ser aplicadas ao termo sendo reescrito. Se não passarmos nenhum número entre colchetes, o comando `rewrite` será aplicado até que um termo final, que não possa mais ser reescrito,

[4] O operador `s` é inicial de "shepherd", pastor; `w` é inicial de "wolf", lobo; `g` é inicial de "goat", cabra e `c` é inicial de "cabbage", couve.

seja obtido. Como o módulo `RIVER-CROSSING` especifica um sistema sem terminação (vide a regra `shepherd-alone`), abaixo exemplificamos a execução do comando `rewrite` limitando sua execução a aplicação de até cinco reescritas sobre o termo `initial`:

```
Maude> rew [5] initial .
rewrite [5] in RIVER-CROSSING : initial .
rewrites: 17 in 1ms cpu (1ms real) (17000 rewrites/second)
result Group: s(right) w(right) g(left)
```

Após executarmos o comando `rewrite`, podemos utilizar o comando `continue X` para continuar a estratégia de reescrita corrente em mais X reescritas. Sendo X um limite especificado pelo usuário.

```
Maude> continue 1 .
rewrites: 3 in 0ms cpu (1ms real) (3003 rewrites/second)
result Group: s(left) w(left) g(left)
```

Se repetirmos três ou quatro vezes o comando `continue` acima, veremos que utilizando a estratégia padrão do comando `rewrite`, apenas as regras `wolf` e `goat` permanecem sendo aplicadas, de forma alternada, no termo que representa o estado do sistema. Loops como este podem ocorrer em módulos de sistema, por isso Maude oferece outro comando de reescrita, o comando `frewrite` ("fair rewrite"). Este comando determina qual regra de reescrita aplicar, garantindo que nenhuma regra seja ignorada.

```
Maude> frew [5] initial .
frewrite [5] in RIVER-CROSSING : initial .
rewrites: 15 in 1ms cpu (1ms real) (15000 rewrites/second)
result Group: s(right) w(right) c(right)
```

Podemos agora acompanhar a estratégia para aplicação das regras de reescrita do comando `frewrite`, executando algumas vezes o comando `continue 1`. Neste caso, veremos que todas as regras do módulo que podem ser aplicadas, passam a ser aplicadas.

Utilizando os comandos `rewrite` e `frewrite` não temos controle sobre como as regras de reescrita de um módulo são aplicadas. No entanto, em geral, este controle pode ser conveniente. Em especificações não confluentes e sem terminação, como o módulo `RIVER-CROSSING`, isto pode ser na realidade não só conveniente como também necessário. O controle na aplicação das regras pode ser obtido pela especificação de estratégias de reescrita [19].

Com os comandos `rewrite` e `frewrite`, apenas uma possível seqüência de reescritas é explorada. No módulo `RIVER-CROSSING` isto não é suficiente, visto que especifica um sistema

não confluente, com diferentes possíveis caminhos de reescritas divergentes. Utilizando o comando `search` de Maude, que será detalhado na Seção 2.4.2, podemos explorar todas as possíveis seqüências de reescrita a partir do estado inicial, confirmando que dentre estas, apenas uma corresponde a uma forma válida do pastor transportar todos os itens para a outra margem do rio. O comando `search`, como o próprio nome diz, busca por caminhos de reescrita entre um estado inicial e um estado final, informados pelo usuário.

```
Maude> search initial =>* w(right) c(right) g(right) s(right) .

Solution 1 (state 27)
states: 28  rewrites: 155 in 2ms cpu (16ms real) (51683 rewrites/second)
empty substitution

No more solutions.
```

Dentre vários outros sistemas concorrentes que podemos especificar com módulos de sistema em Maude, destacam-se os sistemas concorrentes orientados a objetos. Maude oferece uma sintaxe especial para especificação de sistemas concorrentes orientados a objetos, os módulos orientados a objetos, vide Seção 2.3.6.

2.3.3 Hierarquia de módulos

Especificações devem ser construídas em módulos de tamanho relativamente pequeno que facilitem o entendimento de grandes sistemas, aumentem a reutilização dos componentes e mantenham localizados os efeitos de uma mudança. O completo suporte para especificações de forma modular é oferecido apenas por Full Maude, através de sua álgebra de módulos extensível que permite, por exemplo, a especificação de módulos parametrizados. Em Core Maude, no entanto, temos as bases para especificações modulares através da possibilidade de definição de uma hierarquia entre módulos. Isto é, cada módulo funcional ou de sistema pode importar outros módulos como submódulos e como esta relação de submódulo é transitiva, temos uma hierarquia. Matematicamente, esta hierarquia pode ser vista como uma ordem parcial de inclusões de teorias, ou seja, a teoria do módulo que importa outros módulos contém as teorias dos módulos que ele importa como suas subteorias. Core Maude também suporta a soma e renomeação de operações de Módulos. O texto desta seção foi inspirado nos textos em [18, 19, 35, 27].

Lembrando que uma teoria de reescrita é uma tupla $\mathcal{R} = (\Sigma, E, L, R)$ onde (Σ, E) é uma teoria em lógica equacional de pertinência, e que um módulo de sistema é uma teoria com modelo inicial (Seção 2.3.2). Podemos usar a inclusão de lógica equacional de

pertinência em lógica de reescrita para ver um módulo funcional que especifica uma teoria (Σ, E) como uma particular teoria de reescrita $(\Sigma, E, \emptyset, \emptyset)$. De fato, a álgebra inicial de (Σ, E) e o modelo inicial de $(\Sigma, E, \emptyset, \emptyset)$ coincidem [18].

Seja então $\mathcal{R} = (\Sigma, E, L, R)$ uma teoria de reescrita especificada por um módulo de sistema e seja $\mathcal{R}' = (\Sigma', E', L', R')$ a teoria que inclui $\mathcal{R}$, então existe uma inclusão de teorias $\mathcal{R} \subseteq \mathcal{R}'$. Cada modelo $\mathcal{S}'$ de $\mathcal{R}'$ pode ser visto com um modelo $\mathcal{S} \mid_{\mathcal{R}}$ de $\mathcal{R}$ simplesmente a partir do descarte dos tipos de dados, operações, equações, assertivas de pertinência e regras da teoria $\mathcal{R}' - \mathcal{R}$. Dado que as teorias de reescrita $\mathcal{R}'$ e $\mathcal{R}$ têm, ambas, os modelos iniciais $\mathcal{T}_{\mathcal{R}'}$ e $\mathcal{T}_{\mathcal{R}}$, pela inicialidade de $\mathcal{T}_{\mathcal{R}}$, sempre existe um único homomorfismo [5]

$$h : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}'} \mid_{\mathcal{R}}.$$

Em Maude, um modulo pode ser importado como submódulo de outro nos modos: `protecting`, `extending` ou `including`. Com as sintaxes:

`protecting` $< ExpressaoDeModulo >$.
`extending` $< ExpressaoDeModulo >$.
`including` $< ExpressaoDeModulo >$.

Informalmente, a importação do módulo M' pelo módulo M no modo `protecting` determina que não é adicionado a M' termos desnecessários (do inglês "junk") ou confusão (termos que passem a significar coisas diferentes). Tais assertivas semânticas não podem, no entanto, ser verificadas por Maude em tempo de execução, pois necessitam de um verificador indutivo de teoremas [19]. No modo `extending` pode haver a inclusão de termos desnecessários em M', embora ainda seja proibida a adição de confusão em M'. Em contrapartida, no modo `including`, nenhuma destas condições é requerida. No entanto, a inclusão no modo `including` não deve destruir as importações no modo `protecting` ou `extending` feitas nos níveis inferiores da hierarquia dos módulos, formada pela cadeia de importações. Isto é, se M importa M' no modo `including`, mas M' importa M'' no modo `protecting` (ou `extending`), então M ainda importa M' no modo `protecting` (ou `extending`).

Os comandos de importação mostrados acima recebem como argumento expressões de módulos, que podem ser: o nome de um módulo, a soma de duas expressões de módulos, ou ainda, operações de renomeação sobre uma expressão de módulos. Como não utilizaremos estes recursos no decorrer deste trabalho, indicamos o manual de Maude [19] para maiores referências sobre estes casos.

[5] Em [18] a completa formalização destes conceitos é apresentada.

2.3.4 O módulo META-LEVEL

No final da Seção 2.2, apresentamos a "torre de reflexão" construída pelas equivalências formalizadas na equação 2.1. Em Maude, a teoria universal $\mathcal{U}$ é modelada pelo módulo funcional predefinido META-LEVEL, que oferece as funcionalidades básicas para construirmos a torre de reflexão apresentada na Seção 2.2. O módulo META-LEVEL importa os módulos META-TERM e META-MODULE. No módulo META-TERM, termos são meta-representados como elementos do tipo de dados Term, pela assinatura:

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .
op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .
```

O caso base na meta-representação dos termos é dado pelos subtipos Variable e Constant do tipo Qid. Constantes são QIDs ("quoted identifiers") que contêm o nome da constante e seu tipo separados por ".", por exemplo, a constante zero do tipo Nat é representada como 'zero.Nat. De forma similar, variáveis também são QIDs contendo o nome e tipo separados por ":", por exemplo, 'N:Nat é a meta-representação da variável N do tipo Nat.

Termos são então construídos pela aplicação recursiva do operador _[_] a uma lista de termos, construída pelo operador _,_.

```
sort TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc gather (e E) prec 120] .
op _[_] : Qid TermList -> Term [ctor] .
```

Por exemplo, o termo f(a,b) (sendo a e b constantes do tipo S) é representado no módulo META-TERM como 'f['a.S,'b.S] e o termo {f(a,b)} (para {_} definido como um operador com sintaxe in-fixada) é meta-representado por '_['f['a.S,'b.S]]. Dado ainda que termos do módulo META-TERM podem ser meta-representados como qualquer outro termo de outro módulo, a meta-representação de um termo também pode ser meta-representada:

```
'_`[_`][''f.Sort,'_`,_[''a.S.Constant,''b.S.Constant]]
```

No módulo META-MODULE, que importa o módulo META-TERM, módulos de sistemas e módulos funcionais são meta-representados por uma sintaxe muito similar a sua sintaxe original:

```
sorts FModule SModule Module .
subsorts FModule < SModule < Module .
op fmod_is_sorts_.____endfm : Qid ImportList SortSet SubsortDeclSet
   OpDeclSet MembAxSet EquationSet -> FModule [...] .
op mod_is_sorts_._____endm : Qid ImportList SortSet SubsortDeclSet
   OpDeclSet MembAxSet EquationSet RuleSet -> SModule [...] .
```

O operador `fmod_is_sorts_.____endfm`, por exemplo, constrói um termo que representa a meta-representação de um módulo funcional. O primeiro argumento é um QID, a meta-representação do nome do módulo. Os argumentos seguintes são termos construídos por operadores que definem a meta-representação das declarações de tipos, subtipos, operações, assertivas de pertinência e equações. No manual do sistema Maude [19] todos os detalhes sobre a meta-representação de módulos são explicados. Por exemplo, o conjunto de regras de reescrita declaradas em um módulo de sistema é meta-representado pela sintaxe:

```
sorts Rule RuleSet .
subsort Rule < RuleSet .
op rl_=>_[_]. : Term Term AttrSet -> Rule [ctor] .
op crl_=>_if_[_]. : Term Term Condition AttrSet -> Rule [ctor] .
op none : -> RuleSet [ctor] .
op __ : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .
```

No módulo `META-LEVEL` são definidas primitivas de meta-programação, também chamadas "descent functions", que reduzem computações de meta-nível para computações no nível objeto (nível zero da torre de reflexão). Dado que estas operações, em geral, além de seus argumentos, devem receber a meta-representação de módulos, tipos ou termos, o módulo `META-LEVEL` também oferece várias funções para movimentação da representação destes elementos entre os níveis de reflexão.

Para exemplificar, o processo para reduzir um termo para sua forma normal é reificado pela função

```
op metaReduce : Module Term -> Term [special(...)] .
```

onde $metaReduce(\overline{M}, \overline{t})$ retorna a meta-representação $\overline{u}$ da forma normal u do termo t no módulo M. O processo para aplicar uma regra de reescrita é reificado pela função

```
op metaApply : Module Term Qid Substitution Nat ~> ResultTriple? [...] .
```

onde $metaApply(\overline{M}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ é computada da seguinte forma:

- o termo t é primeiramente reduzido para sua forma normal usando as equações de M;
- é feito o casamento de padrões do termo resultante da etapa anterior com todas as regras de M que tenham rótulo l, parcialmente instanciadas pelas substituições σ;
- os primeiros n casamentos bem-sucedidos são descartados; se existir um $(n+1)$'ézimo casamento, esta regra é aplicada usando este casamento e os passos seguintes são executados; caso contrário, a constante `failure` é retornada;
- o termo resultante da aplicação de uma regra com o $(n + 1)$'ézimo casamento é completamente reduzido para usa forma normal usando as equações de M;
- a tripla formada pela meta-representação do termo resultante da etapa anterior, a meta-representação do tipo deste termo e a meta-representação das substituições usadas na redução é retornada.

No módulo `META-LEVEL` temos ainda a função `metaRewrite` que reifica o processo de reescrever um termo usando o interpretador padrão de Maude e a função `metaParse` que reifica o processo de casamento de padrão, dentre outras.

2.3.5 Usando Maude como metalinguagem

A utilização de Maude como uma metalinguagem é uma conseqüência natural das propriedades úteis de lógica de reescrita como lógica e "framework" semântico [22].

Lógicas e linguagens de especificação podem ser mapeadas para lógica de reescrita por uma transformação na forma da Equação 2.2, que mapeia teorias ou módulos de uma linguagem $\mathcal{L}$ em teorias de reescrita.

$$\phi \ : \ \mathcal{L} \rightarrow \mathcal{R} \tag{2.2}$$

Em virtude de lógica de reescrita ser reflexiva, uma linguagem que suporte lógica de reescrita, como Maude, suporta a especificação de mapeamentos entre representações na forma da equação 2.2. Sendo assim, Maude transforma-se em uma metalinguagem onde uma grande variedade de linguagens de programação, especificação e "design", e sistemas lógicos e computacionais podem ser semanticamente definidos e implementados.

Em Maude, utilizando o módulo `META-LEVEL`, podemos tornar o mapeamento ϕ executável. Uma especificação ou programa em $\mathcal{L}$ torna-se executável em Maude a partir da

implementação de uma meta-função que, encapsulando as funções do módulo `META-LEVEL`, transforma esta especificação ou programa em uma teoria de reescrita em Maude. Mais especificamente, podemos reificar o mapeamento ϕ da equação 2.2 a partir da definição de um tipo de dados $Module_{\mathcal{L}}$ para representar módulos da lógica ou linguagem $\mathcal{L}$. Dado que no módulo `META-LEVEL` temos o tipo de dados `Module`, cujos termos representam teorias de reescrita, podemos internalizar o mapeamento ϕ por uma meta-função $\overline{\phi}$ equacionalmente definida

$$\overline{\phi} : Module_{\mathcal{L}} \rightarrow Module.$$

Na realidade, em virtude do resultado geral obtido por Bergstra e Tucker [37], qualquer mapeamento de representações computável ϕ pode ser especificado por um número finito de equações Church-Rosser e com terminação.

Tendo este mapeamento de representações definido em Maude, podemos executar em Maude a teoria de reescrita $\overline{\phi}(M)$ associada ao módulo ou teoria M de $\mathcal{L}$. Isto foi feito, por exemplo, na implementação de Full Maude [29], na implementação da ferramenta Real-Time Maude [30] e na implementação de Maude CBabel tool, que será apresentada no Capítulo 3.

Com a definição de um tipo de dados $Module_{\mathcal{L}}$, em um módulo que estenda `META-LEVEL`, podemos definir a sintaxe de $\mathcal{L}$ em Maude. No entanto, para oferecer um ambiente executável para $\mathcal{L}$ em Maude, são necessários ainda outros recursos como, por exemplo, processamento de entradas e saídas de dados e persistência de estado. Estes recursos devem permitir a interação com o interpretador de $\mathcal{L}$ que desejamos definir. Isto é, devemos ser capazes de carregar definições de módulos, executar comandos e obter resposta dos comandos. O módulo `LOOP-MODE` torna isto possível ao oferecer recursos para manter um estado persistente enquanto executa entrada e saída de dados (um "loop" do tipo leitura-avaliação-escrita). Conseqüentemente, um ambiente para uma linguagem $\mathcal{L}$ em Maude é tipicamente definido por um módulo que contém os módulos `META-LEVEL` e `LOOP-MODE` como submódulos.

2.3.6 Módulos orientados a objetos

Full Maude suporta a notação para orientação a objetos descrita em [38]. Em um sistema concorrente orientado a objetos, o estado corrente, chamado configuração, tem, tipicamente, a estrutura de um *multiset* de objetos e mensagens que evolui a partir de reescritas concorrentes, módulo os axiomas de associatividade, comutatividade e identidade (denominadas reescritas ACI). As regras são usadas para descrever os efeitos dos eventos

de comunicação que ocorrem entre objetos e mensagens. Isto é, podemos intuitivamente pensar que as mensagens "viajam" ao encontro dos objetos para os quais elas foram enviadas, causando assim "eventos de comunicação" pela aplicação das regras de reescrita. Podemos pensar, assim, que a computação orientada a objetos é uma dedução em lógica de reescrita. Desta maneira, as configurações S que podem ser alcançadas a partir de uma configuração inicial S_0, são exatamente aquelas para as quais existe uma seqüência $S_0 \rightarrow S$, dedutível em lógica de reescrita, a partir das regras de reescrita que especificam o comportamento de um dado sistema orientado a objetos.

Em Full Maude, módulos orientados a objetos são declarados pela sintaxe:

omod `<Nome>` **is** `<Declarações>` **endom**

Todo módulo orientado a objetos importa implicitamente o módulo `CONFIGURATION` que define os conceitos básicos dos sistemas orientados a objetos em Full Maude.

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
           [ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Configuration .
  subsort Object Msg < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
          [ctor assoc config comm id: none] .
endm
```

Deve-se observar que nenhuma operação construtora é definida para os tipos de dados `Msg`, `Attribute`, `Oid`, `Cid`. Estes tipos serão completamente definidos pelo módulo orientado a objetos que estender `CONFIGURATION`, definindo um particular sistema orientado a objetos. Na versão 2.1.1 de Maude, os atributos `config` e `object` são utilizados para implementação de um comportamento específico na escolha das regras de reescrita quando o comando `frewrite` é utilizado [19].

Em um módulo orientado a objetos de Maude podemos declarar classes e subclasses. Cada classe é declarada pela sintaxe

$$\texttt{class}\ C \mid a_1 : s_1\ ,\ \ldots\ ,\ a_n : s_n\ .$$

onde C é o nome da classe e para cada $a_1 : s_i$, a_i é o identificador do atributo e s_i é o tipo de dados dos valores possíveis o atributo. A relação de subclasse é definida com a sintaxe

$$\texttt{subclass}\ C < C'.$$

onde C é definida como subclasse de C'. Em geral, no entanto, a classe C pode ser definida como subclasse de várias classes $D_1 \ldots D_n$, isto é, multipla herança é suportada. Um objeto, em um determinado estado do sistema, é representado por um termo

$$< O : C \mid a_1 : v_1 \ , \ \ldots \ , \ a_n : v_n >$$

onde O é o identificador do objeto, C é o identificador da classe do objeto, os a_i são os nomes dos atributos do objeto e v_i seus respectivos valores. Um objeto sem atributos pode ser representado de forma simplificada pelo termo $< O : C \mid \ >$.

Mensagens não têm uma sintaxe fixa. A sintaxe das mensagens pode ser definida da forma mais conveniente pelo usuário para cada aplicação. Uma declaração de mensagem

$$\texttt{msg}\ to\ :\ p_1\ \ldots\ p_n\ \rightarrow\ Message\ .$$

define o nome da mensagem como *to* e os tipos de dados de seus parâmetros. Geralmente, o primeiro parâmetro é um identificador de objeto para o qual a mensagem é destinada.

O multiset de objetos e mensagens que representa o estado concorrente de um sistema orientado a objetos é definido pelo operador associativo-comutativo _ _ (justaposição) definido no módulo `CONFIGURATION`. A associatividade e comutatividade do operador _ _ faz dele uma "sopa de objetos e mensagens" [39], desta forma, qualquer conjunto de objetos e mensagens pode, em um dado instante, participar em conjunto de uma transição concorrente que corresponda a um evento de comunicação. A interação concorrente entre objetos é axiomatizada pelas regras de reescrita. A forma geral destas regras é dada por:

$$\begin{array}{rll} \texttt{crl}\ [r] & : & M_1\ \ldots\ M_n\ \langle\ O_1 : F_1 \mid atts_1\ \rangle\ \ldots\ \langle\ O_m : F_m \mid atts_m\ \rangle \Rightarrow \\ & & \langle\ O_{i_1} : F'_{i_1} \mid atts'_{i_1}\ \rangle\ \ldots\ \langle\ O_{i_k} : F'_{i_k} \mid atts'_{i_k}\ \rangle \\ & & \langle\ Q_1 : D_1 \mid atts''_1\ \rangle\ \ldots\ \langle\ Q_p : D_p \mid atts''_p\ \rangle \\ & & M'_1\ \ldots\ M'_q \\ \quad \texttt{if} & & C\ . \end{array}$$

onde r é o rótulo da regra, cada M_i representa uma mensagem, $i_1, \ldots, i_k$ são números diferentes dos originais $1, \ldots, m$ e C é a condição para aplicação da regra. Ou seja, um determinado número de objetos e mensagens pode participar em conjunto de um transição

onde novos objetos podem ser criados, alguns objetos podem ser destruídos, outros podem ter seus estados alterados (valores de seus atributos), e onde novas mensagens podem ser criadas. Se dois ou mais objetos aparecem no lado esquerdo da regra, dizemos que a regra é *síncrona*, pois, força a participação conjunta destes objetos na transição. Se apenas um objeto aparece no lado esquerdo da regra, dizemos que a regra é *assíncrona*. Por convenção, os atributos que não têm seu valor alterado e não afetam o próximo estado de outros atributos não precisam ser mencionados na regra.

Por exemplo, o módulo orientado a objetos `ACCNT` especifica o comportamento dos objetos da classe `Accnt` de contas bancárias. Os objetos desta classe têm apenas um atributo, seu saldo (`bal`). Os objetos, instâncias de `Accnt`, podem receber as mensagens de depósito, débito e transferência entre contas.

```
(omod ACCNT is
  protecting INT .
  protecting QID .

  subsort Qid < Oid .
  class Accnt | bal : Int .

  msgs credit debit : Oid Int -> Msg .
  msg transfer_from_to_ : Int Oid Oid -> Msg .

  vars A B : Oid .
  vars M N N' : Int .

  rl [credit] :
    credit(A, M) < A : Accnt | bal : N > =>
    < A : Accnt | bal : (N + M) > .

  crl [debit] :
    debit(A, M) < A : Accnt | bal : N > =>
    < A : Accnt | bal : (N - M) >
   if N > M .

  crl [transfer] :
    (transfer M from A to B) < A : Accnt | bal : N > < B : Accnt | bal : N' >
    =>
    < A : Accnt | bal : (N - M) > < B : Accnt | bal : (N' + M) >
   if N > M .
endom)
```

Como mencionado na Seção 2.3.5, Full Maude é uma instância do mapeamento ϕ definido pela equação 2.2. Desta forma, para serem *executáveis* na máquina de reescrita do sistema Maude, os módulos orientados a objetos de Full Maude são transformados em módulos de sistema de Core Maude. Esta transformação é completamente descrita em [29].

2.4 Ferramentas de análise de Maude

Maude é uma implementação de alto desempenho de lógica de reescrita que oferece, em adição aos itens mencionados nas seções anteriores, uma grande variedade de ferramentas de verificação. Nesta seção, inspirados pelos textos de [40, 28, 30, 19], apresentamos as principais ferramentas de verificação de Maude para especificação e análise de sistemas.

Na Seção 2.3.2, vimos que o modelo de um módulo de sistema em Maude é um sistema de transição. Neste modelo, a árvore dos caminhos de um estado inicial S, é uma árvore onde os nós são os estados alcançáveis a partir de S e as arestas são as regras de reescrita que os conectam. Os caminhos correspondem assim às possíveis formas do sistema evoluir e um particular caminho é escolhido quando adotamos uma estratégia de execução. Neste tipo de modelo, diferentes análises podem ser realizadas, como por exemplo:

1. *Análise estática.* Neste processo, analisamos a estrutura do modelo para entender como os elementos são organizados e relacionados. Com a análise estática podemos detectar inconsistências geradas por declarações incorretas ou esquecidas.

2. *Simulação da especificação.* Uma especificação em Maude pode ser simulada para identificação de "bugs". Com este procedimento, aumentamos o grau de confiança na especificação. Em Maude, podemos executar o modelo utilizando a estratégia padrão do sistema Maude de escolha das regras de reescrita a serem aplicadas ao termo inicial.

3. *Exploração dos estados com busca em largura.* Um sistema distribuído não-determinístico exibe diferentes comportamentos a partir de um estado inicial, comportamentos estes que não são capturados por uma única estratégia de execução. A busca em largura examina todos os possíveis caminhos no grafo de transições de um estado inicial até um determinado nível, ou até que um número determinado de estados que satisfaçam alguma propriedade seja encontrado.

4. *Exploração dos estados com busca reversa.* Para modelos que satisfaçam certas condições, a busca reversa pode responder questões como: "A partir de quais estados podemos alcançar este estado S?" Como este tipo de verificação, podemos descobrir, por exemplo, quais os estados predecessores de um particular estado.

5. Buscas restringem-se a localizar estados que satisfaçam determinadas propriedades individualmente. Na *Verificação explícita do modelo*, consideramos também

as propriedades dos caminhos, por exemplo, podemos perguntar: "Se atingirmos um estado que satisfaça a propriedade P, então os estados seguintes sempre irão satisfazer a propriedade Q?"

6. *Provas formais.* Para sistemas críticos, podemos ainda realizar provas formais sobre sua correção do sistema. Tais provas podem ser conduzidas por ferramentas formais como o verificador indutivo de teoremas para Maude desenvolvido por Clavel [24].

É importante destacar que os métodos de verificação podem ser organizados de forma crescente em relação a sua complexidade e custo computacional. Podemos adotar como metodologia a utilização inicial dos métodos menos custosos e complexos, com objetivo de aumentar a confiança na especificação. Posteriormente, quando necessário, podemos adotar métodos mais rigorosos e computacionalmente custosos.

Nas seções seguintes, apresentamos os comandos e ferramentas disponíveis no sistema Maude para realização de análise dos tipos 2, 3 e 5 supra. Indicamos o manual de Maude [19] para a completa referência sobre a sintaxe de tais comandos e ferramentas. Estes tipos de análise serão utilizados nos Capítulos 4 e 5. Os tipos de análise 1, 4 e 6 não foram explorados neste trabalho, embora futuramente possam ser utilizados.

2.4.1 Simulação

A forma mais simples de verificar se a especificação de um sistema está correta é exibir um (dentre os possíveis, ou talvez, infinitos) comportamento a partir de um estado inicial.

Em Maude, os comandos `rewrite` e `frewrite` (reescrita justa) executam *um* arbitrário comportamento do sistema a partir de uma estratégia padrão para escolha das regras a serem aplicadas nas reescritas do termo inicial.

Considerando que a especificação de um sistema pode ser tal que o sistema não termine, o processo de reescrita do termo inicial pode então não terminar. Neste caso, ambos os comandos aceitam, opcionalmente, receber um limite máximo de passos de reescrita a ser executado.

2.4.2 Exploração de estados por busca em largura

Em sistemas não-determinísticos, a exploração de apenas um comportamento, dentre todos os possíveis, nem sempre é suficiente para garantir que uma determinada

propriedade seja válida. Para explorar todos os possíveis comportamentos a partir de um estado inicial, Maude oferece o comando `search`.

O comando `search` em Maude busca por todos os estados alcançáveis a partir de um estado inicial, que satisfaçam uma determinada condição. A busca é realizada em largura, isto é, a partir do termo inicial são visitados primeiramente os termos alcançáveis com um passo de reescrita, em seguida, os termos alcançáveis em dois passos de reescrita etc.

O comando `search` de Maude implementa um sofisticado mecanismo de "cache" dos estados visitados durante uma busca para evitar que durante o procedimento um mesmo estado seja visitado mais de uma vez. Mesmo considerando sua eficiente implementação, o número de estados armazenados no "cache" é limitado obviamente à capacidade de memória do equipamento sendo utilizado e, desta forma, a viabilidade da utilização desta ferramenta está relacionada ao número de estados que precisam ser visitados.

O armazenamento dos estados visitantes é também útil para a exibição dos caminhos de reescrita percorridos pela busca. Após a execução do comando `search`, podemos utilizar o comando `show path N`, onde N representa um número de estado, para visualizar o caminho de reescrita do estado inicial até o estado N. Por exemplo, na execução do comando `search` da Seção 2.3.2, o único estado localizado pela busca é o estado de número 27. Utilizando o comando `show path`, podemos visualizar as reescritas executadas do estado inicial até o estado 27:

```
Maude> show path 27 .
state 0, Group: s(left) w(left) g(left) c(left)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] . ]===>
state 2, Group: s(right) w(left) g(right) c(left)
===[ rl s(S) => s(change(S)) [label shepherd-alone] . ]===>
state 7, Group: s(left) w(left) g(right) c(left)
===[ rl s(S) w(S) => s(change(S)) w(change(S)) [label wolf] . ]===>
state 13, Group: s(right) w(right) g(right) c(left)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] . ]===>
state 20, Group: s(left) w(right) g(left) c(left)
===[ rl s(S) c(S) => s(change(S)) c(change(S)) [label cabbage] . ]===>
state 25, Group: s(right) w(right) g(left) c(right)
===[ rl s(S) => s(change(S)) [label shepherd-alone] . ]===>
state 26, Group: s(left) w(right) g(left) c(right)
===[ rl s(S) g(S) => s(change(S)) g(change(S)) [label goat] . ]===>
state 27, Group: s(right) w(right) g(right) c(right)
```

2.4.3 Verificador de modelos de Maude

Um verificador de modelos tipicamente deve suportar dois níveis de especificação: (i) o nível de especificação do sistema, onde o sistema concorrente que será analisado é formalizado; e, (ii) o nível de especificação das propriedades, onde as propriedades que serão analisadas são especificadas.

Nesta seção, apresentamos primeiramente *lógica linear temporal* (LTL), uma particular lógica temporal para especificação de propriedades. Em seguida, apresentamos como o verificador de modelos de Maude [23] pode ser utilizado para verificação de sistemas especificados por módulos de sistema em Maude. LTL permite a especificação de propriedades ditas "safety" (que asseguram que "alguma coisa ruim" nunca ocorra) e propriedades ditas "liveness" (que asseguram que "alguma coisa boa" eventualmente acontece). Indicamos o manual de Maude [19] e [23] como referência para a completa descrição do verificador de modelos de Maude e do mapeamento de LTL em lógica de reescrita.

2.4.3.1 Fórmulas LTL

Dado um conjunto AP de proposições atômicas, definimos as fórmulas da *lógica proposicional linear temporal* $LTL(AP)$, intuitivamente como:

True $\top \in LTL(AP)$;

Proposições atômicas se $p \in AP$, então $p \in LTL(AP)$;

Operador Next se $\varphi \in LTL(AP)$, então $\bigcirc\varphi \in LTL(AP)$;

Operador Until se $\varphi, \psi \in LTL(AP)$, então $\varphi \,\mathcal{U}\, \psi \in LTL(AP)$;

Conectivos booleanos se $\varphi, \psi \in LTL(AP)$, então $\neg\varphi$ `e` $\varphi \vee \psi \in LTL(AP)$.

Outros conectivos booleanos e operadores temporais podem ser definidos a partir destes. A sintaxe para as fórmulas de LTL em Maude é definida no módulo funcional `LTL` [19], predefinido em Maude, que declara os tipos de dados `Prop` e `Formula`. Neste módulo, são declarados os operadores para as fórmulas em LTL, mas nenhum operador é definido para o tipo `Prop` das proposições atômicas, sendo este apenas declarado como subtipo do tipo `Formula`. Veremos adiante como são definidas as proposições atômicas para um dado módulo de sistema `M`.

2.4.3.2 Estruturas de Kripke e teorias de reescrita

Estruturas de Kripke são modelos para lógica proposicional temporal. Fixando um tipo de dados distinto k, o modelo inicial $\mathcal{T}_{\mathcal{R}}$ de uma teoria de reescrita $R = (\Sigma, E, R)$ tem associada uma estrutura de Kripke. Precisamos então definir como uma estrutura de Kripke é associada a uma teoria de reescrita especificada por um módulo de sistema `M` de Maude.

Uma relação binária $R \subseteq A \times A$ em um conjunto A é chamada total se, e somente se, para cada $a \in A$, existe pelo menos um $a' \in A$ tal que $(a, a') \in R$. Se R não for total, pode ser feita total pela definição de uma $R^{\bullet} = R \cup \{(a, a) \in A^2 \mid \nexists\, a' \in Aa, a') \in R$.

Essencialmente, uma estrutura de Kripke é um sistema de transição (total) não rotulado no qual adicionamos uma coleção de predicados de estado ao seu conjunto de estados. Formalmente:

Definição 2.4.1 *Uma estrutura de Kripke é uma tupla $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ onde A é um conjunto, chamado conjunto de estados, $\rightarrow_{\mathcal{A}}$ é uma relação binária total em A, chamada de relação de transição, e $L : A \rightarrow \mathcal{P}(AP)$ é uma função, chamada função de rotulação que associa a cada estado $a \in A$ o conjunto $L(a)$ das proposições atômicas em AP que são válidas no estado a.*

A semântica de LTL é dada pela relação de satisfação

$$\mathcal{A}, a \models \varphi$$

entre uma estrutura de Kripke $\mathcal{A}$ tendo AP como suas proposições atômicas, um estado $a \in A$, e uma fórmula em LTL $\varphi \in LTL(AP)$. Mais especificamente, $\mathcal{A}, a \models \varphi$ é válida, se, e somente se, para cada caminho $\pi \in Path(\mathcal{A})_a$ a relação de satisfação do caminho

$$\mathcal{A}, \pi \models \varphi$$

é válida. Onde definimos o conjunto $Path(\mathcal{A})_a$ de caminhos de computação iniciados no estado a como um conjunto de funções na forma $\pi : \mathbb{N} \rightarrow A$ tal que $\pi(0) = a$ e, para cada $n \in \mathbb{N}$, temos $\pi(n) \rightarrow_{\mathcal{A}} \pi(n + 1)$.

Definimos então a relação de satisfação de qualquer caminho, iniciado em qualquer estado, como:

- Sempre temos $\mathcal{A}, \pi \models_{LTL} \top$;

- Para $p \in AP$,

$$\mathcal{A}, \pi \models_{LTL} p \Leftrightarrow p \in L(\pi(0))$$

- Para $\bigcirc\varphi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \bigcirc\varphi \Leftrightarrow \mathcal{A}, s; \pi \models_{LTL} \varphi,$$

  onde $s : \mathbb{N} \to \mathbb{N}$ é a função sucessora e onde $s; \pi(n) = \pi(s(n))$.

- Para $\varphi \,\mathcal{U}\, \psi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \,\mathcal{U}\, \psi \;\Leftrightarrow$$
$$(\exists n \in \mathbb{N})((\mathcal{A}, s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N}) m < n \Rightarrow \mathcal{A}, s^m; \pi \models_{LTL} \varphi))$$

- Para $\neg\varphi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \neg\varphi \;\Leftrightarrow\; \mathcal{A}, \pi \not\models_{LTL} \varphi$$

- Para $\varphi \vee \psi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \;\Leftrightarrow\; \mathcal{A}, \pi \models_{LTL} \varphi \text{ ou } \mathcal{A}, \pi \models_{LTL} \psi$$

Para associar uma estrutura de Kripke a uma teoria de reescrita $R = (\Sigma, E, R)$ especificada por um módulo de sistema `M` de Maude, precisamos então definir: (i) o tipo de dados k de estados na assinatura Σ; e, (ii) os predicados de estado relevantes, ou seja, o conjunto AP de proposições atômicas.

Após escolher um tipo de dados em `M` como o tipo de dados para estados, por exemplo, `Foo`, podemos especificar os predicados de estado relevantes em um módulo, por exemplo, `M-PREDS` que importa `M` no modo `protecting`.

```
mod M-PREDS is
 protecting M .
 including SATISFATION .
 subsort Foo < State .
 ...
endm
```

O módulo `SATISFATION`, predefinido no sistema Maude, contém apenas a declaração do tipo `State` e do operador `|=`:

```
fmod SATISFACTION is
  pr LTL .
```

```
  sort State .
  op _|=_ : State Formula ~> Bool .
endfm
```

Com a importação, todos os termos do tipo `Foo` em `M` são também transformados em termos do tipo `State` em `M-PREDS`. Cada predicado é declarado como um operador do tipo `Prop`. Na lógica proposicional LTL convencional, as proposições atômicas são constantes, em Maude, no entanto, podemos definir predicados de estado parametrizados. Ou seja, operadores do tipo `Prop` não precisam ser constantes. Definimos então a semântica de tais predicados a partir de um conjunto de equações que especificam para quais estados um determinado predicado de estado é avaliado como `true`. Assumimos que estas equações, quando adicionadas as já existentes no módulo `M`, são "ground" Church-Rosser [6] e com terminação.

Apenas os casos onde o predicado é válido precisam ser especificados. Ou seja, dado um estado t e um, possivelmente parametrizado, predicado de estado $p(u_1, \ldots, u_n)$, quando a expressão sem variáveis $t \models p(u_1, \ldots, u_n)$ não puder ser simplificada para $true$, então o predicado não é válido em t. Isto significa que para especificar a semântica de um predicado de estado é suficiente a especificação de equações com a forma geral:

$$t \models p(u_1, \ldots, u_n) = true \text{ if } C.$$

Podemos então associar ao módulo de sistema `M` (que especifica uma teoria de reescrita $R = (\Sigma, E, R)$ com um tipo de dados k para estados, e um conjunto de predicados de estado Π definidos por equações D) uma estrutura de Kripke cujos predicados atômicos são especificados pelo conjunto $AP_\Pi = \{\theta(p) \mid p \in \Pi, \theta \text{ substituições sem variáveis}\}$, onde por convenção, usamos a notação simplificada $\theta(p)$ para representar o termo sem variáveis $\theta(p(x_1, \ldots, x_n))$. Isto define a função de rotulação L_Π sobre o conjunto de estados $T_{\Sigma/E,k}$, associando para cada $[t] \in T_{\Sigma/E,k}$ um conjunto de proposições atômicas

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid (E \cup D) \vdash (\forall \emptyset) t \models \theta(p) = true\}.$$

A estrutura de Kripke que estamos interessados é então definida por

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi),$$

onde $(\rightarrow^1_{\mathcal{R}})^\bullet$ especifica a relação total que estende a relação de $\mathcal{R}$-reescritas em um passo

[6] Dizemos que E é "ground" Church-Rosser se a garantia de confluência é válida apenas para termos sem variáveis $t \in T_\Sigma$.

$\rightarrow^1_{\mathcal{R}}$ entre estados do tipo k, ou seja, $[t] \rightarrow^1_{\mathcal{R}} [t']$ é válida se, e somente se, existe um $u \in [t]$ e $u' \in [t']$ tal que u' é o resultado da aplicação de uma das regras de R a algum subtermo de u. Sempre podemos escolher u como a forma canônica de t nas equações E se assumirmos E como "ground" Church-Rosser e com terminação e R "ground" coerente [7] em relação a E.

[7] Dizemos que R é "ground" coerente em relação a E se a garantia de coerência, Seção 2.3.2, é válida apenas para termos sem variáveis $t \in T_\Sigma$.

3 Maude CBabel tool

O objetivo da análise formal de uma arquitetura é auxiliar o arquiteto do sistema a responder questões a respeito da arquitetura antes de sua implementação. Em [41] os autores destacam que a escolha de um "framework" semântico para especificação de arquiteturas de "software" deve levar em consideração:

- A facilidade de entendimento do modelo formal pelos projetistas da arquitetura do sistema.
- A flexibilidade para descrição de diferentes tipos de arquiteturas para uma mesma aplicação, isto porque, espera-se que o projetista da aplicação possa utilizar os resultados das análises formais para escolha da melhor arquitetura para uma dada aplicação.

Lógica de reescrita é um formalismo semântico capaz de representar diversas lógicas, linguagens de especificação e modelos de computação. Como apresentado na Seção 2.2, em lógica de reescrita, dedução e computação são "dois lados da mesma moeda" [18]. A escolha de lógica de reescrita como "framework" semântico para CBabel foi influenciada ainda pelos seguintes fatores:

- Com a utilização de lógica de reescrita temos como importante benefício, conforme apontado por Félix [42], o tratamento *ortogonal* dos aspectos estruturais da arquitetura (tipos de dados), definidos por equações, e dos aspectos comportamentais (concorrência e sincronização), dado pela regras de reescrita. Mais ainda, lógica de reescrita permite que análises sobre propriedades relacionadas a ambos os aspectos de uma arquitetura possam ser avaliadas de maneira homogênea, pois, ambos os aspectos são representados no mesmo formalismo.
- Lógica de reescrita tem uma natural representação algébrica para sistemas distribuídos orientados a objetos [38]. Como veremos a seguir, a representação de

uma arquitetura de "software" nos conceitos de orientação a objetos fornece boa intuição do formalismo a projetistas de sistemas não familiarizados com métodos formais.

- Recordando da Seção 2.2, lógica de reescrita é reflexiva. Sendo assim, podemos definir um mapeamento de CBabel para lógica de reescrita como uma meta-função de D_{CBabel}, o tipo de dados para representação de especificações em CBabel, para $\mathcal{R}$, uma teoria de reescrita.

- Sob condições razoáveis, lógica de reescrita é executável. Especificações em lógica de reescrita podem ser executadas com CafeOBJ [43], Elan [44] e Maude [19]. A executabilidade de lógica de reescrita permitiu o desenvolvimento de Maude CBabel tool, uma ferramenta de suporte para a especificação e análise de arquiteturas de "software" descritas em CBabel.

Algumas evidências dos benefícios na utilização de lógica de reescrita como "framework" semântico podem ser encontradas em [45].

Este capítulo está estruturado da seguinte forma. A semântica de CBabel em lógica de reescrita será apresentada na Seção 3.1, onde cada construção sintática de CBabel terá sua semântica expressa em lógica de reescrita. O leitor deve observar que, com o objetivo de simplificar o mapeamento da linguagem CBabel para lógica de reescrita, uma versão simplificada de CBabel foi considerada. Neste capítulo, quando nos referirmos a linguagem CBabel, estaremos nos referindo a versão apresentada no decorrer da Seção 3.1. Referências à sintaxe original de CBabel definida por Sztajnberg em [46], e ilustrada na Seção 2.1, serão destacadas. No final do capítulo, na Seção 3.2, apresentaremos a ferramenta Maude CBabel tool, uma implementação em Maude da semântica de CBabel, que permite a tradução de descrições em CBabel para módulos orientados a objetos de Maude.

3.1 Semântica orientada a objetos em lógica de reescrita de CBabel

Como a maioria das ADLs, CBabel é uma linguagem essencialmente declarativa. Destacamos abaixo os elementos de CBabel que serão formalizados em lógica de reescrita:

- Um *componente* pode ser um módulo ou conector. Um módulo representa uma

entidade que realiza alguma computação, por exemplo, um objeto ou função. Conectores interligam módulos, selecionando a forma e intermediando as interações destes.

- As *portas* identificam os pontos de acesso pelos quais os módulos e conectores oferecem e solicitam serviços. As portas podem ser de entrada, para oferta de serviço, ou saída, para requisição de serviço.

- Os *contratos* definem os aspectos não-funcionais da aplicação relacionados a: coordenação (controle de concorrência e sincronização no encaminhamento de requisições), distribuição (localização de componentes em ambientes distribuídos), interação (características de interação entre módulos).

- Um *link* estabelece a ligação de duas portas, uma porta de saída de um componente com uma porta de entrada de outro componente. Uma vez ligadas, as portas podem interagir.

- *Variáveis* podem ser locais ou de estado. No último caso, permitem que os componentes da arquitetura troquem informações atomicamente, ou seja, seguindo o modelo de comunicação por memória compartilhada.

- Uma *aplicação* é um módulo especial que instancia os componentes da arquitetura, estabelece a ligação das portas e a amarração das variáveis de estado.

Os conceitos de CBabel podem ser naturalmente interpretados nos conceitos de orientação a objetos. Lógica de reescrita, por sua vez, constitui-se um formalismo bastante flexível, como já mencionado, para especificação de diferentes modelos de concorrência [18] e, em particular, para especificação de sistemas concorrentes orientados a objetos [38]. Escolhemos então a notação para objetos e mensagens de Maude para expressar a semântica de CBabel em lógica de reescrita. Vale destacar, no entanto, que pelo fato da notação para objetos e mensagens de Maude ser na realidade apenas um "açúcar sintático" sobre lógica de reescrita [25, 38], nosso mapeamento não está restrito à Maude, sendo válido para qualquer outra implementação de lógica de reescrita. Por exemplo, em OBJ [47], as equações comportamentais fazem as vezes das regras de reescrita de Maude.

A Tabela 2 fornece uma visão geral do mapeamento das construções de CBabel para os conceitos de orientação a objetos em lógica de reescrita onde, essencialmente, componentes são mapeados para classes, suas instâncias para objetos; portas são declarações

de mensagens e estímulos de portas são representados como envio ou recebimento de mensagens.

componente	→	teoria orientada a objetos
instância de componente	→	objeto
estado da aplicação	→	conjunto de objetos e mensagens
porta	→	declaração de mensagem
estímulo de porta	→	transmissão de mensagem (regra de reescrita)
link	→	equação não condicional
contrato de coordenação	→	regra de reescrita
variáveis locais ou de estado	→	atributos de classes
amarração de variáveis	→	equações

Tabela 2: Mapeamento dos conceitos de CBabel para lógica de reescrita

Como veremos ao final deste capítulo, a ferramenta Maude CBabel tool implementa o mapeamento proposto através de uma função de transformação que, a partir de descrições de componentes CBabel, produz teorias de reescrita, especificamente módulos orientados a objetos de Maude. Esta função de transformação é implementada como uma composição de funções elementartes, que traduzem cada construção sintática CBabel em construções sintáticas de Maude. A Figura 3 apresenta uma intuição destes conceitos. Os módulos orientados a objetos de Maude são representados com suas três partes principais: a assinatura Σ, o conjunto de equações E e o conjunto de regras de reescrita R. As setas indicam que partes dos módulos orientados a objetos terão elementos adicionados na tradução de cada construção CBabel.

Nas seções seguintes, apresentamos a semântica de CBabel em lógica de reescrita. Para cada construção elemento de CBabel, apresentaremos sua sintaxe, a intuição de sua semântica em lógica de reescrita e a formalização desta semântica.

No restante desta seção, adotamos a convenção de utilizar letras minúsculas para elementos de um conjunto e letras maiúsculas para conjuntos.

3.1.1 Componentes

Um componente em CBabel pode ser um módulo ou conector. Um módulo pode declarar variáveis locais, portas de entrada e portas de saída. Um conector, além das mesmas declarações que um módulo, também pode declarar um contrato de coordenação e variáveis de estado. A seguinte sintaxe é utilizada para declaração de conectores e módulos:

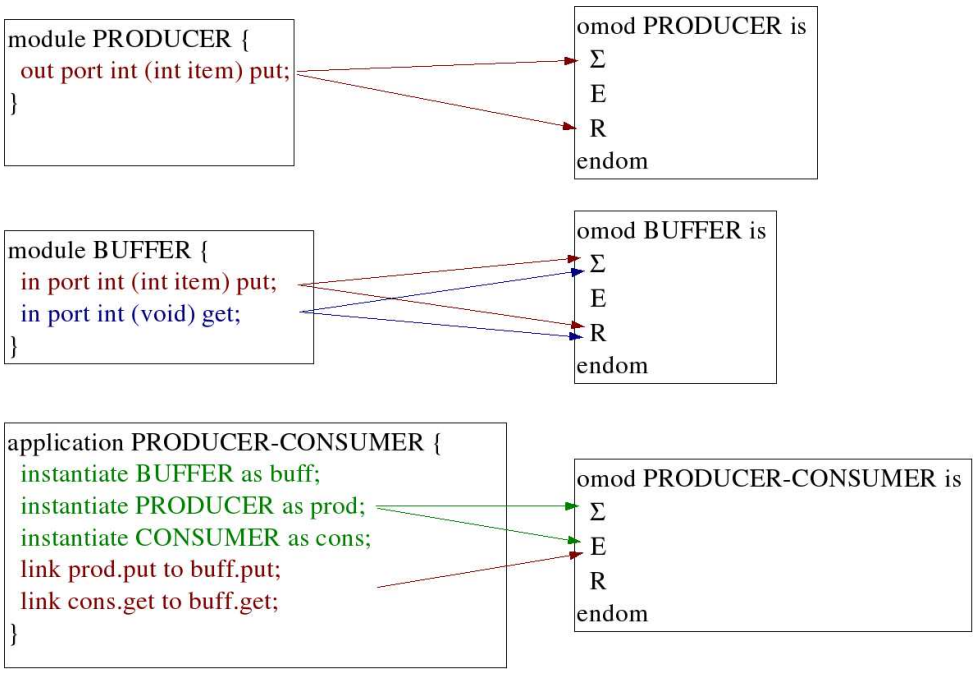

Figura 3: Visão gráfica do mapeamento para a arquitetura `PRODUCER-CONSUMER`

⟨component-decl⟩ →
module ⟨id⟩ **{** ⟨module-decls⟩ **}**
| **connector** ⟨id⟩ **{** ⟨connector-decls⟩ **}**

⟨var-type⟩ → **int** | **bool**
⟨var-decl⟩ →
var ⟨var-type⟩ ⟨id⟩ **;**
| **var** ⟨var-type⟩ ⟨id⟩ = ⟨exp⟩ **;**
| **staterequired** ⟨var-type⟩ ⟨id⟩ **;**

Um componente é mapeado em lógica de reescrita para uma teoria de reescrita. Para cada componente da arquitetura, uma classe com mesmo nome e um método construtor são declarados na assinatura da teoria de reescrita correspondente. Instâncias deste componente são então representadas por objetos desta classe.

A declaração de um módulo CBabel é uma tupla (n, V, I, O) onde n é o identificador do módulo que representa seu nome, V é um conjunto de declarações de variáveis, I o conjunto de declarações de portas de entrada e O o conjunto de declarações de portas de saída. A formalização das declarações de portas será tratada na Seção 3.1.2. Cada declaração de variável local é dada por uma tupla (v, l, t) onde v é o identificador que

representa o nome da variável, l seu valor inicial e t é o seu tipo, que deve ser um dos tipos de dados primitivos de CBabel [46]. A declaração de variáveis de estado será tratada na Seção 3.1.5.

A declaração de um conector CBabel é uma tupla (n, V, I, O, c) onde n é o identificador do conector que representa seu nome. Como na declaração de um módulo, V, I e O correspondem aos conjuntos de declarações de variáveis, portas de entrada e portas de saída. Por fim, c é a declaração de um contrato, assunto que será tratado na Seção 3.1.3.

A semântica em lógica de reescrita de um componente CBabel é dada por uma teoria de reescrita $\mathcal{R} = (\Sigma, E, R)$ cuja assinatura Σ:

- importa as declarações da teoria de reescrita `CBABEL-CONFIGURATION`, que serão mostradas no decorrer do texto;
- inclui uma declaração de classe

$$class\ n \mid S\ .$$

  onde S é o conjunto de atributos da classe. Para cada elemento v_i de V, um atributo s_i é incluindo em S.

O conjunto de equações E inclui a equação 3.1, onde ω é o identificador do objeto, cada a_i corresponde a um identificador de variável v_i em V, e cada l_i um valor de inicialização de v_i em V. Na Seção 3.1.2 trataremos dos elementos de AS, o restante do conjunto de atributos do objeto.

$$eq\ instantiate(\omega, n)\ = < \omega\ :\ n \mid a_1\ :\ l_1,\ \ldots\ , a_n\ :\ l_n\ , AS > \ . \tag{3.1}$$

Sendo que a operação

$$instantiate\ :\ Oid\ Cid\ \rightarrow\ Object$$

é declarada na teoria de reescrita `CBABEL-CONFIGURATION`.

A equação 3.1 especifica que dado um identificador de objeto ω e uma classe n, a operação $instantiate(\omega, n)$ produz uma instância de objeto da classe n com os atributos inicializados para o valor l de cada declaração de variável em V.

3.1.2 Portas

Um componente CBabel pode declarar portas de entrada e portas de saída. Portas de entrada são usadas pelo componente para oferecer seus serviços a outros componentes. Portas de saída são utilizadas pelo componente para solicitar serviços de outros componentes. Uma porta pode ser síncrona ou assíncrona. Portas assíncronas são declaradas com a inclusão da palavra-chave `oneway` na declaração da porta. A ausência da palavra-chave `oneway` caracteriza a declaração de uma porta síncrona. Na sintaxe para declaração de portas apresentada abaixo, deve-se observar que, diferentemente da versão original de CBabel, as portas não declaram parâmetros ou tipo de retorno. (Esta simplificação é resultado de nosso foco inicial no desenvolvimento de uma ferramenta para análise de aspectos relacionados a coordenação das interações entre componentes de uma arquitetura, não sendo relevante assim os dados que efetivamente são trocados pelas instâncias de componentes.)

⟨port-type⟩ → **in** | **out**
⟨port-decl⟩ →
⟨port-type⟩ **port** ⟨id⟩ **;**
| ⟨port-type⟩ **port oneway** ⟨id⟩ **;**

Para que duas instâncias de componentes se comuniquem, uma porta de saída de uma das instâncias deve ser *ligada* a uma porta de entrada da outra instância. A Seção 3.1.4 trata da formalização das ligações de portas.

Cada declaração de portas em um componente CBabel é mapeada para a declaração de uma mensagem na assinatura da teoria de reescrita associada ao componente. No entanto, para simplificar o mapeamento, ao invés de declararmos uma mensagem para cada porta, duas mensagens genéricas, *send* e *ack* são utilizadas. As portas são então mapeadas para constantes que parametrizam estas mensagens genéricas. O estímulo de uma porta é representado pela passagem de uma mensagem *send* para o objeto apropriado, isto é, o objeto que representa a instância do componente cuja porta foi estimulada. De forma análoga, a resposta de um estímulo, quando a porta for síncrona, é representada pela passagem da mensagem *ack* para o objeto apropriado.

Objetos que representam instâncias de módulos devem responder ainda a duas mensagens específicas, *do* e *done*, também parametrizadas por constantes identificadoras de portas. Estas mensagens sinalizam o início e término, respectivamente, do *comportamento*

interno observável do módulo quando no recebimento de estímulos na porta correspondente ao parâmetro da mensagem. Estas mensagens existem para permitir que a análise da arquitetura considere aspectos internos do componente relevantes a uma análise particular. Adiaremos para o Capítulo 4 maiores detalhes sobre estas mensagens. Objetos que representam instâncias de conectores, por sua vez, respondem às mensagens que lhe são direcionadas, estímulos em suas portas, de acordo com seu contrato de coordenação.

As mensagens *send* e *ack*, assim como as mensagens *do* e *done*, carregam a seqüência de identificadores de objetos e portas que compreendem o traço de sua interação, isto é, o caminho percorrido pela mensagem através dos objetos que representam uma instância da arquitetura, também chamada de *topologia*. A seqüência de interações é necessária para que a resposta de uma mensagem possa ser corretamente endereçada quando mais de uma instância de componente está ligada a uma mesma porta.

O conceito de interação, descrito informalmente acima, pode ser formalizado como uma *pilha* de pares onde a primeira projeção é um identificador de objetos e a segunda projeção um identificador de porta. Na teoria `CBABEL-CONFIGURATION` são declarados: o tipo de dados *Interaction*, o tipo de dados *PortId*, das constantes identificadoras de portas, e as mensagens *send*, *ack*, *do* e *done*.

A declaração de portas também gera a inclusão de regras de reescrita na teoria de reescrita associada ao componente. As regras determinam o comportamento das instâncias dos componentes quando no estímulo das portas. No restante desta seção, explicaremos como regras são geradas a partir da declaração de portas em *módulos*. Como em um *conector*, o tratamento das mensagens, que representam os estímulos das portas, é guiado pelo contrato de coordenação nele declarado. Adiaremos para a Seção 3.1.3 a explicação de como regras são geradas a partir da declaração de portas e contratos de coordenação nos conectores.

Existem quatro diferentes possíveis declarações de portas em um *módulo* CBabel, resultantes da combinação dos tipos de portas, entrada e saída, com o tipo de comunicação, síncrona ou assíncrona. Vejamos, para cada caso, quais as regras que devem ser adicionadas ao conjunto de regras da teoria de reescrita associada ao módulo CBabel:

- Quando uma porta de entrada síncrona é declarada, duas regras devem ser adicionadas. A primeira especifica que o recebimento de uma mensagem por aquela porta deve iniciar o comportamento interno do componente associado àquela porta. A segunda regra especifica que tão logo o comportamento interno do componente

termine, uma mensagem de resposta deve ser encaminhada para o componente que estimulou a porta.

O início do comportamento interno do módulo é representado pelo módulo enviando uma mensagem *do* para si mesmo. Ao término do comportamento interno, o módulo envia uma mensagem *done* para si mesmo.

- Quando uma porta de entrada assíncrona é declarada, uma regra deve ser adicionada. Esta regra especifica que o envio de uma mensagem para esta porta deve iniciar o comportamento interno do componente.

- A declaração de uma porta de saída síncrona deve adicionar duas regras. A primeira regra especifica que a execução do comportamento interno do módulo em relação a esta porta deve gerar a requisição de um serviço de outro componente por esta porta. Neste caso, a porta deve ser bloqueada até que a resposta da solicitação seja recebida. A segunda regra especifica exatamente o recebimento da resposta e desbloqueio da porta. O efeito de bloqueio e desbloqueio da porta é capturado pela atualização do valor de um atributo, associado ao *status* da porta.

- A declaração de uma porta de saída assíncrona adiciona uma regra. Esta regra especifica que a execução do comportamento interno do módulo deve gerar a requisição de um serviço de outro componente por esta porta. Neste caso, porém, a porta não precisa ficar bloqueada estando imediatamente disponível para envio de outra solicitação, visto que nenhuma resposta deve ser esperada.

Podemos agora formalizar as explicações supra. Antes, porém, vale reforçar que a declaração de uma porta em um *componente* CBabel, seja ele módulo ou conector, é mapeada da mesma forma na assinatura da teoria de reescrita associada ao componente. Apenas o *comportamento* da porta, mapeado para regras de reescrita, tem um mapeamento diferenciado quando em um módulo ou em um conector. No restante desta seção, formalizaremos como a declaração de uma porta em um componente afeta a assinatura da teoria de reescrita associada ao componente. Formalizaremos também as regras de reescrita geradas para portas em um módulo. Na Seção 3.1.3, mostraremos as regras de reescrita produzidas para portas em um conector.

Dada a declaração de um componente CBabel, um módulo (n, V, I, O) ou um conector (n, V, I, O, c), cada declaração de porta nos conjuntos I e O é uma tupla (m, T), onde m é o identificador da porta e T o tipo, podendo ser uma das constantes *sinc* ou *assinc*. Na assinatura Σ da teoria de reescrita associada ao componente CBabel devem ser incluídas:

- para cada declaração de porta i em I, uma constante i do tipo *PortInId*;
- para cada declaração de porta o em O, uma constante o do tipo *PortOutId*.

Os tipos *PortInId* e *PortOutId* são subtipos de *PortId*, o tipo que parametriza as mensagens:

$$send\ ,\ do\ ,\ done\ :\ Oid\ PortId\ \rightarrow\ Msg.$$

Estes tipos e as mensagens são declarados na teoria `CBABEL-CONFIGURATION` que Σ inclui.

Para formalização de como as portas declaradas em um módulo CBabel geram regras de reescrita, devemos considerar os quatro casos possíveis mostrados supra. Considerando a declaração de um módulo CBabel dada pela tupla (n, V, I, O) e sua teoria de reescrita associada $\mathcal{R} = (\Sigma, E, R)$, temos as seguintes regras adicionadas em R para cada declaração de porta em I e O:

- Porta síncrona $i \in I$ as regras 3.2 e 3.3:

$$rl\ send(\omega, i, \iota)\ <\omega\ :\ n\mid A>\Rightarrow\ do(\omega, i, \iota)\ <\omega\ :\ n\mid A>\ . \tag{3.2}$$

$$rl\ done(\omega, i, \iota)\ <\omega\ :\ n\mid A>\Rightarrow\ ack(\iota)<\omega\ :\ n\mid A>\ . \tag{3.3}$$

  onde ω é o identificador de um objeto que representa uma instância do módulo CBabel, ι é a interação e A o conjunto de atributos do objeto.

- Porta assíncrona $i \in I$ as regras 3.2 e 3.4:

$$rl\ done(\omega, i, \iota)\ <\omega\ :\ n\mid A>\Rightarrow<\omega\ :\ n\mid A>\ . \tag{3.4}$$

- Porta síncrona $o \in O$ as regras 3.5 e 3.6:

$$\begin{aligned} rl\ do(\omega, o, none)\ &<\omega\ :\ n\mid o\text{-}status\ :\ unlocked, A>\Rightarrow \\ &send(\omega, o, [\omega, o])\ <\omega\ :\ n\mid o\text{-}status\ :\ locked, A>. \end{aligned} \tag{3.5}$$

$$\begin{aligned} rl\ ack([\omega, o])\ &<\omega\ :n\mid o\text{-}status : s, A>\Rightarrow \\ &<\omega\ :\ n\mid o\text{-}status\ :\ unlocked, A> done(\omega, o, none)\ . \end{aligned} \tag{3.6}$$

  onde s é uma variável do tipo *PortStatus*, $o{-}status$ é uma "string" formada pela concatenação da "string" em o com a "string" (constante) "status". O tipo *PortStatus* é declarado na teoria de reescrita `CBABEL-CONFIGURATION`, onde também são declaradas as constantes:

$$locked\ ,\ unlocked\ :\rightarrow\ PortStatus$$

e a constante:

$$none \ : \rightarrow \ Interaction \ .$$

- Porta assíncrona $o \in O$ a regra 3.7:

$$rl \ do(\omega, o, none) \ < \omega \ : \ n \mid A > \Rightarrow \ send(\omega, o, [\omega, o]) \ < \omega \ : \ n \mid A > \ . \quad (3.7)$$

Em adição às portas explicitamente declaradas nos componentes, uma porta de saída especial, denominada `ground`, é implicitamente declarada em todos os componentes CBabel. A porta `ground` serve para, em um contrato de interação, finalizarmos o estímulo recebido pela porta de entrada sem encaminhá-lo para outros componentes. Na Seção 4.2, apresentamos um exemplo que utiliza esta porta. A semântica desta porta é definida pela constante identificadora de porta de saída

$$ground \ : \rightarrow \ PortOutId \ .$$

Esta constante é declarada na teoria de reescrita `CBABEL-CONFIGURATION`, juntamente com a equação 3.8, que remove da "sopa de objetos e mensagens" qualquer mensagem que represente estímulo para uma porta de saída `ground`.

$$eq \ send(\omega, ground, \iota) \ = \ none. \quad (3.8)$$

3.1.3 Contratos

Na versão original de CBabel, os contratos podem ser *interação* ou *coordenação*. Um contrato de interação especifica o caminho da interação dentro de um conector, ou seja, quando uma porta de entrada é estimulada para qual, ou quais, portas de saída o estímulo deve ser enviado. Um contrato de coordenação adiciona condições para este encaminhamento de estímulos. Contratos são declarados em conectores, não havendo restrição quanto ao número de contratos por conector.

No entanto, para simplificar a semântica dos conectores e contratos em lógica de reescrita, consideramos que: (i) cada conector declara apenas um contrato; (ii) os contratos de coordenação são declarados no contexto de uma interação, ou seja, entre portas de entrada e portas de saída; e, (iii) as portas relacionadas por um contrato de interação são compatíveis, ou seja, as portas de entrada e as portas de saída devem ser síncronas ou assíncronas.

Deve-se observar que o item (ii) é, em parte, resultado do item (i), pois a declaração de um contrato de coordenação isoladamente em um conector não teria grande valia. Isto porque o contrato de coordenação define, exatamente, "regras" para o encaminhamento dos estímulos recebidos pelas portas de entrada para as portas de saída. Um contrato de interação define exatamente o encaminhamento, isto é, para qual porta de saída deve ser encaminhado um estímulo de uma dada porta de entrada.

Considerando as decisões supra, propomos então uma sintaxe alternativa para declaração de contratos de coordenação que incorpora a especificação da interação. Como resultado, os seguintes contratos podem ser especificados em um conector: seqüencial, exclusão mútua ou guarda.

- O contrato seqüencial é definido sobre uma porta de entrada e uma ou mais portas de saída, especificando que quando a porta de entrada é estimulada, a(s) porta(s) de saída deve(m) ser também estimulada(s).
- O contrato de exclusão mútua é definido sobre interações, ou seja, pares de portas de entrada e saída, especificando que apenas uma das interações é possível por vez.
- O contrato de guarda, também definido sobre um par, porta de entrada e porta de saída, tem uma condição, um bloco de comando *before* e um bloco de comandos *after* e, opcionalmente, uma porta alternativa. Quando a porta de entrada é estimulada, se a condição for *verdadeira*, o bloco *before* será executado e o estímulo encaminhado para a porta de saída. Quando a resposta do estímulo da porta de saída é recebida, o bloco *after* é executado. Por outro lado, quando a porta de entrada é estimulada, se a condição for *falsa*, a requisição será encaminhada para a porta alternativa, caso esta tenha sido definida. Caso uma porta alternativa não tenha sido definida, a requisição é bloqueada e armazenada em uma fila até que a condição do guarda torne-se verdadeira.

A seguinte sintaxe é utilizada para declaração de um contrato em um conector:

⟨guard-decl⟩ →
guard (⟨exp⟩ **)**
| **guard (** ⟨exp⟩ **) {** ⟨guard-body⟩ **}**

⟨guard-body⟩ →
before { ⟨cmd-list⟩ } ⟨guard-body⟩

| **after** **{** ⟨cmd-list⟩ **}** ⟨guard-body⟩
| **alternative** **(** ⟨id⟩ **)** **;** ⟨guard-body⟩
| **none**

⟨port-exp⟩ → ⟨id⟩ | ⟨port-exp⟩ | ⟨port-exp⟩

⟨interaction⟩ → ⟨port-exp⟩ > ⟨port-exp⟩ **;**
⟨interaction-pair⟩ → ⟨id⟩ > ⟨id⟩ **;**
⟨interaction-pair-set⟩ →
⟨interaction-pair⟩ ⟨interaction-pair-set⟩
| **none**

⟨contract-decl⟩ →
interaction { ⟨interaction⟩ **} ;**
| **interaction {** ⟨id⟩ > ⟨guard-decl⟩ > ⟨id⟩ **} ;**
| **exclusive {** ⟨interaction-pair-set⟩ **} ;**

Antes de apresentarmos a semântica em lógica de reescrita para cada contrato, é conveniente descrevermos a intuição do mapeamento para cada contrato.

O contrato seqüencial entre uma porta de entrada e uma porta de saída é uma regra que reescreve a mensagem que representa o estímulo da porta de entrada para uma mensagem que representa o estímulo da porta de saída. No traço da interação da mensagem produzida também é adicionado um novo par formado pelo identificador do objeto, que representa a instância do conector, e o identificador da porta de saída. Se as portas forem síncronas, a chegada da resposta na porta de saída também é especificada por uma regra. Esta regra remove, na mensagem de resposta, o topo da pilha que contém o traço da interação e encaminha a mensagem de resposta para o objeto cujo identificador é a primeira projeção do novo topo da pilha do traço da interação.

Com as explicações acima, tratamos os estilos de interações *1:1* e *n:1*, isto é, quando existe um ligação entre uma porta de uma instância de componente e uma porta de uma instância de conector ou quando várias portas de diferentes instâncias de componentes estão ligadas a uma porta de uma instância de conector. Uma interação *1:n* será tratada por uma regra que reescreve uma mensagem que representa o estímulo da porta de entrada

para n mensagens representando o estímulo em cada uma das n portas de saída. Se as portas de saída forem síncronas, o tratamento das respostas será o mesmo explicado para a interação *1:1*, no entanto, todas as respostas das n portas de saída devem ser recebidas para que uma resposta seja passada adiante.

O contrato de exclusão mútua entre n interações tem a semântica de um semáforo. Este contrato é representado por um atributo *status* no objeto que representa a instância do conector e n regras de reescrita que são aplicadas de forma não determinística para escolha de qual das n interações irá *evoluir*. Quando uma destas regras é aplicada, uma das mensagens que representam os estímulos às portas de entrada das n interações é escolhida para ser reescrita. Se o atributo *status* estiver com valor *unlocked*, a mensagem será então reescrita para uma mensagem que representa o estímulo da porta de saída associada àquela porta de entrada, e o valor do atributo alterado para *locked*. Ou seja, todas as n regras não poderão mais ser aplicadas. A chegada de uma resposta no objeto que representa a instância do conector também é representada por uma regra que reescreve a mensagem resposta de forma semelhante ao explicado para o contrato de interação e também retorna o atributo *status* para o valor *unlocked*. O contrato de exclusão mútua só pode ser definido sobre interações com portas síncronas, caso contrário estaríamos definindo apenas o escalonamento dos tratamentos das mensagens pelo objeto que representa uma instância do conector. Um mapeamento diferente também poderia ser dado para o contrato de exclusão mútua. As portas de entrada do conector poderiam ser mapeadas para constantes que fossem subtipos de *PortId*, o que permitiria então a definição de apenas uma regra de reescrita para tratamento das mensagens que representam estímulos das portas de entrada.

O contrato de guarda é formalizado por três equações e duas regras de reescrita. Uma equação é o predicado que avalia a condição do guarda em relação aos valores dos atributos do objeto que representa a instância do conector. Duas outras equações representam os efeitos dos blocos *before* e *after* sobre o objeto que representa a instância do conector. Estas equações são, na realidade, composições de equações que representam o efeito de cada comando declarado nestes blocos. Uma regra especifica que quando a mensagem que representa o estímulo na porta de entrada é recebida, se a condição do guarda for avaliada para verdadeiro, a equação *before* será aplicada sobre o objeto que representa a instância do conector e a mensagem reescrita para uma mensagem que representa o estímulo na porta de saída do conector. De outro modo, se a condição do guarda for avaliada para falso, a mensagem simplesmente não é reescrita até que a condição torne-se verdadeira. Esta abordagem captura o efeito da mensagem ser armazenada até que o objeto possa

tratá-la, não sendo necessário a definição de um tipo de dados para fila de mensagens, ela simplesmente fica na "sopa" de objetos e mensagens. Quando uma mensagem de resposta é recebida pelo objeto, a segunda regra reescreve a resposta de forma análoga ao caso do contrato seqüencial, aplicando ainda a equação *after* sobre o objeto que representa a instância do conector. Caso o contrato de guarda tenha sido declarado com uma porta alternativa, uma terceira regra é definida para reescrever a mensagem que representa o estímulo da porta de entrada para a mensagem que representa o estímulo na porta alternativa quando a condição do guarda for falsa.

Estamos prontos então para formalizar cada um destes conceitos. Dada a declaração de um conector $C = (n, V, I, O, c)$ onde n é o identificador para o nome do conector, V o conjunto de declarações de variáveis, I o conjunto de declarações de portas de entrada, O o conjunto de declarações de portas de saída e c a declaração de um contrato. Sendo ainda $\mathcal{R} = (\Sigma, E, R)$ a teoria de reescrita gerada pelo mapeamento de C em lógica de reescrita, então:

- Se c é um par (i, o) onde $i \in I$ é uma porta de entrada e $o \in O$ é uma porta de saída, então c especifica um contrato seqüencial e sua semântica em lógica de reescrita é dada pela adição da regra 3.9 em R

$$rl\ send(\omega, i, \iota) < \omega\ :\ n \mid A > \Rightarrow\ send(\omega, o, [\omega, o] :: \iota) < \omega\ :\ n \mid A >\ . \qquad (3.9)$$

  onde os operadores

$$[\ \_\ ,\ \_\ ] : Oid\ PortOutId \rightarrow Interaction$$

  e

$$\_ :: \_ : Interaction\ Interaction \rightarrow Interaction$$

  são operadores construtores do tipo *Interaction*. Estes operadores, assim como o tipo *Interaction*, são declarados na teoria de reescrita `CBABEL-CONFIGURATION`. A variável w é um identificador do objeto que representa a instância do conector, ι a interação que a mensagem carrega e A é o conjunto de atributos do objeto. Se as portas $o \in O$ e $i \in I$ forem síncronas, a regra 3.10 também é adicionada em R.

$$rl\ ack([\omega, o] :: \iota) < \omega\ :\ n \mid A > \Rightarrow\ ack(\iota) < \omega\ :\ n \mid A >\ . \qquad (3.10)$$

- Se c é um par (i, Y) onde $i \in I$ é a porta de entrada e $Y \subseteq O$ é o conjunto de portas de saída, então c é uma variação do contrato seqüencial para o estilo *1:n* e

sua semântica é dada pelas regras 3.11 e 3.12, variações das regras 3.9 e 3.10, com $o_1, \ldots, o_n \in Y$.

$$\begin{aligned} &rl\ send(\omega, i, \iota) < \omega\ :\ n \mid A > \Rightarrow \\ &\quad send(\omega, o_1, [\omega, o_1] :: \iota)\ \ldots\ send(\omega, o_n, [\omega, o_n] :: \iota) < \omega\ :\ n \mid A >\ . \end{aligned} \tag{3.11}$$

$$\begin{aligned} &rl\ ack([\omega, o_1] :: \iota)\ \ldots\ ack([\omega, o_n] :: \iota) < \omega\ :\ n \mid A > \Rightarrow \\ &\quad ack(\iota) < \omega\ :\ n \mid A >\ . \end{aligned} \tag{3.12}$$

- Se c é dado por uma lista $L = [(i_1, o_1), \ldots, (i_n, o_n)]$ onde $i_1 \ldots i_n \in I$ e $o_1 \ldots o_n \in O$, então c corresponde a declaração de um contrato de exclusão mútua. A declaração de c adiciona a declaração da classe em Σ o atributo

$$status\ :\ PortStatus \rightarrow Attribute$$

sendo o tipo $PortStatus$ e duas constantes para este tipo

$$locked\ unlocked\ : \rightarrow\ PortStatus$$

declarados na teoria de reescrita `CBABEL-CONFIGURATION`. Para cada $(i_i, o_i), 1 \leq i \leq n$ de L, as regras 3.13 e 3.14 são adicionadas em R

$$\begin{aligned} &rl\ send(\omega, i_i, \iota) < \omega\ :\ n \mid status\ :\ unlocked, A > \Rightarrow \\ &\quad < \omega\ :\ n \mid status\ : locked, A > send(\omega, o_i, [\omega, o_i] :: \iota)\ . \end{aligned} \tag{3.13}$$

$$\begin{aligned} &rl\ ack([\omega, o_i] :: \iota) < \omega\ :\ n \mid status\ :\ locked, A > \Rightarrow \\ &\quad < \omega\ :\ n \mid status\ :\ unlocked, A > ack(\iota)\ . \end{aligned} \tag{3.14}$$

onde ω é o identificador do objeto que representa uma instância do conector CBabel, ι é a interação e A é o conjunto de atributos do objeto.

- Se c é a declaração de um contrato de guarda sem porta alternativa, então é dado por uma tupla (i, o, b, β, α) onde $i \in I$ é uma porta de entrada, $o \in O$ é uma porta de saída, b é uma expressão booleana, β e α seqüencias de comandos. É suficiente entendermos b, β e α como composições de funções. A expressão b é mapeada para uma função, composição de funções relacionadas a cada operação em b. Uma equação relaciona então esta função, significado de b, à função abstrata declarada

$$open?\ :\ Object\ \rightarrow\ Bool$$

declarada na teoria de reescrita `CBABEL-CONFIGURATION`. Da mesma forma, β e α

também são mapeadas para composições de funções. Duas equações relacionam então estas composições de funções as funções abstratas

$$before\ ,\ after\ :\ Object\ \rightarrow\ Object$$

declaradas em `CBABEL-CONFIGURATION`.

A semântica de c é dada então pela adição das regras 3.15 e 3.16 em R:

$$\begin{aligned} &crl\ send(\omega, i, \iota)\ <\omega : n \mid A> \Rightarrow \\ &\quad before(<\omega : n \mid A>)\ send(\omega, o, [\omega, o] :: \iota) \\ &\ if\ open?(<\omega : n \mid A>)\ . \end{aligned} \tag{3.15}$$

$$rl\ ack([\omega, o] :: \iota) <\omega : n \mid A> \Rightarrow\ after(<\omega : n \mid A>)\ ack(\iota)\ . \tag{3.16}$$

onde ω é o identificador do objeto que representa uma instância do conector CBabel, ι é a interação e A é o conjunto de atributos do objeto. [1]

- Se c é a declaração de um contrato de guarda com porta alternativa, então é dado por uma tupla $(i, o, o_a, b, \beta, \alpha)$ onde $i \in I$ é uma porta de entrada, $o \in O$ é uma porta de saída para a qual a requisição deve ser encaminhada se a expressão b da condição do guarda for avaliada com verdadeira, $o_a \in O$ é a porta alternativa para a qual a requisição será encaminhada se a avaliação de b for falsa e β e α, as seqüencias de comandos dos blocos *before* e *after* do guarda. Neste caso, a semântica de c é dada pelas regras 3.15, 3.16 e 3.17:

$$\begin{aligned} &crl\ send(\omega, i, \iota)\ <\omega : n \mid A> \Rightarrow \\ &\quad before(<\omega : n \mid A>)\ send(\omega, o_a, [\omega, o_a] :: \iota) \\ &\ if\ not\ open?(<\omega : n \mid A>)\ . \end{aligned} \tag{3.17}$$

onde ω é o identificador do objeto que representa uma instância do conector CBabel, ι é a interação e A é o conjunto de atributos do objeto.

3.1.4 Aplicação

Em toda arquitetura CBabel, um módulo especial chamado *módulo de aplicação* especifica como os componentes da arquitetura devem ser utilizados em conjunto. Este

[1] Deve-se observar que os identificadores de objetos são únicos no sistema, garantindo apenas um possível "casamento" de uma mensagem para um objeto. Da mesma forma, em cada componente CBabel, os nomes de portas são únicos. Isto evita, por exemplo, que outra regra reescreva a mensagem *send* que a Regra 3.15 deve reescrever.

módulo deve criar as instâncias dos componentes, ligar as portas e realizar as amarrações de variáveis de estado. Na versão original de CBabel, módulos podem ser declarados de forma aninhada, sendo o módulo mais externo considerado o módulo da aplicação, em nosso mapeamento, definimos uma sintaxe específica para o módulo de aplicação:

⟨module-application⟩ → **application** ⟨id⟩ **{** ⟨app-decls⟩ **}**
⟨app-decls⟩ → ⟨app-decl⟩ **;** ⟨app-decls⟩ | ⟨app-decl⟩
⟨app-decl⟩ →
instantiate ⟨id⟩ **as** ⟨id⟩
| **link** ⟨id⟩**.**⟨id⟩ **to** ⟨id⟩**.**⟨id⟩
| **bind** ⟨id⟩**.**⟨id⟩ **to** ⟨id⟩**.**⟨id⟩

Como nas seções anteriores, é conveniente apresentar a intuição do mapeamento antes de sua formalização. Declarações de instâncias são mapeadas para chamadas às operações construtoras, definidas em cada teoria de reescrita gerada para cada declaração de módulo ou conector. Uma configuração de objetos é então declarada para representar a topologia da arquitetura, ou seja, o conjunto das instâncias dos componentes da arquitetura. Para cada ligação entre uma porta de saída de um componente e uma porta de entrada de outro componente, uma equação é produzida. Esta equação reescreve equacionalmente a mensagem que representa o estímulo da porta de saída para uma mensagem que representa o estímulo da porta de entrada. Lembrando da Seção 2.3, onde vimos que em Maude as equações são aplicadas antes da regras de reescrita, e que é a cada aplicação de regra que o sistema *evolui* em sua computação. Operacionalmente, podemos dizer que com a utilização de uma equação tornamos *iguais* os estados da aplicação: (i) onde ocorre o estímulo da porta de saída e (ii) onde ocorre o estímulo da porta de entrada. Em outras palavras, a transição de um estado A, com o estímulo de uma porta de saída, para outro estado B, com estímulo a uma porta de entrada, onde estas portas são ligadas, ocorre instantaneamente e de forma não observável no grafo de estados da computação do sistema.

Formalmente, um módulo de aplicação CBabel é uma tupla (x, Y, L, B) onde:

- x é o nome do módulo;
- Y é um conjunto de declarações de instâncias de componentes, módulos ou conectores, definidas por uma tupla (ω, c) onde ω é um identificador que representa

uma instância de componente e c um identificador que representa um componente CBabel;

- L é um conjunto de declarações de links definidos por $(\omega_1, o, \omega_2, i)$, onde ω_1 e ω_2 são identificadores representando instâncias de c_1 e c_2; o é o identificador de uma porta de saída declarada em c_1 e i é o identificador de uma porta de entrada em c_2;
- B é um conjunto de declarações de amarrações de variáveis de estado. Na Seção 3.1.5 apresentaremos a formalização das declarações de amarrações de variáveis.

Um módulo de aplicação CBabel é mapeado em lógica de reescrita para uma teoria de reescrita $\mathcal{R} = (\Sigma, E, R)$. Em Σ é incluída a declaração da constante

$$topology \ : \rightarrow Configuration$$

e em E é incluída a semântica para a constante *topology* através da equação 3.18

$$eq\ topology = instantiate(\omega_1, c_1)\ \ldots\ instantiate(\omega_n, c_n)\ . \tag{3.18}$$

onde $n = |Y|$. A constante *topology* representa assim a topologia da arquitetura, uma configuração de objetos onde cada um representa uma instância em Y.

Cada declaração de ligação de portas em L gera a declaração de uma equação 3.19 em E.

$$eq\ send(\omega_1, o, \iota) = send(\omega_2, i, \iota)\ . \tag{3.19}$$

onde ω_1 e ω_2 são identificadores de objetos que representam instâncias de componentes conectados pelas suas portas o e i, respectivamente e ι representa a interação.

3.1.5 Variáveis de estado

Os componentes de uma arquitetura podem declarar dois tipos de variáveis: variáveis locais e variáveis de estado. Variáveis locais podem ser declaradas em módulos ou conectores, como vimos na Seção 3.1.1, enquanto variáveis de estado podem ser declaradas apenas em conectores. Variáveis de estado permitem a comunicação de um conector com outro componente da arquitetura por memória compartilhada, ou seja, se uma variável local (declarada um um módulo ou conector) é alterada, a variável de estado (declarada em outro conector) amarrada a esta variável local, imediatamente também tem seu valor alterado e vice-versa. A amarração de uma variável de estado a uma variável local é feita no módulo de aplicação, como vimos na Seção 3.1.4.

Na Seção 3.1.1 vimos a sintaxe para declaração de variáveis locais e de estado, e na Seção 3.1.4 a sintaxe para declaração de uma amarração de variáveis. Vamos agora apresentar a intuição do mapeamento das variáveis de estado e amarrações de variáveis em lógica de reescrita e, em seguida, sua formalização.

Uma variável de estado é mapeada para um par onde o primeira projeção é seu valor e a segunda projeção é sua situação, podendo ser *alterada* ou *inalterada*. Declarações de amarrações de variáveis no módulo de aplicação são mapeadas para equações que especificam a sincronização das variáveis amarradas. Lembrando que na Seção 2.3 vimos que Maude aplica as equações antes das regras de reescrita, desta forma, as variáveis de estado serão sempre sincronizadas antes que uma regra possa ser aplicada reescrevendo alguma mensagem.

Dada a declaração de um conector $\mathcal{C} = (n, V, I, O, c)$ e sua respectiva teoria de reescrita $\mathcal{R} = (\Sigma, E, R)$, a declaração de uma variável de estado é dada por um par $required(v, t) \in V$. A declaração de uma variável de estado é mapeada na declaração de um atributo na declaração da classe

$$class\ n \mid v\ :\ StateRequired\ .$$

na assinatura Σ. O tipo de dados *StateRequired* é declarado na teoria de reescrita `CBABEL-CONFIGURATION` incluída por Σ. Também na teoria de reescrita `CBABEL-CONFIGURATION` são declaradas as operações construtoras

$$st\ :\ T\ Status\ \rightarrow\ StateRequired\ .$$

para cada tipo primitivo T de CBabel. O tipo *Status*, por sua vez, tem como operações construtoras as constantes:

$$changed, unchanged\ :\rightarrow\ Status\ .$$

Dada a teoria de reescrita $\mathcal{R} = (\Sigma, E, R)$ resultante do mapeamento do módulo de aplicação (x, Y, L, B) em lógica de reescrita. A declaração de uma amarração de variáveis, $b \in B$, é definida por uma tupla $(\omega_1, v_1, \omega_2, v_2)$ onde ω_1 e ω_2 são identificadores representando instâncias dos componentes c_1 e c_2, respectivamente; $required(v_1, t) \in V_{c_1}$ e $(v_2, t) \in V_{c_2}$. As variáveis v_1 e v_2 são do tipo T sendo v_1 uma variável de estado em c_1 e v_2 uma variável local em c_2. A declaração de uma amarração de variáveis gera então as

equações 3.20 e 3.21 em E de $\mathcal{R}$

$$\begin{aligned} eq\ &< \omega_1\ :\ c_1 \mid v_1\ :\ st(V_1, changed)\ , S_1 > < \omega_2\ :\ c_2 \mid v_2\ :\ V_2\ , S_2 > = \\ &< \omega_1\ :\ c_1 \mid v_1\ :\ st(V_1, unchanged)\ , S_1 > < \omega_2\ :\ c_2 \mid v_2\ :\ V_1\ , S_2 > \ . \end{aligned} \tag{3.20}$$

$$\begin{aligned} ceq\ &< \omega_1\ :\ c_1 \mid v_1\ :\ st(V_1, unchanged)\ , S_1 > < \omega_2\ :\ c_2 \mid v_2\ :\ V_2\ , S_2 > = \\ &< \omega_1\ :\ c_1 \mid v_1\ :\ st(V_2, unchanged)\ , S_1 >< \omega_2\ :\ c_2 \mid v_2\ :\ V_2\ , S_2 > \\ &if\ \ V_1 \neq V_2\ . \end{aligned} \tag{3.21}$$

S_1 e S_2 representam o restante do conjunto dos atributos de c_1 e c_2, respectivamente.

3.2 Implementação

Maude CBabel tool é um protótipo de um ambiente para execução e análises de arquiteturas CBabel. Maude CBabel tool é a implementação em Maude da semântica de CBabel em lógica de reescrita, descrita na Seção 3.1, que estende Full Maude. Nesta seção, descreveremos como Full Maude foi estendido para Maude CBabel tool. Também descreveremos como módulos e conectores CBabel são transformados por Maude CBabel tool em módulos de sistema orientados a objetos de Full Maude. A utilização de Maude CBabel tool é o assunto do Capítulo 4.

Na Seção 2.3.5, mostramos como Maude pode ser utilizado como metalinguagem, criando um ambiente executável para uma linguagem L a partir dos recursos de meta-programação disponíveis no sistema Maude e do módulo `LOOP-MODE` [19]. Full Maude utiliza estes recursos para criar um ambiente executável para a *linguagem* Full Maude, uma extensão da linguagem Core Maude que define uma álgebra de módulos.

Dada uma descrição arquitetural em CBabel, Maude CBabel tool produz um módulo de sistema orientado a objetos para cada componente da arquitetura e carrega estes módulos no banco de dados de módulos do Full Maude. Na atual versão, a execução de simulações (reescritas) e a especificação das propriedades da arquitetura a serem analisadas são feitas com a sintaxe de Full Maude, diretamente na interface de comandos do Full Maude. Constitui parte de nosso trabalho futuro criar uma interface de comandos em Maude CBabel tool que entenda *componentes* e *portas* ao invés de *objetos* e *mensagens*.

A implementação de Maude CBabel tool segue as diretrizes apresentadas em [29] para extensão de Full Maude. Em linhas gerais:

- Estendemos a assinatura de Full Maude incluindo nela a definição da sintaxe de

CBabel. Isto foi feito com a declaração do módulo `CBABEL-SIGNATURE` que importa o módulo `FULL-MAUDE-SIGNATURE`.

- Definimos o tipo abstrato de dados `ComponentDecl` para representação das declarações de componentes de CBabel. O tipo `ComponentDecl` foi então definido como subtipo do tipo `Unit` de Full Maude, o tipo de dados dos elementos que podem ser armazenados no banco de dados de Full Maude. Isto foi feito com a declaração do módulo `CBABEL-UNIT` que importa o módulo `UNIT` de Full Maude.

- As funções de Full Maude que realizam o processamento e avaliação de declarações de novos módulos e teorias foram estendidas. O módulo `DATABASE-HANDLING`, onde são declaradas as regras para manipulação do banco de dados, foi estendido pelo módulo `EXT-DATABASE-HANDLING`.

- A transformação de componentes CBabel em módulos orientados a objetos foi implementada na função `cb2omod`, que transforma um elemento do tipo `ComponentDecl` em um elemento do tipo `StrOModule`, o tipo de dados para módulo orientado a objetos de Full Maude. Um termo do tipo `StrOModule` pode então ser inserido no banco de dados de módulos do Full Maude e transformado em um módulo de sistema conforme descrito em [25, 38, 35, 29].

- Finalmente, redefinimos o processo de entrada e saída de Full Maude com a definição de um novo estado inicial para o banco de dados de Full Maude e a preparação do ambiente para aceitar a entrada de declarações CBabel na interface de comandos.

Nas seções seguintes, apresentamos uma visão geral de cada uma destas etapas.

3.2.1 A sintaxe de CBabel

Definimos a sintaxe de CBabel algebricamente, como Goguen em [48]. Maude oferece grande flexibilidade para definição da sintaxe de operadores, que podem ser pré-fixados, pós-fixados, in-fixados ou a combinação de ambos ("mixfix"). Fazendo uso deste recurso, a sintaxe de CBabel pôde ser facilmente especificada no módulo funcional `CBABEL-SIGNATURE`. Para exemplificar como a sintaxe de CBabel é definida, mostramos adiante alguns trechos deste módulo.

```
fmod CBABEL-SIGNATURE is
 inc FULL-MAUDE-SIGN .
 ...
```

```
 sorts eVarDecl eVarType .
 ops int bool : -> eVarType .
 op var_ _;   : eVarType eId -> eVarDecl .
 op var_ _=_; : eVarType eId eExp -> eVarDecl .
 op staterequired_ _; : eVarType eId -> eVarDecl .
 ...
 subsorts eVarDecl ePortDecl < eElement .
 subsort eElement < eElementSet .
 ...
 op mt-element : -> eElementSet .
 op __ : eElementSet eElementSet -> eElementSet
                                    [assoc comm id: mt-element prec 50] .
 subsorts eConnectorDecl eModuleDecl < eComponentDecl .
 op module_{_} : eId eElementSet -> eModuleDecl .
 op application_{_} : eId eElementSet -> eModuleDecl .
 op connector_{_} : eId eElementSet -> eConnectorDecl .
 ...
 subsort eComponentDecl < Input .
 ...
endfm
```

A sintaxe para declaração de variáveis é descrita pelas operações:

```
 op var_ _;   : eVarType eId -> eVarDecl .
 op var_ _=_; : eVarType eId eExp -> eVarDecl .
 op staterequired_ _; : eVarType eId -> eVarDecl .
```

Estas operações constroem termos do tipo `eVarDecl`, subtipo de `eElementSet`.

Em seguida são apresentados os operadores construtores de nível mais alto para termos que representam a declaração de módulos, conectores e módulos de aplicação. Por exemplo, um termo que representa a declaração de um módulo CBabel é construído pela operação `module_{_}`.

O módulo `CBABEL-SIGNATURE` estende o módulo `FULL-MAUDE-SIGN`, que define a sintaxe de Full Maude para comandos e declarações de módulos. Ao tornarmos o tipo `eComponentDecl` subtipo de `Input`, tornamos declarações de componentes CBabel entradas válidas na interface de comandos de Full Maude.

A técnica que Full Maude utiliza para processamento das entradas é descrita em [29]. Vamos aqui abordar apenas os aspectos relevantes para a implementação de CBabel. Para executar a análise sintática de uma entrada, utilizando a função `metaParse` do módulo `META-LEVEL`, Full Maude precisa fornecer a meta-representação de uma assinatura que defina a gramática a ser utilizada na análise sintática. A assinatura de Full Maude é definida no módulo `FULL-MAUDE-SIGN`. No módulo `META-FULL-MAUDE-SIGN` é então definida a constante `GRAMMAR`, do tipo `FModule`, que é atribuída ao módulo `GRAMMAR` (definido em meta-nível). O

módulo `GRAMMAR`, por sua vez, importa o módulo `FULL-MAUDE-SIGN`. A constante `GRAMMAR` é então utilizada nas chamadas da função `metaParse` para a análise sintática das entradas.

Para extensão de Full Maude, definimos então o módulo `EXT-META-FULL-MAUDE-SIGN`, que estende o módulo `META-FULL-MAUDE-SIGN` definindo a nova constante `CB-GRAMMAR` para ser usada pela função `metaParse` na análise sintática das entradas. A constante `CB-GRAMMAR` é equacionalmente atribuída à meta-representação do módulo `GRAMMAR` alterado com a inclusão da importação do módulo `CBABEL-SIGNATURE`.

```
fmod EXT-META-FULL-MAUDE-SIGN is
 pr META-FULL-MAUDE-SIGN .
 pr UNIT .

 op CB-GRAMMAR : -> FModule .
 eq CB-GRAMMAR =
    addImports((including 'CBABEL-SIGNATURE .), GRAMMAR) .
endfm
```

Deve-se observar que o módulo `GRAMMAR` não declara o tipo de dados em que representaremos as entradas. Da chamada da função `metaParse` obtemos um termo que representa a árvore sintática da entrada (Seção 3.2.5). Este termo será então transformado em um termo do tipo de dados `ComponentDecl`, como veremos na Seção 3.2.2.

3.2.2 O tipo abstrato de dados `ComponentDecl`

Em Full Maude o tipo de dados `Unit`, declarado no módulo `UNIT`, é utilizado para representar todos os possíveis módulos que podem ser armazenados no banco de dados de módulos de Full Maude: módulos e teorias de sistema, módulos e teorias funcionais e módulos e teorias orientadas a objetos.

No decorrer da Seção 3.1, cada construção sintática de CBabel foi formalizada por uma representação matemática, a partir da qual, o mapeamento para os elementos de lógica de reescrita foi definido. O tipo de dados `ComponentDecl`, declarado no módulo funcional `CBABEL-UNIT`, corresponde a implementação desta formalização matemática em Maude (ou seja, a especificação algébrica do tipo de dados `ComponentDecl`). A descrição completa do módulo `CBABEL-UNIT` é apresentada no Apêndice A. Para cada construção CBabel um tipo de dados e operações construtoras são definidos. Para exemplificar, a declaração de uma variável, definida na Seção 3.1.1 por uma tupla (v, l, t), é implementada pelo tipo `VariableDecl` e pelas operações construtoras:

```
ops Boolean Integer : -> VarType .
op local : Id VarType -> VariableDecl .
op local : Id VarType Exp -> VariableDecl .
```

Onde as constantes `Boolean` e `Integer` do tipo `VarType` correspondem aos possíveis valores de t. O tipo `Exp` representa uma expressão CBabel que pode ser atribuída ao valor inicial l da variável e o tipo `Id` representa os identificadores válidos de CBabel, correspondendo a v. Uma tupla (v, l, t) pode então ser representada por um termo construído pela operação `local`.

Em adição às operações construtoras para cada construção de Maude CBabel tool, as seguintes funções auxiliares foram definidas no módulo `CBABEL-UNIT`:

- Redefinição da funções `setName` e `getName` definidas no módulo `UNIT` para troca e seleção do nome de um componente.
- Funções para adição de novas declarações no conjunto de declarações já existentes em um componente:

```
op addPort : PortDecl ComponentDecl -> ComponentDecl .
op addVar : VariableDecl ComponentDecl -> ComponentDecl .
op addContract : Contract ComponentDecl -> ComponentDecl .
op addInstance : InstanceDecl ComponentDecl -> ComponentDecl .
op addLink : LinkDecl ComponentDecl -> ComponentDecl .
op addBind : BindDecl ComponentDecl -> ComponentDecl .
```

- Constantes para criação de componentes vazios. Por exemplo, a função `emptyModule` retorna uma declaração de módulo CBabel vazia.

```
op emptyConnector : -> ConnectorDecl .
op emptyModule : -> ModuleDecl .
op emptyApplication : -> ModuleDecl .
```

3.2.3 Processamento das entradas

Em Full Maude, o estado do sistema é mantido em um objeto que contém o banco de dados de unidades (módulos e teorias) e visões carregadas no sistema. Este objeto tem um atributo `db`, que efetivamente armazena o banco de dados; um atributo `default`, que identifica a unidade corrente, dentre as unidades armazenadas; e os atributos `input` e `output`.

```
class database | db : Database , input : Term ,
                 output : QidList, default : ModId .
```

O processamento da entrada corresponde a etapa em que um termo, do tipo `Term`, armazenado no atributo `input`, é processado. O processamento do termo pode corresponder a execução de um comando ou a importação de uma nova unidade para o banco de dados. Em ambos os casos, uma saída pode ser produzida e, neste caso, armazenada no atributo `output`. Nesta seção, iremos descrever como o processamento das entradas foi estendido em Maude CBabel tool. Adiamos para a Seção 3.2.5 a discussão de como é feita a análise sintática de uma declaração CBabel e como a árvore sintática resultante, um termo do tipo `Term`, é armazenada no atributo `input`. A descrição completa dos processos apresentados aqui pode ser obtida em [29].

A classe `database` é declarada no módulo `DATABASE-HANDLING`. Neste módulo também são declaradas as regras de reescrita que descrevem o comportamento do objeto para tratamento das entradas e geração de saídas. O módulo `DATABASE-HANDLING`, em Maude CBabel tool, foi estendido pelo módulo `EXT-DATABASE-HANDLING`, que importa o módulo `INITIAL-DB` e o módulo `CBABEL-PROCESSING`. No módulo `INITIAL-DB`, modificamos a forma como o estado inicial do banco de dados de Full Maude é definida, aumentando dramaticamente o desempenho do ambiente. [2] O módulo `CBABEL-PROCESSING` será explicado a seguir.

```
mod EXT-DATABASE-HANDLING is
 inc DATABASE-HANDLING .
 inc INITIAL-DB .
 pr CBABEL-PROCESSING .
 ...
 crl [module] :
   < O : X@Database | db : DB,  input : (F[T, T']),
        output : nil, default : MN, Atts >
   =>
   < O : X@Database | db : procUnit(F[T, T'], DB), input : nilTermList,
     output :
     ('Introduced 'CBabel 'component modNameToQid(parseModName(T)) '\n),
     default : parseModName(T), Atts >
   if (F == 'module_'{_'}) or-else
      (F == 'connector_'{_'}) or-else
      (F == 'application_'{_'}) .
endm
```

A única regra declarada em `EXT-DATABASE-HANDLING` determina que quando a entrada, um termo no atributo `input`, é reconhecida como a árvore sintática de uma declaração CBabel, o banco de dados deve ser modificado com o resultado da função `ProcUnit`, definida no módulo `CBABEL-PROCESSING`, para a qual é passado o termo na entrada. Além disso, no atributo `output`, é produzida uma saída informando ao usuário que o ambiente reconheceu a entrada como uma declaração CBabel e que esta foi introduzida no banco de dados.

[2] Corrigido em Full Maude 2.1.2.

Deve-se observar que Maude CBabel tool é uma extensão conservativa de Full Maude, isto porque todas as funcionalidades de Full Maude permanecem disponíveis em Maude CBabel tool. Destacamos, por exemplo, que nenhuma modificação foi feita na classe `database`. Além disso, as demais regras para tratamento de comandos e declarações de Full Maude permanecem disponíveis, pois o módulo `EXT-DATABASE-HANDLING` inclui o módulo `DATABASE-HANDLING`.

Em Full Maude, o processamento de um termo resultante da análise sintática de alguma entrada é feito pela função `procUnit`, declarada no módulo `UNIT-PROCESSING`. Esta função recebe como argumento um termo do tipo `Term` e o banco de dados. A função então insere no banco de dados o resultado da transformação de tal termo em um terno do tipo `Unit`. No entanto, o processamento é feito em algumas etapas.

Como Full Maude foi concebido para ser extensível como ambiente de execução para qualquer linguagem L, a função `procUnit` corresponde, na realidade, a uma composição de funções que devem tratar determinadas particularidades. Por exemplo, se em $\mathcal{L}$ a sintaxe pode ser definida pelo usuário (i.e., Maude), módulos de $\mathcal{L}$ podem conter "bubbles". [3] Em linhas gerais, a função `procUnit` em Full Maude corresponde a composição

$$procUnit = procUnit2 \rightarrow procUnit3 \rightarrow procUnit4 \ .$$

Sendo CBabel uma linguagem declarativa sem construções para definições sintáticas, ou seja, ao contrário de Maude por exemplo, que permite a definição de linguagens pelo usuário, apenas as funções `procUnit2`, `procUnit3` e `procUnit4` precisaram ser estendidas em Maude CBabel tool.

As funções `procUnit2` e `procUnit3` foram sobrecarregadas para entenderem declarações CBabel. A função `procUnit2`, ao receber um termo do tipo `Term`, que corresponda a declaração de uma nova unidade, apenas separa o nome da unidade do seu corpo (demais declarações internas), e então chama `procUnit3` passando ainda uma unidade vazia do tipo correto. A função `procUnit3` computa o nome do módulo, atribuindo o nome correto à unidade vazia que foi passada por `procUnit2` e chama `procCbUnit4`.

A função `procUnit4` de Full Maude analisa declaração por declaração da entrada preenchendo o termo do tipo `Unit` que foi recebido de `procUnit3` vazio. A função `procUnit4` utiliza a função auxiliar `parseDecl` para a análise da entrada de forma recursiva e inclusão das declarações apropriadas no termo do tipo `Unit`. Esta função foi substituída

[3] Para lidar com particularidades léxicas de linguagens, Maude suporta a definição de tipos `bubble`, que capturam a noção de "tokens" e identificadores da linguagem em questão.

pela função `procCbUnit4`. Como alguns parâmetros de `procUnit4` não são necessários para o processamento das declarações CBabel, a função `procUnit4` foi substituída por `procCbUnit4`, ao invés de sobrecarregada, como feito para as demais funções `procUnit2` e `procUnit3`. A substituição ao invés da sobrecarga está também relacionada a outros detalhes de funcionamento de `procUnit4` que precisariam ser adaptados, mesmo que desnecessários, para funcionarem em Maude CBabel tool.

```
fmod CBABEL-PROCESSING is
 pr UNIT-PROCESSING .
 pr CBABEL-DECL-PARSING .
 pr CBABEL-EVALUATION .
 ...
 eq procUnit2(T, 'module_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyModule, DB) .
 eq procUnit2(T, 'connector_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyConnector, DB) .
 eq procUnit2(T, 'application_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyApplication, DB) .

 ceq procUnit3(T, 'cbtoken[T'], T'', U, DB)
    = procCbUnit4(T, T'', setName(U, QI), DB)
  if QI := downQid(T') .

 op procCbUnit4 : Term Term Unit Database -> Database .

 ceq procCbUnit4(T, '__[T', T''], PU, DB)
    = procCbUnit4(T, T'', preUnit(PDR), DB)
  if PDR := parseDecl(T', PU, PU, none) .

 ceq procCbUnit4(T, F[TL], PU, DB)
    = evalCbUnit(preUnit(PDR), insertTermUnit(getName(PU), T, DB))
  if F =/= '__  /\
     PDR := parseDecl(F[TL], PU, PU, none) .

endfm
```

Detalhes sobre as assinaturas das funções `procUnit2`, `procUnit3` e `procUnit4` podem ser obtidos em [29]. Aqui eles foram omitidos para simplificar a explicação.

No módulo `CBABEL-DECL-PARSING` a função `parseDecl` é especializada para realizar a análise de declarações CBabel e "preencher" um termo do tipo `ComponentDecl` da forma adequada. Para isto, são usadas as funções auxiliares definidas no módulo `CBABEL-UNIT` (Seção 3.2.2). Para exemplificar, mostramos adiante o trecho do módulo onde a declaração de variáveis é analisada.

```
fmod CBABEL-DECL-PARSING is
 pr UNIT-DECL-PARSING .
```

```
 pr EXT-META-FULL-MAUDE-SIGN .
 pr CBABEL-UNIT .
 ...
 eq parseDecl('var__;[T1, T2], PU, U, VDS)
    = < addVar(local(parseId(T2), parseVarType(T1)), PU) ; U ; VDS > .
 eq parseDecl('staterequired__;[T1, T2], PU, U, VDS)
    = < addVar(required(parseId(T2), parseVarType(T1)), PU) ; U ; VDS > .
 eq parseDecl('var__=_;[T1, T2, T3], PU, U, VDS)
    = < addVar(local(parseId(T2), parseVarType(T1), parseExp(T3)), PU) ;
        U ; VDS > .
 ...
endfm
```

Quanto todo o termo da entrada é processado e, conseqüentemente, o termo do tipo `ComponentDecl` construído, a função `procCbUnit4` chama a função `evalCbUnit`. A função `evalCbUnit` recebe então como primeiro argumento um termo do tipo `ComponentDecl` que corresponde a declaração de um módulo CBabel e o banco de dados modificado pela função `insertTermUnit`.

No início desta seção, vimos que o objeto que constitui o estado do ambiente Full Maude contém um atributo chamado `db`, para armazenar um termo do tipo `Database`. O banco de dados armazenado o atributo `db` é declarado no módulo `DATABASE`. Para nosso propósito, é suficiente entendermos que cada entrada neste banco de dados, que corresponda a uma declaração de uma nova unidade, é do tipo `UnitInfo`, no qual é armazenada a informação sobre a nova unidade. Para cada unidade são armazenadas as seguintes informações principais:

- a forma original, como introduzida no sistema pelo usuário;
- a representação interna da unidade, onde variáveis são renomeadas para evitar conflito com o nome de outras variáveis de outras unidades na mesma hierarquia. Para unidades orientadas a objetos, é armazenada a unidade de sistema equivalente, isto é, o resultado da transformação desta em um módulo de sistema;
- a assinatura da unidade, dada por um módulo funcional do tipo `FModule`;
- a versão não estrutura da unidade, onde todas as importações de outras unidades forma resolvidas, isto é, as declarações nas unidades importadas são incorporadas à unidade.

A função `insertTermUnit`, chamada pela função `proctUnit4`, modifica o banco de dados com a criação de uma nova entrada do tipo `UnitInfo` no banco de dados. Esta nova

entrada irá armazenar as informações do módulo sendo inserido já contendo em um de seus argumentos o termo original recebido como entrada.

Finalmente, a função `evalCbUnit`, declarada no módulo `CBABEL-EVALUATION`, transforma o módulo CBabel inserido em um módulo orientado a objetos de Full Maude, utilizando a função `cb2omod`.

```
fmod CBABEL-EVALUATION is
 pr EVALUATION .
 pr CBABEL-CONVERTER .
 ...
 op evalCbUnit : Unit Database -> Database .

 ceq evalCbUnit(U, DB) = evalUnit(U', none, DB)
  if  U : ComponentDecl /\
     MN := getName(U) /\
     U' := cb2omod(setName(emptyStrOModule, MN), U) .

endfm
```

O restante do processamento da entrada não foi modificado. O módulo orientado a objetos resultante de `cb2omod` é passado para a função `evalUnit` que deverá: (i) transformar o módulo orientado a objetos em módulo de sistema [29]; e, (ii) preencher o restante da nova entrada `UnitInfo` do banco de dados.

3.2.4 Transformação de módulos CBabel em módulos orientados a objetos de Full Maude

Nesta seção, apresentamos a função `cb2omod`, a implementação em Maude do mapeamento das construções de CBabel para lógica de reescrita, apresentado na Seção 3.1. A função `cb2omod` é declarada no módulo `CBABEL-CONVERTER`:

```
op cb2omod : StrOModule ComponentDecl -> StrOModule .
```

A função recebe como primeiro argumento um termo do tipo `StrOModule` de Full Maude, com uma declaração de um módulo orientado a objetos vazio (sem declarações de operações, variáveis, classes etc.) e como segundo argumento um termo do tipo `ComponentDecl`, correspondendo a declaração de um módulo CBabel.

A implementação de `cb2omod` é bastante simples. Basicamente, utiliza funções auxiliares para transformação de cada construção CBabel em declarações de um módulo orientado a objetos de Full Maude [4]:

[4] O tipo `StrOModule` é subtipo de `Unit` [29].

```
op   vars2omod : Unit VariableDeclSet -> Unit .
op ports2omod1 : Unit PortDeclSet -> Unit .
op ports2omod2 : Unit PortDeclSet -> Unit .
op  contract2omod : Unit Contract -> Unit .
op instances2omod : Unit ConfigDeclSet -> Unit .
op links2omod : Unit ConfigDeclSet -> Unit .
op binds2omod : Unit ConfigDeclSet ConfigDeclSet -> Unit .
```

E estas, as funções auxiliares do tipo de dados `Unit` para preencher o termo `StrOModule`:

```
op addImports : List<EImport> Unit -> Unit .
op addSorts : Set<ESort> Unit -> Unit .
op addSubsorts : [Set<ESubsortDecl>] Unit -> Unit .
op addOps : [Set<EOpDecl>] Unit -> Unit .
op addMbs : Set<EMembAx> Unit -> Unit .
op addEqs : Set<EEquation> Unit -> Unit .
op addRls : Set<ERule> Unit -> Unit .
op addClasses : ClassDeclSet Unit -> Unit .
op addSubclasses : SubclassDeclSet Unit -> Unit .
op addMsgs : Set<MsgDecl> Unit -> Unit .
```

A implementação completa de Maude CBabel tool encontra-se no Apêndice A. Para exemplificar, mostramos abaixo implementação do mapeamento de portas em um módulo CBabel (Seção 3.1.2).

Como portas declaradas em conectores têm um mapeamento diferente de portas declaradas em módulos, existem duas funções para mapeamento de portas:

```
op ports2omod1 : Unit PortDeclSet -> Unit .
op ports2omod2 : Unit PortDeclSet -> Unit .
```

Ambas recebem como primeiro argumento um termo do tipo `Unit` que corresponde a declaração de unidade, o módulo orientado a objetos sendo construído a partir das declarações em CBabel, e como segundo argumento um termo do tipo `PortDeclSet`, que corresponde ao conjunto de declarações de portas. A função `ports2omod2` é utilizada para o mapeamento de portas declaradas em um conector e a função `ports2omod1` para portas declaradas em um módulo.

Para portas declaradas em um conector, apenas uma constante do tipo apropriado é declarada na teoria de reescrita resultante. A função `ports2omod2` percorre o conjunto das declarações de portas do conector (capturado pela variável `PDS` do tipo `PortDeclSet`) identificando portas de entrada, `in(I,PT)`, e portas de saída, `out(I,PT)`. Ao identificar o tipo da porta, a função `addOps`, declarada no módulo `UNIT` de Full Maude, é chamada

para incluir uma declaração de operação (constante) do tipo `PortInId` ou `PortOutId` no módulo orientado a objetos que está sendo construído. Cada declaração de porta tratada é removida do conjunto de declarações de porta. Após todas as declarações de portas terem sido tratadas, o segundo argumento terá a constante `mt-port`, e a função retornará o módulo orientado a objetos devidamente alterado.

```
ceq ports2omod2(U, in(I, PT) PDS) = ports2omod2(U', PDS)
 if U' := addOps((op qidId(I) : nil -> 'PortInId [ctor] .), U) .

ceq ports2omod2(U, out(I, PT) PDS) = ports2omod2(U', PDS)
 if U' := addOps((op qidId(I) : nil -> 'PortOutId [ctor] .), U) .

eq ports2omod2(U, mt-port) = U .
```

A função `ports2omod1` é utilizada para o mapeamento de portas declaradas em um módulo. Em adição a função `ports2omod2`, a função `ports2omod1`, utiliza a função auxiliar `prepPortRls` para geração das regras de reescrita que devem ser criadas no módulo orientado a objetos para cada tipo de porta declarado.

```
ceq ports2omod1(U, in(I, PT) PDS)
    = ports2omod1(addRls(RLS, U'), PDS)
 if U'  := addOps((op qidId(I) : nil -> 'PortInId [ctor] .), U) /\
    RLS := prepPortRls(getName(U), in(I, PT)) .

ceq ports2omod1(U, out(I, PT) PDS)
    = ports2omod1(addRls(RLS, U'), PDS)
 if U'  := addOps((op qidId(I) : nil -> 'PortOutId [ctor] .), U) /\
    RLS := prepPortRls(getName(U), out(I, PT)) .

eq ports2omod1(U, mt-port) = U .
```

Cada um dos quatro tipos de declaração de portas em módulos, apresentados na Seção 3.1.2, é considerado pela função `prepPortRls`. No código a seguir, por exemplo, a primeira equação especifica que para uma declaração de porta de entrada síncrona, duas regras de reescrita, instâncias das regras 3.2 e 3.3 (Seção 3.1.2) devem ser retornadas pela função `prepPortRls`. As equações restantes consideram os demais tipos de declarações de portas em módulos.

```
op prepPortRls : ModName PortDecl -> Set<ERule> .

ceq prepPortRls(MN, in(I', sinc)) =
   (rl
     '__[T, 'send[ov, portInC(I'), itv]] => '__[T, 'do[ov, portInC(I'), itv]]
    [labelRule(MN, "recevingAndDo", I')] .)
```

```
   (rl
     '__[T, 'done[ov, portInC(I'), itv]] => '__[T, 'ack[itv]]
    [labelRule(MN, "doneAndAcking", I')] .)
 if T := objt(MN) .

ceq prepPortRls(MN, in(I', assinc)) =
   (rl
     '__[T, 'send[ov, portInC(I'), itv]] => '__[T, 'do[ov, portInC(I'), itv]]
    [labelRule(MN, "receivingAndDo", I')] .)
   (rl
     '__[T, 'done[ov, portInC(I'), itv]] => T  [labelRule(MN,  "done", I')] .)
 if T := objt(MN) .

ceq prepPortRls(MN, out(I', sinc)) =
   (rl
     '__['do[ov, portOutC(I'), 'none.Interaction],
         objt(MN, Q['unlocked.PortStatus])] =>
     '__['send[ov, portOutC(I'), ''[_',_'][ov, portOutC(I')]],
         objt(MN, Q['locked.PortStatus])]
    [labelRule(MN, "sending", I')] .)
   (rl
     '__['ack[''[_',_'][ov, portOutC(I')]],
         objt(MN, Q['locked.PortStatus])] =>
     '__['done[ov, portOutC(I'), 'none.Interaction],
         objt(MN, Q['unlocked.PortStatus])]
    [labelRule(MN,  "receivingAck", I')] .)
  if Q := qid(strId(I') + "-status :_") .

ceq prepPortRls(MN, out(I', assinc)) =
   (rl '__[T, 'do[ov, portOutC(I'), 'none.Interaction]] =>
       '__[T, 'send[ov, portOutC(I'), ''[_',_'][ov, portOutC(I')]]]
    [labelRule(MN, "sending", I')] .)
   (rl
     '__[T, 'ack[''[_',_'][ov, portOutC(I')]]] => T
    [labelRule(MN,  "receivingAck", I')] .)
 if T := objt(MN) .
```

As demais declarações nos módulos e conectores são transformadas de forma similar.

3.2.5 Entrada e saída

Na Seção 2.3.5, mostramos como Maude pode ser utilizado como metalinguagem, criando um ambiente executável para uma linguagem L. Como vimos, isto é possível a partir dos recursos de meta-programação disponíveis no sistema Maude e do módulo `LOOP-MODE` [19]. O módulo `LOOP-MODE` implementa os recursos para persistência de estado e entrada e saída com a declaração de um operador especial, `[_,_,_]`, que pode ser visto como um objeto persistente com um canal de entrada (primeiro argumento), um canal de saída (último argumento) e um estado (segundo argumento). O funcionamento do mecanismo de

loop pode ser basicamente descrito como se segue. Para maiores informações indicamos o manual de Maude [19]. Quando um módulo que estenda `LOOP-MODE` é carregado em Maude, primeiramente precisamos criar uma instância do operador `[_,_,_]`, um objeto *loop*, com a execução do comando especial `loop`. Em seguida, qualquer entrada na interface de comandos de Maude, passada entre parênteses, é convertida em uma seqüência de "tokens" que são representados como QIDs ("quoted identifiers"). A lista de QIDs é então colocada no primeiro argumento do objeto loop. A saída é tratada da forma inversa, qualquer lista de QIDs armazenada no terceiro argumento do objeto loop é convertida em uma lista de "tokens" e impressa no terminal. Estes eventos de entrada e saída podem ser entendidos como reescritas implícitas, que transferem listas de QIDs entre o terminal e o objeto loop.

Para o tratamento das entradas e saídas, o módulo `CBABEL-TOOL` estende o módulo `LOOP-MODE`, substituindo o módulo `FULL-MAUDE`. No módulo `CBABEL-TOOL` define e inicializa o estado persistente e especifica a comunicação entre o objeto *loop* (a entrada e saída do sistema) e o banco de dados.

```
mod CBABEL-TOOL is
  pr EXT-DATABASE-HANDLING .
  pr EXT-META-FULL-MAUDE-SIGN .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  op o : -> Oid .
```

Como vimos na Seção 3.2.3, o estado do sistema é representado por um único objeto da classe `database`, o objeto que mantém o estado do ambiente Full Maude:

```
  subsort Object < State .
```

A constante `init` define o estado inicial do objeto *loop*:

```
  op init : -> System .

  rl [init] : init =>
    [nil,
     < o : Database | db : initial-db, input : nilTermList, output : nil,
                      default : 'CONVERSION >,
     ('\n '\t '\s '\s '\s '\s '\s '\s '\s
      'Cbabel 'Tool '2.4 '\s '`( 'February '14th '`, '\s '2005 '`) '\n)] .
```

Quando algum texto é introduzido no objeto *loop*, o primeiro argumento do operador `[_,_,_]` é diferente de `nil` e a regra a seguir é executada. Uma entrada pode ser um

comando de Full Maude ou uma declaração de Full Maude ou CBabel. A constante `CB-GRAMMAR` (Seção 3.2.1) é utilizada pela meta-função `metaParse` para a análise sintática da entrada.

```
crl [in] :
  [QIL, < O : X@Database | input : nilTermList, Atts >, QIL'] =>
  [nil, < O : X@Database |
              input : getTerm(metaParse(CB-GRAMMAR, QIL, 'Input)), Atts >,
   QIL']
 if QIL =/= nil
    /\ metaParse(CB-GRAMMAR, QIL, 'Input) : ResultPair .
```

A regra acima trata o caso da entrada fornecida ser sintaticamente válida, [5] quando o termo que representa a árvore sintática da entrada é então colocado no atributo `input` do banco de dados. Caso a entrada seja inválida, a regra a seguir é então executada, colocando uma mensagem de erro no último argumento do objeto `loop` (a saída):

```
crl [in] :
  [QIL, < O : X@Database | Atts, output : nil >, QIL'] =>
  [nil,
        < O : X@Database | Atts,
         output : ('\r 'Warning:
                   printSyntaxError(metaParse(CB-GRAMMAR, QIL, 'Input), QIL)
                   '\n
                   '\r 'Error: '\o 'No 'parse 'for 'input. '\n) >,
   QIL']
  if QIL =/= nil
     /\ not metaParse(CB-GRAMMAR, QIL, 'Input) :: ResultPair .
```

Quando o atributo `output` do objeto que mantém o estado de Full Maude contém uma lista não vazia de QIDs, a regra de reescrita condicional `out` move esta lista de QIDs para o terceiro argumento do objeto `loop`. Então o sistema Maude imprime a lista de QIDs no terminal.

```
crl [out] :
  [QIL, < O : X@Database | output : QIL', Atts >, QIL'']
  =>
  [QIL, < O : X@Database | output : nil, Atts >, (QIL' QIL'')]
 if QIL' =/= nil .
```

[5] Na atual versão de Maude CBabel tool, a análise semântica é realizada indiretamente durante a transformação de componentes CBabel em módulos orientados a objetos de Full Maude. Por exemplo, se na declaração de um contrato em um conector, for referenciada uma porta não declarada no conector, a transformação do conector em um módulo orientado a objetos não será bem-sucedida. No futuro, pretendemos adicionar à ferramenta um melhor suporte à análise semântica de declarações CBabel.

4 Estudos de caso

Neste capítulo, apresentaremos alguns estudos de caso onde descrições arquiteturais em CBabel foram implementadas e analisadas com Maude CBabel tool, exemplificando a execução da ferramenta e dos mecanismos e técnicas que Maude CBabel tool oferece ao projetista para aumentar sua confiança na sua especificação.

Para facilitar a comparação de nossa ferramenta com outras abordagens encontradas na literatura para análise formal arquiteturas de "softwares", serão utilizadas as seguintes aplicações de concorrência e compartilhamento de recursos:

- A especificação de uma máquina que vende bolos e maçãs, semelhante as máquinas que vendem refigerantes.
- A aplicação dos produtores e consumidores, onde processos produtores e consumidores acessam um "buffer" para colocar itens recém produzidos, os produtores, ou retirar itens para consumo, os consumidores.
- A aplicação dos leitores e escritores, uma variação da aplicação dos produtores e consumidores onde os processos leitores apenas leem do "buffer", sem alterar seu conteúdo, e os processos escritores alteram o "buffer".
- A aplicação ceia de filósofos, onde um grupo de filósofos alterna entre as atividades de pensar, ter fome e comer, compartilhando recursos comuns, os garfos, para poderem comer.

Este capítulo está organizado da seguinte forma. Na Seção 4.1, descrevemos como utilizar Maude CBabel tool para análise de arquiteturas. Nas seções 4.2 a 4.5, analisamos as aplicações descritas acima.

4.1 Análise de arquiteturas com Maude CBabel tool

Em CBabel o comportamento interno de um componente não é descrito. Segundo Sztajnberg [46], os aspectos funcionais da aplicação não são descritos em CBabel, pois, compreendem a implementação dos módulos em uma linguagem de programação. Da mesma forma, também não fazem parte da descrição de uma arquitetura CBabel:

1. a especificação dos possíveis estados iniciais da aplicação;
2. a especificação das estratégias de escalonamento dos componentes que integram a arquitetura da aplicação;
3. a especificação das propriedades que devem ser garantidas durante a execução da aplicação.

Na versão atual de Maude CBabel tool, tais informações precisam ser descritas diretamente em Maude. Isto porque Maude CBabel tool ainda não oferece uma interface de comandos no nível de abstração de CBabel. [1] Da mesma forma, análises de arquiteturas em Maude CBabel tool são realizadas no nível da linguagem Maude. Ou seja, como vimos no Capítulo 3.1, *componentes* estarão sendo representados por *classes*, *instâncias* por *objetos*, *portas* por *mensagens* etc.

Neste capítulo, para cada arquitetura, dois módulos orientados a objetos em Maude serão definidos. No primeiro módulo, denominado *módulo de execução*, tornamos a arquitetura *executável* com a declaração dos comportamentos internos dos módulos e o estado inicial, ou estados iniciais, da arquitetura. Tornar a arquitetura executável é necessário para que possamos utilizar as ferramentas de análise de Maude apresentadas na Seção 2.4. No segundo módulo, denominado *módulo de análise*, são definidos os predicados relacionados às propriedades que serão analisadas com o verificador de modelos de Maude. Como vimos na Seção 2.4, esta separação entre a especificação do sistema e a especificação das propriedades é normalmente empregada por compreender dois diferentes níveis de especificação de um sistema.

Estratégias de escalonamento para os componentes da arquitetura poderiam ser definidas com auxílio dos recursos de meta-programação de Maude [19, 49]. Neste capítulo, no entanto, a execução das reescritas será feita com a estratégia padrão oferecida por Maude.

[1] Tal interface de comandos corresponderia a implementação do *modelo de gerência de configuração para o "framework" CR–RIO*, definido em [46].

Veremos, através dos exemplos, como são declarados os estados iniciais e as propriedades que serão analisadas. Antes, porém, precisamos apresentar algumas considerações sobre a definição do comportamento interno dos módulos.

4.1.1 Definição do comportamento interno dos módulos

Quando nos referimos ao comportamento interno dos módulos, estamos nos referindo ao comportamento *observável*. Definimos comportamento *observável* como o necessário para as análises que serão executadas na arquitetura. Por exemplo, se desejamos durante uma análise inspecionar a variação de valores de uma determinada variável declarada em um módulo, então na descrição do comportamento dos módulos, as alterações desta variável precisam ser descritas. Ou seja, o comportamento dos módulos deve ser definido em função das análises que serão executadas, o mais simples possível e no nível de abstração adequado para uma arquitetura de "software".

Como vimos na Seção 3.1.1, para cada porta declarada em um *módulo* CBabel, respectivas mensagens *do* e *done* são declaradas no módulo orientado a objetos de Maude, resultante do mapeamento. A interpretação de tais mensagens depende do tipo da porta. O comportamento interno deve determinar quando estímulos nas portas de saída serão gerados e como estímulos nas portas de entrada serão tratados. Desta forma, para uma porta de *entrada*:

- a mensagem *do* representa o início do comportamento interno do módulo ao receber um estímulo naquela porta;
- a mensagem *done*, o término do comportamento interno. Quando a porta de entrada for síncrona, o término do comportamento interno deve ainda gerar uma resposta para o componente que enviou o estímulo.

Para uma uma porta de *saída*:

- a mensagem *do* sinaliza que um estímulo para aquela porta deve ser gerado;
- a mensagem *done* sinaliza o término da interação pela porta. Quando a porta de saída é síncrona, o término da interação corresponde ao recebimento da resposta.

Em Maude, estaremos formalizando o comportamento interno de um módulo M, no recebimento de um estímulo em sua porta de entrada P_i, com a declaração de uma regra

de reescrita no módulo de execução no formato da regra 4.1. A mensagem $do(\omega, P_i, \iota)$ é reescrita para a mensagem $done(\omega, P_i, \iota)$, possivelmente alterando o estado do objeto ω, que representa a instância de M que recebeu o estímulo. Vimos na Seção 3.1.1 que o estado do objeto é constituído pelo valor de seus atributos. Na regra 4.1 os estados inicial e final são representados pelas variáveis A e A' do *sort AttributeSet*.

$$rl\ do(\omega, P_i, \iota)\ <\omega\ :\ M\ |\ A> \Rightarrow\ done(\omega, P_i, \iota)\ <\omega\ :\ M\ |\ A'>\ . \qquad (4.1)$$

As interações entre portas de módulos CBabel, que em nosso mapeamento correspondem a transmissão de mensagens entre objetos (vide Seção 3.1.2), sempre são iniciadas pelas portas de saída. Desta forma, para a análise de uma arquitetura, precisamos definir quais interações e quantas repetições destas interações iremos *simular*. Duas alternativas poderão ser utilizadas: (1) manter a arquitetura em *loop* ou; (2) definir, nos possíveis estados iniciais, os estímulos que devem ser gerados para cada uma das portas de saída dos módulos da arquitetura.

Em algumas arquiteturas, para garantirmos que todas as possíveis seqüências de interações entre os módulos serão consideradas nas análises, todas as portas de saída dos módulos da arquitetura são deixadas em *loop*, isto é, tão logo terminem uma interação, ou seja, recebimento da mensagem *done*, iniciam nova interação, ou seja, criação de uma nova mensagem *do*. Manter o sistema com realimentação de mensagens é uma técnica característica para análise de sistemas reativos. Meseguer, por exemplo, utiliza esta técnica para análises de sistemas concorrentes em [50].

A formalização do comportamento de *loop* em uma porta de saída P_o do módulo M é feita em Maude com a declaração de uma regra de reescrita com a forma da regra 4.2. Onde ω é o objeto que representa uma instância de M, com o estado interno do objeto representado pela variável A podendo ser alterado ou permanecendo constante.

$$rl\ done(\omega, P_o, \iota)\ <\omega\ :\ M\ |\ A> \Rightarrow\ do(\omega, P_o, \iota)\ <\omega\ :\ M\ |\ A'>\ . \qquad (4.2)$$

A segunda alternativa para definição do comportamento interno dos módulos, em relação aos estímulos de suas portas de saída, é inspirada em alguns exemplos da literatura.

Na implementação e análise de protocolos em [51, 52, 53], por exemplo, para cada análise realizada, são definidos estados iniciais específicos que proporcionam, conforme as mensagens e estados dos objetos presentes, uma ou mais interações do protocolo.

As análises propostas procuram então obter, com estes passos, exemplos suficientes de estados que permitam a identificação de um contra-exemplo para alguma propriedade. Obviamente, se somente alguns estados são analisados, uma resposta verdadeira nada significa.

Para arquiteturas com um número finito de estados de computação, geralmente utilizaremos a primeira alternativa para definição do comportamento interno dos módulos em relação às suas portas de saída. Isto porque, em primeiro lugar, esta abordagem permite a exploração de um número maior de possíveis estados da computação. Em segundo lugar, os estados redundantes não serão considerados nas explorações dos estados feitas pelo verificador de modelos ou pelo comando *search* de Maude. Isto porque ambas as ferramentas implementam um mecanismo de "cache" dos estados visitados. Desta forma, a exploração dos estados termina, mesmo com a existência de *ciclos* na transição entre estados.

Para aplicações com um número infinito de estados de computação, a segunda alternativa corresponde a uma possível abordagem para tornar finito o sistema de transição de estados, no sentido de que somente um subconjunto dos estados é analisado. A técnica de abstrações equacionais [54], que será utilizada na Seção 5.1.2, também pode ser empregada com este objetivo.

4.2 Uma máquina de venda

Em CBabel os módulos componentes de uma aplicação são executados de forma concorrente. Da mesma forma, os estímulos às portas destes módulos também podem ser executados concorrentemente entre si [46]. Nesta seção, apresentaremos e analisaremos a arquitetura de uma máquina de venda, [2] exercitando simplesmente a técnica de simulação sobre uma aplicação concorrente.

A máquina de venda que iremos analisar é semelhante a máquina de refrigerantes que, ao invés de vender apenas refrigerantes, vende bolos e maçãs. Uma maçã custa três "quarters" e um bolo, um dólar. A máquina foi construída para aceitar dólares e "quarters", mas, infelizmente, a máquina só é capaz de contar dólares na venda das mercadorias. Para compra de uma maçã, a máquina deve receber um dólar e devolver um "quarter". Para aliviar sua limitação, a máquina é capaz de trocar quatro "quarters"

[2] O exemplo da máquina de venda, em inglês "vending machine", é normalmente utilizado na literatura para apresentar o conceito de concorrência [50, 19].

por um dólar. Esta máquina pode ser representada graficamente por uma Rede de Petri, como mostra a Figura 4.

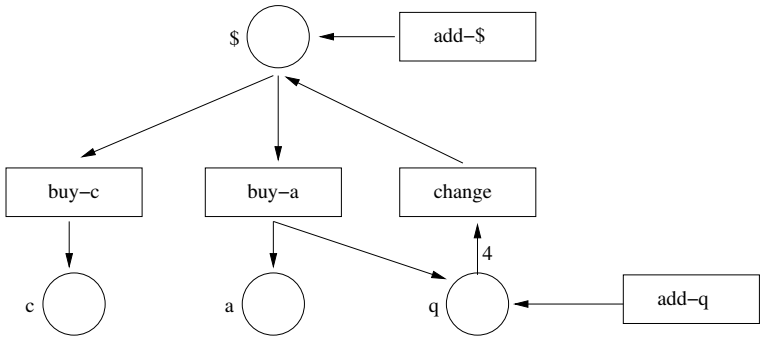

Figura 4: Rede de Petri da máquina de venda

A máquina é concorrente, pois, desde que providos os recursos necessários em cada *local* (os círculos), seus botões (os retângulos) podem ser pressionados ao mesmo tempo.

Na Figura 5, apresentamos a arquitetura CBabel da máquina. Na Figura 6 a representação gráfica (diagrama) informal da mesma arquitetura. No diagrama, os nomes dos componentes e de suas intâncias, bem como os links entre as portas, facilitam o entendimento da descrição textual da arquitetura. Apenas as amarrações de variáveis ("binds") não são graficamente representadas. Os módulos `BUY-APPLE`, `BUY-CAKE`, `ADD-DOLLAR`, `ADD-QUARTER` e `CHANGE`, representam cada botão da máquina. O módulo `SLOT` representa a bandeja de saída das mercadorias. A implementação de cada botão como um módulo permite que os mesmos sejam pressionados simultaneamente. Os conectores `COUNT-DOLLAR` e `COUNT-QUARTER` contabilizam o dinheiro depositado em suas variáveis `dollars` e `quarters`. O conector `MAKE-CHANGE` realiza a troca de "quarters" por dólares, desde que a quantidade de "quarters" seja suficiente. Lembrando da Seção 3.1.2, a porta `ground` serve para, em um contrato de interação, finalizarmos o estímulo recebido pela porta de entrada sem encaminhá-lo para outros componentes. Os conectores `SOLD-APPLE` e `SOLD-CAKE` realizam a venda das mercadorias caso existam dólares na máquina. Para simplificar a arquitetura o troco é representado pelas quantidades de dólares e "quarters" restantes nos conectores `COUNT-DOLLAR` e `COUNT-QUARTER` e admitimos que a máquina dispõe de uma quantidade infinita de produtos para venda. O módulo de aplicação, `VENDING-MACHINE`, são criadas as instâncias dos componentes e estabelecidas as ligações das portas e amarrações das variáveis.

Conforme descrito na Seção 4.1, para analisarmos arquiteturas utilizando Maude CBabel tool, precisamos torná-las executáveis. Vale lembrar que as análises são executadas no nível Maude, isto é, em termos de objetos e mensagens.

```
module BUY-APPLE {
 out port oneway buy-apple ;
}

module BUY-CAKE {
 out port oneway buy-cake ;
}

module ADD-DOLLAR {
 out port oneway add-$ ;
}

module ADD-QUARTER {
 out port oneway add-q ;
}

module CHANGE {
 out port oneway change ;
}

module SLOT {
 var int apples = 0 ;
 var int cakes = 0 ;
 in port oneway put-apple ;
 in port oneway put-cake ;
}

connector MAKE-CHANGE {
 staterequired int ch@dollars ;
 staterequired int ch@quarters ;
 in port oneway change-in ;
 interaction {
  change-in >
    guard(ch@quarters > 3) {
      before {
       ch@dollars = ch@dollars + 1 ;
       ch@quarters = ch@quarters - 4 ;
    } }
  > ground ;
 }
}

connector COUNT-DOLLAR {
 var int dollars = 0 ;
 in port oneway inc-$ ;
 interaction {
  inc-$ >
    guard(TRUE) {
      before {
       dollars = dollars + 1 ;
    } }
  > ground ;
 }
}

connector COUNT-QUARTER {
 var int quarters = 0 ;
 in port oneway inc-q ;
 interaction {
  inc-q >
    guard(TRUE) {
      before {
       quarters = quarters + 1 ;
    } }
  > ground ;
 }
}

connector SOLD-APPLE {
 staterequired int sa@dollars ;
 staterequired int sa@quarters ;
 in port oneway ack-apple ;
 out port oneway give-apple ;
 interaction {
  ack-apple >
    guard(sa@dollars > 0) {
      before {
       sa@dollars = sa@dollars - 1 ;
       sa@quarters = sa@quarters + 1 ;
    } }
  > give-apple ;
 }
}

connector SOLD-CAKE {
 staterequired int sc@dollars ;
 in port oneway ack-cake ;
 out port oneway give-cake ;
 interaction {
  ack-cake >
    guard(sc@dollars > 0) {
      before {
       sc@dollars = sc@dollars - 1 ;
    } }
  > give-cake ;
 }
}


application VENDING-MACHINE {
 instantiate ADD-DOLLAR as bt-ad ;
 ...
 link bt-aq.add-q to con-cq.inc-q ;
 ...
 bind int con-change.ch@dollars to con-cd.dollars ;
 bind int con-change.ch@quarters to con-cq.quarters ;
 bind int con-sa.sa@dollars to con-cd.dollars ;
 bind int con-sc.sc@dollars to con-cd.dollars ;
 bind int con-sa.sa@quarters to con-cq.quarters ;
}
```

Figura 5: Arquitetura para a máquina de venda

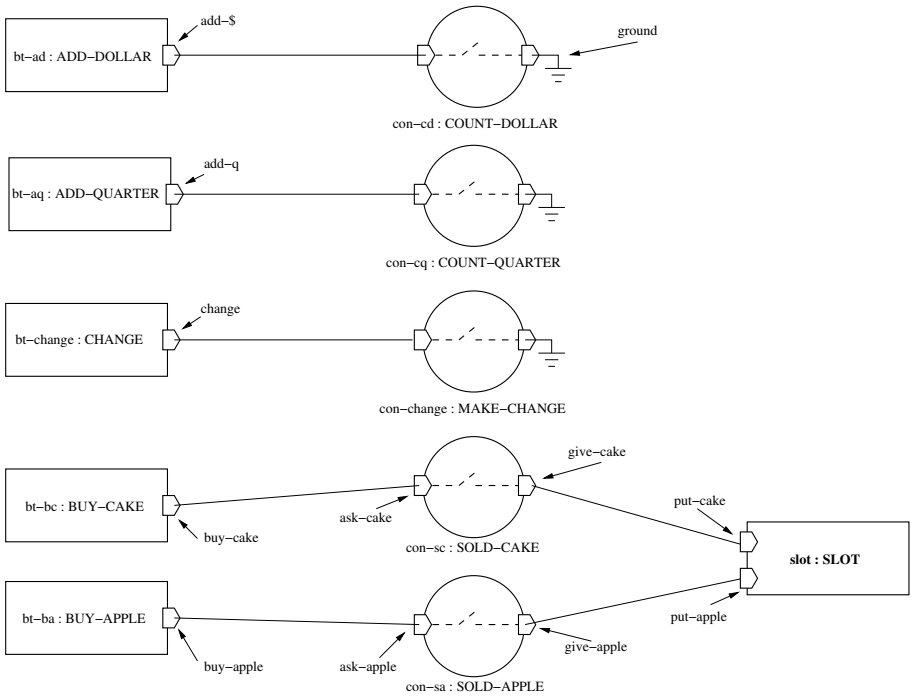

Figura 6: Diagrama da arquitetura `VENDING-MACHINE`

O módulo de execução para a arquitetura `VENDING-MACHINE` é mostrado na Figura 7. As regras `slot-put-apple` e `slot-put-cake` definem o comportamento interno dos objetos da classe `SLOT`. No recebimento das mensagens `put-apple` e `put-cake` os atributos `apples` e `cakes` devem ser incrementados e o comportamento interno finalizado com o envio da mensagem *done*. O comportamento interno das instâncias de `BUY-CAKE`, `BUY-APPLE`, `ADD-DOLLAR` e `ADD-QUARTER` não é relevante. Mensagens *do* para estes objetos serão simplesmente reescritas para mensagens *send*, conforme descrito na Seção 3.1.2. Um estado inicial é definido para a arquitetura, sendo atribuído a constante $op \ : \ initial \ \rightarrow \ Configuration$ equacionalmente. Esta configuração inicial é a topologia da arquitetura acrescida de: uma mensagem para o objeto `bt-ad`, uma mensagem para o objeto `bt-change`, uma mensagem para o objeto `bt-ba` e quatro mensagens para o objeto `bt-aq`. A função `copy` facilita a definição do estado inicial, gerando tantas cópias da mensagem passada como primeiro argumento, quanto o número passado como segundo argumento.

Depois que os módulos CBabel e o módulo `VM-EXEC` são carregados no Maude CBabel tool, podemos iniciar as análises da arquitetura.

Nossa máquina é concorrente porque podemos *apertar* diversos botões da máquina simultaneamente. Por exemplo, se forem colocados um dólar e quatro "quarters" na máquina e, simultaneamente, apertados os botões para compra de uma maçã e troca do dinheiro, a máquina deve responder, também simultaneamente, com uma maçã na

```
(omod VM-EXEC is
 inc VENDING-MACHINE .

 var C : Configuration .
 var O : Oid .
 var IT : Interaction .
 var N : Int .
 var P : PortId .
 var MSG : Msg .

 rl [slot-put-apple] :
    do(O, put-apple, IT) < O : SLOT | apples : N > =>
    done(O, put-apple, IT) < O : SLOT | apples : (N + 1) >  .

 rl [slot-put-cake] :
    do(O, put-cake, IT) < O : SLOT | cakes : N > =>
    done(O, put-cake, IT) < O : SLOT | cakes : (N + 1) >  .

 op copy : Msg Nat -> Configuration .
 eq copy(MSG, N) = MSG copy(MSG, N - 1) .
 eq copy(MSG, 0) = none .

 op initial : -> Configuration .
 eq initial = topology
              do(bt-ad, add-$, none) copy(do(bt-aq, add-q, none), 4)
              do(bt-change, change, none) do(bt-ba, buy-apple, none) .
endom)
```

Figura 7: Módulo de execução para a máquina de venda

bandeja de saída e um dólar e um "quarter" de troco.

Como vimos na Seção 2.4, uma das técnicas de análise que podemos utilizar para garantir que a arquitetura está corretamente implementada é a simulação. Com a simulação podemos, por exemplo, confirmar se todas as ligações de portas dos componentes da arquitetura foram realizadas corretamente. Podemos comprovar o comportamento descrito acima passando o estado inicial, definido no módulo `VM-EXEC` para o comando `rewrite`:

```
Maude> (rewrite initial .)
result Configuration :
 < bt-ad : ADD-DOLLAR | none >
 < bt-aq : ADD-QUARTER | none >
 < bt-ba : BUY-APPLE | none >
 < bt-bc : BUY-CAKE | none >
 < bt-change : CHANGE | none >
 < con-cd : COUNT-DOLLAR | dollars : 1 >
 < con-change : MAKE-CHANGE | ch@dollars : st(1,unchanged),
                              ch@quarters : st(1,unchanged)>
 < con-cq : COUNT-QUARTER | quarters : 1 >
 < con-sa : SOLD-APPLE | sa@dollars : st(1,unchanged),
                         sa@quarters : st(1, unchanged)>
 < con-sc : SOLD-CAKE | sc@dollars : st(1,unchanged)>
 < slot : SLOT | apples : 1,cakes : 0 >
```

O atributo `apples` de `slot` foi incrementado para um, representando a entrega de uma maçã na bandeja. Um dólar e um "quarter" foram devolvidos, conforme constatamos

pelo valor dos atributos `dollars` e `quarters` em `con-cd` e `con-cq`. Embora o resultado obtido corresponda ao esperado, não podemos afirmar que a máquina funcionou *como* foi planejada para funcionar.

Podemos ainda acompanhar a aplicação das regras de reescrita na simulação. O comando `set trace on` habilita a impressão na tela de cada passo de reescrita executado, imprimindo a regra utilizada, termo inicial e final, substituição de variáveis e ainda a resolução das condições na aplicação das regras ou equações condicionais. Como um traço de execução pode gerar um volume muito grande de informações, Maude oferece bastante controle sobre quais informações serão ou não impressas durante a execução das reescritas. Por exemplo, com o comando `set trace eq off`, podemos inibir a impressão das reescritas equacionais.

```
Maude> (rewrite initial .)
*********** rule
rl < O:Oid : V#2:ADD-DOLLAR | V#3:AttributeSet >
   do(O:Oid, add-$, none) =>
   < O:Oid : V#2:ADD-DOLLAR | V#3:AttributeSet >
   send(O:Oid, add-$, [O:Oid,add-$]) [label ADD-DOLLAR-sending-add-$] .

< bt-ad : ADD-DOLLAR | none > < bt-aq : ADD-QUARTER | none >
< bt-ba : BUY-APPLE | none > < bt-bc : BUY-CAKE | none >
< bt-change : CHANGE | none > < con-cd : COUNT-DOLLAR | dollars : 0 >
< con-change : MAKE-CHANGE | ch@dollars : st(0, unchanged),
                             ch@quarters : st(0,unchanged) >
< con-cq : COUNT-QUARTER | quarters : 0 >
< con-sa : SOLD-APPLE | sa@dollars : st(0, unchanged),
                        sa@quarters : st(0,unchanged) >
< con-sc : SOLD-CAKE | sc@dollars : st(0, unchanged) >
< slot : SLOT | apples : 0,cakes : 0 >
do(bt-ad, add-$, none) do(bt-aq, add-q, none)
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-aq, add-q, none) do(bt-ba, buy-apple, none) do(bt-change, change, none)
--->
< bt-aq : ADD-QUARTER | none > < bt-ba : BUY-APPLE | none >
< bt-bc : BUY-CAKE | none > < bt-change : CHANGE | none >
< con-cd : COUNT-DOLLAR | dollars : 0 >
< con-change : MAKE-CHANGE | ch@dollars : st(0, unchanged),
                             ch@quarters : st(0, unchanged) >
< con-cq : COUNT-QUARTER | quarters : 0 >
< con-sa : SOLD-APPLE | sa@dollars : st(0, unchanged),
                        sa@quarters : st(0, unchanged) >
< con-sc : SOLD-CAKE | sc@dollars : st(0, unchanged) >
< slot : SLOT | apples : 0,cakes : 0 >
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-ba, buy-apple, none) do(bt-change, change, none)
< bt-ad : ADD-DOLLAR | none > send(bt-ad, add-$, [bt-ad,add-$])
```

```
********** rule
rl < O:Oid : V#2:ADD-QUARTER | V#3:AttributeSet >
   do(O:Oid, add-q, none) => < O:Oid : V#2:ADD-QUARTER | V#3:AttributeSet >
   send(O:Oid, add-q, [O:Oid,add-q]) [label ADD-QUARTER-sending-add-q] .

...
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-ba, buy-apple, none) do(bt-change, change, none)
send(con-cd, inc-$, [bt-ad,add-$])
--->
...
do(bt-aq, add-q, none) do(bt-aq, add-q, none)
do(bt-aq, add-q, none) do(bt-ba, buy-apple, none)
do(bt-change, change, none) send(con-cd, inc-$, [bt-ad,add-$]))
< bt-aq : ADD-QUARTER | none > send(bt-aq, add-q, [bt-aq,add-q])

...
********** rule
crl < O:Oid : V#7:COUNT-QUARTER | V#9:AttributeSet,quarters : V#8:Int >
    send(O:Oid, inc-q, IT:Interaction) =>
    before(< O:Oid : V#7:COUNT-QUARTER | V#9:AttributeSet,quarters : V#8:Int >)
    send(O:Oid, ground, [O:Oid,ground] :: IT:Interaction)
 if open?(< O:Oid : V#7:COUNT-QUARTER | V#9:AttributeSet,quarters : V#8:Int >)
    = true [label COUNT-QUARTER-sending-inc-q] .

< bt-ad : ADD-DOLLAR | none > < bt-aq : ADD-QUARTER | none >
< bt-ba : BUY-APPLE | none > < bt-bc : BUY-CAKE | none >
< bt-change : CHANGE | none >
< con-cd : COUNT-DOLLAR | dollars : 1 >
< con-change : MAKE-CHANGE | ch@dollars : st(1, unchanged),
  ch@quarters : st(0, unchanged) >
< con-cq : COUNT-QUARTER | quarters : 0 >
< con-sa : SOLD-APPLE | sa@dollars : st(1, unchanged),sa@quarters :
  st(0, unchanged) >
< con-sc : SOLD-CAKE | sc@dollars : st(1, unchanged) >
< slot : SLOT | apples : 1,cakes : 0 > send(con-cq, inc-q, [bt-aq,add-q])
--->
< bt-ad : ADD-DOLLAR | none > < bt-aq : ADD-QUARTER | none > < bt-ba : BUY-APPLE | none >
< bt-bc : BUY-CAKE | none > < bt-change : CHANGE | none >
< con-cd : COUNT-DOLLAR | dollars : 1 >
< con-change : MAKE-CHANGE | ch@dollars : st(1, unchanged),ch@quarters :
  st(0, unchanged) >
< con-sa : SOLD-APPLE | sa@dollars : st(1, unchanged),sa@quarters : st(0, unchanged) >
< con-sc : SOLD-CAKE | sc@dollars : st(1, unchanged) >
< slot : SLOT | apples : 1,cakes : 0 >)
before(< con-cq : COUNT-QUARTER | quarters : 0 >)
send(con-cq, ground, [con-cq,ground] :: [bt-aq,add-q])

rewrite in VM-EXEC :
  initial
result Configuration :
 < bt-ad : ADD-DOLLAR | none > < bt-aq : ADD-QUARTER | none >
 < bt-ba : BUY-APPLE | none > < bt-bc : BUY-CAKE | none >
```

```
 < bt-change : CHANGE | none >
 < con-cd : COUNT-DOLLAR | dollars : 1 >
 < con-change : MAKE-CHANGE | ch@dollars : st(1,unchanged),
   ch@quarters : st(1,unchanged)>
 < con-cq : COUNT-QUARTER | quarters : 1 >
 < con-sa : SOLD-APPLE | sa@dollars : st(1,unchanged),sa@quarters : st(1,unchanged)>
 < con-sc : SOLD-CAKE | sc@dollars : st(1,unchanged)>
 < slot : SLOT | apples : 1,cakes : 0 >
```

Vimos acima a saída completa do comando `rewrite` que foi editada, e apenas as duas primeiras e a última aplicação de regra foram deixadas para ilustrar a simulação. Analisando as regras aplicadas e a evolução dos estados durante a simulação, temos alguma indicação de que o comportamento da arquitetura está adequado, ou seja, as mensagens estão sendo enviadas e recebidas corretamente pelos objetos que representam as intâncias dos componentes da arquitetura. No entanto, a simulação corresponde apenas a um dos possíveis caminhos na execução da arquitetura. Para analisar se nossa máquina está corretamente implementada, precisamos explorar todos os possíveis caminhos da computação a partir de um determinado estado inicial.

Em nossa máquina, com um dólar e três "quarters", devemos ser capazes de comprar uma maçã e um bolo. Abaixo executamos o comando *search* de Maude onde:

- o símbolo `=>!` indica que desejamos localizar apenas os estados *finais*, ou seja, estados que não podem mais ser reescritos;
- o estado inicial correponde a execução do comportamento interno dos objetos `bt-ad`, `bt-change`, `bt-ba`, `bt-bc` e três interações do objeto `bt-aq`;
- o estado final deve ser uma configuração de objetos e mensagens qualquer, sem atender a nenhuma condição adicional. Por isso a variável `C`, do *sort* `Configuration`, foi utilizada.

```
Maude> (search topology do(bt-change,change,none)
               do(bt-ad,add-$,none) copy(do(bt-aq,add-q,none), 3)
               do(bt-bc,buy-cake,none) do(bt-ba,buy-apple, none)
         =>! C:Configuration .)

Solution 1
C:Configuration <- < bt-ad : ADD-DOLLAR | none >
 < bt-aq : ADD-QUARTER | none >
 < bt-ba : BUY-APPLE | none >
 < bt-bc : BUY-CAKE | none >
 < bt-change : CHANGE | none >
 < con-cd : COUNT-DOLLAR | dollars : 0 >
```

```
 < con-change : MAKE-CHANGE | ch@dollars : st(0,unchanged),
                              ch@quarters : st(3,unchanged)>
 < con-cq : COUNT-QUARTER | quarters : 3 >
 < con-sa : SOLD-APPLE | sa@dollars : st(0,unchanged),
                         sa@quarters : st(3,unchanged)>
 < con-sc : SOLD-CAKE | sc@dollars : st(0,unchanged)>
 < slot : SLOT | apples : 0,cakes : 1 >
 send(con-change,change-in,[bt-change,change])
 send(con-sa,ack-apple,[bt-ba,buy-apple])

Solution 2
C:Configuration <- < bt-ad : ADD-DOLLAR | none >
 < bt-aq : ADD-QUARTER | none >
 < bt-ba : BUY-APPLE | none >
 < bt-bc : BUY-CAKE | none >
 < bt-change : CHANGE | none >
 < con-cd : COUNT-DOLLAR | dollars : 0 >
 < con-change : MAKE-CHANGE | ch@dollars : st(0,unchanged),
                ch@quarters : st(0, unchanged)>
 < con-cq : COUNT-QUARTER | quarters : 0 >
 < con-sa : SOLD-APPLE | sa@dollars : st(0,unchanged),
            sa@quarters : st(0,unchanged)>
 < con-sc : SOLD-CAKE | sc@dollars : st(0,unchanged)>
 < slot : SLOT | apples : 1,cakes : 1 >

No more solutions.
```

Na primeira solução encontrada, o dólar colocado na máquina foi primeiro consumido na compra de um bolo. Devido a limitação já mencionada da máquina, os três "quarters" restantes não foram considerados para a compra da maçã. As mensagens `change-in` e `ack-apple`, resultantes da execução dos objetos `bt-change` e `bt-ba`, terminaram bloqueadas nos objetos `con-change` e `con-sa`, respectivamente. Na segunda solução, o dólar foi utilizado na compra de uma maçã. Os três "quarters", acrescidos do "quarter" que sobrou no troco da venda da maçã, foram trocados para um dólar e um bolo pôde ser entregue.

Para comprovar que um bolo não pode ser vendido quando apenas três "quarters" são fornecidos para a máquina, podemos realizar uma exploração dos estados com o comando *search*. No comando abaixo, informamos que a busca deve iniciar a partir de um estado onde, além da topologia da arquitetura, existam três estímulos ao objeto `bt-aq` e um estímulo ao objeto `bt-bc`. O símbolo `=>*` indica que não apenas os estados finais devem ser considerados, mas qualquer estado atingível com zero ou mais passos de reescrita a partir do estado de início da busca. Desejamos que seja localizado pelo menos um estado, por isso o parâmetro `[1]`, onde o valor do atributo `cakes`, do objeto `slot`, seja maior que zero.

```
Maude> (search [1] topology copy(do(bt-aq, add-q, none), 3)
                   do(bt-bc, buy-cake, none)
         =>* C:Configuration < slot : SLOT | apples : N:Int , cakes : M:Int >
         such that M:Int > 0 .)

No solution.
```

4.3 A aplicação produtores e consumidores

O problema dos produtores e consumidores é comumente utilizado na literatura, por exemplo em [26], para descrição de aspectos de coordenação entre processos concorrentes que disputam acesso a um recurso comum. Esta aplicação foi também extensamente utilizada em [46] para apresentação das primitivas da linguagem CBabel.

Nesta aplicação, um recurso denominado "buffer" é disputado entre processos produtores e consumidores. Os produtores acessam o "buffer" para depositar itens que eles acabaram de produzir. Os consumidores acessam o "buffer" para remover itens que eles consumirão. O "buffer" pode ser limitado ou ilimitado. Quando limitado, a aplicação deve garantir que não serão depositados mais itens que o limite máximo permitido, e não serão removidos itens quando o "buffer" estiver vazio. Nesta seção, estaremos implementando e analisando três diferentes arquiteturas para o problema dos produtores e consumidores. Estas arquiteturas foram inspiradas nas arquiteturas propostas por Sztajnberg em [46].

A primeira arquitetura, e sua respectiva representação gráfica, é apresentada na Figura 8. No diagrama os nomes das portas e módulos são destacados para facilitar a interpretação da descrição textual. São declarados três módulos, `PRODUCER`, `CONSUMER` e `BUFFER` para representar, respectivamente, os produtores, consumidores e "buffer". O connector `DEFAULT` implementa o contrato de interação de CBabel, descrito na Seção 3.1.3. O módulo da aplicação, `PC-DEFAULT`, declara duas instâncias de cada um dos módulos `PRODUCER` e `CONSUMER`, duas instâncias do conector `DEFAULT` e uma instância do módulo `BUFFER`. Em seguida, as portas de cada instância são ligadas. Na Figura 8, `prod1` e `prod2`, instâncias do módulo `PRODUCER`, acessam `buff`, instância do módulo `BUFFER`, através de `default1`, instância do connector `DEFAULT`. Da mesma forma, `cons1` e `cons2`, instâncias do módulo `CONSUMER`, acessam `buff` através de `default2`, outra instância do conector `DEFAULT`. Esta arquitetura não implementa qualquer controle de concorrência sobre as instâncias dos módulos. Assim sendo, acessos concorrentes à instância do módulo `BUFFER` poderão ocorrer.

Na Figura 9, apresentamos a segunda arquitetura para o problema. O conector `DEFAULT` foi substituído pelo conector `MUTEX` e um novo módulo da aplicação foi declarado, `PC-MUTEX`.

```
module PRODUCER {
 out port producer@put ;
}

module CONSUMER {
 out port consumer@get ;
}

connector DEFAULT {
 in port default@in ;
 out port default@out ;
 interaction{
   default@in > default@out ;
 }
}
```

```
application PC-DEFAULT {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod1 ;
 instantiate PRODUCER as prod2 ;
 instantiate CONSUMER as cons1 ;
 instantiate CONSUMER as cons2 ;
 instantiate DEFAULT as default1 ;
 instantiate DEFAULT as default2 ;
 link prod1.producer@put to default1.default@in ;
 link prod2.producer@put to default1.default@in ;
 link default1.default@out to buff.buffer@put ;
 link cons1.consumer@get to default2.default@in ;
 link cons2.consumer@get to default2.default@in ;
 link default2.default@out to buff.buffer@get ;
}

module BUFFER {
 var int items = 0 ;
 var int maxitems = 2 ;
 in port buffer@put ;
 in port buffer@get ;
}
```

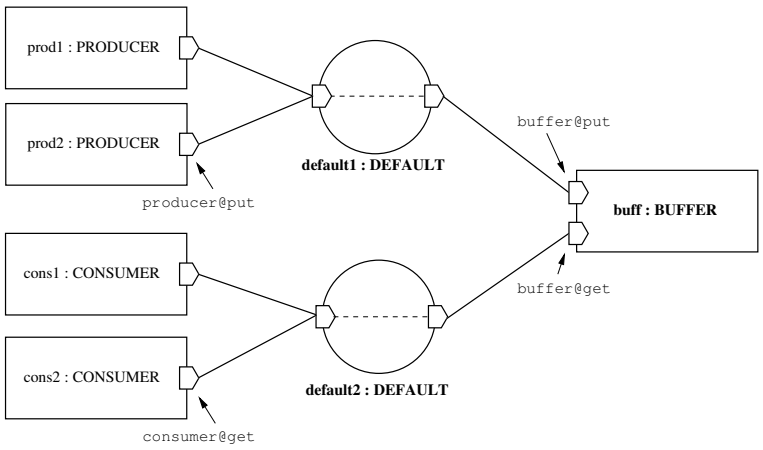

Figura 8: Arquitetura `PC-DEFAULT` para os produtores e consumidores

As declarações dos demais módulos, PRODUCER, CONSUMER e BUFFER, não são apresentadas por terem permanecido inalteradas em relação a Figura 8. Nesta nova arquitetura, os acessos concorrentes à instância do módulo BUFFER devem ser evitados pelo conector MUTEX, que implementa o contrato de exclusão mútua de CBabel.

```
connector MUTEX {
 in port mutex@in1 ;
 in port mutex@in2 ;
 out port mutex@out1 ;
 out port mutex@out2 ;

 exclusive{
  mutex@in1 > mutex@out1 ;
  mutex@in2 > mutex@out2 ;
 }
}
```

```
application PC-MUTEX {
 instantiate BUFFER as buff ;
 instantiate PRODUCER as prod1 ;
 instantiate PRODUCER as prod2 ;
 instantiate CONSUMER as cons1 ;
 instantiate CONSUMER as cons2 ;
 instantiate MUTEX as mutx ;
 link prod1.producer@put to mutx.mutex@in1 ;
 link prod2.producer@put to mutx.mutex@in1 ;
 link cons1.consumer@get to mutx.mutex@in2 ;
 link cons2.consumer@get to mutx.mutex@in2 ;
 link mutx.mutex@out1 to buff.buffer@put ;
 link mutx.mutex@out2 to buff.buffer@get ;
}
```

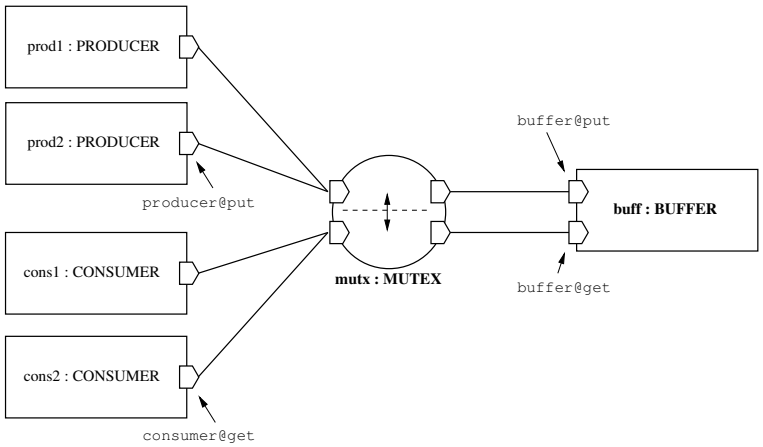

Figura 9: Arquitetura PC-MUTEX para os produtores e consumidores

Nas duas arquiteturas anteriores, os limites de armazenamento do "buffer" não são controlados. Na arquitetura apresentada na Figura 10 este problema é resolvido. Os conectores GUARD-GET e GUARD-PUT controlam os acessos ao módulo BUFFER por meio dos contratos de interação guarda, que inspecionam as variáveis de estado Full e Empty. Tais variáveis correspondem a semáforos [6]. O incremento das variáveis pode ser interpretado com a primitiva *signal*, e o decremento como a primitiva *wait*. Nesta arquitetura, o conector MUTEX é mantido para continuar a impedir acessos concorrentes ao "buffer".

Importadas as arquiteturas em Maude CBabel tool, devemos construir os módulos de verificação e análise. Como as três arquiteturas, cujos módulos de aplicaçao são PC-DEFAULT, PC-MUTEX e PC-GUARDS-MUTEX, compartilham os mesmos módulos, e em todas, desejamos verificar as mesmas propriedades, apenas um módulo de execução e um módulo de análise serão necessários.

Na Figura 11 é apresentado o módulo de execução PCB-EXEC. Este módulo importa,

```
connector GUARD-PUT {
 var int Full = 0 ;
 staterequired int gp@Empty ;
 in port gp@in ;
 out port gp@out ;
 interaction {
  gp@in > guard(gp@Empty > 0) {
    before {
      gp@Empty = gp@Empty - 1 ;
    }
    after {
      Full = Full + 1 ;
    }
  } > gp@out ;
 }
}
```

```
connector GUARD-GET {
 var int Empty = 2 ;
 staterequired int gg@Full ;
 in port gg@in ;
 out port gg@out ;
 interaction {
  gg@in > guard(gg@Full > 0) {
    before {
      gg@Full = gg@Full - 1 ;
    }
    after {
      Empty = Empty + 1 ;
    }
  } > gg@out ;
 }
}
```

```
application PC-GUARDS-MUTEX {
 instantiate BUFFER as buff ;
 ...
 link prod1.producer@put to gput.gp@in ;
 link prod2.producer@put to gput.gp@in ;
 ...
 bind int gget.gg@Full to gput.Full ;
 bind int gput.gp@Empty to gget.Empty ;
}
```

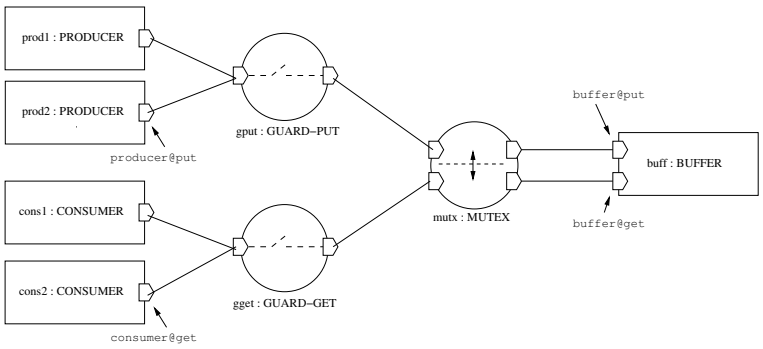

Figura 10: Arquitetura `PC-GUARDS-MUTEX` para os produtores e consumidores

inicialmente, o módulo `PC-DEFAULT`, módulo Maude gerado por Maude CBabel tool para o módulo `PC-DEFAULT` em CBabel. Para a análise das arquiteturas `PC-MUTEX` e `PC-MUTEX-GUARDS`, este módulo precisará ser redefinido de forma a importar a arquitetura correspondente. Em seguida, as regras `producer-do` e `consumer-do` definem que as instâncias de `PRODUCER` e `CONSUMER` serão deixadas em *loop*, continuamente enviando mensagens de `producer@put` e `producer-get`, respectivamente. As regras `buffer-do-put` e `buffer-do-get` definem o comportamento da instância do módulo `BUFFER`, quando no recebimento de mensagens `buffer@put` e `buffer@get`. O recebimento da mensagem `buffer@put` deverá gerar o incremento do valor do atributo `items`, que representa o número de itens armazenados no "buffer", e o recebimento da mensagem `buffer@get` deverá gerar o decremento do valor de `items`. Para tornar finito o número de estados alcançáveis na execução da arquitetura, o número de itens armazenados é mantido no intervalo $[-1, maxitems + 1]$. Na realidade, como estamos interessados apenas em determinar se o número de itens armazenados, em algum instante, ultrapassa o limite máximo, valor do atributo `maxitems`, ou torna-se negativo, poderíamos mesmo definir uma constante para representar o intervalo $[0, maxitems]$. No entanto, considerando que o número de estados de computação das arquiteturas é pequeno, tal simplificação não se faz necessária.

```
(omod PCB-EXEC is
 inc PC-DEFAULT .

 var C : Configuration .
 var O : Oid .
 var IT : Interaction .
 vars M N : Int .

 op initial : -> Configuration .
 eq initial = topology
              do(cons1, consumer@get, none) do(prod1, producer@put, none)
              do(cons2, consumer@get, none) do(prod2, producer@put, none) .

 rl [producer-do] : done(O, producer@put, IT) => do(O, producer@put, none) .
 rl [consumer-do] : done(O, consumer@get, IT) => do(O, consumer@get, none) .

 rl [buffer-do-put] :
    do(O, buffer@put, IT) < O : BUFFER | buffer@items : N, buffer@maxitems : M >
    =>
    done(O, buffer@put, IT)
    < O : BUFFER | buffer@maxitems : M ,
          buffer@items : (if (N + 1) > (M + 1) then (M + 1) else (N + 1) fi) > .

 rl [buffer-do-get] :
    do(O, buffer@get, IT) < O : BUFFER | buffer@items : N, buffer@maxitems : M >
    =>
    done(O, buffer@get, IT)
    < O : BUFFER | buffer@items : (if (N - 1) < -1 then -1 else (N - 1) fi),
                   buffer@maxitems : M > .
endom)
```

Figura 11: Módulo de execução para a aplicação produtores e consumidores

Em todas as três arquiteturas, estaremos considerando o caso do "buffer" limitado e

analisando as seguintes propriedades.

Propriedade 4.3.1 *Em nenhum estado da computação deve ser verdade que todas as instâncias de módulos estão bloqueadas e a computação impedida de prosseguir. Isto é, a arquitetura deve estar livre de "deadlock".*

Propriedade 4.3.2 *Em nenhum estado da computação, o objeto* `buff`*, instância do módulo* `BUFFER` *pode receber mensagens de* `buffer@put` *e* `buffer@get` *simultaneamente. Isto é, não devem existir acessos concorrentes de produtores e consumidores ao "buffer" ("race condition").*

Propriedade 4.3.3 *Em nenhum estado da computação o valor do atributo* `items` *do objeto* `buff` *deve ser superior ao valor do atributo* `maxitems`*. Isto é, a arquitetura não deve permitir "overflow" do "buffer", quando o número de itens armazenados no "buffer" ultrapassa o limite máximo estabelecido para o tamanho do "buffer".*

Propriedade 4.3.4 *Em nenhum estado da computação o valor do atributo* `items` *do objeto* `buff` *deve ser negativo. Isto é, a arquitetura não deve permitir "underflow" do "buffer", que ocorre quando o número de itens armazenados no "buffer" torna-se negativo.*

Todas as propriedades apresentadas acima referem-se a existência ou não de estados que satisfaçam determinada propriedade individualmente, isto é, são propriedades de estados. Recordando da Seção 2.4, este tipo de propriedade pode ser validada com uma busca pela árvore de estados que representa as possíveis computações na arquitetura. Ou seja, com o comando *search* de Maude podemos analisar todas as propriedades acima. No entanto, para ilustrar a utilização do verificador de modelos de Maude, estaremos definido algumas proposições e utilizando o verificador de modelos para análise das propriedades 4.3.2, 4.3.3 e 4.3.4. Pretendemos assim mostrar que o verificador de modelos pode não só analisar propriedades relacionadas aos caminhos da árvore de estados, mas também propriedades de estados.

Na Figura 12 é apresentado o módulo `PCB-VER`. Nele são declaradas as proposições e fórmulas temporais que serão utilizadas nas análises de cada arquitetura. As proposições `putting` e `getting`, definidas como constantes do *sort* `Prop`, se verificam nos estados onde alguma instância de `BUFFER` esteja recebendo, respectivamente, as mensagens `buffer@put` ou `buffer@get`. As proposições `overflow` e `underflow`, também declaradas como constantes do *sort* `Prop`, se verificam nos estados onde o número de itens armazenados em alguma

instância de `BUFFER` tenha, respectivamente, ultrapassado o limite máximo ou se tornado negativo. Com auxílio destas proposições, podemos expressar as propriedades 4.3.2, 4.3.3 e 4.3.4 através de fórmulas em LTL (vide Seção 2.4). A propriedade 4.3.2 é expressa pela fórmula `[]~ (putting /\ getting)`, atribuída à constante `raceCond` (do inglês "race condition"), que define que em nenhum estado da árvore de estados das computações da arquitetura alguma instância de `BUFFER` estará recebendo, simultaneamente, as mensagens `buffer@put` e `buffer@get`. A propriedade 4.3.3 é expressa pela fórmula `[]~ overflow`, isto é, em nenhum estado da árvore de estados das computações da arquitetura a proposição `overflow` é válida. A propriedade 4.3.4 é expressa pela fórmula `[]~ underflow`, isto é, em nenhum estado das árvores de estados das computações da arquitetura a proposição `underflow` é válida.

```
(omod PCB-VER is
  pr PCB-EXEC .
  pr MODEL-CHECKING .

  var C : Configuration .
  var O : Oid .
  var IT : Interaction .
  vars N M : Int .

  subsort Configuration < State .

  ops putting getting : -> Prop .
  eq < O : BUFFER | > send(O, buffer@put, IT) C |= putting = true .
  eq < O : BUFFER | > send(O, buffer@get, IT) C |= getting = true .

  op raceCond : -> Formula .
  eq raceCond = [] ~ (putting /\ getting) .

  ops overflow underflow : -> Prop .
  ceq < O : BUFFER | buffer@items : N, buffer@maxitems : M >
      C |= overflow = true if N > M .

  ceq < O : BUFFER | buffer@items : N >
      C |= underflow = true if N < 0 .
endom)
```

Figura 12: Módulo de análise para a aplicação produtores e consumidores

Após carregar no Maude CBabel tool as arquiteturas e os módulos de execução e análise, podemos iniciar as análises.

Podemos garantir que a propriedade 4.3.1 é satisfeita pela arquitetura `PC-DEFAULT` com a execução de uma busca por estados *finais* em largura pela árvore de estados gerada na execução da arquitetura. Na busca abaixo, `initial` é o estado inicial declarado no módulo `PCB-EXEC`. O símbolo `=>!` informa que desejamos localizar estados *finais*, ou seja, estados que não possam ser mais reescritos, ou seja, folhas da árvore de estados que representa as possíveis computações na arquitetura. O padrão do estado que buscamos é uma configuração qualquer, por isso utilizamos a variável `C:Configuration`.

```
Maude> (search initial =>! C:Configuration .)

No solution.
```

No módulo `PCB-EXEC` as regras `producer-do` e `consumer-do` especificam que as instâncias de `PRODUCER` e `CONSUMER` devem permanecer em *loop*, isto é, constantemente reenviando mensagens tão logo recebam uma resposta. Sendo assim, se a busca acima localizasse algum estado que não pudesse ser reescrito, este estado seria um estado de "deadlock".

Como já foi dito, para verificarmos a propriedade 4.3.2 na arquitetura `PC-DEFAULT`, submetemos a fórmula `raceCond`, definida no módulo `PCB-VER` na Figura 12, ao verificador de modelos de Maude. No comando abaixo, o verificador de modelos irá verificar a fórmula `raceCond` em todos os estados da árvore de estados das computações da arquitetura, alcançáveis a partir do estado inicial definido pela constante `initial`, declarada no módulo `PCB-EXEC` na Figura 11:

```
Maude> (red modelCheck(initial, raceCond) . )
result ModelCheckResult :
  counterexample({< buff : BUFFER | items : 0,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< default1 : DEFAULT | none >
< default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons1,consumer@get,none)
do(cons2,consumer@get,none)
do(prod1,producer@put,none)
do(prod2,producer@put,none),'CONSUMER-sending-consumer@get}
...
{< buff : BUFFER | items : -1,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : locked >
< default1 : DEFAULT | none >
< default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : locked >
< prod2 : PRODUCER | producer@put-status : unlocked >
done(cons1,consumer@get,none)
done(prod2,producer@put,none)
send(buff,buffer@get,[default2,default@out]::[cons2,consumer@get])
send(buff,buffer@put,[default1,default@out]::[prod1,producer@put]),
    'BUFFER-recevingAndDo-buffer@get}...)
```

O contra-exemplo acima mostra um caminho de execução onde existe um estado (o último estado exibido após "...") em que o objeto `buff` recebe, simultaneamente, uma mensagem `buffer@put` e uma mensagem `buffer@get`, ou seja, a arquitetura `PC-DEFAULT` não

garante exclusão mútua nos acessos ao *buffer*. O verificador de modelos de Maude explora os estados da computação utilizando uma busca em profundidade, desta forma, um contra-exemplo pode ser bastante extenso. A saída acima foi editada para que apenas o estado inicial e o estado com as duas mensagens para o objeto `buff` fossem exibidos.

O comando *search* de Maude também pode ser utilizado para comprovar que a propriedade 4.3.2 não é válida na arquitetura `PC-DEFAULT`. Na busca abaixo, desejamos localizar algum estado (apenas um, por isso o limite `[1]`), alcançável a partir do estado inicial definido pela constante `initial`, onde o objeto `buff` esteja recebendo, simultaneamente, mensagens de `buffer@put` e `buffer@get`. O símbolo `=>*`, relembrando, indica que estamos interessados em estados que possam ser alcançados com zero ou mais passos de reescritas a partir do estado inicial.

```
Maude> (search [1] initial =>* C:Configuration
                               send(buff, buffer@put, IT1:Interaction)
                               send(buff, buffer@get, IT2:Interaction) .)

Solution 1
C:Configuration <- < buff : BUFFER | items : 0,maxitems : 2 >
 < cons1 : CONSUMER | consumer@get-status : locked >
 < cons2 : CONSUMER | consumer@get-status : unlocked >
 < default1 : DEFAULT | none > < default2 : DEFAULT | none >
 < prod1 : PRODUCER | producer@put-status : locked >
 < prod2 : PRODUCER | producer@put-status : unlocked >
 do(cons2,consumer@get,none) do(prod2,producer@put,none);
IT1:Interaction <- [default1,default@out]::[prod1,producer@put];
IT2:Interaction <- [default2,default@out]::[cons1,consumer@get]
```

Acima, o comando *search* encontrou um estado cujo padrão corresponde ao desejado. No estado localizado, o objeto `buff` está recebendo uma mensagem de `buffer@put`, produzida a partir da mensagem inicial de `prod1` e uma mensagem de `buffer@get`, produzida a partir da mensagem inicial de `cons1`, conforme mostra os "valores" atribuídos às variáveis `IT1` e `IT2`, variáveis para interações (do tipo `Interaction`). A existência de uma solução na busca acima é condição suficiente para garantir que a propriedade 4.3.2 não é válida na arquitetura `PC-DEFAULT`.

Na arquitetura `PC-DEFAULT` nenhum controle é implementado para evitar que os limites de armazenamento do "buffer" sejam ultrapassados. Logo, se submetermos a fórmula `[]~ underflow` ao verificador de modelos de Maude, encontramos um contra-exemplo para a propriedade 4.3.4:

```
Maude> (red modelCheck(initial, [] ~ underflow) .)
result ModelCheckResult :
```

```
  counterexample({< buff : BUFFER | items : 0,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< default1 : DEFAULT | none >
< default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons1,consumer@get,none)
do(cons2,consumer@get,none)
do(prod1,producer@put,none)
do(prod2,producer@put,none),'CONSUMER-sending-consumer@get}
...
{< buff : BUFFER | items : -1, maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : locked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< default1 : DEFAULT | none > < default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
done(buff,buffer@get,[default2,default@out]::[cons1,consumer@get])
done(cons2,consumer@get,none) done(prod1,producer@put,none)
done(prod2,producer@put,none),'BUFFER-doneAndAcking-buffer@get} ...)
```

O contra-exemplo acima mostra um traço de execução onde existe um estado em que o valor do atributo `items` do objeto `buff` é negativo (o último após ...), isto é, em que a proposição `underflow`, definida no módulo de análise, é válida. A mesma análise pode ser feita para a propriedade 4.3.3. O verificador de modelos também encontra um conta-exemplo para a fórmula `[]~ overflow`:

```
Maude> (red modelCheck(initial, [] ~ overflow) .)
result ModelCheckResult :
  counterexample({< buff : BUFFER | items : 0,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< default1 : DEFAULT | none >
< default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons1,consumer@get,none)
do(cons2,consumer@get,none)
do(prod1,producer@put,none)
do(prod2,producer@put,none),'CONSUMER-sending-consumer@get}
...
{< buff : BUFFER | items : 3, maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< default1 : DEFAULT | none > < default2 : DEFAULT | none >
< prod1 : PRODUCER | producer@put-status : locked >
< prod2 : PRODUCER | producer@put-status : unlocked >
done(buff,buffer@put,[default1,default@out]::[prod1,producer@put])
done(cons1,consumer@get,none) done(cons2,consumer@get,none)
done(prod2,producer@put,none),'BUFFER-doneAndAcking-buffer@put} ...)
```

No contra-exemplo acima, no segundo estado exibido, o valor do atributo `items` do objeto `buff` é superior ao valor do atributo `maxitems`, isto é, este estado comprova que a propriedade 4.3.3 não é válida na arquitetura `PC-DEFAULT`.

Para a análise da arquitetura `PC-MUTEX`, precisamos primeiro editar e reimportar para Maude CBabel tool o módulo `PCB-EXEC`. A importação do módulo `PC-DEFAULT` será substituída pela importação do módulo `PC-MUTEX`, permanecendo inalterado o restante do módulo:

```
(omod PCB-EXEC is
  inc PC-MUTEX .
  ...
omod)
```

A propriedade 4.3.1 também é válida na arquitetura `PC-MUTEX`:

```
Maude> (search initial =>! C:Configuration .)

No solution.
```

Na arquitetura `PC-MUTEX`, os acessos concorrentes ao *buffer* são evitados com o conector `MUTEX`, que implementa o contrato de exclusão mútua. Ao submetermos a fórmula temporal `raceCond`, ao verificador de modelos, temos a comprovação de que a propriedade 4.3.2 é válida em todos os caminhos de execução da arquitetura `PC-MUTEX`.

```
Maude> (red modelCheck(initial, raceCond) .)
result Bool :
  true
```

Da mesma forma, os limites de armazenamento do *buffer* continuam a não ser controlados. Ao verificarmos a propriedade 4.3.3, encontramos um contra-exemplo para a fórmula `[]~overflow`:

```
Maude> (red modelCheck(initial, []~ overflow) .)
result ModelCheckResult :
  counterexample({< buff : BUFFER | items : 0,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< mutx : MUTEX | status : unlocked >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons1,consumer@get,none) do(cons2,consumer@get,none)
do(prod1,producer@put,none)
do(prod2,producer@put,none),'CONSUMER-sending-consumer@get}
```

```
...
{< buff : BUFFER | items : 3, maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< mutx : MUTEX | status : locked >
< prod1 : PRODUCER | producer@put-status : locked >
< prod2 : PRODUCER | producer@put-status : unlocked >
done(buff,buffer@put,[mutx,mutex@out1]::[prod1,producer@put])
done(cons1,consumer@get,none) done(cons2,consumer@get,none)
done(prod2,producer@put,none),'BUFFER-doneAndAcking-buffer@put}...)
```

No contra-exemplo supra, no segundo estado listado, o valor do atributo `items` é superior ao valor do atributo `maxitems` do objeto `buff`, comprovando que a propriedade 4.3.3 não é válida na arquitetura `PC-MUTEX`.

Ao verificarmos a propriedade 4.3.4, o verificador de modelos encontra um contra-exemplo para a fórmula `[]~underflow`:

```
Maude> (red modelCheck(initial, []~ underflow) .)
result ModelCheckResult :
  counterexample({< buff : BUFFER | items : 0,maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : unlocked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< mutx : MUTEX | status : unlocked >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons1,consumer@get,none)
do(cons2,consumer@get,none)
do(prod1,producer@put,none)
do(prod2,producer@put,none),'CONSUMER-sending-consumer@get}
...
{< buff : BUFFER | items : -1, maxitems : 2 >
< cons1 : CONSUMER | consumer@get-status : locked >
< cons2 : CONSUMER | consumer@get-status : unlocked >
< mutx : MUTEX | status : locked >
< prod1 : PRODUCER | producer@put-status : unlocked >
< prod2 : PRODUCER | producer@put-status : unlocked >
do(cons2,consumer@get,none) do(prod1,producer@put,none)
do(prod2,producer@put,none)
done(buff,buffer@get,[mutx,mutex@out2]::[cons1,consumer@get]),
 'BUFFER-doneAndAcking-buffer@get}...)
```

No contra-exemplo acima, no segundo estado listado, o valor do atributo `items` do objeto `buff` é negativo, comprovando que na arquitetura `PC-MUTEX` a propriedade 4.3.4 não é válida.

Finalmente, na arquitetura `PC-MUTEX-GUARDS`, as propriedades 4.3.1, 4.3.2, 4.3.3 e 4.3.4, listadas no início da seção são atendidas. Com a mesma busca já realizada para as demais arquiteturas, comprovamos que a propriedade 4.3.1 é válida em `PC-MUTEX-GUARDS`:

```
Maude> (search initial =>! C:Configuration .)

No solution.
```

A propriedade 4.3.2 continua a ser garantida pela presença do objeto `mutex` na topologia:

```
Maude> (red modelCheck(initial, raceCond) .)
result Bool :
  true
```

Finalmente, as propriedades 4.3.3 e 4.3.4 também são garantidas na arquitetura `PC-GUARDS-MUTEX`. Os objetos `guard-put` e `guard-get` controlam os acessos ao objeto `buff`, garantindo que os limites de armazenamento do "buffer" não sejam ultrapassados.

```
Maude> (red modelCheck(initial, []~ overflow) .)
result Bool :
  true

Maude> (red modelCheck(initial, []~ underflow) .)
result Bool :
  true
```

4.4 A aplicação leitores e escritores

Outra aplicação relacionada ao compartilhamento de recursos entre processos concorrentes é a aplicação dos leitores e escritores [55]. Nesta aplicação, leitores e escritores disputam acesso a um recurso comum chamado "buffer". Os leitores acessam o "buffer" para ler itens nele armazenados. Os leitores podem acessar o "buffer" de forma concorrente com outros leitores. Os escritores acessam o "buffer" para escrever itens neste. Os escritores devem acessar o "buffer" de forma exclusiva, ou seja, quando um escritor estiver acessando o "buffer", nenhum outro escritor ou leitor pode acessar o "buffer". Na realidade, esta aplicação é uma variação da aplicação dos produtores e consumidores. Na aplicação dos produtores e consumidores os produtores não são apenas escritores, eles precisam ler a posição livre no "buffer" para incluir um novo item e precisam determinar se o "buffer" tem espaço livre para uma nova inclusão. Analogamente, os consumidores não são apenas leitores, ao remover um item eles precisam ajustar as posições livres do "buffer" para permitir novas inclusões. [3] Como na aplicação dos leitores e escritores, os

[3]Obviamente os aspectos de implementação não são tratados no nível da arquitetura mas constituem, neste caso, a motivação para a apresentação de ambas as variações de problemas relacionados ao compartilhamento de recursos.

leitores não escrevem no "buffer", podemos permitir acessos simulatâneos de leitores, o que nos permite mais eficiência na utilização do recurso. Os escritores, por sua vez, por alterarem o estado do "buffer", continuam sendo obrigados a acessá-lo de forma exclusiva.

Na Figura 13, apresentamos a arquitetura `READERS-WRITERS`, uma possível arquitetura para esta aplicação. Foram declarados três módulos, `READER`, `WRITER` e `BUFFER`, que representam, respectivamente, os processos leitores, escritores e o recurso compartilhado. Os conectores `WANT-WRITE` e `WANT-READ` contabilizam o número de leitores e escritores que desejam acessar o "buffer" por meio das variáveis `want-read` e `want-write`. Os connectores `COUNT-READ` e `COUNT-WRITE` controlam o acesso ao "buffer" da seguinte forma:

- Acessos concorrentes de leitores ao "buffer" são permitidos. Quando um novo leitor acessa o "buffer", a variável `want-read` é decrementada e a variável `readers`, que controla o número de leitores que estão acessado o "buffer", é incrementada. Ao término do acesso de um leitor, a variável `readers` é decrementada.

- Os escritores devem acessar o "buffer" de forma exclusiva. Isto é, apenas um escritor pode acessar o "buffer" por vez, e nenhum leitor pode estar lendo do "buffer" enquanto um escritor estiver escrevendo no "buffer". A variável *bool* `writers` declarada no conector `COUNT-WRITE`, quando verdadeira, indica que um escritor está escrevendo no "buffer".

- A prioridade no acesso ao "buffer" é dos leitores, até que não existam mais leitores interessados em ler. Este comportamento é implementado no bloco *before* do guarda declarado no connector `COUNT-READ`. O valor da variável `turn` será alterado para um, passando a prioridade para os escritores apenas quando não houverem mais leitores interessados em acessar o "buffer", ou seja, `want-read == 0`.

- Sempre que os escritores têm a prioridade no acesso, `turn == 1`, ou não existam leitores interessados em ler, `want-read == 0`, um escritor pode acessar o "buffer". Esta restrição é implementada na expressão do guarda declarado no conector `COUNT-WRITE`.

- Sempre que um escritor termina de escrever no "buffer", a prioridade do acesso é retornada aos leitores, ou seja, o valor da variável `turn` é alterado para zero.

Na Figura 14, apresentamos o módulo de execução `RW-EXEC`. As regras de reescrita `reading` e `writing` determinam que as instâncias de `READER` e `WRITER` devem iniciar nova interação tão logo recebam a mensagem de *done*. Para as análises que serão apresentadas

```
module READER {
 out port r@read ;
}

module WRITER {
 out port w@write ;
}

module BUFFER {
 in port buffer@read ;
 in port buffer@write ;
}

connector WANT-READ {
 var int want-read = 0 ;

 in port in-want-read ;
 out port out-want-read ;

 interaction {
  in-want-read > guard( TRUE ) {
    before { want-read = want-read + 1 ; }
  }
  > out-want-read ;
 }
}

connector WANT-WRITE {
 var int want-write = 0 ;

 in port in-want-write ;
 out port out-want-write ;

 interaction {
  in-want-write > guard( TRUE ) {
   before { want-write = want-write + 1 ; }
  }
  > out-want-write ;
 }
}

application READERS-WRITERS {
 instantiate WRITER as writer1 ;
 ...
 link reader1.r@read to wr.in-want-read ;
 ...
 bind int cr.cr@want-read to wr.want-read ;
 bind int cr.cr@want-write to ww.want-write ;
 bind bool cr.cr@writing to cw.writing ;
 bind int cw.cw@want-read to wr.want-read ;
 bind int cw.cw@want-write to ww.want-write ;
 bind int cw.cw@readers to cr.readers ;
 bind int cw.cw@turn to cr.turn ;
}
```

```
connector COUNT-READ {
  var int turn = 0 ;
  var int readers = 0 ;

  staterequired int cr@want-read ;
  staterequired int cr@want-write ;
  staterequired bool cr@writing ;

  in port in-count-read ;
  out port out-count-read ;

  interaction {
   in-count-read >
   guard(cr@writing == FALSE &&
    (cr@want-write == 0 || turn == 0)) {
    before {
     cr@want-read = cr@want-read - 1 ;
     readers = readers + 1 ;
     if (cr@want-read == 0) { turn = 1 ; }
    }
    after { readers = readers - 1 ; }
   } > out-count-read ;
 }
}

connector COUNT-WRITE {
 var bool writing = FALSE ;

 staterequired int cw@want-read ;
 staterequired int cw@want-write ;
 staterequired int cw@readers ;
 staterequired int cw@turn ;

 in port in-count-write ;
 out port out-count-write ;

 interaction {
  in-count-write >
  guard((cw@readers == 0 && writing == FALSE)
        && (cw@want-read == 0 || cw@turn == 1)) {
   before {
    cw@want-write = cw@want-write - 1 ;
    writing = TRUE ;
   }
   after {
    writing = FALSE ;
    cw@turn = 0 ;
   }
  } > out-count-write ;
 }
}
```

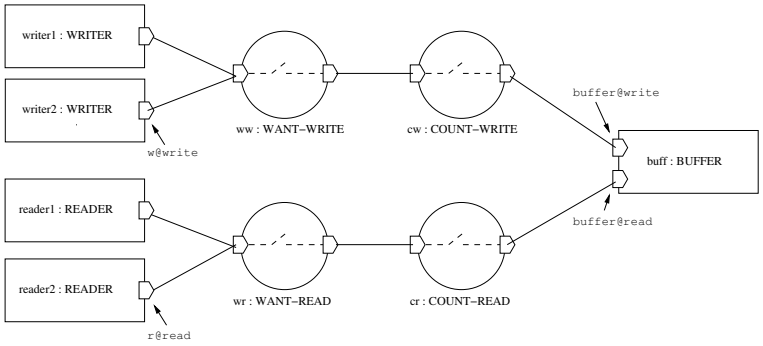

Figura 13: Arquitetura para a aplicação dos leitores e escritores

o estado interno da instância de `BUFFER` é irrelevante, sendo assim, as regras `buffer-write` e `buffer-read` apenas definem que mensagens *do* serão reescritas para mensagens *done*, sem qualquer alteração no estado do objeto que recebeu a mensagem. Por fim, o estado inicial da aplicação é declarado como sendo a topologia da arquitetura com as mensagens iniciais de *do* para as instâncias dos módulos `READER` e `WRITER`.

```
(omod RW-EXEC is
 inc READERS-WRITERS .

 var O : Oid .
 var IT : Interaction .

 rl [reading] :
    < O : READER | > done(O, r@read, IT)  =>
    < O : READER | > do(O, r@read, none) .

 rl [writing] :
    < O : WRITER | > done(O, w@write, IT) =>
    < O : WRITER | > do(O, w@write, none) .

 rl [buffer-write] :
    < O : BUFFER | > do(O, buffer@write, IT) =>
    < O : BUFFER | > done(O, buffer@write, IT) .

 rl [buffer-read] :
    < O : BUFFER | > do(O, buffer@read, IT)  =>
    < O : BUFFER | > done(O, buffer@read, IT) .

 op initial : -> Configuration .
 eq initial = topology
              do(writer1, w@write, none) do(writer2, w@write, none)
              do(reader1, r@read, none) do(reader2, r@read, none) .
endom)
```

Figura 14: Módulo de execução da aplicação leitores e escritores

Vamos analisar agora a arquitetura `READERS-WRITERS` em relação às seguintes propriedades:

Propriedade 4.4.1 *O objeto* `buff`*, instância de* `BUFFER`*, não pode receber, simultaneamente, mensagens de* `buffer@write` *e* `buffer@read`*. Mensagens simultâneas de* `buffer@write` *também não podem ser recebidas por* `buff`*. Ou seja, o "buffer" não pode receber acessos concorrentes de leitores e escritores e também não pode receber acessos concorrentes de escritores.*

Propriedade 4.4.2 *O objeto* `buff` *pode receber, simultaneamente, mais de uma mensagem de* `buffer@read`*. Ou seja, acessos concorrentes de leitores ao* buffer *são permitidos.*

Propriedade 4.4.3 *Em nenhum estado da computação deve ser verdade que todas as instâncias de módulos estão bloqueadas e a computação impedida de prosseguir. Isto é, a arquitetura* `READERS-WRITERS` *deve estar livre de "deadlock".*

Propriedade 4.4.4 *Se* `buff` *está recebendo alguma mensagem de alguma instância de* `READER` *e não existem mensagens de instâncias de* `WRITER` *bloqueadas, então não existe motivo para novas mensagens de instâncias de* `READER` *serem bloqueadas. Ou seja, caso não existam escritores desejando acessar o "buffer", então os leitores permanecem com a prioridade no acesso.*

Antes de analisarmos as propriedades acima, precisamos definir o módulo de análise para a arquitetura `READERS-WRITERS`. O módulo de análise `RW-VER` é apresentado na Figura 15. As proposições `writing` e `reading`, parametrizadas por um identificador de objeto, se verificam nos estados onde o objeto `buff`, instância do módulo `BUFFER`, esteja recebendo, respectivamente, mensagem de `buffer@write` ou `buffer@reader` do objeto que parametriza a proposição. Com auxílio das proposições parametrizadas `writing` e `reading`, definimos as fórmulas em LTL `writing` e `reading`, constantes do tipo `Formula`, que se verificam nos estados onde alguma instância de `WRITER` ou `READER` esteja, respectivamente, enviando mensagem de `buffer@write` ou `buffer@read` para o objeto `buff`. A proposição `suspendWriters` é verdadeira nos estados onde o atributo `want-read` for maior que zero, ou seja, exista no sistema uma ou mais mensagens do objeto `ww`, que representa uma instância do conector `WANT-WRITE`, encaminhadas para o objeto `cw`, que representa uma instância do conector `COUNT-WRITE`. A proposição `suspendReaders` é verdadeira nos estados onde, de forma semelhante, existam mensagens do objeto `wr`, que representa uma instância do conector `WANT-READ`, encaminhadas para o objeto `cr`, que representa uma instância do conector `COUNT-READ`. A propriedade 4.4.1 pode então ser expressa pela fórmula `[]~ raceCond`, onde `raceCond`, definida equacionalmente com auxílio da proposição parametrizada `writing` e das fórmulas `writing` e `reading`, é válida nos estados onde o objeto `buff` esteja recebendo duas mensagens de `buffer@write`, dos objetos `writer1` e `writer2`, ou uma mensagem de `buffer@write`, de alguma instância de `WRITER`, e uma mensagem de `buffer@read`, de alguma instância de `READER`. Para análise da propriedade 4.4.4, definimos a fórmula em LTL atribuída a costante `priority-readers`, do tipo `formula`, expressa que em todos os caminhos de execução da arquitetura, sempre que houver um estado onde:

1. existam mensagens encaminhadas pelo objeto `cr` para `buff` ainda não respondidas, ou seja, a proposição `reading` seja válida;

2. o objeto `ww` não tenha recebido mensagens, `~suspendWriters`;

3. existam mensagens encaminhadas do objeto `wr` ainda não recebidas pelo objeto `cr`, ou seja, a proposição `suspendReaders` seja válida.

Então, deste estado em diante, o objeto `cr` não aceitará novas mensagens até que receba as respostas de suas mensagens já encaminhadas para `buff`.

```
(omod RW-VER is
 inc RW-EXEC .
 inc MODEL-CHECKER .

 subsort Configuration < State .

 var C : Configuration .
 var O : Oid .
 var P : PortOutId .
 var IT : Interaction .
 vars N M : Int .

 ops writing reading : Oid -> Prop .
 eq C < buff : BUFFER | > send(buff, buffer@write, IT :: [O, P])
    |= writing(O) = true .
 eq C < buff : BUFFER | > send(buff, buffer@read, IT :: [O, P])
    |= reading(O) = true .

 ops writing reading : -> Formula .
 eq writing = (writing(writer1) \/ writing(writer2)) .
 eq reading = (reading(reader1) \/ reading(reader2)) .

 op raceCond : -> Formula .
 eq raceCond = ((writing(writer1) /\ writing(writer2)) \/
                (writing /\ reading)) .

 ops suspendWriters suspendReaders : -> Prop .
 ceq C < wr : WANT-READ | want-read : N > |= suspendReaders = true   if N > 0 .
 ceq C < ww : WANT-WRITE | want-write : N > |= suspendWriters = true if N > 0 .

 op priority-readers : -> Formula .
 eq priority-readers =
    [] ((reading /\ ~ suspendWriters /\ suspendReaders) ->
        (suspendReaders U ~ reading)) .
endom)
```

Figura 15: Módulo de análise da arquitetura `READERS-WRITERS`

Para analisarmos a arquitetura `READERS-WRITERS` em relação a propriedade 4.4.1, podemos utilizar uma busca em largura. A busca abaixo, após explorar todos os estados da árvore de estados das computações da arquitetura, não localiza nenhum estado onde o objeto `buff` esteja recebendo mensagens das instâncias de `READER`, isto é, $readers > 0$, e também esteja recebendo mensagens de alguma instância de `WRITER`, isto é, $writing = true$. O parâmetro `[1]`, passado para o comando *search*, limita a uma o número de soluções que desejamos obter. Isto porque bastaria a localização de um estado por esta busca para invalidar a propriedade 4.4.1.

```
Maude> (search [1] initial =>* C:Configuration
         < cw : COUNT-WRITE | writing : true , AS:AttributeSet >
         < cr : COUNT-READ | readers : N:Int , AS':AttributeSet >
        such that N:Int > 0  .)

No solution.
```

Na busca supra não comprovamos uma das restrições da propriedade 4.4.1, qual seja, de que mensagens simultâneas de `buffer@write` das duas instâncias de `WRITER` também não podem ser recebidas pelo objeto `buff`. Para efetivamente comprovar se a propriedade 4.4.1 é válida na arquitetura, podemos submeter a fórmula `[]~raceCond` ao verificador de modelos de Maude:

```
Maude> (red modelCheck(initial, []~ raceCond) .)
result Bool :
  true
```

A propriedade 4.4.2 é verificada com uma busca por estados onde existam pelo menos duas mensagens `buffer@read` encaminhadas ao objeto `buff`:

```
Maude> (search [1] initial =>* C:Configuration
                   send(buff, buffer@read, IT1:Interaction)
                   send(buff, buffer@read, IT2:Interaction) .)

Solution 1
C:Configuration <- < buff : BUFFER | none >
< cr : COUNT-READ | cr@want-read : st(0,unchanged),cr@want-write : st(0,
    unchanged),cr@writing : st(false,unchanged),readers : 2,turn : 1 >
< cw : COUNT-WRITE | cw@readers : st(2,unchanged),cw@turn : st(1,unchanged),
    cw@want-read : st(0,unchanged),cw@want-write : st(0,unchanged),writing :
    false >
< reader1 : READER | r@read-status : locked >
< reader2 : READER | r@read-status : locked >
< wr : WANT-READ | want-read : 0 >
< writer1 : WRITER | w@write-status : unlocked >
< writer2 : WRITER | w@write-status : unlocked >
< ww : WANT-WRITE | want-write : 0 >
do(writer1,w@write,none)
do(writer2,w@write,none);
IT1:Interaction <- [cr,out-count-read]::[wr,out-want-read]::[reader1,r@read];
IT2:Interaction <- [cr,out-count-read]::[wr,out-want-read]::[reader2,r@read]
```

Na busca acima, foi localizado pelo menos um estado onde o objeto `buff` está recebendo duas mensagens de `buffer@read`. A existência deste estado é suficiente para comprovar que a propriedade 4.4.2 é válida na arquitetura.

Como apresentado na Seção 4.3, a propriedade 4.4.3 pode ser verificada com uma busca por estados *canônicos finais*. Isto porque, também na arquitetura `READERS-WRITERS`, o comportamento interno das instâncias de `WRITER` e `READER` é constantemente realimentar o sistema com novas mensagens, sempre que recebem a resposta da última mensagem encaminhada. Na busca a seguir nenhum estado canônico final foi localizado. Desta forma, a propriedade 4.4.3 é válida na arquitetura.

```
Maude> (search initial =>! C:Configuration . )

No solution.
```

A propriedade 4.4.4 descreve uma característica desejável para todos os possíveis caminhos de execução da arquitetura, onde a ocorrência de um estado determina os estados que podem ocorrer a seguir. Neste caso, conforme descrito na Seção 2.4, a exploração de estados com o comando *search* não seria útil. Para a análise de propriedades de caminhos de execução, precisamos utilizar o verificador de modelos.

A fórmula `priority-readers`, definida no módulo `RW-VER` na Figura 15, descreve a negação da propriedade 4.4.4. Então, ao submetermos a fórmula `priority-readers` ao verificador de modelos, podemos obter a comprovação da propriedade 4.4.4 através de um contra-exemplo:

```
Maude> (red modelCheck(initial, priority-readers) .)
result ModelCheckResult :
  counterexample(...
{< buff : BUFFER | none >
< cr : COUNT-READ | cr@want-read : st(1,unchanged),cr@want-write : st(0,
    unchanged),cr@writing : st(false,unchanged),readers : 1,turn : 1 >
< cw : COUNT-WRITE | cw@readers : st(1,unchanged),cw@turn : st(1,unchanged),
    cw@want-read : st(1,unchanged),cw@want-write : st(0,unchanged),writing :
    false >
< reader1 : READER | r@read-status : locked >
< reader2 : READER | r@read-status : locked >
< wr : WANT-READ | want-read : 1 >
< writer1 : WRITER | w@write-status : locked >
< writer2 : WRITER | w@write-status : locked >
< ww : WANT-WRITE | want-write : 0 >
send(buff,buffer@read,[cr,out-count-read]::[wr,out-want-read]::[reader1,r@read])
send(cr,in-count-read,[wr,out-want-read]::[reader2,r@read])
send(ww,in-want-write,[writer1,w@write])
send(ww,in-want-write,[writer2,w@write]),'BUFFER-recevingAndDo-buffer@read}
...
{ack([cr,out-count-read]::[wr,out-want-read]::[reader1,r@read])
< buff : BUFFER | none >
< cr : COUNT-READ | cr@want-read : st(0,unchanged),cr@want-write : st(0,
    unchanged),cr@writing : st(false,unchanged),readers : 2,turn : 1 >
< cw : COUNT-WRITE | cw@readers : st(2,unchanged),cw@turn : st(1,unchanged),
    cw@want-read : st(0,unchanged),cw@want-write : st(0,unchanged),writing :
    false >
< reader1 : READER | r@read-status : locked >
< reader2 : READER | r@read-status : locked >
< wr : WANT-READ | want-read : 0 >
< writer1 : WRITER | w@write-status : locked >
< writer2 : WRITER | w@write-status : locked >
< ww : WANT-WRITE | want-write : 0 >
send(buff,buffer@read,[cr,out-count-read]::[wr,out-want-read]::[reader2,
```

```
    r@read])
send(ww,in-want-write,[writer1,w@write])
send(ww,in-want-write,[writer2,w@write]),'BUFFER-recevingAndDo-buffer@read}..)
```

O primeiro estado do contra-exemplo acima contém uma mensagem `buffer@read`, enviada pelo objeto `reader1`, sendo recebida pelo objeto `buff`. Enquanto isso, neste mesmo estado, outra mensagem de leitura, encaminhada pelo `reader2` aguarda para ser recebida pelo objeto `cr`. No estado seguinte, a mensagem de `reader2` foi encaminhada por `cr` para `buff` antes que a resposta de `buff` para a mensagem de `reader1` fosse recebida por `cr`. Devido ao tamanho do contra-exemplo, os demais estados listados no contra-exemplo foram removidos. Outra leitura do contra-exemplo pode ser feita. O atributo `readers` também foi incrementado para dois, mostrando que uma nova mensagem de `buffer@read` foi encaminhada para `buff` por `cr`, antes que a última mensagem `buffer@read` fosse respondida. Nos estados destacados, podemos observar também que não existem mensagens de instâncias de `WRITER` bloqueadas.

A existência de mensagens de instâncias de `READER`, encaminhadas para `cr`, enquanto `buff` recebe uma mensagem de `buffer@read`, não obriga que tais mensagens sejam recebidas por `buff` antes que a mensagem `buffer@read` seja respondida. A propriedade 4.4.4 determina apenas que tais mensagens *possam* ser recebidas, ou seja, não estejam bloqueadas por `cr`. Isto pode ser comprovado pelo contra-exemplo que obtemos se a fórmula

```
[] ( (reading /\ ~ suspendWriters /\ suspendReaders) -> (~ suspendReaders R reading) )
```

for submetida ao verificador de modelos.

4.5 A aplicação ceia de filósofos

Tratamos, nas seções anteriores, de dois casos onde processos concorrentes disputam acesso a um mesmo recurso. No caso dos produtores e consumidores, todos os acessos ocorrem de forma exclusiva. No caso dos leitores e escritores é permitido o acesso simultâneo apenas dos leitores. A aplicação da ceia de filósofos pode ser entendida com um caso geral de compartilhamento de recursos [26] onde, ao invés de um único recurso estar sendo compartilhado entre vários processos, temos conjuntos de recursos compartilhados, cada um, por um grupo determinado de processos.

Esta aplicação pode assim ser descrita. Um grupo de filósofos permanece, indefinidamente, alternando entre as atividades de pensar, ter fome e comer. Para comer, um

filósofo precisa sentar-se a uma mesa circular, pegar o garfo da sua esquerda e pegar o garfo da sua direita. Após comer, o filósofo larga os garfos, levanta da mesa e volta a pensar. Uma vez sentado à mesa, um filósofo só largará os garfos e voltará a pensar após comer.

Na Figura 16, apresentamos uma arquitetura possível para esta aplicação. Na Figura 17 a respectiva representação gráfica. Foram declarados os módulos `TABLE`, `FOOD`, `FORK` e `PHILOSOPHER` e os conectores `GET-FORK` e `GET-TABLE`. Nos módulos `FORK` e `TABLE`, que correspondem aos recursos compartilhados mesa e garfos, nenhuma porta de saída ou entrada é declarada. Cada instância destes módulos representará um recurso sendo compartilhado e sua utilização será controlada através dos conectores `GET-TABLE` e `GET-FORK` através de memória compartilhada. No módulo `TABLE` a variável `places` é iniciada com uma unidade a menos que o número de filósofos existentes. Como todos os filósofos precisam sentar-se à mesa para disputar os garfos, a falta de um lugar irá garantir que na arquitetura não ocorra "deadlock" [46, 50].

Ainda na Figura 16, o módulo `PHILOSOPHER` representa um filósofo, e sua porta de saída `phi@eat` será estimulada significando a vontade de comer do filósofo. O módulo `FOOD` representa a comida que os filósofos desejam acessar. Desta forma, os conectores estabelecem a ligação das portas de saída dos filósofos, com a porta de entrada `food@eat` de `FOOD`. No módulo da aplicação, `4-PHILOSOPHERS`, são criadas quatro instâncias de `FORK` e `PHILOSOPHER`, a instância `table` de `TABLE` e `food` de `FOOD`. São criadas também quatro instâncias do conector `GET-TABLE`. Cada uma destas, ao receber um estímulo em sua porta de entrada, passará a requisição adiante apenas após inspecionar a variável `places` de `table` e certificar-se que ainda existem lugares disponíveis na mesa. Como cada garfo será compartilhado entre dois filósofos, para cada instância de `FORK` são criadas duas instâncias de `GET-FORK`. O módulo `fork1` terá sua variável `available` inspecionada pelas instâncias `lfork1` e `rfork2` de `GET-FORK`. A amarração dos conectores `lfork1` e `rfork2` ao módulo `fork1` representa que o garfo à esquerda do filósofo 1, `phi1`, é o mesmo que está à direita do filósofo 2, `phi2`.

O módulo de execução para esta arquitetura é mostrado na Figura 18. O comportamento interno do módulo `PHILOSOPHER` é implementado pela regra `eat`. Os objetos instâncias de `PHILOSOPHER` são deixados em *loop*, constantemente iniciando uma interação ao término da anterior. O comportamento interno do módulo `FOOD` ao receber uma mensagem `food@eat` é simplesmente responder a mensagem sem qualquer alteração em seu estado interno.

Na Figura 19, apresentamos o módulo de análise para a arquitetura. Neste módulo

```
module PHILOSOPHER {
 out port phi@eat ;
}

connector GET-TABLE {
 staterequired int get-table@places ;
 in port get-table@in ;
 out port get-table@out ;

 interaction {
  get-table@in >
   guard(get-table@places > 0) {
    before {
     get-table@places = get-table@places - 1 ;
    }
    after {
     get-table@places = get-table@places + 1 ;
    }
   }
  > get-table@out ;
 }
}

connector GET-FORK {
 staterequired bool get-fork@available ;
 in port get-fork@in ;
 out port get-fork@out ;

 interaction {
  get-fork@in >
   guard(get-fork@available == TRUE) {
    before {
     get-fork@available = FALSE ;
    }
    after {
     get-fork@available = TRUE ;
    }
   }
  > get-fork@out ;
 }
}
```

```
module FOOD {
 in port food@eat ;
}

module TABLE {
 var int places = 3 ;
}

module FORK {
 var bool available = TRUE ;
}

application 4-PHILOSOPHERS {
 instantiate PHILOSOPHER as phi1 ;
 ...
 instantiate FOOD as food ;
 instantiate TABLE as table ;
 instantiate FORK as fork1 ;
 ...
 instantiate GET-TABLE as gtable1 ;
 ...
 instantiate GET-FORK as lfork1 ;
 ...
 instantiate GET-FORK as rfork1 ;
 ...
 link phi1.phi@eat to gtable1.get-table@in ;
 link gtable1.get-table@out to lfork1.get-fork@in ;
 link lfork1.get-fork@out to rfork1.get-fork@in ;
 link rfork1.get-fork@out to food.food@eat ;
 ...
 bind bool lfork1.get-fork@available to fork1.available ;
 bind bool rfork1.get-fork@available to fork4.available ;
 bind bool lfork2.get-fork@available to fork2.available ;
 bind bool rfork2.get-fork@available to fork1.available ;
 bind bool lfork3.get-fork@available to fork3.available ;
 bind bool rfork3.get-fork@available to fork2.available ;
 bind bool lfork4.get-fork@available to fork4.available ;
 bind bool rfork4.get-fork@available to fork3.available ;
 bind int gtable1.get-table@places to table.places ;
 bind int gtable2.get-table@places to table.places ;
 bind int gtable3.get-table@places to table.places ;
 bind int gtable4.get-table@places to table.places ;
}
```

Figura 16: Arquitetura 4-PHILOSOPHERS para aplicação ceia de filósofos

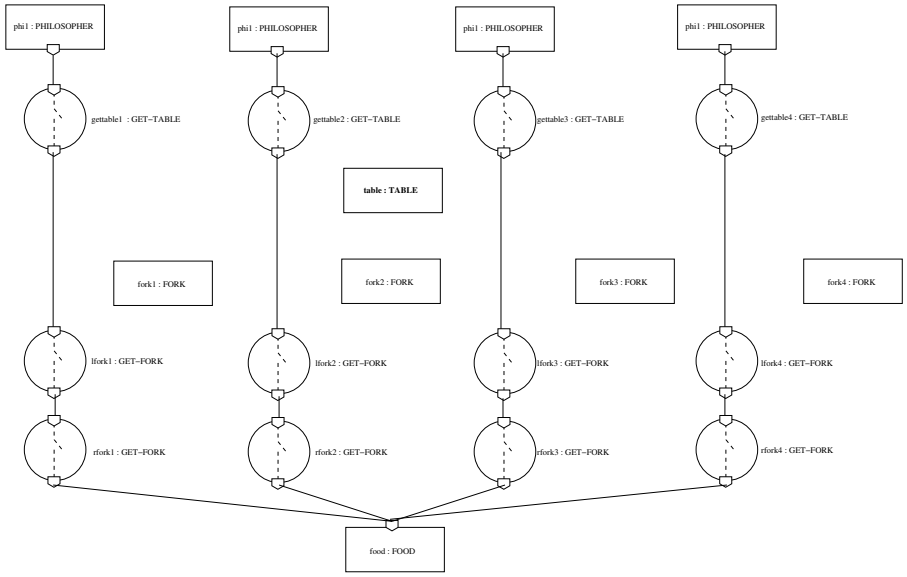

Figura 17: Diagrama da arquitetura `4-PHILOSOPHERS`

```
(omod PHI-EXEC is
 inc 4-PHILOSOPHERS .

 var C : Configuration .
 var O : Oid .
 var IT : Interaction .

 op initial : -> Configuration .
 op init : Oid -> Configuration .

 eq initial = topology init(phi1) init(phi2) init(phi3) init(phi4) .

 eq init(O) = do(O, phi@eat, none) .

 rl [eat] :
   < O : PHILOSOPHER | > done(O, phi@eat, IT) =>
   < O : PHILOSOPHER | > init(O) .

 rl [eating] :
   < food : FOOD | > do(food, food@eat, IT) =>
   < food : FOOD | > done(food, food@eat, IT) .
endom)
```

Figura 18: Módulo de execução da aplicação ceia de filósofos

é declarada apenas a proposição `eating`, parametrizada pelo identificador de um objeto. A proposição `eating(O)` é válida nos estados onde o objeto `food` esteja recebendo uma mensagem originada do objeto `O`.

```
(omod PHI-VER is
 inc PHI-EXEC .
 inc MODEL-CHECKER .

 subsort Configuration < State .

 var C : Configuration .
 var O : Oid .
 var P : PortId .
 var IT : Interaction .

 op eating : Oid -> Prop .
 eq C < food : FOOD | > send(food, food@eat, IT :: [O, P]) |= eating(O) = true .
endom)
```

Figura 19: Módulo de análise da aplicação ceia de filósofos

Como para as arquiteturas apresentadas nas seções anteriores, desejamos garantir as seguintes propriedades.

Propriedade 4.5.1 *O objeto* `food`*, instância do módulo* `FOOD`*, não pode receber, simultaneamente, mensagens de duas instâncias de* `PHILOSOPHER` *adjacentes. Isto é, cada garfo só poderá estar sendo utilizado por um filósofo a cada instante.*

Como cada garfo é compartilhado por filósofos adjacentes, a proposição `eating` não pode ser válida para dois objetos adjacentes em um mesmo instante.

```
Maude>(red modelCheck(initial,[]~(eating(phi1) /\ eating(phi2))) .)
result Bool :
  true
```

A utilização dos objetos `phi1` e `phi2` é completamente intercambiável, isto é, qualquer par de identificadores de objetos que corresponda a filósofos adjacentes poderia ser usado. Logo, comprovamos o acesso exclusivo aos recursos.

Propriedade 4.5.2 *Em nenhum estado da computação deve ser verdade que todas as instâncias de módulos estão bloqueadas e a computação impedida de prosseguir. Isto é, a arquitetura deve estar livre de "deadlock".*

Como todas as instâncias de `PHILOSOPHER` são deixadas em *loop*, reenviando mensagens tão logo recebam uma resposta, podemos utilizar a mesma técnica usada nas seções anteriores para comprovar a ausência de "deadlock":

```
Maude> (search initial =>! C:Configuration .)

No solution.
```

Na busca por estados finais mostrada acima, nenhum estado final é encontrado, comprovando a propriedade 4.5.2.

5 Avaliação de Maude CBabel tool e trabalhos relacionados

Neste capítulo, apresentamos uma avaliação da ferramenta Maude CBabel tool. Os comentários aqui apresentados são resultantes de nossas experiências com o uso da ferramenta durante a implementação dos estudos de caso apresentados no Capítulo 4 e de outros experimentos realizados durante nossa pesquisa.

Este capítulo está estruturado da seguinte forma. No decorrer da Seção 5.1:

- Nas Seções 5.1.1, 5.1.2 e 5.1.4, destacamos a preservação a modularidade de descrições arquiteturais em CBabel no mapeamento destas para lógica de reescrita e mostramos o resultado positivo desta preservação da modularidade da descrição arquitetural: (i) na direta interpretação dos resultados das análises para adaptação da descrição arquitetural e (ii) na possibilidade de realização das análises de arquiteturas de forma modular.

- Na Seção 5.1.2, mostramos como a técnica de abstração equacional [54] pode ser utilizada para contornar possíveis problemas de explosão de estados durante as análises de arquiteturas em Maude CBabel tool.

- Ainda na Seção 5.1.2, mostramos os cuidados que devem ser tomados na adaptação de arquiteturas descritas na sintaxe original de CBabel, proposta por Sztajnberg em [46], para a sintaxe aceita por Maude CBabel tool.

- Na Seção 5.1.3, utilizando como exemplo a arquitetura dos leitores e escritores apresentada na Seção 4.4, descrevemos como realizar a análise de "starvation" com o verificador de modelos de Maude.

- Utilizando como exemplo a arquitetura para o problema de Ceia dos Filósofos apresenta por Sztajnberg em [46], mostramos na Seção 5.1.4 como avaliar se um componente pode ser removido da arquitetura.

Na Seção 5.2, apresentamos alguns trabalhos encontrados na literatura relacionados à nossa pesquisa. Finalmente, na Seção 5.3, comparamos nossa abordagem com estes trabalhos.

5.1 Avaliação dos estudos de caso

Durante a implementação e análise dos estudos de caso apresentados no Capítulo 4, diferentes aspectos da ferramenta Maude CBabel tool foram analisados. Como nosso objetivo naquele capítulo era apresentar a ferramenta, adiamos para esta seção a análise crítica destes aspectos. Procuramos aqui identificar os pontos positivos e negativos de ferramenta revisitando os exemplos apresentados no Capítulo 4.

5.1.1 Máquina de venda

Um dos aspectos importantes de Maude CBabel tool é sua capacidade de preservar a modularidade da descrição da arquitetura no nível de abstração onde são realizadas as análises. Modularidade é um fator importante para qualquer linguagem de descrição de arquiteturas de "software", pois, permite, dentre outras coisas, reconfiguração da arquitetura e reaproveitamento de componentes entre arquiteturas [2]. Ao preservar a modularidade da descrição arquitetural, Maude CBabel tool permite que as análises sobre a arquitetura possam ser feitas de forma modular, aplicando técnicas de prova modular. A preservação da modularidade facilita ainda a compreensão dos resultados das simulações e análises. O preço pago pela preservação da modularidade é, no entanto, o elevado número de interações entre componentes e, conseqüentemente, um elevado número de estados de computação atingíveis durante as simulações da arquitetura e durante as explorações dos estados nas análises realizadas.

Na Seção 4.2, apresentamos uma arquitetura em CBabel para uma máquina de venda de bolos e maçãs. A arquitetura apresentada foi inspirada em uma especificação em Maude da mesma máquina [19]. Para ilustrar o impacto da modularidade no número de estados de computação durante as simulações da arquitetura, faremos uma comparação entre a especificação em Maude produzida por CBabel com a especificação original em Maude. A comparação tem por objetivo ilustrar a diferença entre o nível de abstração de cada uma delas e como o elevado número de interações entre componentes CBabel, devido a preservação da modularidade da descrição arquitetural, resulta em um elevado número de estados de computação.

Em Maude, a especificação da máquina de vendas pode ser dividida em dois módulos. O módulo funcional `VENDING-MACHINE-SIGNATURE` declara os tipos de dados e os aspectos estáticos da máquina, sua assinatura:

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item Marking .
  subsorts Coin Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: null] .
  op null : -> Marking .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
  op c : -> Item [format (b! o)] .
endfm
```

O módulo de sistema `VENDING-MACHINE` importa o módulo `VENDING-MACHINE-SIGNATURE` e declara as regras de reescrita que definem o comportamento da máquina:

```
mod VENDING-MACHINE is
  protecting VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```

A separação em um módulo funcional e um módulo de sistema facilita a especificação da máquina. Além de tornar explícita a assinatura da máquina, permite que a mesma assinatura possa ser compartilhada por diferentes módulos de sistemas que, possivelmente, definam diferentes comportamentos para a máquina.

Os módulos apresentados acima especificam uma máquina de vendas concorrente, de funcionamento idêntico ao descrito na Seção 4.2. Da mesma forma, um bolo custa um dólar e uma maçã três "quarters". A mesma limitação quanto à contabilização apenas de dólares para venda de mercadorias existe, ou seja, quatro "quarters" precisam primeiro ser trocados por um dólar, antes da venda de um bolo ou maçã (neste último caso, com um troco de um "quarter"). Os *botões* da máquina são representados por cada uma das regras de reescrita do módulo `VENDING-MACHINE`. A máquina é dita concorrente, pois permite, se providos os *recursos* necessários, que mais de um botão possa ser pressionado de uma vez, ou seja, várias regras podem ser aplicadas simultaneamente. O *estado* do sistema é representado por um termo do *sort* `Marking`, um *multiset* de itens e moedas, construído pela operação associativa e comutativa, `op __ : Marking Marking -> Marking [assoc comm id : null]`.

Deve-se observar que a máquina é concorrente, embora o interpretador de Maude seja seqüencial. As transições concorrentes de estado são simuladas pelas correspondentes reescritas em "interleaving". Uma implementação concorrente de Maude permitiria que várias regras de reescrita fossem efetivamente aplicadas concorrentemente [19].

Na Seção 4.2, para testar o funcionamento da máquina, executamos uma busca, a partir de um estado inicial onde um dólar e três "quarters" foram fornecidos, por estados onde um bolo e uma maçã tenham sido entregues pela máquina. Para analisar o número de estados percorridos durante esta busca, precisamos contornar a limitação do comando *search* de Full Maude em não exibir o número de estados percorridos na busca. Para isso, utilizando o comando `show all` de Full Maude, geramos a versão (core) Maude do módulo `VM-EXEC`. Após carregar no (core) Maude o módulo resultante, podemos realizar a busca. Lembramos da Seção 4.2, que as mensagens no termo inicial da busca correspondem ao pressionamento simultâneo dos botões.

```
Maude> search in VM-EXEC : topology do(bt-change,change,none) do(bt-ad,add-$,none)
       copy(do(bt-aq,add-q,none), 3) do(bt-bc,buy-cake,none)
       do(bt-ba,buy-apple, none) =>!
       < slot : SLOT | apples : 1 , cakes : 1 > C:Configuration .

Solution 1 (state 583)
states: 584  rewrites: 38789 in 1249ms cpu (1277ms real) (31035 rewrites/second)
C:Configuration --> < bt-ad : ADD-DOLLAR | none >
< bt-aq : ADD-QUARTER | none >
< bt-ba : BUY-APPLE | none >
< bt-bc : BUY-CAKE | none >
< bt-change : CHANGE | none >
< con-cd : COUNT-DOLLAR | dollars : 0 >
< con-change : MAKE-CHANGE | ch@dollars : st(0, unchanged),ch@quarters : st(0,
    unchanged) >
< con-cq : COUNT-QUARTER | quarters : 0 >
< con-sa : SOLD-APPLE | sa@dollars : st(0, unchanged),sa@quarters : st(0,
    unchanged) >
< con-sc : SOLD-CAKE | sc@dollars : st(0, unchanged) >

No more solutions.
states: 584  rewrites: 38789 in 1251ms cpu (1390ms real) (30986 rewrites/second)
```

Para localizar as duas soluções acima, a busca percorreu 584 estados de computação. Cada um dos 584 estados corresponde a uma troca de mensagem entre os objetos no sistema, as instâncias dos módulos e conectores da arquitetura. Podemos reproduzir a busca acima no módulo `VENDING-MACHINE`:

```
Maude> search in VENDING-MACHINE : $ q q q =>! a c M:Marking .
search in VENDING-MACHINE-0 : $ q q q =>! a c M:Marking .
```

```
Solution 1 (state 4)
states: 6  rewrites: 5 in 2ms cpu (2ms real) (2500 rewrites/second)
M:Marking --> null

No more solutions.
states: 6  rewrites: 5 in 4ms cpu (4ms real) (1250 rewrites/second)
```

No módulo `VENDING-MACHINE` foram percorridos apenas 6 estados para obtenção da mesma comprovação de que um bolo e uma maçã podem ser obtidos quando são fornecidos um dólar e três "quarters".

As especificações certamente diferem em certos aspectos. Por exemplo, a troca de quatro "quarters" por um dólar, no módulo `VENDING-MACHINE`, ocorre espontaneamente, isto é, sempre que existirem quatro "quarters" no termo sendo reescrito. Em compensação, no módulo `VM-EXEC`, uma troca só será tentada se for *solicitada*, isto é, se existir uma mensagem `do(change, change, none)` para a instância `change` do módulo `CHANGE`. Mesmo assim, pode-se observar que a diferença do número de estados no sistema é de duas ordens de grandeza.

Para ilustrar mais ainda esta diferença, vamos considerar a compra de um bolo com o fornecimento de um dólar. No módulo `VENDING-MACHINE`, o procedimento é realizado com apenas uma reescrita (transição de estado):

```
Maude> rew in VENDING-MACHINE : $ .
rewrites: 1 in 1ms cpu (2ms real) (500 rewrites/second)
result Item: c
```

No módulo `VM-EXEC`, o mesmo procedimento é feito com 186 passos de reescrita:

```
Maude> rew in VM-EXEC : topology do(bt-bc, buy-cake,none) do(bt-ad,add-$,none) .
rewrites: 186 in 27ms cpu (27ms real) (6643 rewrites/second)
result Configuration: < bt-ad : ADD-DOLLAR | none >
 < bt-aq : ADD-QUARTER | none >
 < bt-ba : BUY-APPLE | none >
 < bt-bc : BUY-CAKE | none >
 < bt-change : CHANGE | none >
 < con-cd : COUNT-DOLLAR | dollars : 0 >
 < con-change : MAKE-CHANGE | ch@dollars : st(0, unchanged),
                ch@quarters : st(0, unchanged)>
 < con-cq : COUNT-QUARTER | quarters : 0 >
 < con-sa : SOLD-APPLE | sa@dollars : st(0, unchanged),
            sa@quarters : st(0, unchanged) >
 < con-sc : SOLD-CAKE | sc@dollars : st(0, unchanged) >
 < slot : SLOT | apples : 0,cakes : 1 >
```

A diferença no número de estados de computação nas simulações de cada uma das especificações reflete o nível de abstração de cada uma. No módulo `VENDING-MACHINE`,

por exemplo, os *componentes* da máquina não são especificados, tão pouco as interações possíveis entre eles. O único aspecto capturado pela especificação Maude é o estado do sistema, representado pelo conjunto de moedas, isto é, dólares e "quarters", e itens vendidos, isto é, bolos e maçãs. Mais ainda, como todos os componentes da arquitetura CBabel são mantidos separados com a preservação da modularidade da arquitetura na sua transformação para lógica de reescrita, as buscas no módulo `VM-EXEC` devem explorar um elevado número de estados de computação, resultado do elevado número de interações entre os componentes da arquitetura.

5.1.2 Produtores e consumidores

Vamos avaliar aqui a escalabilidade de Maude CBabel tool com o aumento do número de instâncias de módulos produtores e consumidores nas arquiteturas apresentadas na Seção 4.3. Mostramos como a técnica de abstrações equacionais [54] pode ser utilizada para a realização de análises de forma modular evitando-se assim o problema de explosão de estados durante as análises. Em seguida, mostramos os cuidados na adaptação das descrições arquiteturas na sintaxe original de CBabel [46] para a sintaxe aceita por Maude CBabel tool.

5.1.2.1 Análise abstrata e modular em Maude CBabel tool

Na Seção 4.3 analisamos três arquiteturas para representar o problema dos produtores e consumidores. Com a execução de buscas e verificações de modelo, as arquiteturas foram analisadas em relação às propriedades de "deadlock" (Propriedade 4.3.1), "race condition" (Propriedade 4.3.2), "overflow" (Propriedade 4.3.3) e "underflow" (Propriedade 4.3.4). Embora as análises realizadas tenham sido úteis para detecção de erros e obtenção de um determinado grau de confiança sobre o funcionamento das arquiteturas, todas as análises foram executadas a partir de um determinado estado inicial, dois produtores e dois consumidores iniciando, simultaneamente, acessos a um "buffer".

Na implementação do comportamento interno dos módulos produtores e consumidores, não restringimos a quantidade de estímulos gerados nas portas de saída, ou seja, não limitamos o número de interações que devem ser realizadas nas portas dos módulos `PRODUCER` e `CONSUMER`. Com isso, tratamos todas as possíveis situações de sincronização entre as interações. Mesmo assim, ainda nos resta a dúvida sobre o que aconteceria se mais instâncias de produtores e consumidores estivessem presentes.

Para as propriedades apresentadas na Seção 4.3, assumimos que dois produtores e dois consumidores são suficientes para comprovação das propriedades. Embora sem nenhuma prova formal, consideramos que a presença de mais de duas instâncias de produtores ou consumidores não acrescenta, às análises realizadas, qualquer situação que já não exista com apenas dois produtores e dois consumidores. No entanto, por esta afirmação ser apenas de uma hipótese, realizamos alguns testes com aumento gradativo do número de instâncias de produtores e consumidores.

Com quatro produtores e quatro consumidores, para explorar todos os estados na árvore de computação da arquitetura, Maude necessita de aproximadamente um "gigabyte" de memória principal. Em um computador com pouca memória principal, o tempo de processamento torna-se elevado, na medida em que a memória secundária é utilizada para "swap".

A razão de Maude consumir tanta memória é o número de estados de computação que precisam ser explorados à medida que aumentamos as instâncias de produtores e consumidores. Na Tabela 3, apresentamos a relação do número de instâncias de produtores e consumidores, e a quantidade de estados na árvore de computação da arquitetura da Figura 9. Como dito na Seção 2.4, o comando *search* de Maude explora os estados da computação utilizando uma busca em largura, isto é, primeiro são visitados os estados atingíveis em um passo de reescrita a partir do estado inicial, depois os estados atingíveis em dois passos de reescrita e assim por diante. Para evitar que um estado seja percorrido mais de uma vez, Maude precisa guardar na memória todos os estados visitados. Para cada estado, também é guardado o *caminho* até ele a partir do estado inicial, informação necessária para o comando *show path* de Maude [19]. Mesmo considerando a eficiente implementação do comando *search* de Maude, uma busca pode estourar o limite disponível de memória da máquina.

Produtores	Consumidores	Estados
1	0	20
0	1	12
1	1	176
2	2	4.608
3	3	102.400

Tabela 3: Relação do número de instâncias de produtores e consumidores com os estados de computação da arquitetura `PC-MUTEX`

Em [53], na implementação e análises do protocolo NSPK, Ølveczky apresenta uma avaliação sobre o consumo de memória durante buscas em Maude. Assumindo que a cada

estado seja possível realizar pelo menos n diferentes passos de reescrita para um novo estado, então podemos atingir $n+1$ estados em um passo de reescrita. Em dois passos de reescrita, podemos atingir então n^2+n+1. Em três passos de reescrita n^3+n^2+n+1 etc. Em geral, $\sum_{i=0}^{d}(n^i)$, o que é maior que n^d em d passos de reescrita. Na aplicação dos produtores e consumidores, como as portas de saída dos módulos produtores e consumidores são síncronas, cada nova interação de uma instância deve aguardar o término da interação anterior, logo, podemos considerar que a cada estado o número de diferentes possibilidades de reescrita (n) é igual ao número de instâncias de produtores e consumidores em execução (número de interações em execução que podem *evoluir*). Considerando que uma interação completa, de um produtor ou consumidor, ocorre em aproximadamente 9 passos de reescrita, podemos estipular que na maioria das análises realizadas na Seção 4.3 foram necessários, pelo menos, quinze passos de reescrita para atingirmos um estado *ruim*, $d = 15$ (para completar duas interações executando concorrentemente a partir de reescritas em "interleaving", seriam necessárias 18 reescritas, escolhemos quinze como arredondamento). A árvore de estados da computação contém assim, para uma profundidade de quinze reescritas, 4^{15} estados, pouco mais de um bilhão de estados. Conforme reportado por Ølveczky e comprovado em nossos experimentos, com um milhão de estados já é possível ocorrer um estouro no limite de armazenamento da memória durante uma busca. Além disso, à medida que o consumo de memória aumenta, Maude passa a gastar mais tempo realizando a troca de dados entre memória principal e memória secundária (paginação), do que efetivamente realizando a exploração dos estados.

Nas arquiteturas analisadas para a aplicação produtores e consumidores, o aumento de uma até três instâncias de cada um dos módulos `PRODUCER` e `CONSUMER`, não alteraram os resultados das buscas ou verificações de modelos. Este resultado sugere que o número de instâncias dos módulos não interferem nos aspectos de coordenação da arquitetura. Todavia, o consumo de memória e tempo de processamento nas buscas realizadas nas arquiteturas com mais de três instâncias de cada módulo parecem limitar a utilização de Maude CBabel tool para arquiteturas com um número elevado de instâncias de módulos e conectores. No entanto, nestes casos, as análises podem ser realizadas a partir de *etapas* mais específicas das interações, focando assim em alguma *parte* específica da arquitetura. Podemos ilustrar a idéia com uma alternativa para a análise da arquitetura `PC-GUARDS-MUTEX`, apresentada na Figura 10, em relação a propriedade de que o limite máximo de armazenamento do "buffer" não deve ser ultrapassado.

Na busca a seguir, a exploração dos estados parte de um estado que representa uma etapa intermediária de uma interação iniciada por uma instância qualquer de `PRODUCER`. Ou

seja, ao invés de explorarmos os estados da computação a partir do início das interações nos módulos `PRODUCER` e `CONSUMER`, partirmos de um estado intermediário, onde a instância `gput` do conector `GUARD-PUT` está recebendo uma mensagem encaminhada por qualquer instância de `PRODUCER`. Como é irrelevante de qual instância de `PRODUCER` partiu a interação, o caminho da interação na mensagem é atribuído à variável do *sort* `Interaction`. O atributo `empty` igual a zero, recordando da Seção 4.3, significa que `buff` não pode receber mensagens para colocação de novos itens. O atributo `full` igual a dois significa que `buff` não pode receber mensagens para retirada de itens. A partir deste estado de partida, não deve ser possível atingirmos, com zero ou mais passos de reescrita, um estado onde exista uma mensagem `buffer@put` para o objeto `buff`:

```
Maude> (search [1]
        < gget : GUARD-GET | empty : 0 , gg@full : st(2,unchanged) >
        < gput : GUARD-PUT | full : 2 , gp@empty : st(0,unchanged) >
        < mutx : MUTEX | status : unlocked >
        send(gput,gp@in, IT1:Interaction)
        =>*
        C2:Configuration send(buff, buffer@put, IT3:Interaction) .)

No solution.
```

Mostramos, assim, que o conector `GUARD-PUT` impede que o limite máximo de armazenamento do "buffer" seja ultrapassado. A partir deste mesmo estado, no entanto, a retirada de itens não estará bloqueada pelo objeto `gget`, instância do conector `GUARD-GET`:

```
Maude> (search [1]
          < gget : GUARD-GET | empty : 0 , gg@full : st(2,unchanged) >
          < gput : GUARD-PUT | full : 2 , gp@empty : st(0,unchanged) >
          < mutx : MUTEX | status : unlocked >
          send(gget,gg@in, IT1:Interaction)
          =>*
          C2:Configuration send(buff, buffer@get, IT2:Interaction) .)

Solution 1
C2:Configuration <- < gget : GUARD-GET | empty : 0,gg@full : st(1,unchanged)>
  < gput : GUARD-PUT | full : 1,gp@empty : st(0,unchanged)>
  < mutx : MUTEX | status : locked > ;
IT2:Interaction <- [mutx,mutex@out2]::[gget,gg@out] :: IT1:Interaction
```

É importante observar que as duas últimas buscas percorrem o mesmo número de estados de computação, independentemente do número de instâncias de `PRODUCER` e `CONSUMER` na arquitetura.

A execução de buscas a partir de termo com variáveis (não "ground"), que represente um estado intermediário de alguma interação, é bastante conveniente para análise de um

particular *pedaço* da arquitetura. No entanto, a especificação deste estado intermediário deve ser feita com bastante cautela. Em primeiro lugar, deve-se garantir que todos os objetos necessários para as próximas etapas da interação foram incluídos no termo. Em segundo lugar, os estados de cada objeto, representados pelos valores de seus atributos, devem estar consistentes. Por exemplo, nas buscas supra, o valor do atributo `gg@full` de `gget` (uma variável de estado declarada em `GUARD-GET`), deve ser igual ao valor do atributo `full` no objeto `gput`.

Utilizando a técnica de abstrações equacionais [54], podemos realizar o corte do número de estados da árvore de computação de uma maneira mais segura e elegante. A técnica consiste basicamente em agrupar, com equações, diferentes estados da computação na mesma *classe de equivalência*. Lembrando da Seção 2.3.2, que Maude aplica as regras de reescritas sobre um termo *módulo* às equações, isto é, um termo é primeiramente reduzido à sua forma canônica com as equações e axiomas antes de ser reescrito com as regras de reescrita disponíveis.

Para exemplificar a utilização da técnica, considere a análise das propriedades de "overflow" e "underflow" na arquitetura `PC-GUARDS-MUTEX`, Figura 10, para garantir que os limites de armazenamento do "buffer" não são violados, precisamos apenas observar os estados internos das instâncias dos conectores `GUARD-GET` e `GUARD-PUT`. Os valores dos atributos `full` e `empty`, dos objetos que representam as instâncias dos conectores da arquitetura, variam apenas no intervalo $[0, 2]$, durante o recebimento e envio de mensagens por estes objetos. Para esta análise, o início das interações nas instâncias dos módulos `PRODUCER` e `CONSUMER`, e *chegada* das interações na instância de `BUFFER` são irrelevantes. Podemos então aplicar a técnica de abstrações equacionais para desconsiderar, durante a exploração dos estados, os estados correspondentes a estas etapas das interações.

O módulo `PCB-EXEC-ABS` adiante, estende o módulo `PCB-EXEC` com três equações. A primeira equação define que termos com mensagens `ack([O, producer@put] :: I)`, onde `O` é uma variável do *sort* `Oid` (qualquer identificador de objeto) e `I` é uma variável do *sort* `Interaction` (qualquer interação), estão na mesma classe de equivalência de termos com mensagens `send(gput, gp@in, [O, producer@put])`. Operacionalmente, isto significa que qualquer mensagem `ack([O, producer@put] :: I)`, presente na configuração de objetos e mensagens, será equacionalmente reescrita para uma mensagem `send(gput, gp@in, [O, producer@put])`, antes que o sistema evolua para um novo estado. Por isso, os estados para os quais o sistema evoluiria, a partir da reescrita das mensagens `ack([O, producer@put] :: I)`, deixam de existir na árvore de computação da arquitetura. Nas equações se-

guintes, o mesmo tratamento é dado para as mensagens `ack([O, consumer@get] :: I)` e `send(buff,P,I)`.

```
(omod PCB-EXEC-ABS is
  ex PCB-EXEC .

  var O : Oid .
  var I : Interaction .
  var P : PortId .

  eq ack([O, producer@put] :: I) = send(gput, gp@in, [O, producer@put]) .
  eq ack([O, consumer@get] :: I) = send(gget, gg@in, [O, consumer@get]) .

  eq send(buff, P, I) = ack(I) .
endom)
```

O sistema resultante é diferente do sistema original e, logo, em buscas realizadas no novo sistema não teremos os mesmos resultados que teríamos no sistema original. Por exemplo, enquanto uma busca executada no módulo `PCB-EXEC` localiza pelo menos um estado com uma mensagem `send(O,buffer@put,I)`:

```
Maude> (search [1] in PCB-EXEC : initial =>* C:Configuration
                                        send(O, buffer@put, I:Interaction) .)

Solution 1
C:Configuration <- < buff : BUFFER | items : 0,maxitems : 2 >
 < cons1 : CONSUMER | consumer@get-status : unlocked >
 < cons2 : CONSUMER | consumer@get-status : unlocked >
 < gget : GUARD-GET | empty : 1,gg@full : st(0,unchanged)>
 < gput : GUARD-PUT | full : 0,gp@empty : st(1,unchanged)>
 < mutx : MUTEX | status : locked >
 < prod1 : PRODUCER | producer@put-status : locked >
 < prod2 : PRODUCER | producer@put-status : unlocked >
 do(cons1,consumer@get,none)
 do(cons2,consumer@get,none)
 do(prod2,producer@put,none);
I:Interaction <- [mutx,mutex@out1]::[gput,gp@out]::[prod1,producer@put];
O:Oid <- buff
```

O mesmo não é verdade para o novo sistema definido pelo módulo `PCB-EXEC-ABS`. Neste novo sistema, qualquer mensagem para `buff` será equacionalmente reescrita para uma mensagem `ack`, antes que esta possa ser recebida por `buff`.

```
Maude> (search [1] in PCB-EXEC-ABS : initial =>* C:Configuration
                                            send(O, buffer@put, I:Interaction) .)
No solution.
```

No entanto, como dito anteriormente, para verificar se os conectores na arquitetura PC-GUARDS-MUTEX funcionam corretamente, nossa abstração é adequada, pois, todos os possíveis recebimentos e envios de mensagens para os objetos que representam as instâncias dos conectores continuam a existir. Podemos então analisar o comportamento dos conectores com a busca abaixo, realizada primeiramente no módulo PCB-EXEC.

```
Maude> (search [1] in PCB-EXEC : initial =>* C:Configuration
              < gget : GUARD-GET | empty : N:Int , A1:AttributeSet >
              < gput : GUARD-PUT | full : M:Int , A2:AttributeSet >
         such that (N:Int > 2 or M:Int > 2 or N:Int < 0 or M:Int < 0) .)

No solution.
```

Teremos o mesmo resultado desta busca no sistema abstrato. Com a diferença que, Maude percorre 5760 estados na busca realizada no módulo PCB-EXEC e apenas 392 quando realizamos a busca no módulo PCB-EXEC-ABS.

```
Maude> (search [1] in PCB-EXEC-ABS : initial =>* C:Configuration
              < gget : GUARD-GET | empty : N:Int , A1:AttributeSet >
              < gput : GUARD-PUT | full : M:Int , A2:AttributeSet >
         such that (N:Int > 2 or M:Int > 2 or N:Int < 0 or M:Int < 0) .)

No solution.
```

5.1.2.2 Adaptação de arquiteturas para Maude CBabel tool

Em CBabel, nenhuma limitação é imposta com relação ao número de contratos declarados em um conector [46]. No entanto, com objetivo de simplificar a semântica dos conectores e contratos e, conseqüentemente, dar semântica a um subconjunto representativo da linguagem, consideramos em nosso mapeamento de CBabel para lógica de reescrita, que cada conector implementa apenas um contrato. Conforme descrito na Seção 3.1.3, as seguintes decisões (empíricas) de "design" foram adotadas:

1. Cada conector só pode declarar um único contrato. Esta decisão foi tomada pois, em [46], não foi definida composição de contratos. Destacamos que composicionalidade de componentes não é uma questão trivial e seu tratamento foge ao escopo deste trabalho.

2. Os contratos de exclusão mútua e interação guardada são declarados no contexto de uma *interação*, ou seja, entre portas de entrada e portas de saída. Esta decisão foi tomada por dois motivos. Em primeiro lugar, conforme destacado na Seção 3.1.3,

tendo em vista a função dos conectores de encaminhar requisições, a declaração de contratos de coordenação isoladamente em um conector não faria sentido. Como o encaminhamento de requisições é definido por um contrato de interação, então todo conector deve declarar pelo menos um contrato de interação entre suas portas de entrada e saída. Como no item anterior restringimos a declaração de apenas um contrato por conector, os contratos de coordenação foram então redefinidos para serem declarados sob interações, e não mais sobre portas. Em segundo lugar, esta decisão foi tomada baseada na própria intuição destes contratos, que dão significado a tipos de interação e coordenação de interação entre portas, e não somente a declaração de uma porta, como colocado em [46]. Mais ainda, como veremos a seguir, esta intuição fica capturada exatamente por regras de reescrita na teoria de reescrita gerada.

3. Os tipos das portas em cada interação são compatíveis, ou seja, as portas de entrada e as portas de saída devem ser síncronas ou assíncronas. Embora tal restrição não apareça na especificação original da linguagem CBabel [46], esta decisão foi tomada levando-se em consideração que portas de entrada e portas de saída de uma interação devem ter assinaturas compatíveis. Na realidade, este tipo de questionamento sobre a especificação da linguagem CBabel corresponde exatamente a uma das contribuições de nosso trabalho na especificação de uma semântica formal de CBabel.

Em relação ao item 1, imaginou-se que a composição de contratos pudesse ser feita na forma de macros que produzissem conectores conectados ao invés de uma composição de contratos. Percebeu-se, no entanto, que tal transformação nem sempre é intuitiva, como mostraremos no exemplo adiante. Em relação aos itens 2 e 3, as simplificações adotadas constituem também sugestões para o (re)design da linguagem. O item 2 torna a sintaxe dos contratos de coordenação mais próxima da intuição do significado destes contratos, conforme explicado. O item 3 constitui uma contribuição para a redefinição da gramática da linguagem, eliminando destas, construções sintáticas que não teriam uma semântica adequada.

Inicialmente, para ilustrar a simplificação obtida na semântica dos contratos com a restrição de apenas um contrato por conector, vamos analisar a semântica intencional para os conectores `MutexPCcon` e `SincMutexPCcon`, apresentados por Sztajnberg em [46].

No conector `MutexPCcon`, mostrado a seguir, embora nenhum contrato de interação tenha sido definido, as portas de entrada e saída do tipo `GetT` estão implicitamente liga-

das por um contrato de interação. O mesmo ocorrendo para as portas do tipo `PutT`. A semântica esperada para o contrato de exclusão mútua neste conector é de que as portas de saída `GetT` e `PutT` não podem ser estimuladas juntas. Caso as portas de entrada do conector `MutexPCcon` sejam estimuladas simultaneamente, apenas uma das requisições recebidas será transmitida para a porta de saída correspondente. A outra requisição será bloqueada até que a resposta da primeira requisição seja recebida pelo conector.

```
1  connector {
2   in port GetT ;
3   in port PutT ;
4   exclusive {
5     out port GetT ;
6     out port PutT ;
7   }
8  } MutexPCcon;
```

No conector `SincMutexPCcon`, o contrato de exclusão mútua, quando aninhado a portas guardadas, deveria, intencionalmente, apresentar a semântica de um monitor [56]. Ou seja, caso o conector `SincMutexPCcon` receba estímulos simultâneos em suas portas de entrada, `Get` e `Put`, o contrato de exclusão mútua deve garantir que um dos estímulos seja bloqueado enquanto o outro é transmitido para uma das portas de saída. Considere que o estímulo da porta `Put` seja transmitido inicialmente para a porta `GPut`. Se o guarda da porta `GPut` estiver *fechado*, o estímulo de `GPut` deverá ser bloqueado pelo guarda e o contrato de exclusão mútua deve liberar o estímulo de `Get`, dando a vez para que a porta `GGet` seja estimulada.

```
1  connector {
2   condition vazio = true, cheio = false;
3   staterequired int n_itens;
4   int MAX_ITENS = 10;
5   exclusive {
6     out port PutT {
7       guard (vazio) {
8         after {
9           cheio = true;
10          if (n_itens == MAX_ITENS){
11            vazio = false;
12      } } }
13    } GPut;
14    out port GetT {
15      guard (cheio) {
16        after {
17          vazio = true;
18          if (n_itens == 0) {
19            cheio = false;
```

```
20        } } }
21      } GGet ;
22    }
23    in port GetT Get;
24    in port PutT Put;
25  } SincMutexPCcon;
```

Ou seja, considerando o conector `SincMutexPCon`, a semântica do contrato de exclusão mútua deve levar em consideração a semântica das portas guardas. Não sendo possível assim definir a semântica do contrato de coordenação *exclusive*, sem considerar cada caso: ele estar aninhado a portas guardadas, ou, ele estar coordenando apenas a portas de saída. A partir da restrição de um contrato por conector, a semântica de cada contrato de coordenação, exclusão mútua e guardas, pode ser dada individualmente, não sendo sobrecarregada com qualquer informação do contexto onde o contrato está sendo utilizado.

Operacionalmente, o resultado da restrição de um contrato por conector em Maude CBabel tool é a maior complexidade na descrição das arquiteturas, pela necessidade de declaração e criação de instâncias de um número maior de conectores. Além disso, na atual versão de Maude CBabel tool, arquiteturas CBabel onde mais de um contrato tenha sido definido em um conector, precisam ser ajustadas manualmente.

Na implementação e análises das arquiteturas para a aplicação dos produtores e consumidores, por exemplo, as arquiteturas apresentadas em [46] precisaram ser adaptadas para que pudessem ser executadas em Maude CBabel tool.

Três tentativas de desmembramento do conector `SincMutexPCcon` foram experimentadas. Na primeira tentativa, os conectores `MUTEX`, `GUARD-PUT` e `GUARD-GET` foram declarados conforme apresentado na Figura 20. A ligação dos conectores no módulo da aplicação foi feita procurando representar a intenção do projetista com o aninhamento dos contratos no conector `SincMutexPCcon` original. Isto é, conectando-se as saídas do `MUTEX` às entradas dos guardas.

Durante as análises, no entanto, identificamos que a arquitetura da Figura 20 apresenta um estado de "deadlock". Isto porque, após a instância de `MUTEX` transmitir uma requisição, permanece bloqueada até obter uma resposta daquela requisição. No entanto, caso a requisição transmitida pela instância de `MUTEX` seja bloqueada pelas instâncias de `GUARD-PUT` ou `GUARD-GET`, o sistema estará em um estado de "deadlock".

Numa segunda tentativa, a ligação dos conectores conectando as saídas dos guardas às entradas do `MUTEX`, conforme descrito pelo módulo de aplicação apresentado na Figura 21, evita a ocorrência do "deadlock", mas impede que os conectores `GUARD-PUT` e `GUARD-GET`

```
connector GUARD-PUT {
  var bool vazio = TRUE ;
  var int maxitems = 2 ;
  staterequired int gp@items ;
  staterequired bool gp@cheio ;

  in port gp@in ;
  out port gp@out ;

  interaction {
    gp@in >
    guard(vazio == TRUE) {
     after {
      gp@cheio = TRUE ;
      if (gp@items == maxitems) {
        vazio = FALSE ;
      }
     }
    } > gp@out ;
  }
}

connector MUTEX {
 in port mutex@in1 ;
 in port mutex@in2 ;
 out port mutex@out1 ;
 out port mutex@out2 ;

 exclusive{
  mutex@in1 > mutex@out1 ;
  mutex@in2 > mutex@out2 ;
 }
}
```

```
connector GUARD-GET {
  var bool cheio = FALSE ;
  staterequired int gg@items ;
  staterequired bool gg@vazio ;

  in port gg@in ;
  out port gg@out ;

  interaction {
    gg@in >
    guard(cheio == TRUE) {
     after {
      gg@vazio = TRUE ;
      if (gg@items == 0) {
        cheio = FALSE ;
      }
     }
    } > gg@out ;
  }
}

application PC-MUTEX-GUARDS {
 instantiate BUFFER as buff ;
 ...
 link prod.producer@put to mutx.mutex@in1 ;
 link cons.consumer@get to mutx.mutex@in2 ;
 ...
 bind int gget.gg@items to buff.buffer@items ;
 bind int gput.gp@items to buff.buffer@items ;
 bind bool gput.gp@cheio to gget.gg@cheio ;
 bind bool gget.gg@vazio to gput.gp@vazio ;
}
```

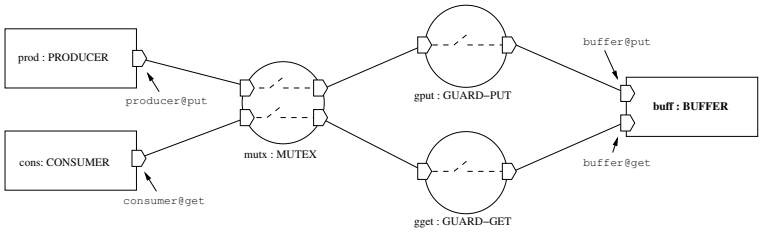

Figura 20: Primeira tentativa de desmembramento do conector `SincMutexPCcon`

```
application PC-GUARDS-MUTEX {
 instantiate BUFFER as buff ;
 ...
 link prod.producer@put to gput.gp@in ;
 link cons.consumer@get to gget.gg@in ;
 ...
 bind int gget.gg@nitems to gput.gp@nitems ;
 bind int gget.gg@maxitems to gput.gp@maxitems ;
 bind bool gput.gp@cheio to gget.gg@cheio ;
 bind bool gget.gg@vazio to gput.gp@vazio ;
}
```

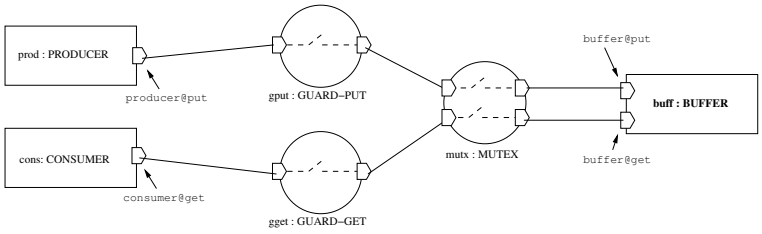

Figura 21: Segunda tentativa de desmembramento do conector `SincMutexPCcon`

funcionem como esperado, ou seja, controlando os limites do "buffer". Neste caso, com o conector MUTEX após os conectores GUARD-PUT e GUARD-GET, a avaliação das expressões dos guardas e as atualizações das variáveis cheio e vazio não ocorrem em exclusão mútua, permitindo assim estados inconsistentes na arquitetura.

A terceira tentativa para representar o conector SincMutexPCcon é apresentada na Figura 22. Na figura, os conectores GUARD-PUT e GUARD-GET incorporam o controle de acessos concorrentes através da variável semaphoro. Na arquitetura da Figura 22 todas as propriedades apresentadas na Seção 4.3 são válidas. No entanto, isto não é suficiente para afirmarmos que a arquitetura da Figura 22 é *equivalente* a arquitetura da Figura 10. Na realidade, a verificação de equivalência de arquiteturas está fora do escopo deste trabalho, assim como um estudo mais detalhado sobre a composição de contratos.

5.1.3 Leitores e escritores

Na Seção 4.4, apresentamos a arquitetura READERS-WRITERS para a aplicação leitores e escritores e mostramos que esta arquitetura está livre de "deadlock" (Propriedade 4.4.3). No entanto, adiamos para esta seção a análise de outra propriedade importante para um sistema distribuído, a ausência de "livelock". Na existência de "livelock" ou "deadlock", alguns componentes podem permanecer constantemente impedidos de prosseguir sua computação, caracterizando assim a situação de "starvation" destes.

Na arquitetura READERS-WRITERS, as instâncias dos módulos READER e WRITER enviam mensagens para a instância buff do módulo BUFER. Precisamos garantir que todos os leitores e escritores conseguirão, eventualmente, acessar o "buffer". Caso algum leitor ou escritor permaneça sempre impedido de acessar o "buffer", dizemos que a arquitetura READERS-WRITERS apresenta "livelock". No módulo RW-VER, Figura 15, definimos a proposição parametrizada reading(O) verdadeira nos estados onde buff esteja recebendo uma mensagem do objeto O. A ausência de "starvation" da instância reader1 na arquitetura pode então ser descrita pela fórmula LTL:

```
[] <> reading(reader1)
```

Se submetermos a fórmula acima ao verificador de modelos, teremos um contra-exemplo, mostrando um caminho de execução onde nenhuma mensagem de reader1 é recebida por buff. No fragmento do contra-exemplo a seguir, a mensagem de reader1 chega ao objeto cr, mas não é consumida por ele, permanencendo no sistema indefinidamente.

```
connector GUARD-GET {
  var bool gg@cheio = FALSE ;
  staterequired int gg@items ;
  staterequired bool gg@vazio ;
  staterequired bool gg@semaphoro ;
  in port gg@in ;
  out port gg@out ;
  interaction {
    gg@in >
    guard(gg@semaphoro == TRUE &&
          gg@cheio == TRUE) {
     before {
      gg@semaphoro = FALSE ;
     }
     after {
      gg@vazio = TRUE ;
      if (gg@items == 0) {
        gg@cheio = FALSE ;
      }
      gg@semaphoro = TRUE ;
     }
    } > gg@out ;
  }
}

module PRODUCER {
 out port producer@put ;
}

module CONSUMER {
 out port consumer@get ;
}

module BUFFER {
 var int buffer@items = 0 ;
 in port buffer@put ;
 in port buffer@get ;
}
```

```
connector GUARD-PUT {
  var bool gp@vazio = TRUE ;
  var bool gp@semaphoro = TRUE ;
  var int gp@maxitems = 2 ;
  staterequired int gp@items ;
  staterequired bool gp@cheio ;
  in port gp@in ;
  out port gp@out ;
  interaction {
   gp@in >
    guard(gp@semaphoro == TRUE &&
          gp@vazio == TRUE) {
     before {
      gp@semaphoro = FALSE ;
     }
     after {
      gp@cheio = TRUE ;
      if (gp@items == gp@maxitems) {
        gp@vazio = FALSE ;
      }
      gp@semaphoro = TRUE ;
     }
    } > gp@out ;
  }
}

application PC-MONITOR {
 instantiate BUFFER as buff ;
 ...
 link prod.producer@put to gput.gp@in ;
 link cons.consumer@get to gget.gg@in ;
 ...
 bind int gput.gp@items to buff.buffer@items ;
 bind int gget.gg@items to buff.buffer@items ;
 bind bool gput.gp@cheio to gget.gg@cheio ;
 bind bool gget.gg@vazio to gput.gp@vazio ;
 bind bool gget.gg@semaphoro to gput.gp@semaphoro ;
}
```

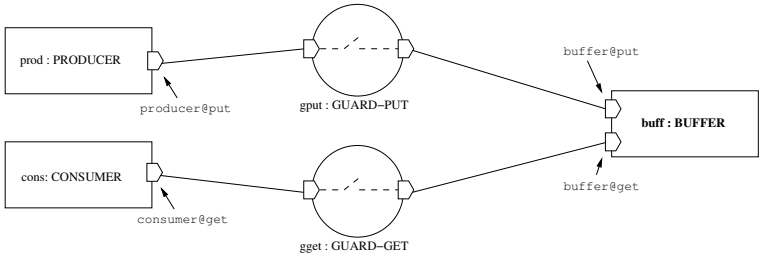

Figura 22: Terceria tentativa de desmembramento do conector `SincMutexPCcon`

O segundo argumento do operador `counterexample`, editado para melhor legibilidade, corresponde ao ciclo de uma interação completa da instância `reader2`: a regra `reading`, do módulo `RW-EXEC` reinicia uma interação e a regra `READER-receivingAck-r@read` termina uma interação e reinicia o ciclo.

```
Maude> (red modelCheck(initial, []<> reading(reader1)) .)
result ModelCheckResult :
  counterexample(
   {< buff : BUFFER | none >
    < cr : COUNT-READ | cr@want-read : st(0,unchanged),
      cr@want-write : st(0,unchanged),cr@writing :
      st(false,unchanged), readers : 0,turn : 0 >
    < cw : COUNT-WRITE | cw@readers : st(0,unchanged),cw@turn : st(0,unchanged),
      cw@want-read : st(0,unchanged), cw@want-write : st(0,unchanged),
      writing : false >
    < reader1 : READER | r@read-status : unlocked >
    < reader2 : READER | r@read-status : unlocked >
    < wr : WANT-READ | want-read : 0 > < ww : WANT-WRITE | want-write : 0 >
    < writer1 : WRITER | w@write-status : unlocked >
    < writer2 : WRITER | w@write-status : unlocked >
    do(reader1,r@read,none) do(reader2,r@read,none)
    do(writer1,w@write,none) do(writer2,w@write,none), 'READER-sending-r@read}
  ...
  {< buff : BUFFER | none >
   < cr : COUNT-READ | cr@want-read : st(1,unchanged),
     cr@want-write : st(0,unchanged), cr@writing : st(false,unchanged),
     readers : 1,turn : 0 >
   < cw : COUNT-WRITE | cw@readers : st(1,unchanged),
     cw@turn : st(0,unchanged), cw@want-read : st(1,unchanged),
     cw@want-write : st(0,unchanged),writing : false >
   < reader1 : READER | r@read-status : locked >
   < reader2 : READER | r@read-status : locked >
   < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 0 >
   < writer1 : WRITER | w@write-status : locked >
   < writer2 : WRITER | w@write-status : locked >
   send(buff,buffer@read,[cr,out-count-read]::[wr,out-want-read]::[reader2,r@read])
   send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
   send(ww,in-want-write,[writer1,w@write])
   send(ww,in-want-write,[writer2,w@write]),'BUFFER-recevingAndDo-buffer@read}
  ...
  {< buff : BUFFER | none >
   < cr : COUNT-READ | cr@want-read : st(1,unchanged),
     cr@want-write : st(1,unchanged),cr@writing : st(false,unchanged),
     readers : 0,turn : 0 >
   < cw : COUNT-WRITE | cw@readers : st(0,unchanged), cw@turn : st(0,unchanged),
     cw@want-read : st(1,unchanged),cw@want-write : st(1,unchanged),
     writing : false >
   < reader1 : READER | r@read-status : locked >
   < reader2 : READER | r@read-status : unlocked >
   < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 1 >
   < writer1 : WRITER | w@write-status : locked >
   < writer2 : WRITER | w@write-status : locked >
```

```
  done(reader2,r@read,none)
  send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
  send(cw,in-count-write,[ww,out-want-write]::[writer1,w@write])
  send(ww,in-want-write,[writer2,w@write]),'WANT-WRITE-sending-in-want-write}
  ...,
 {< buff : BUFFER | none >
  < cr : COUNT-READ | cr@want-read : st(1,unchanged),cr@want-write :
    st(2,unchanged), cr@writing : st(false,unchanged),readers : 0,turn : 0 >
  < cw : COUNT-WRITE | cw@readers : st(0,unchanged),cw@turn : st(0,unchanged),
    cw@want-read : st(1,unchanged),cw@want-write : st(2,unchanged),
    writing : false >
  < reader1 : READER | r@read-status : locked >
  < reader2 : READER | r@read-status : unlocked >
  < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 2 >
  < writer1 : WRITER | w@write-status : locked >
  < writer2 : WRITER | w@write-status : locked >
  done(reader2,r@read,none)
  send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
  send(cw,in-count-write,[ww,out-want-write]::[writer1,w@write])
  send(cw,in-count-write,[ww,out-want-write]::[writer2,w@write]),'reading}
  ...
 {ack([reader2,r@read])
  < buff : BUFFER | none >
  < cr : COUNT-READ | cr@want-read : st(1,unchanged),cr@want-write :
    st(2,unchanged),cr@writing : st(false,unchanged),readers : 0,turn : 0 >
  < cw : COUNT-WRITE | cw@readers : st(0,unchanged),cw@turn : st(0,unchanged),
    cw@want-read : st(1,unchanged),cw@want-write : st(2,unchanged),
    writing : false >
  < reader1 : READER | r@read-status : locked >
  < reader2 : READER | r@read-status : locked >
  < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 2 >
  < writer1 : WRITER | w@write-status : locked >
  < writer2 : WRITER | w@write-status : locked >
  send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
  send(cw,in-count-write,[ww,out-want-write]::[writer1,w@write])
  send(cw,in-count-write,[ww,out-want-write]::[writer2,w@write]),
    'READER-receivingAck-r@read})
```

Se tentarmos verificar se pelo menos uma vez alguma mensagem de `reader1` é recebida por `buff`, situação que pode ser descrita pela fórmula LTL `<> reading(reader1)`, o verificador de modelos também consegue um contra-exemplo. O fragmento abaixo exibe o contra-exemplo obtido, que por sua similaridade com o contra-exemplo anterior, teve o segundo argumento removido.

```
Maude> (red modelCheck(initial, <> reading(reader1)) .)
result ModelCheckResult :
  counterexample({< buff : BUFFER | none >
    < cr : COUNT-READ | cr@want-read : st(0,unchanged),
      cr@want-write : st(0,unchanged),cr@writing :
      st(false,unchanged), readers : 0,turn : 0 >
    < cw : COUNT-WRITE | cw@readers : st(0,unchanged),cw@turn : st(0,unchanged),
```

```
     cw@want-read : st(0,unchanged), cw@want-write : st(0,unchanged),
     writing : false >
   < reader1 : READER | r@read-status : unlocked >
   < reader2 : READER | r@read-status : unlocked >
   < wr : WANT-READ | want-read : 0 > < ww : WANT-WRITE | want-write : 0 >
   < writer1 : WRITER | w@write-status : unlocked >
   < writer2 : WRITER | w@write-status : unlocked >
   do(reader1,r@read,none) do(reader2,r@read,none)
   do(writer1,w@write,none) do(writer2,w@write,none),
   'READER-sending-r@read}
...
{< buff : BUFFER | none >
 < cr : COUNT-READ | cr@want-read : st(1,unchanged),
   cr@want-write : st(0,unchanged), cr@writing : st(false,unchanged),
   readers : 1,turn : 0 >
 < cw : COUNT-WRITE | cw@readers : st(1,unchanged),
   cw@turn : st(0,unchanged), cw@want-read : st(1,unchanged),
   cw@want-write : st(0,unchanged),writing : false >
 < reader1 : READER | r@read-status : locked >
 < reader2 : READER | r@read-status : locked >
 < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 0 >
 < writer1 : WRITER | w@write-status : locked >
 < writer2 : WRITER | w@write-status : locked >
 send(buff,buffer@read,[cr,out-count-read]::[wr,out-want-read]::[reader2,r@read])
 send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
 send(ww,in-want-write,[writer1,w@write])
 send(ww,in-want-write,[writer2,w@write]),'BUFFER-recevingAndDo-buffer@read}
...
{< buff : BUFFER | none >
 < cr : COUNT-READ | cr@want-read : st(1,unchanged),
   cr@want-write : st(1,unchanged),cr@writing : st(false,unchanged),
   readers : 0,turn : 0 >
 < cw : COUNT-WRITE | cw@readers : st(0,unchanged), cw@turn : st(0,unchanged),
   cw@want-read : st(1,unchanged),cw@want-write : st(1,unchanged),
   writing : false >
 < reader1 : READER | r@read-status : locked >
 < reader2 : READER | r@read-status : unlocked >
 < wr : WANT-READ | want-read : 1 > < ww : WANT-WRITE | want-write : 1 >
 < writer1 : WRITER | w@write-status : locked >
 < writer2 : WRITER | w@write-status : locked >
 done(reader2,r@read,none)
 send(cr,in-count-read,[wr,out-want-read]::[reader1,r@read])
 send(cw,in-count-write,[ww,out-want-write]::[writer1,w@write])
 send(ww,in-want-write,[writer2,w@write]),'WANT-WRITE-sending-in-want-write}
 ..., [CICLO])
```

No entanto, também podemos mostrar que não é verdade que nunca uma mensagem de `reader1` é recebida por `buff`. Abaixo, o verificador de modelos consegue encontrar um contra-exemplo para a fórmula LTL `[] ~ reading(reader1)`, que descreve que uma mensagem de `reader1` nunca é recebida por `buff`. No fragmento abaixo, o segundo estado mostrado contém uma mensagem para `buff`, enviada por `reader1`.

```
Maude> (red modelCheck(initial, [] ~ reading(reader1)) .)
result ModelCheckResult :
  counterexample({< buff : BUFFER | none >
    < cr : COUNT-READ | cr@want-read : st(0,unchanged),
      cr@want-write : st(0,unchanged),cr@writing :
      st(false,unchanged), readers : 0,turn : 0 >
    < cw : COUNT-WRITE | cw@readers : st(0,unchanged),cw@turn : st(0,unchanged),
      cw@want-read : st(0,unchanged), cw@want-write : st(0,unchanged),
      writing : false >
    < reader1 : READER | r@read-status : unlocked >
    < reader2 : READER | r@read-status : unlocked >
    < wr : WANT-READ | want-read : 0 > < ww : WANT-WRITE | want-write : 0 >
    < writer1 : WRITER | w@write-status : unlocked >
    < writer2 : WRITER | w@write-status : unlocked >
    do(reader1,r@read,none)do(reader2,r@read,none)
    do(writer1,w@write,none)do(writer2,w@write,none),'READER-sending-r@read}
    ...
   {< buff : BUFFER | none >
    < cr : COUNT-READ | cr@want-read : st(0,unchanged),cr@want-write :
      st(0,unchanged),cr@writing : st(false,unchanged),readers : 1,turn : 1 >
    < cw : COUNT-WRITE | cw@readers : st(1,unchanged),cw@turn : st(1,unchanged),
      cw@want-read : st(0,unchanged),cw@want-write : st(0,unchanged),
      writing : false >
    < reader1 : READER | r@read-status : locked >
    < reader2 : READER | r@read-status : locked >
    < wr : WANT-READ | want-read : 0 > < ww : WANT-WRITE | want-write : 0 >
    < writer1 : WRITER | w@write-status : locked >
    < writer2 : WRITER | w@write-status : locked >
    send(buff,buffer@read,[cr,out-count-read]::[wr,out-want-read]::[reader1,r@read])
    send(wr,in-want-read,[reader2,r@read])
    send(ww,in-want-write,[writer1,w@write])
    send(ww,in-want-write,[writer2,w@write]),'BUFFER-recevingAndDo-buffer@read}
   ...,[CICLO])
```

A aparente contradição dos resultados supra pode ser explicada. A descrição informal da aplicação dos leitores e escritores diz que eventualmente um leitor irá acessar o "buffer". Da mesma forma, uma vez acessando o "buffer", eventualmente este será liberado. Ou seja, nenhum leitor permanecerá na mesma *etapa* de sua computação por tempo indeterminado. Esta condição de *justiça* não é capturada pelo modelo de Maude, que por sua generalidade, considera a possibilidade de uma mensagem poder ficar por tempo indeterminado sem ser consumida pelo objeto destinatário [53]. Em Maude, podemos impor a condição de *justiça* com a implementação de uma estratégia para aplicação das regras de reescrita, como em [49].

No entanto, é interessante observar que todo caminho de execução *finito*, simulado pelo comando `rewrite [n]`, captura precisamente a descrição informal, apresentada na Seção 4.4, da aplicação leitores e escritores. Isto porque corresponde ao prefixo de um caminho de execução onde nenhuma mensagem permanece indefinidamente sem ser rece-

bida ou transmitida. Portanto, a estratégia padrão de Maude para aplicação das regras não afeta a análise da existência de um estado de *deadlock* (Propriedade 4.4.3), feita na Seção 4.4. Análise semelhante é apresentada por Ølveczky em [53].

Como a situação de "livelock" compreende um cenário infinito, devemos então levar em consideração certas condições de *justiça* para análise de "livelock". Neste caso, dizemos que a análise é feita em relação a uma forma mais fraca de "livelock", chamada "weak liveness".

Na arquitetura `READERS-WRITERS`, podemos analisar a propriedade de "livelock" em sua forma mais fraca. Para isso, alteramos o módulo de análise `RW-VER` incluindo a sintaxe e a semântica para um novo predicado de estado. A sintaxe deste novo predicado é definida pelo operador `there-is`, parametrizado por dois identificadores de objetos e um identificador de porta. Sua semântica, dada pela equação, define que este predicado é verdadeiro nos estados onde exista no sistema uma mensagem destinada ao objeto `O'`, para sua porta `P`, tendo sido originada no objeto `O`.

```
(omod RW-VER is
 ...
 op there-is : Oid Oid PortId -> Prop .
 eq C send(O', P, IT :: [O, P']) |= there-is(O, O', P) = true .
 ...
endom)
```

Após reimportar o módulo `RW-VER` em Maude CBabel tool, podemos submeter a fórmula LTL abaixo ao verificador de modelos:

```
([]<> there-is(reader1, wr, in-want-read)
 /\ []<> there-is(reader2, wr,in-want-read)
 /\ []<> there-is(writer1, ww, in-want-write)
 /\ []<> there-is(writer2, ww,in-want-write)) ->
 ([]<> there-is(reader1, cr,in-count-read)
  /\ []<> there-is(reader2, cr, in-count-read)
  /\ []<> there-is(writer1, cw, in-count-write)
  /\ []<> there-is(writer2, cw,  in-count-write)) -> []<> reading(reader1)
```

A fórmula especifica que, se for verdade que sempre eventualmente mensagens das instâncias de `READER` e `WRITER` estarão sendo recebidas pelas instâncias dos conectores `WANT-READ` e `WANT-WRITE`, respectivamente, então se também for verdade que sempre eventualmente mensagens estarão sendo recebidas pelas instâncias dos conectores `COUNT-READ` e `COUNT-WRITE`, então uma mensagem da instância `reader1` sempre eventualmente conseguirá ser recebida por `buff`. Submetendo a fórmula ao verificador de modelos de Maude:

```
Maude> (red modelCheck(initial,
([]<> there-is(reader1, wr, in-want-read)
 /\ []<> there-is(reader2, wr,in-want-read)
 /\ []<> there-is(writer1, ww, in-want-write)
 /\ []<> there-is(writer2, ww,in-want-write)) ->
 ([]<> there-is(reader1, cr,in-count-read)
  /\ []<> there-is(reader2, cr, in-count-read)
  /\ []<> there-is(writer1, cw, in-count-write)
  /\ []<> there-is(writer2, cw,  in-count-write)) -> []<> reading(reader1)) .)

result Bool :
  true
```

Afirmações simétricas para as outras instâncias de `READER` e `WRITER` também valem. Análise semelhante foi realizada em [23] para o algoritmo de Dekker.

Critérios de justiça sobre eventos eventuais não podem ser *implementados* em sua forma mais genérica, dado que não existe uma fronteira para quando tal evento deve ocorrer. No entanto, vários modelos de concorrência não executáveis implementam critérios de justiça como: "nenhuma regra pode permanecer continuamente sem ser aplicada se ela pode ser aplicada." Toda reescrita infinita de Maude é justa, no sentido de que uma regra de reescrita que pode continuamente ser aplicada, eventualmente será aplicada. No entanto, a justiça na aplicação das regras não é suficiente; deve-se ainda garantir justiça sobre os objetos sobre os quais as regras são aplicadas. Ou seja, justiça para cada uma das instâncias das regras [53, 50].

Para encerrar esta seção, um último comentário torna-se bastante relevante. Foi possível observarmos nesta seção que os contra-exemplos apresentados pelo verificador de modelos podem, geralmente, se tornar bastante extensos, tornando difícil a sua compreensão. Isto porque tais contra-exemplos compreendem um traço completo de execução a partir do estado inicial onde a propriedade analisada é invalidada. Durante nossa pesquisa, uma abordagem que utilizamos para análise dos contra-exemplos foi a definição de funções de projeção sobre partes dos contra-exemplos.

Na Figura 23, exemplificamos este procedimento. No módulo `MC-SIMPLE` as funções `getRuleNames` e `getPathMsg` recebem como entrada um termo do tipo `TransitionList`. O tipo `TransitionList`, e suas operações construtoras, é declarado no módulo `MODEL-CHECHER`, para representar uma lista de transições onde cada transição é uma tupla formada por um estado e o nome da regra de reescrita utilizada para o sistema atingir aquele estado. A função `getRulenames` captura, de cada transição, apenas o nome da regra de reescrita associada. Sendo assim, a partir de uma lista de transições (um caminho de execução), esta função produz uma lista dos nomes das regras de reescrita aplicadas naquele caminho.

A função `getPathMsg` captura, de cada transição, apenas as mensagens presentes no estado da transição, produzindo como saída uma lista de estados com mensagens apenas. Para capturar apenas as mensagens de um estado, definimos o tipo de dados `MsgConfiguration`, subtipo de `Configuration`.

```
(omod MC-SIMPLE is
  inc MODEL-CHECKER .

  sorts EmptyConfiguration NEConfiguration MsgConfiguration
        NEMsgConfiguration ObjectConfiguration NEObjectConfiguration .

  subsorts Msg < NEMsgConfiguration < MsgConfiguration NEConfiguration .
  subsorts Object < NEObjectConfiguration < ObjectConfiguration NEConfiguration .
  subsort NEConfiguration < Configuration .
  ...

  sort RuleNameList .
  subsort RuleName < RuleNameList .
  op nil : -> RuleNameList .
  op _->_ : RuleNameList RuleNameList -> RuleNameList [assoc id: nil] .

  ops getPre getPost : ModelCheckResult -> TransitionList .
  eq getPre(b:Bool) = nil .
  eq getPost(b:Bool) = nil .
  eq getPre(counterexample(tl0:TransitionList,tl1:TransitionList))
     = tl0:TransitionList .
  eq getPost(counterexample(tl0:TransitionList,tl1:TransitionList))
     = tl1:TransitionList .

  op getRuleNames : TransitionList -> RuleNameList .
  eq getRuleNames(nil) = nil .
  eq getRuleNames({s:State,rn:RuleName} tl:TransitionList)
     = rn:RuleName -> getRuleNames(tl:TransitionList) .

  op getPathMsg : TransitionList -> TransitionList .
  eq getPathMsg(nil) = nil .
  eq getPathMsg({MC:MsgConfiguration C:Configuration, rn:RuleName} t1:TransitionList)
     = {MC:MsgConfiguration, rn:RuleName} getPathMsg(t1:TransitionList) .
endom)
```

Figura 23: Funções de projeção de partes de um contra-exemplo

As funções `getPre` e `getPost` facilitam a escolha de qual das listas de transições do contra-exemplo deve ser utilizada. Lembrando que a primeira lista de transições do contra-exemplo corresponde a um caminho finito iniciado no estado inicial, e a segunda a um *loop* [19].

Uma vez definidas estas funções, podemos utilizá-las para facilitar interpretação dos contra-exemplos exibidos no início desta seção:

```
Maude> (red getRuleNames(getPre(modelCheck(initial, [] ~ reading(reader1)))) .)
result RuleNameList :
  'READER-sending-r@read -> 'READER-sending-r@read -> 'WRITER-sending-w@write
    -> 'WRITER-sending-w@write -> 'WANT-READ-sending-in-want-read -> ...
```

5.1.4 Ceia de filósofos

Na Seção 4.5 apresentamos uma arquitetura para o problema da Ceia dos Filósofos. Sztajnberg propôs em [46] outra arquitetura para o mesmo problema, a arquitetura `CeiaFilosofos`. Aqui, vamos analisar a arquitetura `CeiaFilosofos`, inicialmente em relação à sua estrutura mostrando como uma análise formal permite a identificação de *excessos* na especificação. Isto é, mostrarmos como, a partir da intuição de que um contrato da arquitetura seria desnecessário, conseguimos utilizar Maude CBabel tool para comprovar esta intuição. Destacamos, no entanto, que nosso objetivo não é apresentar uma prova de equivalência entre arquiteturas, o que está fora do escopo deste trabalho. Em seguida, apresentamos uma discussão sobre o elevado número de estados de computação desta arquitetura e algumas alternativas para análise de arquiteturas com esta característica. Destacamos que na Seção 4.5 outra arquitetura para a aplicação Ceia de Filósofos foi utilizada, exatamente com o objetivo de adiar para esta seção as discussões supracitadas.

A arquitetura `CeiaFilosofos` é mostrada na Figura 24 (diagrama correspondente na Figura 25). O módulo `filosofo` possui quatro portas de saída para requisitar e liberar cada um dos recursos que utiliza: mesa, garfo da esquerda e garfo da direita. Os módulos `garfo` e `mesa` correspondem aos recursos sendo compartilhados. Cada um destes módulos dispõe de duas portas de entrada que devem ser acessadas em exclusão mútua, uma porta para requisição e outra para liberação do recurso. O conector `RequestTable` interliga os filósofos à mesa, ele garante que os acessos à mesa ocorram em exclusão mútua e serialmente. Também garante, através do guarda, que apenas $N-1$ filósofos consigam acesso à mesa, onde N é o número de filósofos na arquitetura. O guarda é fechado quando a mesa está ocupada por $N-1$ filósofos. Um filósofo deve, antes de requisitar os garfos, sentar-se à mesa. O conector `RequestTable` controla o número de lugares ocupados na mesa pela variável de estado `usedPlaces`, declarada no módulo `mesa`. O conector `RequestFork` interliga os filósofos aos garfos. Este conector também garante o acesso em exclusão mútua e serialmente ao garfo. O guarda em `RequestFork` utiliza a variável `Free` para manter o estado do garfo, bloqueando os acessos quando o garfo está ocupado ou liberando quando o garfo está desocupado.

Na Figura 26, apresentamos a arquitetura `NOVA-CEIA-FILOSOFOS` (o diagrama correspondente é apresentado na Figura 27), uma adaptação da arquitetura `CeiaFilosofos` onde:

- os nomes de portas e módulos foram modificados por conveniência;
- para atender a restrição de um contrato por conector, o conector `RequestTable` foi

```
port int ResrvT (int MakeReserv);
port int LevT (int ReleaseReserv);

module filosofo {
 out port ResrvT  GetLeftFork;
 out port LevT    ReleaseLeftFork;
 out port ResrvT  GetRightFork;
 out port LevT    ReleaseRightFork;
 out port ResrvT  GetTable;
 out port LevT    LeaveTable;
}

module garfo {
 in port ResrvT Reserve;
 in port LevT   Release;
}

module mesa {
 int usedPlaces;
 in port ResrvT Reserve;
 in port LevT   Release;
}

connector RequestFork {
 condition Free = true;
 exclusive, selfexclusive {
   out port ResrvT {
     guard ( Free ) {
       before { Free = false; }
     }
   }
   out port LevT {
     guard (true) {
       after { Free = true; }
     }
   }
 }
 in port ResrvT;
 in port LevT;
} ReqFork [i];
```

```
connector RequestTable (int n) {
 int MAX_PLACES = n - 1;
 condition PlaceAvailable = true;
 staterequired (int usedPlaces);
 exclusive, selfexclusive {
   out port ResrvT {
     guard (PlaceAvailable) {
       after {
        if (usedPlaces = MAX_PLACES)
             PlaceAvailable = false;
   } } }
   out port LevT {
     guard (true) {
       after {
        PlaceAvailable = true;
   } } }
 }
 in port ResrvT;
 in port LevT;
}

module CeiaFilosofos (int N) {
 instantiate mesa Mesa;
 for (i = 0 to N) {
  instantiate filosofo Filo[i];
  instantiate garfo Garfo[i];
 }
 for(i = 0 to N) {
  link Filo[i].GetLeftFork to Garfo[i]
     by RequestFork[i];
  link Filo[i].ReleaseLeftFork to Garfo[i]
     by RequestFork[i];
  link Filo[i].GetRightFork to Garfo[(i+1) mod 5]
     by RequestFork[(i+1) mod N];
  link Filo[i].ReleaseRightFork to Garfo[(i+1) mod 5]
     by RequestFork[(i+1) mod N];
  link Filo[i] to Mesa
     by RequestTable (N);
 }
 instantiate ceiaFilosofos as 5F;
}
```

Figura 24: Arquitetura `CeiaFilosofos` proposta por Sztajnberg

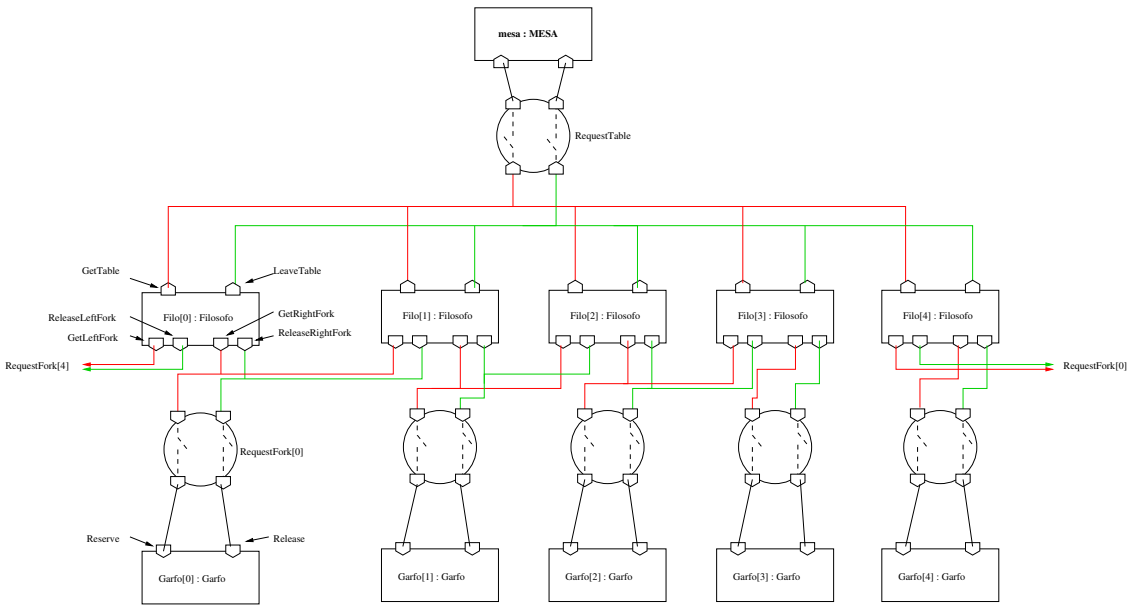

Figura 25: Diagrama da arquitetura `CeiaFilosofos`

desmembrado nos conectores REQUEST-TABLE, LEAVE-TABLE e EXCLUSIVE-COMM-TABLE; e o conector RequestFork foi desmembrado nos conectores REQUEST-FORK e LEAVE-FORK;

- outras modificações sintáticas foram feitas para atender a BNF da linguagem CBabel de Maude CBabel tool. Por exemplo, Maude CBabel tool ainda não aceita o comando *for* para "loops" na instanciação de módulos e conectores, passagem de parâmetros para instanciação de módulos e conectores, passagem de parâmetros para portas etc.

As diferenças sintáticas da arquitetura CeiaFilosofos para NOVA-CEIA-FILOSOFOS não afetam as características que desejamos analisar nas arquiteturas. No entanto, comparando o conector RequestFork, Figura 24, com os conectores da Figura 26, podemos observar que o contrato de exclusão mútua do conector RequestFork foi removido na arquitetura NOVA-CEIA-FILOSOFOS. Para mostrar que o contrato de exclusão mútua do conector RequestFork é desnecessário na arquitetura CeiaFilosofos, precisamos comprovar se na arquitetura NOVA-CEIA-FILOSOFOS os garfos são acessados em exclusão mútua. O módulo de execução para a arquitetura NOVA-CEIA-FILOSOFOS é apresentado na Figura 28.

No módulo NCF-EXEC, as regras de reescrita fork-request e fork-release definem o comportamento das instâncias de FORK, objetos da classe FORK. Estes objetos devem apenas responder as mensagens do tipo fork@request e fork@release sem alterar seu estado interno. As regras de reescrita table-request e table-release definem que a instância única de TABLE deve responder as mensagens do tipo table@request e table@release atualizando seu atributo used-places de acordo. Este atributo será consultado pelos conectores REQUEST-TABLE e RELEASE-TABLE por memória compartilhada. A mensagem init, parametrizada por um identificador de objeto, é definida para auxiliar na especificação do comportamento interno das instâncias de PHILOSOPHER. Cada instância de PHILOSOPHER comporta-se como uma máquina de estado, transitando para um novo estado tão logo uma interação iniciada no estado atual seja completada. Finalmente, o estado inicial do sistema é atribuído à constante initial, sendo a topologia da arquitetura acompanhada de uma mensagem init para cada instância de PHILOSOPHER.

Após carregar o módulo NCF-EXEC em Maude CBabel tool, estamos prontos para verificar se o contrato de exclusão mútua definido no conector RequestFork da arquitetura CeiaFilosofos é realmente necessário.

Cada par de filósofos compartilha um mesmo garfo. Estamos interessados em verificar se é possível, em algum estado, um garfo estar recebendo mais de uma requisição

```
connector REQUEST-FORK {
  staterequired bool rf@free ;
  in port rf@in ;
  out port rf@out ;
  interaction {
    rf@in > guard (rf@free) {
       before { rf@free = FALSE ; }
      }
    > rf@out ;
  }
}

connector LEAVE-FORK {
  staterequired bool lf@free ;
  in port lf@in ;
  out port lf@out ;
  interaction {
   lf@in > guard(TRUE) {
             after { lf@free = TRUE ; }
           }
   > lf@out ;
  }
}

connector EXCLUSIVE-COMM-TABLE {
 in port et@in1 ;
 in port et@in2 ;
 out port et@out1 ;
 out port et@out2 ;
 exclusive{
  et@in1 > et@out1 ; et@in2 > et@out2 ;
 }
}

connector LEAVE-TABLE {
 staterequired bool lt@place-available ;
 in port lt@in ;
 out port lt@out ;
 interaction {
  lt@in >
   guard(TRUE) { after {
     lt@place-available = TRUE ; }
    }
  > lt@out ;
 }
}

module TABLE {
 var int used-places = 0 ;
 in port table@request ;
 in port table@release ;
}
```

```
connector REQUEST-TABLE {
  var int max-places = 3 ;
  var bool place-available = TRUE ;
  staterequired int rt@used-places ;
  in port rt@in ;
  out port rt@out ;
  interaction {
   rt@in >
    guard (place-available) { after {
      if(rt@used-places == max-places){
       place-available = FALSE ;
    }}}
   > rt@out ;
  }
}

module PHILOSOPHER {
 var int state = 0 ;
 out port get-lfork ;
 out port rel-lfork ;
 out port get-rfork ;
 out port rel-rfork ;
 out port get-table ;
 out port rel-table ;
}

module FORK {
 var bool free ;
 in port fork@request ;
 in port fork@release ;
}

application NOVA-CEIA-FILOSOFOS {
 instantiate PHILOSOPHER as phi1 ;
 ...
 link phi1.get-table to et.et@in1 ;
 ...
 bind bool lfc1.lf@free to fork1.free ;
 bind bool lfc2.lf@free to fork2.free ;
 bind bool lfc3.lf@free to fork3.free ;
 bind bool lfc4.lf@free to fork4.free ;
 bind bool rfc1.rf@free to fork1.free ;
 bind bool rfc2.rf@free to fork2.free ;
 bind bool rfc3.rf@free to fork3.free ;
 bind bool rfc4.rf@free to fork4.free ;
 bind bool lt.lt@place-available to rt.place-available ;
 bind int rt.rt@used-places to table.used-places ;
}
```

Figura 26: Arquitetura `NOVA-CEIA-FILOSOFOS`

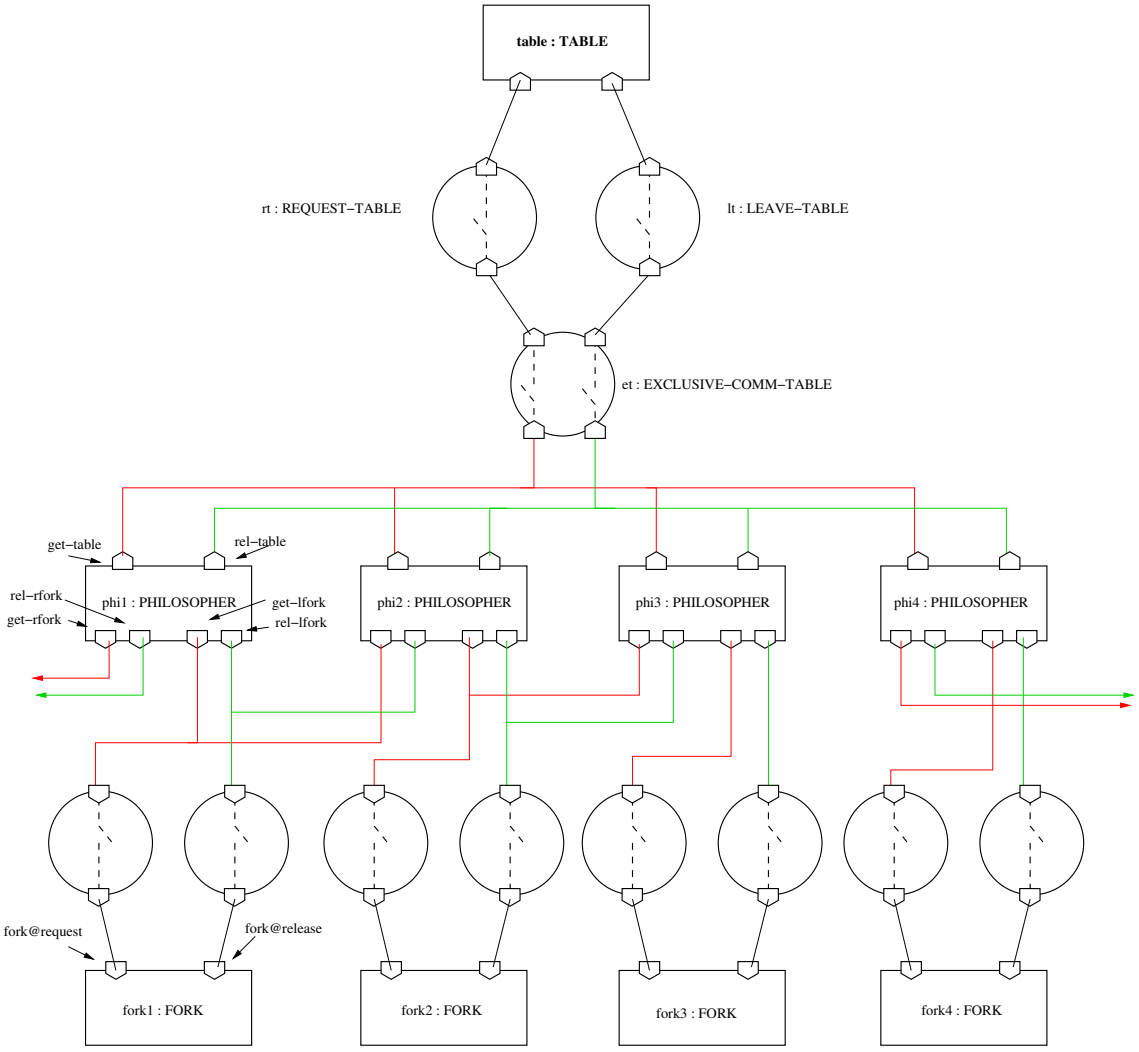

Figura 27: Diagrama da arquitetura `NOVA-CEIA-FILOSOFOS`

```
(omod NCF-EXEC is
 inc NOVA-CEIA-FILOSOFOS .

 var C : Configuration .
 var O : Oid .
 vars IT IT1 IT2 : Interaction .
 var N : Int .

 rl [fork-request] :
   < O : FORK | > do(O, fork@request, IT) =>
   < O : FORK | > done(O, fork@request, IT) .

 rl [fork-release] :
    < O : FORK | > do(O, fork@release, IT) =>
    < O : FORK | > done(O, fork@release, IT) .

 rl [table-request] :
    < O : TABLE | used-places : N > do(O, table@request, IT) =>
    < O : TABLE | used-places : (N + 1) > done(O, table@request, IT) .

 rl [table-release] :
    < O : TABLE | used-places : N > do(O, table@release, IT) =>
    < O : TABLE | used-places : (N - 1) > done(O, table@release, IT) .

 op init : Oid -> Msg .
 rl [phi-begining] :
    init(O) < O : PHILOSOPHER | state : 0 > =>
    < O : PHILOSOPHER | state : 1 > .

 rl [phi-getting-table] :
    < O : PHILOSOPHER | state : 1 > =>
    < O : PHILOSOPHER | state : 2 > do(O, get-table, none) .

 rl [phi-getting-forks] :
    done(O, get-table, IT1) < O : PHILOSOPHER | state : 2 > =>
    < O : PHILOSOPHER | state : 3 >
    do(O, get-lfork, none) do(O, get-rfork, none) .

 rl [phi-eating-releasing-forks] :
    done(O, get-lfork, IT1) done(O, get-rfork, IT2)
    < O : PHILOSOPHER | state : 3 > =>
    < O : PHILOSOPHER | state : 4 >
    do(O, rel-lfork, none) do(O, rel-rfork, none) .

 rl [phi-releasing-table] :
    done(O, rel-lfork, IT1) done(O, rel-rfork, IT2)
    < O : PHILOSOPHER | state : 4 > =>
    < O : PHILOSOPHER | state : 5 > do(O, rel-table, none) .

 rl [phi-thinking] :
    done(O, rel-table, IT) < O : PHILOSOPHER | state : 5 > =>
    < O : PHILOSOPHER | state : 0 > init(O) .

 op initial : -> Configuration .
 eq initial = topology init(phi1) init(phi2) init(phi3) init(phi4) .
endom)
```

Figura 28: Módulo de execução da arquitetura DINING-PHILOSOPHERS

simultânea. Para esta análise, podemos escolher qualquer par de filósofos que esteja conectado a um mesmo garfo. Para os demais filósofos o comportamento será o mesmo. Escolhidas as instâncias `phi1` e `phi2` de `PHILOSOPHER`, que compartilham o garfo `fork1`, podemos explorar todos os estados de computação atingíveis a partir do estado onde `phi1` estimula sua porta de saída `get-lfork` e `phi2`, estimula sua porta de saída `get-rfork`. Ambos os estímulos correspondem a mensagens para `fork1`. Podemos ainda observar que como desejamos analisar apenas as mensagens que chegam a `fork1`, não precisamos explorar os estados a partir do momento onde `phi1` e `phi2` iniciam suas transições de estado enviando mensagem para a instância `mesa`. A busca abaixo mostra que a partir do estado onde `phi1` requisita seu garfo da esquerda, iniciando o comportamento interno de sua porta `get-lfork`, e `phi2` requisita seu garfo da direita, iniciando o comportamento interno de sua porta `get-rfork`, não existem estados onde `fork1` receba, no mesmo instante, duas mensagens em alguma de suas portas.

```
Maude> (search [1] topology do(phi1,get-lfork,none) do(phi2,get-rfork,none) =>*
         C:Configuration send(fork1, P:PortId, IT1:Interaction)
                         send(fork1, P:PortId, IT2:Interaction) .)

No solution.
```

Repetindo a busca a partir do mesmo estado inicial, conseguimos mostrar que embora não cheguem juntas, as mensagens de `phi1` e `phi2` podem chegar, separadamente, a `fork1`. Veja adiante o caso para a mensagem de `phi2`:

```
Maude> (search [1] topology do(phi1,get-lfork,none) do(phi2,get-rfork,none) =>*
                   C:Configuration
                   send(fork1, fork@request, IT1:Interaction :: [phi2, P:PortId]) .)

Solution 1
C:Configuration <- < et : EXCLUSIVE-COMM-TABLE | status : unlocked >
 < fork1 : FORK | free : false >
 < fork2 : FORK | free : true >
 < fork3 : FORK | free : true >
 < fork4 : FORK | free : true >
 < lfc1 : LEAVE-FORK | lf@free : st(false,unchanged)>
 < lfc2 : LEAVE-FORK | lf@free : st(true,unchanged)>
 < lfc3 : LEAVE-FORK | lf@free : st(true,unchanged)>
 < lfc4 : LEAVE-FORK | lf@free : st(true,unchanged)>
 < lt : LEAVE-TABLE | lt@place-available : st(true,unchanged)>
 < phi1 : PHILOSOPHER | get-lfork-status : unlocked,get-rfork-status : unlocked,
    get-table-status : unlocked,rel-lfork-status : unlocked,rel-rfork-status :
    unlocked,rel-table-status : unlocked,state : 0 >
 < phi2 : PHILOSOPHER | get-lfork-status : unlocked,get-rfork-status : locked,
    get-table-status : unlocked,rel-lfork-status : unlocked,rel-rfork-status :
    unlocked,rel-table-status : unlocked,state : 0 >
```

```
 < phi3 : PHILOSOPHER | get-lfork-status : unlocked,get-rfork-status : unlocked,
    get-table-status : unlocked,rel-lfork-status : unlocked,rel-rfork-status :
    unlocked,rel-table-status : unlocked,state : 0 >
 < phi4 : PHILOSOPHER | get-lfork-status : unlocked,get-rfork-status : unlocked,
    get-table-status : unlocked,rel-lfork-status : unlocked,rel-rfork-status :
    unlocked,rel-table-status : unlocked,state : 0 >
 < rfc1 : REQUEST-FORK | rf@free : st(false,unchanged)>
 < rfc2 : REQUEST-FORK | rf@free : st(true,unchanged)>
 < rfc3 : REQUEST-FORK | rf@free : st(true,unchanged)>
 < rfc4 : REQUEST-FORK | rf@free : st(true,unchanged)>
 < rt : REQUEST-TABLE | max-places : 3,place-available : true,rt@used-places :
    st(0,unchanged)>
 < table : TABLE | used-places : 0 >
 do(phi1,get-lfork,none);
IT1:Interaction <- [rfc1,rf@out];
P:PortId <- get-rfork
```

Mostramos, assim, que na arquitetura `NOVA-CEIA-FILOSOFOS` os garfos são acessados em exclusão mútua, sem a necessidade do contrato de exclusão mútua declarado no conector `RequestFork` da arquitetura `CeiaFilosofos`.

O resultado da última busca reforça os comentários já apresentados na Seção 5.1.2, em relação a perda de clareza da especificação arquitetural quando esta é executada em Maude CBabel tool. A maior quantidade de conectores na arquitetura `NOVA-CEIA-FILOSOFOS`, resultante da restrição de um contrato por conector e, conseqüentemente, desmembramento dos contratos em diferentes conectores, certamente dificulta a compreensão da arquitetura.

Nos parágrafos acima, mostramos como analisar o comportamento de uma *parte* da arquitetura `NOVA-CEIA-FILOSOFOS` sem "executar", simultaneamente, todas as instâncias do módulo `PHILOSOPHER`. No entanto, para comprovar que esta arquitetura está livre de "deadlock", precisamos explorar os estados da computação considerando todas as instâncias de `PHILOSOPHER` em execução. Uma alternativa para análise de ausência de "deadlock" já foi apresentada na Seção 4.5. Naquela seção, executamos uma busca por estados finais de computação a partir do estado inicial da arquitetura, definido no módulo `NCF-EXEC`, Figura 28:

```
 op initial : -> Configuration .
 eq initial = topology init(phi1) init(phi2) init(phi3) init(phi4) .
```

A busca executada foi:

```
Maude> (search initial =>! C:Configuration .)
```

Em um equipamento Pentium Xeon 2.4GHz, após consumir quatro gigabytes de memória em aproximadamente três horas de processamento, a busca acima foi interrompida e o sistema Maude fechado. O alto consumo de memória é resultado do número de estados que são explorados pela busca. Na Tabela 4, apresentamos uma relação do número de estados explorados para cada combinação de instâncias de filósofos sendo iniciadas. Com apenas a instância `phi1` iniciada, o comando *search* de Maude explora 146 estados. Com as instâncias `phi1` e `phi2` sendo iniciadas são explorados 12.036 estados. Este valor está na mesma ordem de grandeza do número de estados explorados quando as instâncias `phi1` `phi3` são iniciadas, equivalente a $146^2 = 21.316$. Ou seja, para cada estado da execução de uma interação de `phi1`, `phi2` ou `phi3` poderão estar executando uma etapa de suas interações. A diferença no valor absoluto é resultado de alguns trechos de interações ocorrerem em exclusão mútua. Por exemplo, quando `phi1` e `phi2` são iniciados, não existe um estado onde as mensagens de ambos tenham passadas, juntas, pela instância `et` do conector `EXCLUSIVE-COMM-TABLE`.

Termo inicial	Estados
topology init(phi1)	146
topology init(phi1) init(phi2)	12.036
topology init(phi1) init(phi3)	21.252

Tabela 4: Número de estados na arquitetura `NOVA-CEIA-FILOSOFOS`

Da Tabela 4, concluímos que a ordem de grandeza do número de estados de computação da arquitetura é 146^n, onde n é o número de instâncias de `PHILOSOPHER` iniciadas. Não é surpresa a busca por estados finais mostrada acima ter sido abortada; seriam pesquisados 454 milhões de estados. No entanto, conforme destacado por Meseguer [57], o método de análise é completo, na medida em que se o equipamento utilizado tivesse quantidade suficiente de memória, a busca terminaria.

A falha na realização da busca acima sugere que outro método de exploração da *árvore de computação* da arquitetura seja tentado. Existem dois métodos clássicos para exploração de uma árvore, busca em largura e busca em profundidade.

A busca em largura tem como propriedade principal o fato de sempre encontrar um estado que satisfaça uma determinada propriedade, desde que este estado seja alcançável a partir do estado inicial. Além disso, o estado retornado pela busca sempre será, dentre aqueles que satisfazem a propriedade desejada, o mais próximo do estado inicial da busca. A principal desvantagem da busca em largura é o consumo de memória gerado pela necessidade de armazenar, ao menos, os estados da *fronteira* [1] da árvore visitada até

[1] As folhas da árvore até então atingidas.

o momento.

Em contrapartida, na busca em profundidade, apenas os estados no caminho do estado inicial até o estado corrente, e não todos os estados na *fronteira* da árvore, precisam ser guardados. Em uma árvore de computação com caminhos infinitos, a grande desvantagem deste método é a possibilidade do estado desejado não ser encontrado. Isto porque, a busca pode facilmente encontrar um caminho infinito, onde o estado desejado não ocorre.

Como a árvore de computação da arquitetura `NOVA-CEIA-FILOSOFOS` não é infinita, poderíamos considerar o emprego do verificador de modelos de Maude, que realiza a exploração dos estados em profundidade, para evitar o alto consumo de memória do comando *search*. Podemos, por exemplo, verificar se a partir do estado inicial, em todos os caminhos da computação, sempre existe um estado onde algum filósofo esteja comendo. Esta propriedade pode ser descrita pela seguinte fórmula LTL:

```
[]<> (eating(phi1) \/ eating(phi2) \/ eating(phi3) \/ eating(phi4))
```

onde a proposição `eating` foi definida no módulo `NCF-PROPS`, na Figura 29.

```
(omod NCF-PROPS is
  ex NCF-EXEC .
  inc MODEL-CHECKER .

 subsort Configuration < State .

 var C : Configuration .
 var O : Oid .

 op eating : Oid -> Prop .
 eq < O : PHILOSOPHER | state : 5 > C |= eating(O) = true .

 op hungry : Oid -> Prop .
 eq < O : PHILOSOPHER | state : 1 > C |= hungry(O) = true .
endom)
```

Figura 29: Módulo de análise da arquitetura `NOVA-CEIA-FILOSOFOS`

Se submetida a fórmula acima ao verificador de modelos de Maude, após duas horas de processamento o sistema Maude é novamente fechado pelo sistema operacional e a exploração de estados interrompida. Embora efetuando uma busca em profundidade, o verificador de modelos também consome grande quantidade de memória, além do limite disponível no equipamento utilizado. Este resultado não é surpreendente, pois, a árvore de estados da computação da arquitetura `NOVA-CEIA-FILOSOFOS` é mais *profunda* do que *larga*, isto é, os 454 milhões de estados da computação desta arquitetura estão distribuídos em poucos *ramos* ou *caminhos* de computação, refletindo o baixo nível de não-determinísmo da arquitetura. Isto pode ser ilustrado com o comando *show search graph* de Maude, que

nos permite imprimir a árvore de estados gerada pela última execução do comando *search*. Podemos então repetir a busca

```
Maude> search initial =>! C:Configuration .
```

interrompendo seu processamento após alguns segundos teclando control-C. A saída completa do comando *show search graph* pode ser bastante extensa. Abaixo mostramos um trecho das aproximadamente 154 mil linhas geradas em poucos segundos.

```
Maude> show search graph .
state 0, Configuration: init(phi1) init(phi2) init(phi3) init(phi4) ...
arc 0 ===> state 1 (rl init(O:Oid)
< O:Oid : V#50:PHILOSOPHER | V#57:AttributeSet,get-lfork-status :
    V#51:PortStatus,get-rfork-status : V#52:PortStatus,get-table-status :
    V#53:PortStatus,rel-lfork-status : V#54:PortStatus,rel-rfork-status :
    V#55:PortStatus,rel-table-status : V#56:PortStatus,state : 0 > => < O:Oid :
    V#50:PHILOSOPHER | state : 1,get-lfork-status : V#51:PortStatus,
    get-rfork-status : V#52:PortStatus,get-table-status : V#53:PortStatus,
    rel-lfork-status : V#54:PortStatus,rel-rfork-status : V#55:PortStatus,
    V#57:AttributeSet,rel-table-status : V#56:PortStatus > [label phi-begining]
    .)
arc 1 ===> state 2 (rl init(O:Oid)
< O:Oid : V#50:PHILOSOPHER | V#57:AttributeSet,get-lfork-status :
    V#51:PortStatus,get-rfork-status : V#52:PortStatus,get-table-status :
    V#53:PortStatus,rel-lfork-status : V#54:PortStatus,rel-rfork-status :
    V#55:PortStatus,rel-table-status : V#56:PortStatus,state : 0 > => < O:Oid :
    V#50:PHILOSOPHER | state : 1,get-lfork-status : V#51:PortStatus,
    get-rfork-status : V#52:PortStatus,get-table-status : V#53:PortStatus,
    rel-lfork-status : V#54:PortStatus,rel-rfork-status : V#55:PortStatus,
    V#57:AttributeSet,rel-table-status : V#56:PortStatus > [label phi-begining]
    .)
arc 2 ===> state 3 (rl init(O:Oid)
< O:Oid : V#50:PHILOSOPHER | V#57:AttributeSet,get-lfork-status :
    V#51:PortStatus,get-rfork-status : V#52:PortStatus,get-table-status :
    V#53:PortStatus,rel-lfork-status : V#54:PortStatus,rel-rfork-status :
    V#55:PortStatus,rel-table-status : V#56:PortStatus,state : 0 > => < O:Oid :
    V#50:PHILOSOPHER | state : 1,get-lfork-status : V#51:PortStatus,
    get-rfork-status : V#52:PortStatus,get-table-status : V#53:PortStatus,
    rel-lfork-status : V#54:PortStatus,rel-rfork-status : V#55:PortStatus,
    V#57:AttributeSet,rel-table-status : V#56:PortStatus > [label phi-begining]
    .)
arc 3 ===> state 4 (rl init(O:Oid)
< O:Oid : V#50:PHILOSOPHER | V#57:AttributeSet,get-lfork-status :
    V#51:PortStatus,get-rfork-status : V#52:PortStatus,get-table-status :
    V#53:PortStatus,rel-lfork-status : V#54:PortStatus,rel-rfork-status :
    V#55:PortStatus,rel-table-status : V#56:PortStatus,state : 0 > => < O:Oid :
    V#50:PHILOSOPHER | state : 1,get-lfork-status : V#51:PortStatus,
    get-rfork-status : V#52:PortStatus,get-table-status : V#53:PortStatus,
    rel-lfork-status : V#54:PortStatus,rel-rfork-status : V#55:PortStatus,
    V#57:AttributeSet,rel-table-status : V#56:PortStatus > [label phi-begining]
    .)
```

```
state 1, Configuration: ...
...
```

Na saída acima, pode-se observar o primeiro nível da árvore, formado pelos quatro estados atingíveis, com uma reescrita, a partir do estado inicial. O estado inicial é o estado zero, `state 0`. Os estados um, dois, três e quatro são gerados pelas diferentes possíveis aplicações da regra de reescrita `phi-begining`, do módulo `NCF-EXEC`:

```
rl [phi-begining] :
   init(O) < O : PHILOSOPHER | state : 0 > => < O : PHILOSOPHER | state : 1 > .
```

Analisando a saída completa, podemos observar que, em média, são geradas entre quatro a cinco ramificações a cada nível da árvore. Como durante a busca em profundidade o verificador de modelos armazena os estados do caminho sendo percorrido, a profundidade da árvore é grande o suficiente para causar o estouro de memória durante a exploração de um único caminho.

Fazendo uso da característica reflexiva de lógica de reescrita, e sua implementação nos recursos de meta-programação de Maude, seria possível combinarmos as boas propriedades da busca em largura e da busca em profundidade. Em [49], para análise do protocolo NSPK, os autores sugerem uma estratégia de busca em profundidade interativa:

- a árvore é percorrida em profundidade até uma profundidade d_0;
- se o estado desejado não for encontrado, repete-se a busca na árvore até uma profundidade maior, digamos $d_0 + 1$. Este processo é repetido até que o estado desejado seja localizado.

A definição de estratégias de reescrita [58] pode reduzir o número de estados da aplicação a serem explorados. Nas análises de arquiteturas, por exemplo, poderíamos definir estratégias que garantissem justiça na evolução das interações de cada módulo da arquitetura, neste caso, evitaríamos estados *injustos* onde, por exemplo, uma mensagem de `phi1` seja recebida por `table` antes que qualquer outra instância de `PHILOSOPHER` tenha encaminhado alguma mensagem.

5.2 Trabalhos relacionados

Um estudo abrangente sobre arquiteturas de "software" e ADLs é apresentado em [1] e [17]. As vantagens de prover uma ADL com uma semântica formal e ferramentas

para realização de análises formais é também assunto largamente encontrado na literatura [1, 17, 42, 59]. Como destacado em [17], uma ADL deve prover, além de uma sintaxe concreta para descrição de elementos arquiteturais (módulos, conectores, portas etc), um "framework" conceitual para caracterização de arquiteturas. Desta forma, várias ADLs estão relacionadas, ou são extensões de algum formalismo, e oferecem ferramentas para suporte nas tarefas de especificação e análise, por exemplo:

- Wright [3] é uma ADL utilizada largamente para análise de protocolos de conectores. Wright utiliza álgebra de processos, especificamente CSP [60], como notação formal para a descrição dos aspectos dinâmicos de uma particular arquitetura ou estilo arquitetural. Em Wright, um conjunto padrão de verificações de consistência e completude (por exemplo, verificação de "deadlock" dos conectores) é definido para cada arquitetura. Estas verificações são definidas precisamente em termos do modelo semântico de Wright em CSP e podem ser realizadas com o auxílio de verificadores de modelos. O verificador de modelos mais utilizado é o FDR (Failure/Divergences Refinement) [61]. Atualmente, as ferramentas disponíveis para Wright compreendem: um analisador sintático de descrições arquiteturais; um tradutor de Wright para CSP; um tradutor de Wright para Acme e um tradutor de Acme para Wright. Para a especificação de arquiteturas em Wright um projetista deve estar familiarizado com a notação de CSP. Para análise da arquitetura, deve-se ainda conhecer o modelo semântico de CSP, "failure-divergence", adotado pelo verificador de modelos FDR. As análises de descrições Wright em FDR são facilitadas pela capacidade do tradutor Wright-CSP gerar automaticamente testes predefinidos na linguagem do verificador de modelos FDR. Exemplos destes testes são: possibilidade de"deadlock" dos conectores, consistência da computação descrita para as portas, possibilidade de "deadlock" das *roles* dos conectores e teste de compatibilidade entre portas dos módulos e *roles* dos conectores.

- Em Rapide [62] os tipos de conectores são predefinidos. Essencialmente, um conector determina como eventos de saída são produzidos em uma porta (interface) quando eventos de entrada ocorrem em outra porta. Os aspectos dinâmicos da arquitetura são descritos por padrões de eventos de forma reativa ou no estilo de programação procedural. O modelo de computação é, desta forma, baseado em eventos. Cada atividade significativa da computação de um componente é descrita por um evento que pode ser temporizado por uma duração. A relação de dependência entre eventos também pode ser representada através de padrões e ex-

pressões. As ferramentas de análise disponíveis para Rapide baseiam-se na análise do comportamento da arquitetura em tempo de execução. Ou seja, Rapide pode funcionar como um tipo de linguagem de simulação de arquiteturas. Conjuntos de traços de eventos ("posets") podem ser examinados para determinar se satisfazem uma relação de ordem desejada. Dentre as ferramentas de análise oferecidas para Rapide estão: RapArch, o editor gráfico de arquiteturas e *Rapide Compiler*, capaz de criar arquiteturas executáveis (programas Rapide). A execução de um programa Rapide produz um arquivo de saída com uma ordem casual de eventos. Outras ferramentas podem ser utilizadas para análise deste arquivo de saída: POV, um visualizador gráfico da ordem de eventos simulada; Raptor, um animador que mostra graficamente a execução do programa Rapide; e, um verificador de assertivas que analisa se as assertivas definidas na arquitetura se verificaram em sua simulação. A utilização de "posets" de eventos como modelo para descrição do comportamento dos componentes permite que projetistas de arquiteturas menos familiarizados com métodos formais possam facilmente utilizar as ferramentas de análise da Rapide.

- Acme [63] é uma ADL genérica, proposta para ser utilizada como linguagem intermediária para tradução de descrições arquiteturais entre ADLs. Por esta razão, Acme não está relacionada a nenhum modelo formal específico. Acme é extensível. Armani [64], por exemplo, é uma extensão de Acme para modelagem de "constraints" arquiteturais. Dentre as ferramentas de especificação e análise disponíveis para Acme está o AcmeStudio [65], que permite a modelagem de sistemas em um editor gráfico que também realiza alguns tipos de verificações estáticas e semânticas. No AcmeStudio, as análises são realizadas por "plug-ins", sendo que na versão atual, o único "plug-in" disponibilizado é o "Intergrated Armani constraint checker", para verificação de regras de "design", isto é, teste das assertivas a respeito da estrutura da arquitetura, por exemplo: quantidade de portas em um componente, ligações de portas compatíveis e verificação de portas não conectadas. Acme não dispõe de construções para especificação dos aspectos dinâmicos da arquitetura. No entanto, tais aspectos podem ser guardados em *propriedades*, quando capturados na tradução de descrições em outras ADLs para Acme. AcmeStudio é um ambiente de fácil utilização, sendo acessível mesmo a projetistas sem familiaridade em métodos formais.

Em relação a utilização de métodos formais na análise de arquiteturas de "software", Shaw e Garlan [1] enumeram quatro itens principais sujeitos à formalização na área de

arquitetura de "software":

- A arquitetura de um sistema específico.
- Um estilo ou uma abstração arquitetural.
- a teoria da arquitetura de "softwares", isto é, os conceitos relacionados a arquiteturas de "software" como: componentes, portas, conectores etc.
- Semântica de ADLs.

Embora nosso trabalho esteja enquadrado no último dos itens acima, relacionamos inicialmente alguns trabalhos relativos aos demais itens citados.

Uma grande variedade de diferentes formalismos foram propostos para descrição de arquiteturas de "software" ou formalização de conceitos relacionados a ADLs. Inverardi e Wolf [41] propuseram a utilização de "Chemical Abstract Machine Model" (CHAM) para a descrição de uma arquitetura em termos de sua semântica operacional. Métayer [66] propôs a descrição de estilos arquiteturais com gramáticas de grafos e instâncias de descrições arquiteturais como grafos. A linguagem Z [67] é usada para descrição de estilos arquiteturais em [68, 69]. Em [70], o modelo de componentes Enterprise JavaBeans foi formalizado em Promela [71] e sua especificação comportamental foi analisada no verificador de modelos SPIN [71]. Fiadeiro e Lopes apresentam em [72] a formalização de conectores em *categorias*, mostrando como outros formalismos podem ser utilizados para substituir CSP em especificações de arquiteturas em Wright [3]. Os autores apresentam ainda uma extensão da linguagem COMMUNITY [73] para definição de conectores. Em [45, 74] são apresentados alguns trabalhos que utilizam lógica de reescrita para formalização de arquiteturas e sistemas distribuídos.

Existem ainda na literatura trabalhos que utilizam diretamente "framework" semânticos para especificação de arquiteturas de "software". Maude [19], por exemplo, foi utilizado: (i) na especificação formal do "framework" MDS ("Mission Data System") da NASA [75], utilizado para especificação e implementação dos sistemas das missões espaciais; (ii) na especificação formal das Redes Ativas e seus protocolos [28] e na formalização do "framework" *Reference Model of Open Distributed Processing* (RM-ODP) [76]. Embora a capacidade de modelagem de alguns formalismos como lógica de reescrita, Redes de Petri, Statecharts e CHAM sejam similares a de algumas ADLs, alguns trabalhos como Medvidovic e Taylor [17] e Allen e Garlan [3] destacam os benefícios de se prover

uma notação com construções explícitas para abstrações arquiteturais como componentes, portas, conectores e configurações.

Em uma linha mais abrangente, Félix [42] apresenta uma metodologia para utilização de métodos formais na arquitetura de "software". Uma de suas contribuições é a definição de uma notação formal, chamada NDA, para expressar abstrações arquiteturais, e uma semântica para esta notação, expressa como regras de transformação das estruturas de NDA em CCS [77]. Como sua pesquisa não é orientada a ADLs, nenhuma ADL em especial é considerada por Félix ou mapeada para a notação NDA.

Dos trabalhos mais diretamente relacionados à nossa pesquisa, isto é, relacionados à semântica de ADLs e análises de arquiteturas a partir de descrições arquiteturais em ADLs, podemos citar:

Lichtner, Alencar e Cowan descrevem em [78] um "framework" para a realização de análises em arquiteturas de "software" descritas por uma ADL. A proposta é baseada na tradução da descrição arquitetural em alguma ADL [2] para um modelo matemático e, posteriormente, a mecanização das análises com o provador de teoremas PVS [79]. O modelo matemático para descrição dos elementos arquiteturais e seus relacionamentos é definido em [16]. Para a definição deste modelo foi utilizada a linguagem Z [67]. Em [78] os autores descrevem como uma descrição arquitetural pode ser traduzida para a linguagem do provador de teoremas PVS, baseada em lógica de alta-ordem, usando como base o modelo matemático por eles definido. A tradução da descrição arquitetural em uma ADL para a linguagem do PVS é guiada pelo modelo, mas não automática. Sendo assim, requer do projetista do sistema conhecimentos básicos sobre o PVS. Nos exemplos apresentados em [78], todas as análises apresentadas são análises estáticas da arquitetura, isto é, restringem-se à análise da estrutura da arquitetura, por exemplo: verificação dos tipos das instâncias de componentes ou compatibilidade das portas nas conexões. Não são apresentadas análises a respeito do comportamento dinâmico da arquitetura.

Stafford, Richardson e Wolf apresentam uma técnica, chamada "chaining", e a ferramenta Aladdin, para análise de dependências em arquiteturas de "softwares" [80]. Originalmente, a análise de dependências é aplicada no nível da implementação, para otimização de códigos a partir da identificação das reestruturações do código que podem ser realizadas com segurança. Os autores apresentam a aplicação da análise de dependências no nível de abstração da arquitetura de "software" para análise de aspectos estáticos e comportamentais da arquitetura, procurando responder questões como: quais componen-

[2] O "framework" não é dependente de uma ADL específica.

tes do sistema podem receber notificação de determinados eventos; quais componentes serão afetados pela remoção ou substituição de um componente; quais componentes do sistema não são necessários por nunca serem utilizados por outros componentes, dentre outras. As análises de arquiteturas são realizadas em dois passos. Primeiro, uma representação intermediária da arquitetura é gerada e em seguida as análises são realizadas nesta representação. A representação utilizada é um conjunto de células onde cada célula representa um conjunto de relacionamentos entre pares de elementos da arquitetura. Na versão descrita da ferramenta Aladdin, apenas a tradução de especificações em Rapide são apresentadas, embora a tradução de outras ADLs seja apresentada como tarefa trivial.

Em contrapartida aos dois trabalhos apresentados acima, em nossa abordagem, tanto os aspectos estáticos quando dinâmicos podem ser analisados, embora nesta dissertação apenas análises de comportamento tenham sido apresentadas.

Steggles e Kosiuczenko apresentam em [81] uma semântica para SDL ("Specification and Description Language") em TRL ("lógica de reescrita temporalizada"). SDL [82] é uma linguagem formal para descrição de sistemas distribuídos de tempo real largamente utilizada pela indústria para descrição de sistemas de telecomunicação. Pelo seu nível de abstração na descrição de sistemas e outras características usuais de uma ADL, SDL pode ser considerada uma ADL. TRL [83] é um formalismo algébrico que estende lógica de reescrita com a adição de restrições de tempo nas regras de reescrita. Em TRL, o comportamento dinâmico de sistemas de tempo real é especificado por regras de reescrita temporizadas. Na inexistência de uma ferramenta que implementasse TRL, TRL foi mapeada para lógica de reescrita [18] na forma de uma função de transformação. Uma prova de equivalência detalhada é apresentada mostrando a correção da função de transformação. Das ferramentas disponíveis que implementam lógica de reescrita, os autores escolheram Elan [44] para mecanização das análises no exemplo apresentado. A abordagem de Steggles e Kosiuczenko para formalização de SDL é muito próxima da nossa abordagem para formalização de CBabel, na medida em que ambas as propostas se baseiam na formulação de uma semântica transformacional da ADL para uma teoria no "framework" semântico.

Recentemente, pesquisas na área de arquitetura de "software" têm se concentrado na forma de utilização de ADLs para especificação de aspectos não funcionais da arquitetura [84, 85] e em como garantir que, em tempo de execução, tais aspectos sejam atendidos. Sendo assim, aspectos que envolvam coordenação de interações complexa entre componentes, tempo real e qualidade de serviço (QoS), tornam-se parte da descrição

arquitetural em uma ADL. Com a utilização dos conceitos de contratos, CBabel permite o tratamento de aspectos de coordenação de maneira mais flexível que outras ADLs [84]. CBabel também oferece contratos de QoS para capturar aspectos não-funcionais da arquitetura [86]. Embora os contratos de QoS de CBabel não tenham sido considerados nesta dissertação, a extensão de nosso mapeamento e da ferramenta Maude CBabel tool, para suporte destes contratos constitui parte de nossos trabalhos futuros (vide Capítulo 6).

5.3 Comparação com os trabalhos relacionados

Na Seção 4.1, vimos que para analisar uma arquitetura em Maude CBabel tool, precisamos definir em Maude o comportamento interno observável dos módulos, isto é, o comportamento dos módulos necessário para as análises que serão realizadas na arquitetura. Vimos ainda que a especificação dos comportamentos internos dos módulos torna "executável" o modelo de objetos e mensagens gerado a partir das descrições CBabel. Isto porque, ao definir como os objetos irão responder ou enviar mensagens, estamos definindo como as instâncias dos componentes da arquitetura irão responder a estímulos em suas portas de entrada e quando irão gerar estímulos em suas portas de saída. Mas como outras ADLs tratam da especificação do comportamento dos componentes?

Para Félix [42], a descrição de um módulo compreende a especificação de seu nome e interface, dada pela lista de portas nomeadas e definidas como processos interativos, e ainda:

- A ordem de ativação das portas, do ponto de vista externo, dada por meio de esquemas de autômatos.
- Seus atributos, através de um especificação algébrica dos tipos de dados abstratos utilizados.
- A sua atividade interna que realiza mudanças em seus atributos, dada por um autômato que descreve um processo seqüencial. Esta informação é opcional, mas necessária para a realização de análises formais de propriedades sobre dados.

Félix descreve ainda que é preciso que haja alguma compatibilidade (equivalência observacional) entre a ordem de ativação das portas e o fluxo de controle das atividades, pois são entre os passos de computação (atualização de valores dos atributos) que o módulo realiza interações com o ambiente (demais componentes) por meio de suas portas.

Em Wright, um módulo compreende a especificação de sua interface, lista de portas e sua computação. A computação do componente descreve seu comportamento interativo com o ambiente através de suas portas, ou seja, representa precisamente o comportamento observável descrito por Félix. Como dito anteriormente, em Wright este comportamento observável do componente é descrito em CSP [60], não existindo construções para descrição de atributos do componente ou do comportamento interno do componente (manipulação destes atributos). No entanto, para análises de propriedades sobre os dados, descrições Wright podem ser combinadas a especificações na notação Z [67], conforme descrito em [3]. Esta combinação permite o mapeamento do modelo de estados na execução do sistema, construindo uma seqüência de mudanças de estado que corresponde a traços do modelo de eventos de Wright, especificado em CSP.

Em Rapide, a interface de um componente é descrita através de *actions*, que podem ser de entrada e saída, e especificam a habilidade do componente de observar ou emitir determinados eventos. A seção "behavior" da descrição de cada componente pode conter declarações de variáveis locais e a descrição da computação que o componente realiza, isto é, como o componente deve reagir a *actions* de entrada e como deve gerar *actions* de saída. Como dito anteriormente, computações são descritas por padrões de eventos. Padrões de eventos são conjuntos de eventos relacionados através de uma ordem parcial (poset).

Em Acme não existem construções para descrição dos comportamentos (internos ou observáveis), estes comportamentos, quando capturados na tradução de descrições arquiteturais de outras ADLs em Acme, são meramente armazenados em *propriedades*.

Em CBabel também não existem construções sintáticas para descrição do comportamento observável da interface (padrão de comportamento interativo) do módulo. Da mesma forma, não existem construções para descrição da computação interna do componente, relativas às ações internas do componente de manipulação de seus atributos. Sendo assim, o comportamento interno e o comportamento observável da interface dos módulos, que em Rapide são especificados por padrões de eventos, precisam em nossa abordagem ser especificados de maneira *ad-hoc*, em Maude, no módulo de execução. Este comportamento é definido através de regras de reescrita para o tratamento e envio das mensagens `do` e `done`. Tais regras correspondem à especificação do comportamento interno dos componentes, pois, podem envolver a alteração do estado do objeto e também correspondem à especificação do comportamento observável da interface do módulo, uma vez que especificam a ordem de ativação das portas. No entanto, do ponto de vista do

comportamento observável, é especificado tanto quanto o necessário para que exista uma compatibilidade (simulação) entre o comportamento do módulo e o comportamento dos conectores a ele ligados. Do ponto de vista do comportamento interno, é especificado tanto quanto o necessário para as análises de propriedades sobre dados (atributos dos objetos). Em suma, completamos em Maude a especificação da arquitetura, uma vez que CBabel não oferece construções para a especificação do comportamento dos componentes.

A especificação do comportamento observável do componente é *ad-hoc*, pois, não existe uma verificação de suficiência desta especificação contra a análise a ser realizada. Neste sentido, em Félix [42], deve haver uma simulação entre o comportamento observável especificado no componente e o comportamento dos conectores a ele ligados. Em Allen [3], o transformador de Wright para CSP/FDR é capaz de gerar, para cada porta de componente conectada a um *papel* de conector, o teste de compatibilidade entre a porta e o papel. No entanto, é especificado em Maude tanto quando em Rapide é especificado na seção *behavior* de cada componente. Em relação a Wright, nossa abordagem tem ainda como benefício a utilização de um único formalismo, lógica de reescrita, para descrição do modelo de estados e do modelo de comportamentos dos componentes.

Na Seção 5.1.4, vimos que nosso estudo não tratou de equivalência de arquiteturas. Isto porque, para abordar tal questão, seria necessária a existência de uma semântica de CBabel em um modelo formal básico, como um sistema de transição (ST). Se tal semântica existisse, poderíamos traduzir duas arquiteturas a serem comparadas em seus respectivos sistemas de transição e então avaliar a existência de uma bissimulação entre estes sistemas de transição. A existência de uma semântica para CBabel em um sistema de transição também permitiria a prova de corretude de nosso mapeamento de CBabel para lógica de reescrita, uma vez que poderíamos avaliar a existência de uma bisimulação entre o sistema de transição de uma descrição CBabel e o modelo da teoria de reescrita gerada para esta mesma descrição arquitetural.

Como vimos na Seção 5.1.4, em uma arquitetura que contenha uma quantidade razoável de instâncias de módulos e conectores, a representação de um estado de sua computação, no nível Maude, envolve um quantidade grande de objetos e mensagens, dificultando sua interpretação. A atual restrição do mapeamento em relação à descrição de apenas um contrato por conector contribui mais ainda para o aumento de objetos e mensagens na representação do estado. A implementação de uma interface de comandos no nível de abstração de CBabel permitiria uma representação mais atraente e simplificada para os estados da execução da arquitetura e, conseqüentemente, para os caminhos de uma

simulação. Neste aspecto, as ferramentas Rapitor e POV de Rapide são uma referência, oferecendo visualização gráfica das simulações de uma arquitetura. No aprimoramento da interface de comandos de Maude CBabel tool, um visualizador gráfico dos contra-exemplos do verificador de modelos de Maude poderia ser desenvolvido. Este visualizador poderia ser desenvolvido seguindo o mesmo princípio de funcionamento do Rapitor, onde a visualização dos eventos é gerada a partir de um arquivo de saída produzido durante a simulação da arquitetura.

Em relação à escalabilidade da ferramenta, durante a implementação dos estudos de caso, identificamos que, em algumas arquiteturas, menos de uma dezena de instâncias de módulos e conectores é suficiente para tornar o sistema de transição de estados da computação da arquitetura suficientemente grande para as ferramentas de análise utilizadas. Isto porque tanto o verificador de modelos como o comando *search* de Maude baseiam-se na exploração exaustiva dos estados da arquitetura para comprovação de alguma propriedade. Embora seja resultado conhecido na literatura que a eficiência da técnica de verificação de modelos depende da complexidade da aplicação [23, 87], destacamos:

- Conseguimos com sucesso aplicar a verificação de modelos e buscas em arquiteturas com até 500 mil estados de computação. Nossos resultados, em termos de consumo de memória e tempo de processamento, são compatíveis aos resultados alcançados por Eker [23] durante a análise de desempenho do verificador de modelos de Maude.

- Nakajima [70] apresenta análises de comportamento do modelo de componentes Enterprise JavaBeans. Também neste trabalho, é reportado como o aumento do número de estados durante a verificação de modelos é expressivo para relativamente poucos *objetos* em execução. Como nós, Nakajima sugere como solução a aplicação de abstrações para contornar o problema da explosão de estados.

- Como dito na Seção 5.2, Wright adota CSP como linguagem formal para descrição do comportamento dos conectores. Análises de comportamento de arquiteturas descritas em Wright são geralmente feitas com o verificador de modelos FDR [61]. Em sua tese [3], Allen também descreve o problema da explosão de estados na verificação de modelo de arquiteturas de "softwares" complexas. A abordagem sugerida por Allen é a verificação modular da arquitetura. Na realidade, Wright é largamente utilizada para análise de protocolos de conectores, por isso, conforme reportado em [70], em todos os exemplos encontrados na literatura de análises de comportamento de arquiteturas descritas em Wright, apenas um único conector é declarado e analisado. Na Seção 5.1.4, mostramos que com Maude CBabel tool

também é possível analisarmos *partes* da arquitetura de cada vez. No entanto, na maioria das análises efetuadas nas seções anteriores, toda a arquitetura foi simulada de uma vez. Entretanto, o verificador de modelos FDR, utilizado nas análises de descrições CSP, apresenta melhor resultado que nossa ferramenta. No tutorial de Wright [88], James Ivers relata que o FDR foi capaz de verificar um modelo com três milhões de estados em aproximadamente 3 horas, em um computador de médio porte.

6 Conclusão

Apresentamos aqui alguns comentários finais e os resultados alcançados por nossa pesquisa, comparando nossa abordagem com as outras abordagens encontradas na literatura apresentadas na Seção 5.2. Vamos sumarizar as principais contribuições de nossa pesquisa e apontar algumas direções para investigações futuras que poderão continuar a pesquisa iniciada com esta dissertação.

6.1 Resultados alcançados e contribuições

Nesta dissertação apresentamos uma semântica para a ADL CBabel em lógica de reescrita. Componentes CBabel foram mapeados para teorias de reescrita, mais precisamente, para módulos orientados a objetos em Maude. Também apresentamos a ferramenta Maude CBabel tool, uma implementação direta em Maude da semântica de CBabel. Mostramos através dos estudos de caso apresentados nos Capítulos 4 e 5 que com Maude CBabel tool podemos executar e analisar descrições arquiteturais em CBabel transformadas em teorias de reescrita em Maude. Nesta seção, destacamos os resultados alcançados e contribuições de nossa pesquisa.

O encapsulamento do formalismo. Como a transformação de CBabel para lógica de reescrita é de fato a semântica de CBabel, nossa abordagem tem a interessante propriedade de efetivamente *executar* descrições arquiteturais em CBabel para a realização de simulações (isto é, reescritas da topologia) e realização de análises. Mais ainda, a sintaxe orientada a objetos de Maude fornece uma interpretação intuitiva para os conceitos de CBabel mapeados para lógica de reescrita, sendo ainda facilmente entendida pela maioria dos projetistas de "softwares". Acreditamos que nossa abordagem consegue assim atender a dois propósitos importantes. Por um lado, ao apresentarmos uma semântica formal para CBabel, dando significado preciso e não-ambíguo às construções da linguagem, tornamos precisa a interpretação de uma descrição arquitetural em CBabel e tornamos factível a realização de análises formais nesta arquitetura. Ao mesmo tempo, facilitamos a adoção

de métodos formais por parte dos projetistas de sistemas, nem sempre acostumados com ferramentas formais como: (i) provadores de teoremas como o PVS [79], adotado por Lichtner, Alencar e Cowan [16, 78]; (ii) álgebras de processos como CSP, modelo formal da Wright [3]; ou, (iii) notações matemáticas para especificação de modelos como a linguagem Z [67], utilizada em [68, 69]. Nakajima e Tamai [70] observam por exemplo que a utilização do verificador de modelos FDR (Failure/Divergences Refinement) [1], para análise do comportamento de componentes de uma arquitetura descrita em Wright, requer do projetista da arquitetura alguma familiaridade com a semântica de CSP no modelo "failure-divergence" [61]. Assim como é necessária familiaridade em expressar as propriedades que serão verificadas em termos de relações de refinamentos [61], conceitos menos acessíveis à maioria dos projetistas de sistemas. No entanto, em nossa ferramenta, a especificação das propriedades relacionadas ao comportamento da arquitetura exige do projetista conhecimentos de lógica temporal, especificamente, lógica linear temporal (LTL) (Seção 2.4.3). Em versões futuras de Maude CBabel tool, pretendemos estudar a possibilidade de geração automática, a partir da descrição arquitetural, das propriedades em LTL necessárias para a análise da arquitetura. Isto seria equivalente aos testes padrões gerados pela ferramenta de tradução de Wright para CSP (vide Seção 5.2).

O tratamento de aspectos estruturais e comportamentais dentro do mesmo formalismo. A escolha de lógica de reescrita como "framework" semântico tem como importante benefício o tratamento *ortogonal* que esta dá aos aspectos estruturais da arquitetura (tipos de dados), definidos por equações, e os aspectos comportamentais (concorrência e sincronização), dado pela regras de reescrita. Félix [42] também adota esta separação para formalização de sistemas utilizando, no entanto, dois diferentes formalismos: álgebra de processos, para formalização do comportamento interativo e concorrência; e lógica equacional, para formalização das propriedades sobre dados. Wright utiliza CSP para descrição do comportamento e a notação Z [67] para descrição dos estados de computação. A adoção de lógica de reescrita permite que análises sobre propriedades relacionadas a ambos os aspectos de uma arquitetura possam ser executadas de maneira homogênea, pois, ambos os aspectos são representados no mesmo formalismo.

A tradução modular. Um importante aspecto da transformação de descrições CBabel para lógica de reescrita é sua modularidade. Um componente CBabel pode ser completamente transformado em uma teoria de reescrita sem qualquer informação sobre os demais componentes da arquitetura. As mensagens *do* e *done* são utilizadas com este objetivo. Elas permitem o encapsulamento, na teoria de reescrita que representa o módulo

[1] Um verificador de modelos para CSP.

CBabel, do tratamento da liberação ("unlock") e travamento ("lock") das portas. De outra maneira, as equações que dão semântica às declarações de ligações de portas seriam mais complexas do que simplesmente a renomeação de mensagens. Ainda com o objetivo de preservar a modularidade, utilizamos as mensagens *send* e *ack* para implementação da comunicação síncrona entre componentes. Conforme apresentado na Seção 2.3.6, a comunicação síncrona entre objetos é geralmente implementada em Maude por uma regra de reescrita que contenha os objetos que se comunicam de forma síncrona no seu lado esquerdo. Se utilizássemos este modelo para nosso mapeamento, não poderíamos definir a semântica de uma porta síncrona sem considerar suas ligações, o que claramente não seria uma semântica modular. Modularidade permite ainda que técnicas de prova modular [87, Capítulo 12] possam ser aplicadas, pois existe uma relação 1-1 entre os objetos e instâncias de módulos e conectores. Mais ainda, a preservação da modularidade da descrição arquitetural no modelo formal facilita o ciclo de análise da arquitetura e ajustes que possam ser necessários. Em adição ao fato de que modularidade é uma importante propriedade para especificações, acreditamos que modularidade será bastante relevante para tratamento de questões relacionadas à reconfiguração de arquiteturas [2], um importante conceito em arquitetura de "software" relacionado como um dos trabalhos futuros propostos. A preocupação com modularidade também está presente no trabalho de Steggles e Kosiuczenko [81] (vide Seção 5.2).

A versatilidade da ferramenta. Nos Capítulos 4 e 5 foram analisadas propriedades relacionadas a sincronização e coordenação entre componentes de uma arquitetura como: condição de corrida (acessos concorrentes a um recurso comum que devem ser evitados), "deadlock" e "livelock". Com a utilização de simulação, também analisamos aspectos sobre a estrutura da arquitetura com sua correta especificação das ligações de portas e amarrações de variáveis. Neste contexto, acreditamos que Maude CBabel tool constitui-se uma ferramenta de propósito mais amplo, não sendo restrita a nenhum tipo particular de análise, como é o caso da ferramenta Alladin [80], exclusivamente voltada para análise de dependências de componentes de uma arquitetura.

As contribuições para o (re)design da linguagem CBabel. Durante a especificação formal de CBabel, vários aspectos da linguagem foram discutidos, contribuindo para seu aperfeiçoamento a partir da remoção de suas ambigüidades. Na formalização do contrato de interação, foi discutida a validade de se permitir que portas síncronas e assíncronas possam participar de um mesmo contrato de interação na especificação de um conector. Optamos em nossa semântica pela restrição a portas de mesmo tipo, considerando assim os casos: entrada síncrona/saída síncrona e entrada assíncrona/saída

assíncrona. Os casos restantes, no entanto, foram relacionados para investigação futura, [2] podendo ou não serem considerados válidos em versões futuras da linguagem. Ainda durante a formalização dos contratos de CBabel, algumas contribuições sobre a gramática da versão inicial de CBabel, apresentada em [46], foram dadas. A definição do contrato de exclusão mútua sobre portas assíncronas, permitida pela gramática, foi questionada. Na semântica que apresentamos (vide Seção 3.1.3), a liberação do semáforo não seria possível neste caso, ou por outro lado, se a semântica fosse modificada para liberação do semáforo instantaneamente, estaríamos definindo apenas um escalonador (vide Seção 3.1.3). Outra mudança proposta foi a definição dos contratos sobre interações, e não mais sobre portas, como proposto inicialmente por Sztajnberg. Esta mudança aproxima a sintaxe da própria intuição dos contratos, que dão significado a tipos de interação e coordenação de interação entre portas e não somente a portas. Ainda em relação a gramática original de CBabel, diferentes composições de contratos em um conector são permitidas. Durante nosso estudo, identificamos que nem todas as composições têm semântica consistente. Embora tenhamos adotado em nosso estudo a restrição de um contrato por conector para simplificarmos a semântica dos contratos e conectores, constatamos durante nossos estudos que a gramática de CBabel precisa ser revista para restringir as possibilidade de composições de contratos apenas aos casos válidos, isto é, que tenham algum significado. Por fim, em comparação à outras ADLs como Rapide e Wright, CBabel apresenta como limitação a falta de construções para descrição do comportamento interno e observável dos componentes. Para a realização de análises que envolvam a exploração dos estados de computação das arquiteturas, isto é, a simulação das arquiteturas, a descrição do comportamento dos componentes é fundamental. Em Maude CBabel tool, o comportamento dos módulos é descrito diretamente em Maude. Acreditamos que versões futuras de CBabel poderiam prever uma forma de descrição do comportamento dos módulos da arquitetura, possivelmente utilizando alguma abordagem próxima da utilizada por Rapide.

Mais uma evidência para lógica de reescrita como "framework" semântico. Como destacado em [17], os tipos de análise para os quais uma ADL é adequada depende do modelo semântico adotado por esta ADL. Nosso trabalho contribui com uma semântica para CBabel bastante flexível e abrangente, capaz de representar diferentes aspectos da linguagem CBabel e propriedades a serem analisadas sobre as arquiteturas descritas em CBabel. A flexibilidade e abrangência são devidas às características de lógica de reescrita como lógica e um formalismo semântico, comprovadas por extensa literatura já publi-

[2] A conexão de portas síncronas e assíncronas em um contrato de interação envolve questões relacionadas a casamento de tipos de portas. Ou seja, questões que podem ser tratadas por um analisador sintático da linguagem.

cada [45] a seu respeito. O fato de lógica de reescrita ser, sobre condições razoáveis, executável, permite ainda que sistemas especificados em lógica de reescrita possam ser submetidos a uma relação de métodos formais de análise. Os métodos formais suportados por lógica de reescrita variam, desde métodos menos custosos e complexos ("lighter") (como especificação formal, simulação e exploração de estados), até métodos mais formais e rigorosos ("heavier"), como prova de teorema. Com a adoção de Maude, tornamos todas as ferramentas de análise formal de Maude disponíveis para serem utilizadas na validação de arquiteturas de "software" descritas em CBabel. Desta forma, fornecemos uma ferramenta formal de análise de amplo domínio e extensível para CBabel. Acrescentamos assim à vasta literatura de lógica de reescrita mais um estudo de caso sobre sua aplicação como formalismo semântico para linguagens de especificação.

Contribuições à implementação de Full Maude. Em relação a Maude e Full Maude, destacamos as contribuições realizadas durante a implementação de Maude CBabel tool como: identificação e correção de alguns "bugs" na implementação de Full Maude relacionados à implementação do comando *search* e *pretty-print*, solução para a limitação da utilização do verificador de modelos no ambiente Full Maude. Todas as contribuições foram reportadas à equipe que mantém o sistema Maude. Também destacamos nossas dificuldades como forma de contribuição ao desenvolvimento de suas versões futuras. Em primeiro lugar, falta documentação atualizada sobre Full Maude que facilite sua extensão. No desenvolvimento de Maude CBabel tool a fonte mais atualizada de informação foi o próprio código de Full Maude e os comentários neste. Em segundo lugar, Full Maude foi concebido como uma álgebra de módulos extensível, desta forma, sua utilização como "framework" semântico para uma linguagem $\mathcal{L}$ é facilitada quando $\mathcal{L}$ é uma linguagem de especificação algébrica. Como CBabel não é uma linguagem de especificação algébrica, a transformação dos módulos CBabel em módulos orientados a objetos não pode ser implementada como uma extensão da função de transformação definida por Full Maude para ser estendida [25]. A função `cb2omod`, descrita na Seção 3.2, teve de ser aplicada em uma etapa anterior ao momento em que Full Maude aplica sua função de transformação sobre as entradas.

6.2 Trabalhos futuros

No nosso entendimento, a nossa abordagem é inovadora ao prover um ambiente executável, que inclui ferramentas de verificação, para uma ADL, baseado na semântica formal desta ADL. Existem, no entanto, vários itens a serem ainda investigados.

- A definição de uma interface de comandos completa que entenda termos relacionados à arquitetura de "software" como componentes, portas e conectores e não classes, objetos e mensagens. Esta interface também poderia traduzir as respostas das ferramentas de análise para termos relacionados a arquitetura de "software", no nível de abstração de CBabel. Além disso, também poderia permitir o controle das simulações através de diferentes políticas de escalonamento para os componentes, especificadas propriamente através de comandos no nível de abstração de CBabel.

- Nossa semântica permite apenas um contrato por conector. Esta escolha permitiu tornar a semântica mais simples, conforme destacado na Seção 3.1.3. No entanto, a descrição de uma arquitetura torna-se muito mais simples quando permitimos que um conector especifique mais de um contrato. Em versões futuras de Maude CBabel tool isto deverá ser possível a partir da definição de uma semântica para a composição de contratos.

- A atual sintaxe concreta de CBabel em Maude CBabel tool é bastante simples, [3] utilizada quase que diretamente pelo transformador (a função `cd2omod` mostrada na Seção 3.2.4) na implementação de Maude CBabel tool. Versões futuras de Maude CBabel tool permitirão declarações de forma mais flexível como, por exemplo, loops e vetores para declaração de instâncias no módulo da aplicação.

- Investigar como aplicar a técnica de provas modulares (verificação composicional) para análise de arquiteturas mais complexas, como o exemplo "cruise control" [89].

- O desenvolvimento e aplicação de outras técnicas de análise que podem utilizar, por exemplo, abstrações equacionais [54] ou definição de estratégias [90] no contexto de arquiteturas de "software". As técnicas de abstrações equacionais e definição de estratégias para aplicação das regras de reescrita podem ser úteis para realização de provas modulares, no sentido apresentado por Clarke em [87, Capítulo 12], evitando-se assim o problema da explosão de estados. A explosão de estados geralmente ocorre quando consideramos o comportamento de todos os componentes da arquitetura em uma mesma análise [42]. Para a aplicação da técnica de abstrações equacionais, pode ser investigada a utilização do gerador de abstrações proposto por Palomino [91]. Para a aplicação da técnica de definição de estratégias pode ser utilizado o interpretador de estratégias de Meseguer, Martí-Oliet e Verdejo [90]. A definição de estratégias facilitaria ainda a análise de propriedades como "livelock", como vimos

[3] Muito próxima da representação matemática que utilizamos na formalização das construções da linguagem no Capítulo 3.

na Seção 5.1.3, que só podem ser analisadas tendo em vista alguma condição de justiça na execução dos componentes.

Três comentários finais merecem destaque. Nesta dissertação não foram considerados os contratos de qualidade de serviços (QoS) de CBabel, bem como não foram exploradas questões relacionadas a *arquiteturas dinâmicas*, isto é, arquiteturas que podem ser reconfiguradas dinamicamente (i.e., remoção e inclusão de componentes, troca de ligações entre portas). A formalização destes conceitos e o desenvolvimento de técnicas de análise para estes casos devem ser exploradas. Para formalização dos contratos de QoS, aspectos de tempo-real podem ser necessários, podendo ser analisados e formalizados pela ferramenta Real-Time Maude [30]. Como Maude CBabel tool e Real-Time Maude são ambas extensões conservativas de Full Maude, a incorporação de aspectos de tempo-real em Maude CBabel tool pode ser bastante facilitada.

A abordagem adotada para semântica de CBabel em lógica de reescrita não é dependente da implementação de lógica de reescrita escolhida, em nosso caso Maude, podendo constituir um dos trabalhos futuros a implementação da semântica de CBabel apresentada nesta dissertação em outra implementação de lógica de reescrita. Em OBJ [47], por exemplo, os aspectos dinâmicos podem ser representados por equações comportamentais e a notação de objetos e mensagens pode ser facilmente simulada por não utilizarmos herança de classes. (Para implementar Herança de classes, Full Maude realiza transformação de teorias.) Nossa preocupação na definição da semântica sobre uma sintaxe reduzida e simples para CBabel também facilita esta portabilidade. Deve-se observar, no entanto, que a sintaxe definida não foi provada minimal sob nenhuma condição de minimalidade. Da mesma forma, embora nossa abordagem possa ser utilizada para semântica de outras ADLs, não foi exercitada com outras sintaxes concretas de outras ADLs que não CBabel.

Para validação da uma arquitetura de "software", devemos considerar que o passo criativo para identificar análises necessárias, relacionando as propriedades acerca dos dados e estrutura (estáticas) com as propriedades comportamentais (sobre coordenação e interações dos componentes), depende do projetista, não podendo ser automatizado [42]. Esta dificuldade sugere que as propriedades a serem analisadas sejam agrupadas em classes, para as quais determinados métodos de análise seriam relacionados como mais adequados. Nesta linha, Félix sugere a necessidade de serem desenvolvidas técnicas de análise para domínios específicos.

Referências

[1] SHAW, M.; GARLAN, D. *Software architecture: perspectives on an emerging discipline*. USA: Prentice-Hall Inc., 1996.

[2] LOQUES, O. et al. On the integration of meta-level programming and configuration programming. In: *Reflection and Software Engineering (special edition)*. Heidelberg, Germany: Springer-Verlag, 2000. (Lecture Notes in Computer Science, v. 1826), p. 191–210.

[3] ALLEN, R. J. *A Formal Approach to Software Architecture*. Tese (Doutorado) — School of Computer Science, Carnegie Mellon University, EUA, May 1997.

[4] LÖHR, K.-P. Concurrency annotations for reusable software. *Communications of the ACM*, ACM Press, v. 36, n. 9, p. 81–89, 1993. ISSN 0001-0782.

[5] LEA, D. *Concurrent Programming in Java: Design Principles and Patterns*. Second edition. USA: Addison-Wesley, 2003. (The Java Series).

[6] BEN-ARI, M. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[7] BACH, M. J. *The Design of the UNIX Operation System*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[8] FRöLUND, S. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In: *Proceedings of the ECOOP'92 - 6th European Conference on Object-oriented Programming*. [S.l.]: Springer-Verlag, 1992. (Lecture Notes in Computer Science, v. 615), p. 185–196.

[9] MATSUOKA, S.; YONEZAWA, A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In: *Research directions in concurrent object-oriented programming*. [S.l.]: MIT Press, 1993. p. 107–150. ISBN 0-262-01139-5.

[10] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.

[11] SCHMIDT, D. C. et al. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. [S.l.]: John Wiley & Sons, Inc., 2000. ISBN 0471606952.

[12] LOQUES, O.; SZTAJNBERG, A. Customizing component-based architectures by contract. In: EMMERICH, W.; WOLF, A. L. (Ed.). *Component Deployment, Second International Working Conference*. Edinburgh, UK: [s.n.], 2004. (Lecture Notes in Computer Science, v. 3083), p. 18–34. ISBN 3-540-22059-3.

[13] MEYER, B. Applying design by contract. *IEEE Computer*, v. 25, n. 10, p. 40–51, October 1992.

[14] HELM, R.; HOLLAND, I. M.; GANGOPADHYAY, D. Contracts: Specifying behavioral compositions in object-oriented systems. In: *Proceedings of OOPSLA/ECOOP '90 Conference on Object-Oriented Programming Systems*. Ottawa: [s.n.], 1990. (Languages and Application), p. 169–180.

[15] BEUGNARD, A. et al. Making components contract aware. *IEEE Computer*, p. 38–45, July 1999.

[16] LICHTNER, K.; ALENCAR, P.; COWAN, D. An extensible model of architecture description. In: *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2000. p. 156–165. ISBN 1-58113-240-9.

[17] MEDVIDOVIC, N.; TAYLOR, R. N. A framework for classifying and comparing architecture description languages. In: *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. Zurich, Switzerland: Springer-Verlag New York, Inc., 1997. p. 60–76. ISBN 3-540-63531-9. `http://doi.acm.org/10.1145/267895.267903`.

[18] MESEGUER, J. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, Elsevier, v. 96, n. 1, p. 73–155, April 1992.

[19] CLAVEL, M. et al. The maude 2.0 system. In: NIEUWENHUIS, R. (Ed.). *Rewriting Techniques and Applications (RTA 2003)*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, 2706), p. 76–87. `http://maude.cs.uiuc.edu`. Acesso em 9/05/2005.

[20] BRAGA, C.; SZTAJNBERG, A. Towards a rewriting semantics to a software architecture description language. In: CAVALCANTI, A.; MACHADO, P. (Ed.). *Proceedings of WMF 2003, 6th Workshop on Formal Methods, Campina Grande, Brazil*. [S.l.]: Elsevier, 2003. (Electronic Notes in Theoretical Computer Science, v. 95), p. 148–168.

[21] RADEMAKER, A.; BRAGA, C.; SZTAJNBERG, A. A rewriting semantics for a software architecture description language. In: MOTA, A.; MOURA, A. (Ed.). *Proceedings of SBMF 2004, 7o. Simpósio Brasileiro de Métodos Formais, Recife, Pernambuco, Brazil, 2004*. [S.l.]: Elsevier, 2005. (Electronic Notes in Theoretical Computer Science, v. 130), p. 345–377.

[22] MARTÍ-OLIET, N.; MESEGUER, J. Handbook of philosophical logic. In: _____. Dordrecht: Kluwer Academic Publishers, 2002. v. 9, cap. Rewriting Logic as a Logical and Semantic Framework. `http://maude.cs.uiuc.edu/papers`. Acesso em 30/06/2005.

[23] EKER, S.; MESEGUER, J.; SRIDHARANARAYANAN, A. The Maude LTL model checker. In: GADDUCCI, F.; MONTANARI, U. (Ed.). *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*. [S.l.]: Elsevier, 2002. (Electronic Notes in Theoretical Computer Science, v. 71), p. 230–234.

[24] CLAVEL, M. *Reflection in rewriting logic: metalogical foundations and metaprogramming applications*. [S.l.]: CSLI Publications, 2000.

[25] DURÁN, F.; MESEGUER, J. An extensible module algebra for maude. In: *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*. [S.l.]: Elsevier, 1998. (Electronic Notes in Theoretical Computer Science, v. 15).

[26] TANENBAUM, A. S. *Operating Systems: Design and Implementation*. Englewood Cliffs, NJ 07632, USA: Prentice-Hall, 1987. ISBN 0-13-637331-3.

[27] BRAGA, C. de O. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`. Acesso em 9/05/2005.

[28] DENKER, G.; MESEGUER, J.; TALCOTT, C. Formal specification and analysis of active networks and communication protocols: The Maude experience. In: *DISCEX 2000, Proc. Darpa Information Survivability Conference and Exposition, Hilton Head, South Carolina*. [S.l.]: IEEE Computer Society Press, 2000. v. 1, p. 251–265.

[29] DURÁN, F. *Reflective Module Algebra with Applications to the Maude Language*. Tese (Doutorado) — University of Málaga, 1999.

[30] ÖLVECZKY, P. C. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. Tese (Doutorado) — University of Bergen, 2000.

[31] GOGUEN, J.; MESEGUER, J. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, v. 105, p. 217–273, 1992.

[32] MESEGUER, J. Membership algebra as a logical framework for equational specification. In: *In 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*. [S.l.]: Springer-Verlag, 1998. (Lecture Notes in Computer Science, v. 1376), p. 18–61.

[33] BRUNI, R.; MESEGUER, J. Generalized rewrite theories. In: BAETEN, J. C. M. et al. (Ed.). *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2719), p. 252–266.

[34] CLAVEL, M. *Reflection in General Logics and in Rewriting Logic*. Tese (Doutorado) — Univesity of Navarre, 1998.

[35] CLAVEL, M. et al. *Maude: Specification and Programming in Rewriting Logic*. `http://maude.csl.sri.com`, January 1999.

[36] PALOMINO, M.; MARTÍ-OLIET, N.; VERDEJO, A. Playing with Maude. In: ABDENNADHER, S.; RINGEISSEN, C. (Ed.). *Fifth International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany*. [S.l.]: Elsevier, 2005. (Electronic Notes in Theoretical Computer Science, v. 124), p. 3–23.

[37] BERGSTRA, J.; TUCKER, J. Characterization of computable data types by means of a finite equational specification method. In: BAKKER, J. W. de; LEEUWEN, J. van (Ed.). *Seventh Colloquium on Automata, Languages, and Programming*. [S.l.]: Springer-Verlag, 1980. (Lecture Notes in Computer Science, v. 81), p. 76–90.

[38] MESEGUER, J. A logical theory of concurrent objects. In: *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 1990. p. 101–115. ISBN 0-201-52430-X.

[39] MESEGUER, J.; TALCOTT, C. Semantic models for distributed object reflection. In: *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*. [S.l.]: Springer, 2002. (Lecture Notes in Computer Science, v. 2374), p. 1–36. ISBN 3-540-43759-2.

[40] EKER, S. et al. Pathway logic: Executable models of biological networks. In: *Fourth International Workshop on Rewriting Logic and Its Applications (WRLA'2002)*. [S.l.]: Elsevier, 2002. (Electronic Notes in Theoretical Computer Science, v. 71). `http://www.csl.sri.com/papers/eker-etal-02wrla/`. Acesso em 9/05/2005.

[41] INVERARDI, P.; WOLF, A. L. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, v. 21, n. 4, 1995.

[42] FELIX, M. F. *Análise Formal de Modelos de Software Orientados por Abstrações Arquiteturais*. Tese (Doutorado) — Departamento de Informática, PUC-Rio, Rio de Janeiro, Brazil, 2004. In portuguese.

[43] DIACONESCU, R.; FUTATSUGI, K. Logical foundations of CafeOBJ. *Theoretical Computer Science*, Elsevier Science Publishers Ltd., Essex, UK, v. 285, n. 2, p. 289–318, 2002. ISSN 0304-3975.

[44] BOROVANSKÝ, P. et al. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, v. 285, n. 2, p. 155–185, 2002.

[45] MARTÍ-OLIET, N.; MESEGUER, J. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, v. 285, n. 2, p. 121–154, 2002.

[46] SZTAJNBERG, A. *Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos*. Tese (Doutorado) — COPPE/UFRJ, Maio 2002.

[47] Software engineering with OBJ: Algebraic specification in action. In: GOGUEN, J. A.; MALCOLM, G. (Ed.). Boston: Kluwer Academic Publishers, 2000. cap. A Comprehensive Introduction to OBJ. ISSN: 0-7923-7757-5.

[48] GOGUEN, J. A.; MALCOLM, G. *Algebraic Semantics of Imperative Programs*. [S.l.]: The MIT Press, 1996.

[49] DENKER, G.; MESEGUER, J.; TALCOTT, C. Protocol specification and analysis in Maude. In: *Proc. of Workshop on Formal Methods and Security Protocols*. [S.l.: s.n.], 1998. `www.cs.bell-labs.com/who/nch/fmsp/index.html`. Acesso em 9/5/2005.

[50] MESEGUER, J. Program verification. Lecture Notes. `http://www-courses.cs.uiuc.edu/~cs476/`. Acesso em: 9/05/2005.

[51] GOODLOE, A. et al. Formal specification of sectrace. In: *Workshop on Context Sensitive Systems Assurance (Contessa'03)*. Philadelphia: [s.n.], 2003. `http://formal.cs.uiuc.edu/stehr/sectrace.html`. Acesso em 9/05/2005.

[52] GOODLOE, A.; GUNTER, C. A.; STEHR, M.-O. Formal specification of the layer 3 accounting (l3a) protocol. `http://formal.cs.uiuc.edu/stehr/l3a/l3a.html`. Acesso em 9/05/2005. August 2004.

[53] ÖLVECZKY, P. C. Formal modeling and analysis of distributed systems in maude. Lecture Notes `http://heim.ifi.uio.no/~peterol/`. January 2005.

[54] MESEGUER, J.; PALOMINO, M.; MARTÍ-OLIET, N. Equational abstractions. In: BAADER, F. (Ed.). *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction*. Miami Beach, Florida, USA: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2741), p. 2–16.

[55] COURTOIS, P. J.; HEYMANS, F.; PARNAS, D. L. Concurrent control with "readers" and "writers". *Commun. ACM*, New York, NY, USA, v. 14, n. 10, p. 667–668, 1971.

[56] BUHR, P. A.; FORTIER, M.; COFFIN, M. H. Monitor classification. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 27, n. 1, p. 63–107, 1995. ISSN 0360-0300.

[57] MESEGUER, J. Software specification and verification in rewriting logic. Lecture notes. `http://maude.cs.uiuc.edu/papers/`. Acesso em 30/06/2005. 2003.

[58] CLAVEL, M.; MESEGUER, J. Reflection and strategies in rewriting logic. In: MESEGUER, J. (Ed.). *Electronic Notes in Theoretical Computer Science*. [S.l.]: Elsevier Science Publishers, 2000. v. 4.

[59] CUESTA, C. E.; FUENTE, P. de la; BARRIO-SOLORZANO, M. Dynamics and reflection in software architecture. In: *Proceedings of 4th Workshop on M'etodos, Entornosy Nuevas Herramientas en Ingenier'ya de Requisitos*. [S.l.: s.n.], 1999.

[60] HOARE, C. A. R. *Communicating sequential processes*. [S.l.]: Prentice-Hall, Inc., 1985.

[61] FDR Model Checker. `http://www.fsel.com/software.html`. Acesso em 9/05/2005.

[62] LUCKHAM, D. C.; AL et. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, v. 21, n. 4, p. 336–355, April 1995.

[63] GARLAN, D.; MONROE, R.; WILE, D. Foundations of component-based systems. In: _____. [S.l.]: Cambridge Univ. Press, 2000. cap. Acme: Architectural Descriptions of Component-Based Systems, p. 47–68.

[64] MONROE, R. T. *Capturing Software Architecture Design Expertise With Armani*. [S.l.], 1998. `http://www-2.cs.cmu.edu/~able/paper_abstracts/armani-lrm.html`. Acesso em 24/06/2005.

[65] ACME Studio 2.2 User Manual. May 2005. `http://www-2.cs.cmu.edu/~acme/acme_documentation.html`. Acesso em: 4/05/2005.

[66] MéTAYER, D. L. Software architecture styles as graph grammars. In: *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 1996. p. 15–23. ISBN 0-89791-797-9.

[67] SPIVEY, M. *The Z Notation: A Reference Manual*. [S.l.]: Prentice-Hall, 1992.

[68] ABOWD, G. D.; ALLEN, R.; GARLAN, D. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, ACM Press, New York, NY, USA, v. 4, n. 4, p. 319–364, 1995. ISSN 1049-331X.

[69] MEDVIDOVIC, N.; TAYLOR, R. N.; WHITEHEAD, J. E. J. Formal modeling of software architectures at multiple levels of abstraction. In: *Proceedings of the California Software Symposium 1996*. Los Angeles: [s.n.], 1996. p. 28–40.

[70] NAKAJIMA, S.; TAMAI, T. Behavioural analysis of the enterprise JavaBeans component architecture. In: *SPIN'01: Proceedings of the 8th international SPIN workshop on Model checking of software*. New York, NY, USA: Springer-Verlag New York, Inc., 2001. (Lecture Notes in Computer Science, v. 2057), p. 163–182.

[71] HOLZMANN, G. J. The model checker Spin. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 23, n. 5, p. 279–295, 1997. ISSN 0098-5589.

[72] FIADEIRO, J. L.; LOPES, A. Semantics of architectural connectors. In: *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*. London, UK: Springer-Verlag, 1997. p. 505–519. ISBN 3-540-62781-2.

[73] FIADEIRO, J. L.; MAIBAUM, T. Categorical semantics of parallel program design. *Science of Computer Programming*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 28, n. 2-3, p. 111–138, 1997. ISSN 0167-6423.

[74] CLAVEL, M. et al. Maude as a formal meta-tool. In: WING, J. M.; WOODCOCK, J.; DAVIES, J. (Ed.). *Proceedings of World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999, Vol. II*. [S.l.]: Springer-Verlag, 1999. (Lecture Notes in Computer Science, v. 1709), p. 1684–1703.

[75] DENKER, G.; TALCOTT, C. *Maude Specification of the MDS Architecture and Examples*. [S.l.], October 2003. `http://www.csl.sri.com/users/denker/pub_03.html`. Acesso em 9/05/2005.

[76] DURAN, F.; VALLECILLO, A. Formalizing ODP enterprise specifications in maude. In: *Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*. Monterey, California: [s.n.], 2004. (Computer Standard and Interfaces), p. 20–24. `http://csdl.computer.org/comp/proceedings/edoc/2004/2214/00/2214toc.htm`, Acesso em 9/05/2005.

[77] MILNER, R. *Communication and Concurrency*. [S.l.]: Prentice-Hall International, 1989. (Prentice-Hall International Series in Computer Science).

[78] LICHTNER, K.; ALENCAR, P.; COWAN, D. A framework for software architecture verification. In: *ASWEC '00: Proceedings of the 2000 Australian Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2000. p. 149. ISBN 0-7695-0631-3.

[79] OWRE, S.; RUSHBY, J. M.; SHANKAR, N. PVS: A prototype verification system. In: KAPUR, D. (Ed.). *11th International Conference on Automated Deduction (CADE)*. Saratoga, NY: Springer-Verlag, 1992. (Lecture Notes in Artificial Intelligence, v. 607), p. 748–752.

[80] STAFFORD, J. A.; WOLF, A. L. Architecture-level dependence analysis in support of software maintenance. In: *ISAW '98: Proceedings of the third international workshop on Software architecture*. New York, NY, USA: ACM Press, 1998. p. 129–132. ISBN 1-58113-081-3.

[81] STEGGLES, L. J.; KOSIUCZENKO, P. A formal model for sdl specifications based on timed rewriting logic. *Automated Software Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 7, n. 1, p. 61–90, 2000. ISSN 0928-8910.

[82] FÆRGEMAND, O.; OLSEN, A. Introduction to SDL-92. *Computer Network ISDN System*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 26, n. 9, p. 1143–1167, 1994. ISSN 0169-7552.

[83] KOSIUCZENKO, P.; WIRSING, M. Timed rewriting logic with an application to object-based specification. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 28, n. 2-3, p. 225–246, 1997. ISSN 0167-6423.

[84] DOBRICA, L.; NIEMELA, E. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, v. 28, n. 7, p. 638–653, July 2002.

[85] LEWIS, B.; FEILER, P. An overview of the SAE architecture analysis design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. In: *Workshop on Architecture Description Languages*. Toulouse, France: [s.n.], 2004.

[86] LOQUES, O. et al. A contract-based approach to describe and deploy non-functional adaptations in software architectures. *Journal of the Brazilian Computer Society*, SBC, Rio de Janeiro, RJ, Brasil, v. 10, p. 5–18, July 2004.

[87] CLARKE, E. M.; GRUMBERG, O.; PELED, D. *Model Checking*. [S.l.]: MIT Press, 2000.

[88] IVERS, J. Wright tutorial. `http://www-2.cs.cmu.edu/afs/cs/project/able/www/wright/index.html`. Acesso em 30/06/2005. September 1998.

[89] MAGEE, J.; KRAMER, J. *Concurrency: state models & Java programs*. UK: John Wiley & Sons, Inc., 1999. ISBN 0-471-98710-7.

[90] MARTí-OLIET, N.; MESEGUER, J.; VERDEJO, A. Towards a strategy language for maude. In: *In Proceedings of the WRLA'04*. Barcelona, Spain: Elsevier, 2004. (Electronic Notes in Theoretical Computer Science). `http://maude.sip.ucm.es/strategies/`. Acesso em 9/05/2005.

[91] PALOMINO, M. *Reflexión, abstracción y simulación en la lógica de reescritura*. Tese (Doutorado) — Universidad Complutense de Madrid, Spain, Mar 2005.

APÊNDICE A – Especificação da ferramenta Maude CBabel tool

```
--- edited full-maude.maude
in my-fm.maude
in model-checker.maude

fmod CBABEL-SIGNATURE is
 pr INT .
 pr QID-LIST .

 *** IDS
 sorts eId .

 op cbtoken : Qid -> eId [special(id-hook Bubble  (1 1)
                           op-hook qidSymbol (<Qids> : ~> Qid)
                           id-hook Exclude ( . ; TRUE FALSE ))] .

 *** STMTS
 sorts eBoolean eExp .
 subsorts eBoolean eId < eExp .

 ops TRUE FALSE : -> eBoolean .
 ops _!=_ _==_ _<_ _>_ _+_ _-_ : eExp eExp -> eExp [prec 32] .
 ops _&&_ _||_ : eExp eExp -> eExp [prec 33] .
 op ((_)) : eExp -> eExp .

 sorts eStmt eStmtSeq .
 subsort eStmt < eStmtSeq .

 op skip : -> eStmt .
 op _=_; : eId eExp -> eStmt [prec 13] .
 op if (_) {_} : eExp eStmtSeq  -> eStmt .
 op if (_) {_} else {_} : eExp eStmtSeq eStmtSeq  -> eStmt .
 op __ : eStmtSeq eStmtSeq -> eStmtSeq [prec 45 assoc] .

 *** PORTS
 sort ePortDecl .

 op in port_; : eId -> ePortDecl .
 op out port_; : eId -> ePortDecl .
 op in port oneway_; : eId -> ePortDecl .
```

```
op out port oneway_; : eId -> ePortDecl .

*** VARS
sorts eVarDecl eVarType .

ops int bool : -> eVarType .
op var_ _;   : eVarType eId -> eVarDecl .
op var_ _=_; : eVarType eId eExp -> eVarDecl .
op staterequired_ _; : eVarType eId -> eVarDecl .


*** MODULES
sorts eModuleDecl eInstantiateDecl eLinkDecl eBindDecl eElement eElementSet .
subsorts eVarDecl ePortDecl < eElement .
subsorts eLinkDecl eInstantiateDecl eBindDecl < eElement .
subsort eElement < eElementSet .

op mt-element : -> eElementSet .
op __ : eElementSet eElementSet -> eElementSet
                                   [assoc comm id: mt-element prec 50] .

op module_{_} : eId eElementSet -> eModuleDecl .
op application_{_} : eId eElementSet -> eModuleDecl .

op instantiate_as_; : eId eId -> eInstantiateDecl [prec 49] .
op link_to_; : eId eId -> eLinkDecl [prec 49] .
op bind_ _to_; : eVarType eId eId -> eBindDecl [prec 49] .


*** CONTRACTS
sorts GuardBody GuardDecl PortExp
      InteractionDecl InteractionDeclSet ContractDecl .
subsort InteractionDecl < InteractionDeclSet .
subsort ContractDecl < eElement .
subsort eId < PortExp .

op before{_} : eStmtSeq -> GuardBody .
op after{_} : eStmtSeq -> GuardBody .
op alternative(_); : eId -> GuardBody .
op __ : GuardBody GuardBody -> GuardBody [assoc comm] .

op guard(_) : eExp -> GuardDecl .
op guard(_){_} : eExp GuardBody -> GuardDecl .
op _>_>_; : eId GuardDecl eId -> InteractionDecl [prec 15] .

op _|_ : PortExp PortExp -> PortExp [assoc comm prec 13] .
op _>_; : PortExp PortExp -> InteractionDecl [prec 15] .
op __ : InteractionDeclSet InteractionDeclSet -> InteractionDeclSet [assoc comm] .

op interaction{_} : InteractionDecl -> ContractDecl .
op exclusive{_} : InteractionDeclSet -> ContractDecl .

*** CONNECTORS
sort eConnectorDecl .
```

```
 op connector_{_} : eId eElementSet -> eConnectorDecl .

 *** CBABEL
 sort eComponentDecl .
 subsorts eConnectorDecl eModuleDecl < eComponentDecl .
endfm

fmod CBABEL-ID is
 pr QID .

 sort Id IdSet .
 subsort Id < IdSet .

 op ID : Qid -> Id [ctor] .
 op mt-id : -> IdSet [ctor] .
 op _;_ : IdSet IdSet -> IdSet [ctor assoc comm id: mt-id] .
endfm

fmod CBABEL-UNIT is
 pr INT .
 inc CBABEL-ID .
 inc UNIT .

 sorts Assign Conditional Stmt StmtSeq Value Integer Boolean Exp .

 subsorts Integer Boolean < Value .
 subsorts Value Id < Exp .
 subsorts Assign Conditional < Stmt < StmtSeq .

 op VL : Int -> Integer .
 ops TRUE FALSE : -> Boolean .

 ops not-equal equal less-than greater-than plus minus : Exp Exp -> Exp .
 ops land lor : Exp Exp -> Exp .
 op ((_)) : Exp -> Exp .

 op skip : -> Stmt .
 op stmt-seq : StmtSeq StmtSeq -> StmtSeq .
 op assign : Id Exp -> Assign .
 op if-then-else : Exp StmtSeq StmtSeq  -> Conditional .

 sorts  PortDecl PortDeclSet PortType
        VariableDecl VariableDeclSet VarType
        Interaction InteractionSet Contract
        InstanceDecl LinkDecl BindDecl ConfigDecl ConfigDeclSet
        ModuleDecl ConnectorDecl ComponentDecl .

 subsort PortDecl < PortDeclSet .
 subsort VariableDecl < VariableDeclSet .
 subsort Interaction < InteractionSet .
 subsorts InstanceDecl LinkDecl BindDecl < ConfigDecl < ConfigDeclSet .
 subsorts ModuleDecl ConnectorDecl < ComponentDecl < Unit .

 ops sinc assinc : -> PortType .
```

```
op out : Id PortType -> PortDecl [ctor] .
op in : Id PortType -> PortDecl [ctor] .
op mt-port : -> PortDeclSet [ctor] .
op __ : PortDeclSet PortDeclSet -> PortDeclSet [assoc comm id: mt-port] .

ops Boolean Integer : -> VarType .
ops local required : Id VarType -> VariableDecl .
op local : Id VarType Exp -> VariableDecl .
op mt-var : -> VariableDeclSet .
op __ : VariableDeclSet VariableDeclSet -> VariableDeclSet
                                          [assoc comm id: mt-var] .

op seq : IdSet IdSet -> Interaction .
op guard : Id Id Exp StmtSeq StmtSeq -> Interaction .
op guard : Id Id Id Exp StmtSeq StmtSeq -> Interaction .
op mt-interaction : -> InteractionSet [ctor] .
op __ : InteractionSet InteractionSet -> InteractionSet
                                        [assoc comm id: mt-interaction] .

op interaction : Interaction -> Contract .
op exclusive : InteractionSet -> Contract .
op mt-contract : -> Contract .

op instantiate : Id Id -> InstanceDecl .
op link : Id Id Id Id -> LinkDecl .
op bind : Id Id Id Id VarType -> BindDecl .
op mt-config : -> ConfigDeclSet .
op __ : ConfigDeclSet ConfigDeclSet -> ConfigDeclSet [assoc comm id: mt-config] .

op module : Id VariableDeclSet PortDeclSet -> ModuleDecl .
op connector : Id VariableDeclSet PortDeclSet Contract -> ConnectorDecl .
op application : Id ConfigDeclSet ConfigDeclSet ConfigDeclSet -> ModuleDecl .

var VDS : VariableDeclSet .
var VD : VariableDecl .
var PDS : PortDeclSet .
var PD : PortDecl .
var CD CD' : Contract .
vars CDS CDS' CDS'' : ConfigDeclSet .
vars I I' : Id .
var U : Unit .
var ID : InstanceDecl .
var LD : LinkDecl .
var BD : BindDecl .
var MN : ModName .

op emptyConnector : -> ConnectorDecl .
op emptyModule : -> ModuleDecl .
op emptyApplication : -> ModuleDecl .
op noName : -> Id .
eq noName = ID('noName) .

eq emptyConnector = connector(noName, mt-var, mt-port, mt-contract) .
eq emptyApplication = application(noName, mt-config, mt-config, mt-config) .
```

```
 eq emptyModule = module(noName, mt-var, mt-port) .

 eq setName( module(I, VDS, PDS) , MN)
    = module(ID(MN), VDS, PDS) .
 eq setName( connector(I, VDS, PDS, CD), MN)
    = connector(ID(MN), VDS, PDS, CD) .
 eq setName( application(I, CDS, CDS', CDS''), MN)
    = application(ID(MN), CDS, CDS', CDS'') .

 eq getName( module(ID(MN), VDS, PDS) ) = MN .
 eq getName( connector(ID(MN), VDS, PDS, CD) ) = MN .
 eq getName( application(ID(MN), CDS, CDS', CDS'') ) = MN .

 op addPort : PortDecl Unit -> Unit .
 eq addPort(PD, module(I, VDS, PDS)) = module(I, VDS, PDS PD) .
 eq addPort(PD, connector(I, VDS, PDS, CD))
    = connector(I, VDS, PDS PD, CD) .

 op addVar : VariableDecl Unit -> Unit .
 eq addVar(VD, module(I, VDS, PDS)) = module(I, VDS VD, PDS) .
 eq addVar(VD, connector(I, VDS, PDS, CD))
    = connector(I, VDS VD, PDS, CD) .

 op addContract : Contract Unit -> Unit .
 eq addContract(CD', connector(I, VDS, PDS, CD))
    = connector(I, VDS, PDS, CD') .

 op addInstance : InstanceDecl Unit -> Unit .
 eq addInstance(ID, application(I, CDS, CDS', CDS''))
    = application(I, ID CDS, CDS', CDS'') .

 op addLink : LinkDecl Unit -> Unit .
 eq addLink(LD, application(I, CDS, CDS', CDS''))
    = application(I, CDS, LD CDS', CDS'') .

 op addBind : BindDecl Unit -> Unit .
 eq addBind(BD, application(I, CDS, CDS', CDS''))
    = application(I, CDS, CDS', BD CDS'') .
endfm

fmod CBABEL-SIGN is
 inc FULL-MAUDE-SIGN .
 inc CBABEL-SIGNATURE .

 subsort eComponentDecl < Input .
endfm

fmod EXT-META-FULL-MAUDE-SIGN is
 pr META-FULL-MAUDE-SIGN .
 pr UNIT .

 op CB-GRAMMAR : -> FModule .
 eq CB-GRAMMAR =
    addImports((including 'CBABEL-SIGN .), GRAMMAR) .
```

```
endfm

mod INITIAL-DB is
  pr META-FULL-MAUDE-SIGN .
  pr DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

 op importFromMetaModules : List<Unit> Database -> Database .
 op importFromCoreMaude : QidList Database -> Database .

 var  Q : Qid .
 var  QL : QidList .
 var  U : Unit .
 var  UL : List<Unit> .
 var DB : Database .

 ceq importFromCoreMaude (Q QL, DB)
 = importFromCoreMaude (QL,
    importFromCoreMaude (Q, DB))
 if QL =/= nil .

 eq importFromCoreMaude (Q, DB)
 = importFromMetaModules (upModule (Q, false), DB) .

 ceq importFromMetaModules (U UL, DB)
 = importFromMetaModules (UL,
    importFromMetaModules (U, DB))
 if UL =/= nil .

 eq importFromMetaModules (U, DB)
 = evalUnit (U, none, DB) .

 op initial-db : -> Database .
 eq initial-db =
    importFromMetaModules (TRIV UP CONFIGURATION+,
     importFromCoreMaude
       ('TRUTH-VALUE 'TRUTH 'BOOL 'EXT-BOOL
        'IDENTICAL
        'NAT 'INT 'RAT 'FLOAT 'STRING 'CONVERSION
        'QID 'QID-LIST 'META-TERM 'META-MODULE 'META-LEVEL
        'LTL 'LTL-SIMPLIFIER 'SATISFACTION 'MODEL-CHECKER
        'CONFIGURATION, emptyDatabase)) .
endm

fmod CBABEL-CONVERTER-AUX is
 pr CBABEL-UNIT .
 pr CONVERSION .

 vars I I' : Id .
 var Q : Qid .
 var S : String .
 var TL : TermList .
 var N : Nat .
```

```
var MN : ModName .

op makeVar : Qid Nat -> Variable .
eq makeVar(Q, N) = qid("V#" + string(N, 10) + ":" + string(Q)) .

op strId : Id -> String .
eq strId(ID(Q)) = string(Q) .

op qidId : Id -> Qid .
eq qidId(ID(Q)) = Q .

ops itv ov : -> Variable .
eq itv = 'IT:Interaction .
eq ov  = 'O:Oid .

op consId : Id String -> Constant .
eq consId(I, S) = qid(strId(I) + "." + S) .

op  varId : Id String -> Variable .
eq  varId(I, S) = qid(strId(I) + ":" + S) .

ops portInC portOutC classC varNameC : Id -> Constant .
eq  portInC(I) = consId(I, "PortInId") .
eq portOutC(I) = consId(I, "PortOutId") .
eq   classC(I) = consId(I, strId(I)) .

op classC : ModName -> Constant .
eq classC(MN) = qid(string(MN) + "." + string(MN)) .

op objt : ModName -> Term .
eq objt(MN) = '<_:_|`>[ov, classC(MN)] .

op objt : ModName TermList -> Term .
eq objt(MN, TL) =  '<_:_|_>[ov, classC(MN), TL] .

op objt : Id Id TermList -> Term .
eq objt(I, I', TL) = '<_:_|_>[consId(I, "Oid"), classC(I'), TL] .

op labelRule : ModName String Id -> Attr .
eq labelRule(MN, S, I) = label(qid(string(MN) + "-" + S + "-" + strId(I))) .

op labelRule : Id String Id -> Attr .
eq labelRule(I, S, I') = label(qid(strId(I) + "-" + S + "-" + strId(I'))) .

op labelRule : ModName String -> Attr .
eq labelRule(MN, S) = label(qid(string(MN) + "-" + S)) .


op value? : Value -> Constant .
eq value?( TRUE  ) = 'true.Bool .
eq value?( FALSE ) = 'false.Bool .
eq value?( VL(I:Int) ) = upTerm(I:Int) .

op defaultValue : VarType -> Constant .
```

```
 eq defaultValue(Integer) = value?(VL(0)) .
 eq defaultValue(Boolean) = value?(TRUE) .

 op type? : VarType -> Qid .
 eq type?( Boolean ) = 'Bool .
 eq type?( Integer ) = 'Int .
endfm

fmod CBABEL-STMT-CONVERTER is
 pr CBABEL-CONVERTER-AUX .

 op st2t : StmtSeq -> Term .
 op st2t : Term StmtSeq -> Term .
 op  e2t : Exp -> Term .
 op  e2t : Term Exp -> Term .

 var T : Term .
 var I : Id .
 var VL : Value .
 vars SS1 SS2 SS : StmtSeq .
 vars E E1 E2 : Exp .

 eq st2t(SS) = st2t('OBJ:Object, SS) .

 eq st2t(T, skip) = T .

 eq st2t(T, stmt-seq(SS1, SS2)) =
    st2t(st2t(T, SS1), SS2) .

 eq st2t(T, assign(I, E)) =
    qid("set-" + strId(I))[T, e2t(T, E)] .

 eq st2t(T, if-then-else(E, SS1, SS2)) =
    'if_then_else_fi[e2t(T, E), st2t(T, SS1), st2t(T, SS2)] .

 eq e2t(E) = e2t('OBJ:Object, E) .

 eq e2t(T, equal(E, E))   = 'true.Bool .
 eq e2t(T, equal(E1, E2)) =  '_==_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, not-equal(E1, E2)) =  '_=/=_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, less-than(E1, E2))    = '_<_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, greater-than(E1, E2)) = '_>_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, plus(E1, E2))    = '_+_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, minus(E1, E2))    = '_-_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, land(E1, E2)) = '_and_[e2t(T, E1), e2t(T, E2)] .
 eq e2t(T, lor(E1, E2))  =  '_or_[e2t(T, E1), e2t(T, E2)] .

 eq e2t(T, VL) = value?(VL) .
 eq e2t(T, I) = qid("get-" + strId(I))[T] .
endfm

fmod CBABEL-CONVERTER is
 pr CBABEL-STMT-CONVERTER .
```

```
 vars I I' I'' I1 I2 I3 : Id .
 vars IDS IDS' : IdSet .
 vars OId1 PId1 OId2 PId2 : Id .
 var S : String .
 vars Q Q' QA : Qid .
 var TL : TermList .
 vars T T' T1 T2 T3 T4 : Term .
 var VT : VarType .
 var VL : Value .
 vars U U' U'' : Unit .
 var VDS : VariableDeclSet .
 var VD : VariableDecl .
 var PT : PortType .
 var PDS : PortDeclSet .
 var PD : PortDecl .
 var RLS : Set<ERule> .
 var EQS : Set<EEquation> .
 var CTD : Contract .
 *** vars M M' : IMap .
 var E : Exp .
 vars SS1 SS2 : StmtSeq .
 var CLD : ClassDecl .
 var AD : AttrDecl .
 var ADS : AttrDeclSet .
 vars CDS CDS' CDS'' CDS1 CDS2 CDS3 : ConfigDeclSet .
 var MN : ModName .


------------------------------------------------------------
--- Variable's Semantics
------------------------------------------------------------
 op getName : VariableDecl -> Id .
 eq getName(local(I, VT)) = I .
 eq getName(local(I, VT, E)) = I .
 eq getName(required(I, VT)) = I .

 op getType : VariableDecl -> VarType .
 eq getType(local(I, VT)) = VT .
 eq getType(local(I, VT, E)) = VT .
 eq getType(required(I, VT)) = VT .

 op varOpGetName : VariableDecl -> Qid .
 eq varOpGetName(VD) = qid("get-" + strId(getName(VD))) .

 op varOpSetName : VariableDecl -> Qid .
 eq varOpSetName(VD) = qid("set-" + strId(getName(VD))) .

 op varOpType : VariableDecl -> Qid .
 eq varOpType(VD) = type?(getType(VD)) .

 op  vars2omod : Unit VariableDeclSet -> Unit .

 eq vars2omod(U, VD VDS) =
    vars2omod(
```

```
    prepGetSetEq(
     addOps(
      (op varOpGetName(VD) : 'Object -> varOpType(VD) [none] .)
      (op varOpSetName(VD) : 'Object varOpType(VD) -> 'Object [none] .)
      , U)
     , VD)
    , VDS) .

 eq vars2omod(U, mt-var) = U .

 op  prepGetSetEq : Unit VariableDecl -> Unit .

 ceq prepGetSetEq(U, required(I, VT)) =
    addEqs(
     (eq varOpGetName(VD)[objt(MN, QA['st[Q, 'S:Status]])] = Q [none] .)
     (eq varOpSetName(VD)[objt(MN, QA['st[Q, 'S:Status]]), Q']
         = objt(MN, QA['st[Q', 'changed.Status]]) [none] .)
     , U)
  if VD := required(I, VT) /\
    MN := getName(U) /\
    Q  := qid("V:" + string(type?(VT))) /\
    Q' := qid("V':" + string(type?(VT))) /\
    QA := qid(strId(I) + " :_") .

 ceq prepGetSetEq(U, VD) =
    addEqs(
     (eq varOpGetName(VD)[objt(MN, QA[Q])] = Q [none] .)
     (eq varOpSetName(VD)[objt(MN, QA[Q]), Q'] = objt(MN, QA[Q']) [none] .)
     , U)
  if MN := getName(U) /\
    Q  := qid("V:" + string(type?(getType(VD))))  /\
    Q' := qid("V':" + string(type?(getType(VD)))) /\
    QA := qid(strId(getName(VD)) + " :_") [owise] .


---------------------------------------------------------------------------
--- Port's Semantics

--- we declare two functions to convert port declarations. The
--- ports2omod1 convert ports declarations in module and ports2omod2
--- convert ports declarations in connectors. The difference is that
--- in modules each port declaration generate a set of rules, but in
--- connectors the port declaration do not generate rules.
---------------------------------------------------------------------------
 op ports2omod1 : Unit PortDeclSet -> Unit .
 op ports2omod2 : Unit PortDeclSet -> Unit .

 ceq ports2omod1(U, in(I, PT) PDS)
    = ports2omod1(addRls(RLS, U'), PDS)
  if U'  := addOps((op qidId(I) : nil -> 'PortInId [ctor] .), U) /\
    RLS := prepPortRls(getName(U), in(I, PT)) .

 ceq ports2omod1(U, out(I, PT) PDS)
    = ports2omod1(addRls(RLS, U'), PDS)
```

```
 if U'  := addOps((op qidId(I) : nil -> 'PortOutId [ctor] .), U) /\
    RLS := prepPortRls(getName(U), out(I, PT)) .

ceq ports2omod2(U, in(I, PT) PDS) = ports2omod2(U', PDS)
 if U' := addOps((op qidId(I) : nil -> 'PortInId [ctor] .), U) .

ceq ports2omod2(U, out(I, PT) PDS) = ports2omod2(U', PDS)
 if U' := addOps((op qidId(I) : nil -> 'PortOutId [ctor] .), U) .

eq ports2omod1(U, mt-port) = U .
eq ports2omod2(U, mt-port) = U .

op prepPortRls : ModName PortDecl -> Set<ERule> .

ceq prepPortRls(MN, in(I', sinc)) =
   (rl
     '__[T, 'send[ov, portInC(I'), itv]] => '__[T, 'do[ov, portInC(I'), itv]]
    [labelRule(MN, "recevingAndDo", I')] .)
   (rl
     '__[T, 'done[ov, portInC(I'), itv]] => '__[T, 'ack[itv]]
    [labelRule(MN, "doneAndAcking", I')] .)
 if T := objt(MN) .

ceq prepPortRls(MN, in(I', assinc)) =
   (rl
     '__[T, 'send[ov, portInC(I'), itv]] => '__[T, 'do[ov, portInC(I'), itv]]
    [labelRule(MN, "receivingAndDo", I')] .)
   (rl
     '__[T, 'done[ov, portInC(I'), itv]] => T  [labelRule(MN,  "done", I')] .)
 if T := objt(MN) .

ceq prepPortRls(MN, out(I', sinc)) =
   (rl
     '__['do[ov, portOutC(I'), 'none.Interaction],
         objt(MN, Q['unlocked.PortStatus])] =>
     '__['send[ov, portOutC(I'), ''[_',_'][ov, portOutC(I')]],
         objt(MN, Q['locked.PortStatus])]
    [labelRule(MN, "sending", I')] .)
   (rl
     '__['ack[''[_',_'][ov, portOutC(I')]],
         objt(MN, Q['locked.PortStatus])] =>
     '__['done[ov, portOutC(I'), 'none.Interaction],
         objt(MN, Q['unlocked.PortStatus])]
    [labelRule(MN,  "receivingAck", I')] .)
  if Q := qid(strId(I') + "-status :_") .

ceq prepPortRls(MN, out(I', assinc)) =
   (rl '__[T, 'do[ov, portOutC(I'), 'none.Interaction]] =>
       '__[T, 'send[ov, portOutC(I'), ''[_',_'][ov, portOutC(I')]]]
    [labelRule(MN, "sending", I')] .)
   (rl
     '__[T, 'ack[''[_',_'][ov, portOutC(I')]]] => T
    [labelRule(MN,  "receivingAck", I')] .)
 if T := objt(MN) .
```

```
---------------------------------------------------------------------
--- The functions below convert Contract declaration
---------------------------------------------------------------------
 sort Tuple<Term|Term> .
 op <_;_> : Term Term -> Tuple<Term|Term> .

 op contract2omod : Unit Contract -> Unit .

 op prepMsgsLHS : Tuple<Term|Term> Nat IdSet -> Tuple<Term|Term> .
 op prepMsgsRHS : Term IdSet -> Term .
 op prepAcksLHS : Term IdSet -> Term .
 op prepAcksRHS : Term -> Term .

 vars C C' : Constant .
 var V : Variable .
 var N : Nat .
 var IT : Interaction .
 var ITS : InteractionSet .

 ceq prepMsgsLHS(< noTerm ; noTerm >, N, I ; IDS) =
     prepMsgsLHS(< 'send[ov, portInC(I), V] ; V >, s(N), IDS)
  if V := makeVar('Interaction, N) .

 ceq prepMsgsLHS(< T ; T' >, N, I ; IDS) =
  prepMsgsLHS(< '__[T, 'send[ov, portInC(I), V]] ; 'iset[V, T'] >, s(N), IDS)
  if V := makeVar('Interaction, N) .

 eq  prepMsgsLHS(< T ; T' >, N, mt-id) = < T ; T' > .


 ceq prepMsgsRHS(T, I ; IDS) =
     '__['send[ov, C, '_::_['`[_`,_`][ov, C], T]], prepMsgsRHS(T, IDS)]
  if C := portOutC(I) .

 eq prepMsgsRHS(T, mt-id) = 'none.Configuration .


 eq prepAcksLHS(T, I ; IDS) =
    '__['ack['_::_['`[_`,_`][ov, portOutC(I)], T]], prepAcksLHS(T, IDS)] .

 eq prepAcksLHS(T', mt-id) = 'none.Configuration .


 eq prepAcksRHS('iset[V, T]) = '__['ack[V], prepAcksRHS(T)] .
 eq prepAcksRHS(C) = 'ack[C] .
 eq prepAcksRHS(V) = 'ack[V] .

 ceq contract2omod(U, interaction(seq(IDS, IDS'))) =
     addRls(
      (rl '__[T1, T] => '__[T, T2] [labelRule(MN, "sending")] .)
      (rl '__[T3, T] => '__[T, T4] [labelRule(MN, "acking")] .)
      , U)
```

```
 if
   MN := getName(U) /\
    T := objt(MN) /\
   < T1 ; T' > := prepMsgsLHS(< noTerm ; noTerm >, 0, IDS) /\
   T2 := prepMsgsRHS(T', IDS') /\
   T3 := prepAcksLHS(T', IDS') /\
   T4 := prepAcksRHS(T') .


op prepExclusiveRls : Set<ERule> ModName InteractionSet -> Set<ERule> .

ceq prepExclusiveRls(RLS, MN, seq(I1, I2) ITS) =
    prepExclusiveRls(RLS
     (rl '__[T , T1] => '__[T', T2]  [labelRule(MN, "sending", I1)] .)
     (rl '__[T', T3] => '__[T , T4]  [labelRule(MN, "acking", I2)] .)
     , MN, ITS)
 if T  := objt(MN, 'status`:_['unlocked.PortStatus]) /\
    T' := objt(MN, 'status`:_['locked.PortStatus]) /\
    T1 := 'send[ov, portInC(I1), itv] /\
    T2 := 'send[ov, portOutC(I2), '_::_['``[_`,_`][ov, portOutC(I2)], itv]] /\
    T3 := 'ack['_::_['``[_`,_`][ov, portOutC(I2)], itv]] /\
    T4 := 'ack[itv] .

eq prepExclusiveRls(RLS, MN, mt-interaction) = RLS .

eq contract2omod(U, exclusive(ITS))
   = addRls(prepExclusiveRls(none, getName(U), ITS), U) .


op prepGuardEqs : ModName Exp StmtSeq StmtSeq -> Set<EEquation> .

eq prepGuardEqs(MN, E, SS1, SS2) =
   (ceq 'before['OBJ:Object] = st2t(SS1)
     if  'class['OBJ:Object] = classC(MN) [none] .)
   (ceq  'after['OBJ:Object] = st2t(SS2)
     if  'class['OBJ:Object] = classC(MN) [none] .)
   (ceq  'open?['OBJ:Object] = e2t(E)
     if  'class['OBJ:Object] = classC(MN) [none] .) .

*** retirei o lock do status apos aplicacao da regra mantendo sempre o
*** obj unlocked

ceq contract2omod(U, interaction(guard(I1, I2, E, SS1, SS2)))
    = addRls(RLS, addEqs(EQS, U))
 if MN := getName(U) /\
    T  := objt(MN) /\
  EQS := prepGuardEqs(MN, E, SS1, SS2) /\
  RLS :=
    (crl
      '__[T, 'send[ov, portInC(I1), itv]] =>
      '__['before[T],
          'send[ov, portOutC(I2), '_::_['``[_`,_`][ov, portOutC(I2)], itv]]]
      if 'open?[T] = 'true.Bool [labelRule(MN, "sending", I1)] .)
    (rl
```

```
      '__[T, 'ack['_::_['`[_`,_`][ov, portOutC(I2)], itv]]] =>
      '__['after[T], 'ack[itv]] [labelRule(MN, "acking", I2)] .) .


 ceq contract2omod(U, interaction(guard(I1, I2, I3, E, SS1, SS2)))
     = addRls(RLS, addEqs(EQS, U))
  if MN := getName(U) /\
     T  := objt(MN) /\
   EQS := prepGuardEqs(MN, E, SS1, SS2) /\
   RLS :=
     (crl
       '__[T, 'send[ov, portInC(I1), itv]] =>
       '__['before[T],
           'send[ov, portOutC(I2), '_::_['`[_`,_`][ov, portOutC(I2)], itv]]]
       if 'open?[T] = 'true.Bool [labelRule(MN, "sending", I1)] .)
     (crl
       '__[T, 'send[ov, portInC(I1), itv]] =>
       '__['before[T],
           'send[ov, portOutC(I3), '_::_['`[_`,_`][ov, portOutC(I3)], itv]]]
       if 'open?[T] = 'false.Bool [labelRule(MN, "sending", I3)] .)
     (rl
       '__[T, 'ack['_::_['`[_`,_`][ov, portOutC(I2)], itv]]] =>
       '__['after[T], 'ack[itv]] [labelRule(MN, "acking", I2)] .) .


--------------------------------------------------------------------------------
--- Class Declaration
--------------------------------------------------------------------------------
 op  makeClass : ClassDecl VariableDeclSet Contract -> ClassDecl .
 op  makeClass : ClassDecl VariableDeclSet PortDeclSet -> ClassDecl .
 op makeClass1 : ClassDecl VariableDeclSet -> ClassDecl .
 op makeClass2 : ClassDecl PortDeclSet -> ClassDecl .
 op makeClass3 : ClassDecl Contract -> ClassDecl .

 eq makeClass(CLD, VDS, CTD)
    = makeClass3(makeClass1(CLD, VDS), CTD) .
 eq makeClass(CLD, VDS, PDS)
    = makeClass2(makeClass1(CLD, VDS), PDS) .

 eq makeClass1(class MN | ADS ., required(I, VT) VDS)
    = makeClass1((class MN | (attr qidId(I) : 'StateRequired) , ADS .), VDS) .
 eq makeClass1(class MN | ADS ., local(I, VT) VDS)
    = makeClass1((class MN | (attr qidId(I) : type?(VT)) , ADS .), VDS) .
 eq makeClass1(class MN | ADS ., local(I, VT, E) VDS)
    = makeClass1((class MN | (attr qidId(I) : type?(VT)) , ADS .), VDS) .
 eq makeClass1(CLD, mt-var) = CLD .

 ceq makeClass2(class MN | ADS ., out(I, sinc) PDS)
     = makeClass2(class MN | AD , ADS ., PDS)
  if AD := (attr qid(strId(I) + "-status") : 'PortStatus) .
 eq makeClass2(CLD, mt-port) = CLD .
 eq makeClass2(CLD, PD PDS) = makeClass2(CLD, PDS) [owise] .

 eq makeClass3(class MN | ADS ., exclusive(ITS))
```

```
    = (class MN | (attr 'status : 'PortStatus) , ADS .) .
 eq makeClass3(class MN | ADS ., CTD) = (class MN | ADS .) [owise] .


------------------------------------------------------------------------------
--- Make instance of Classes
------------------------------------------------------------------------------
 op prepAttrTerm : Term PortDeclSet -> Term .
 op prepAttrTerm : Term VariableDeclSet -> Term .
 op prepAttrTerm : Term Contract -> Term .

 ceq prepAttrTerm(T, out(I,sinc) PDS)
     = prepAttrTerm('_`,_[T, T'], PDS)
  if T' := qid(strId(I) + "-status :_")['unlocked.PortStatus] .
 eq prepAttrTerm(T, mt-port) = T .
 eq prepAttrTerm(T, PD PDS) = prepAttrTerm(T, PDS) [owise] .

 ceq prepAttrTerm(T, local(I, VT) VDS)
     = prepAttrTerm('_`,_[T, T'], VDS)
  if T' := qid(strId(I) + " :_")[defaultValue(VT)] .

 ceq prepAttrTerm(T, local(I, VT, E) VDS)
     = prepAttrTerm('_`,_[T, T'], VDS)
  if T' := qid(strId(I) + " :_")[e2t(E)] .

 ceq prepAttrTerm(T, required(I, VT) VDS)
     = prepAttrTerm('_`,_[T, T'], VDS)
  if T' := qid(strId(I) + " :_")['st[defaultValue(VT), 'unchanged.Status]] .
 eq prepAttrTerm(T, mt-var) = T .

 eq prepAttrTerm(T, exclusive(ITS)) =
    '_`,_[T, 'status`:_['unlocked.PortStatus]] .
 eq prepAttrTerm(T, CTD) = T [owise] .

 op makeEqInstance : ModName VariableDeclSet PortDeclSet -> EEquation .
 op makeEqInstance : ModName VariableDeclSet Contract -> EEquation .

 ceq makeEqInstance(MN, VDS, PDS) =
     (eq 'instantiate[ov, classC(MN)] = '<_:_|_>[ov, classC(MN), T] [none] .)
  if T := prepAttrTerm(prepAttrTerm('none.AttributeSet, PDS), VDS) .

 ceq makeEqInstance(MN, VDS, CTD) =
     (eq 'instantiate[ov, classC(MN)] = '<_:_|_>[ov, classC(MN), T'] [none] .)
  if T  := prepAttrTerm('none.AttributeSet, CTD) /\
     T' := prepAttrTerm(T, VDS) .

------------------------------------------------------------------------------
--- Binds, Links and Instances
------------------------------------------------------------------------------

--- the functions below generate the equation for the application
--- topology, it receives the instances declarations

 op prepEqTopology : ConfigDeclSet -> EEquation .
```

```
 op prepEqTopoRHS : ConfigDeclSet -> Term .

 eq prepEqTopology(CDS)
   = (eq 'topology.Configuration = prepEqTopoRHS(CDS) [none] .) .

 eq prepEqTopoRHS(instantiate(I1, I2) CDS)
   = '__['instantiate[consId(I1,"Oid"), classC(I2)], prepEqTopoRHS(CDS)] .
 eq prepEqTopoRHS(mt-config) = 'none.Configuration .


--- abaixo as funcoes que transformam uma unidade recebida de acordo com
--- as declaracoes de binds, instancias e links

 op instances2omod : Unit ConfigDeclSet -> Unit .
 op links2omod : Unit ConfigDeclSet -> Unit .
 op binds2omod : Unit ConfigDeclSet ConfigDeclSet -> Unit .

 vars Q1 Q1' Q2 Q2' : Qid .
 vars CI1 CI2 V1 V2 OI1 OI2 PI1 PI2 : Id .

 eq instances2omod(U, instantiate(I1, I2) CDS) =
    instances2omod(
     addOps((op qidId(I1) : nil -> 'Oid [none] .),
      addImports((including qidId(I2) .), U)), CDS) .

 eq instances2omod(U, mt-config) = U .


 eq links2omod(U, link(OI1,PI1,OI2,PI2) CDS) =
    links2omod(
      addEqs(
       (eq 'send[consId(OI1,"Oid"), portOutC(PI1), itv] =
           'send[consId(OI2,"Oid"),  portInC(PI2), itv]
        [labelRule(PI1,"linking",PI2)] .)
       , U), CDS) .

 eq links2omod(U, mt-config) = U .


 ceq binds2omod(U, CDS', bind(I1,V1,I2,V2,VT) CDS) =
     binds2omod(
      addEqs(
        (eq '__[objt(I1, CI1, Q1'['st[Q1, 'changed.Status]]),
                objt(I2, CI2, Q2'[Q2])] =
            '__[objt(I1, CI1, Q1'['st[Q1, 'unchanged.Status]]),
                objt(I2, CI2, Q2'[Q1])] [none] .)
        (ceq '__[objt(I1, CI1, Q1'['st[Q1, 'unchanged.Status]]),
                 objt(I2, CI2, Q2'[Q2])] =
             '__[objt(I1, CI1, Q1'['st[Q2, 'unchanged.Status]]),
                 objt(I2, CI2, Q2'[Q2])]
          if '_=/=_[Q1, Q2] = 'true.Bool [none] .)
        , U)
      , CDS', CDS)
  if instantiate(I1, CI1) instantiate(I2, CI2) CDS'' := CDS' /\
```

```
     Q1  := qid("V1:" + string(type?(VT))) /\
     Q1' := qid(strId(V1) + " :_")         /\
     Q2  := qid("V2:" + string(type?(VT))) /\
     Q2' := qid(strId(V2) + " :_") .

 eq binds2omod(U, CDS, mt-config) = U .

------------------------------------------------------------------------------
--- Top level function to convert, modules, connector and applications
--- to their representation in RWL
------------------------------------------------------------------------------
 op cb2omod : StrOModule ComponentDecl -> StrOModule .

 ceq cb2omod(U, module(I, VDS, PDS))
    = vars2omod(ports2omod1(addEqs(EQS, addClasses(CLD, U')), PDS), VDS)
  if  MN := getName(U) /\
      CLD := makeClass(class MN | none ., VDS, PDS) /\
      EQS := makeEqInstance(MN, VDS, PDS) /\
      U'  := addImports((including 'CBABEL-CONFIGURATION .), U) .

 ceq cb2omod(U, connector(I, VDS, PDS, CTD))
    = contract2omod(
       vars2omod(ports2omod2(addEqs(EQS, addClasses(CLD, U')), PDS), VDS)
       , CTD)
  if  MN := getName(U) /\
      CLD := makeClass(class MN | none ., VDS, CTD) /\
      EQS := makeEqInstance(MN, VDS, CTD) /\
      U'  := addImports((including 'CBABEL-CONFIGURATION .), U) .

 ceq cb2omod(U, application(I, CDS1, CDS2, CDS3))
    = links2omod(binds2omod(instances2omod(addEqs(EQS, U'), CDS1)
                            , CDS1, CDS3), CDS2)
  if EQS := prepEqTopology(CDS1) /\
      U' := addOps((op 'topology : nil -> 'Configuration [none] .),
             addImports((including 'CBABEL-CONFIGURATION .), U)) .

endfm

fmod CBABEL-EVALUATION is
 pr EVALUATION .
 pr CBABEL-CONVERTER .

 var MN : ModName .
 vars U U' U'' U''' : Unit .
 var UL : List<Unit> .
 var VDS : Set<EOpDecl> .
 vars DB DB' : Database .
 var EIL : List<EImport> .

 op evalCbUnit : Unit Database -> Database .

 ceq evalCbUnit(U, DB) = evalUnit(U', none, DB)
  if  U : ComponentDecl /\
      MN := getName(U) /\
```

```
     U' := cb2omod(setName(emptyStrOModule, MN), U) .

endfm

fmod CBABEL-DECL-PARSING is
 pr UNIT-DECL-PARSING .
 pr EXT-META-FULL-MAUDE-SIGN .
 pr CBABEL-UNIT .

 sort TupleGuard .
 op noId : -> Id .
 op <_;_;_;_> : Exp StmtSeq StmtSeq Id -> TupleGuard .
 op emptyGuard : -> TupleGuard .
 eq emptyGuard = < TRUE ; skip ; skip ; noId > .

 sort Tuple<Id|Id> .
 op <_;_> : Id Id -> Tuple<Id|Id> .

 op parseId : Term -> Id .
 op parseExp : Term -> Exp .
 op parseStmt : Term -> StmtSeq .
 op parseVarType : Term -> VarType .
 op parseGuard : Term TupleGuard -> TupleGuard .
 op splitId : Term -> Tuple<Id|Id> .

 vars T1 T2 T3 T4 T T' T'' : Term .
 vars PU U : Unit .
 var C : Constant .
 vars VDS : Set<EOpDecl> .
 vars SS1 SS2 : StmtSeq .
 var E : Exp .
 vars I I1 I2 I3 I4 : Id .
 vars GT GT' : TupleGuard .
 var S : String .
 var N : Nat .
 vars IS IS' : IdSet .
 var ITS : InteractionSet .
 var Q : Qid .
 var RP : ResultPair? .

--- the function splitIds is used to convert bind's and link's
--- declarations: bind x.y to z.t
---
--- neste versao, nenhum tratamento de erro e feito, ou seja,
--- assume-se que as descricoes cbabel dadas como entrada estao
--- corretas e bem formatadas.

 ceq splitId(T) = < ID(qid(substr(S, 0, N))) ;
                    ID(qid(substr(S, s(N), length(S)))) >
  if ID(Q) := parseId(T) /\ S := string(Q) /\ N := find(S, ".", 0) .

 ceq parseId('cbtoken[C]) =
    if RP :: ResultPair then
      VL(downTerm(getTerm(RP), error))
```

```
   else
     ID(downQid(C))
   fi
 if Q := downQid(C) /\
    RP := metaParse(CB-GRAMMAR, Q, anyType) .

eq parseVarType('int.eVarType) = Integer .
eq parseVarType('bool.eVarType) = Boolean .

eq parseExp('TRUE.eBoolean) = (TRUE).Exp .
eq parseExp('FALSE.eBoolean) = (FALSE).Exp .
eq parseExp('cbtoken[C]) = parseId('cbtoken[C]) .
op error : -> [Int] .

eq parseExp('_<_[T1, T2])  = less-than(parseExp(T1), parseExp(T2)) .
eq parseExp('_>_[T1, T2])  = greater-than(parseExp(T1), parseExp(T2)) .
eq parseExp('_-_[T1, T2])  = minus(parseExp(T1), parseExp(T2)) .
eq parseExp('_+_[T1, T2])  = plus(parseExp(T1), parseExp(T2)) .
eq parseExp('_==_[T1, T2]) = equal(parseExp(T1), parseExp(T2)) .
eq parseExp('_!=_[T1, T2]) = not-equal(parseExp(T1), parseExp(T2)) .
eq parseExp('_&&_[T1, T2]) = land(parseExp(T1), parseExp(T2)) .
eq parseExp('_||_[T1, T2]) = lor(parseExp(T1), parseExp(T2)) .
eq parseExp('`(_`)[T]) = (parseExp(T)).Exp .

eq parseStmt('skip.eStmtSeq) = skip .
eq parseStmt('_=_;[T1, T2]) = assign(parseId(T1), parseExp(T2)) .

eq parseStmt('if`(_`)`{_`}[T1, T2])
   = if-then-else(parseExp(T1), parseStmt(T2), skip) .
eq parseStmt('if`(_`)`{_`}else`{_`}[T1, T2, T3])
   = if-then-else(parseExp(T1), parseStmt(T2), parseStmt(T3)) .
eq parseStmt('__[T1, T2]) = stmt-seq(parseStmt(T1), parseStmt(T2)) .


ceq parseGuard('__[T1, T2], GT) = parseGuard(T2, GT')
 if GT' := parseGuard(T1, GT) .
eq parseGuard('before`{_`}[T], < E ; SS1 ; SS2 ; I >)
   = < E ; parseStmt(T) ; SS2 ; I > .
eq parseGuard('after`{_`}[T], < E ; SS1 ; SS2 ; I >)
   = < E ; SS1 ; parseStmt(T) ; I > .
eq parseGuard('alternative`(_`);[T], < E ; SS1 ; SS2 ; I >)
   = < E ; SS1 ; SS2 ; parseId(T) > .
eq parseGuard('guard`(_`)[T], < E ; SS1 ; SS2 ; I >)
   = < parseExp(T) ; SS1 ; SS2 ; I > .
eq parseGuard('guard`(_`)`{_`}[T, T1], < E ; SS1 ; SS2 ; I >)
   = parseGuard(T1, < parseExp(T) ; SS1 ; SS2 ; I >) .


eq parseDecl('in`port_;[T], PU, U, VDS)
   = < addPort(in(parseId(T), sinc), PU) ; U ; VDS > .
eq parseDecl('out`port_;[T], PU, U, VDS)
   = < addPort(out(parseId(T), sinc), PU) ; U ; VDS > .
eq parseDecl('in`port`oneway_;[T], PU, U, VDS)
   = < addPort(in(parseId(T), assinc), PU) ; U ; VDS > .
```

```
 eq parseDecl('out‘port‘oneway_;[T], PU, U, VDS)
    = < addPort(out(parseId(T), assinc), PU) ; U ; VDS > .

 eq parseDecl('var__;[T1, T2], PU, U, VDS)
    = < addVar(local(parseId(T2), parseVarType(T1)), PU) ; U ; VDS > .
 eq parseDecl('staterequired__;[T1, T2], PU, U, VDS)
    = < addVar(required(parseId(T2), parseVarType(T1)), PU) ; U ; VDS > .
 eq parseDecl('var__=_;[T1, T2, T3], PU, U, VDS)
    = < addVar(local(parseId(T2), parseVarType(T1), parseExp(T3)), PU) ;
        U ; VDS > .

------------------------------------------------------------------------
--- the following equations define de parsing of contracts that could
--- be defined in connector modules.
------------------------------------------------------------------------
 op parseIdSet : Term -> IdSet .
 op parseInterSet : Term InteractionSet -> InteractionSet .
 op parseInter : Term -> Interaction .

 eq parseIdSet('_|_[T, T']) = parseId(T) ; parseIdSet(T') .
 eq parseIdSet(T) = parseId(T) [owise] .

 ceq parseInterSet('__[T, T'], ITS)
     = parseInterSet(T', seq(parseIdSet(T1), parseIdSet(T2)) ITS)
  if '_>_;[T1, T2] := T .

 eq parseInterSet('_>_;[T, T'], ITS)
    = seq(parseIdSet(T), parseIdSet(T')) ITS [owise] .

 eq parseDecl('interaction‘{_‘}[T], PU, U, VDS)
    = < addContract(interaction(parseInter(T)), PU) ; U ; VDS > .

 eq parseInter('_>_;[T, T']) = seq(parseIdSet(T), parseIdSet(T')) .

 ceq parseInter('_>_>_;[T1, T2, T3])
     = if I == noId then
        guard(parseId(T1), parseId(T3), E, SS1, SS2)
       else
        guard(parseId(T1), parseId(T3), I, E, SS1, SS2)
       fi
  if < E ; SS1 ; SS2 ; I > := parseGuard(T2, emptyGuard) .

 ceq parseDecl('exclusive‘{_‘}[T], PU, U, VDS)
     = < addContract(exclusive(ITS), PU) ; U ; VDS >
  if ITS := parseInterSet(T, mt-interaction) .


------------------------------------------------------------------------
--- the following equations define de parsing of instances, links and
--- binds of application modules.
------------------------------------------------------------------------

 eq parseDecl('instantiate_as_;[T1, T2], PU, U, VDS)
    = < addInstance(instantiate(parseId(T2), parseId(T1)), PU) ; U ; VDS > .
```

```
 ceq parseDecl('link_to_;[T1, T2], PU, U, VDS)
     = < addLink(link(I1, I2, I3, I4), PU) ; U ; VDS >
  if < I1 ; I2 > := splitId(T1) /\
     < I3 ; I4 > := splitId(T2) .

 ceq parseDecl('bind__to_;[T3, T1, T2], PU, U, VDS)
     = < addBind(bind(I1, I2, I3, I4, parseVarType(T3)), PU) ; U ; VDS >
  if < I1 ; I2 > := splitId(T1) /\
     < I3 ; I4 > := splitId(T2) .
endfm


fmod CBABEL-PROCESSING is
 pr UNIT-PROCESSING .
 pr CBABEL-DECL-PARSING .
 pr CBABEL-EVALUATION .

 var PDR : ParseDeclResult .
 var VDS : Set<EOpDecl> .
 vars PU PU' U U' : Unit .
 vars T T' T'' T''' : Term .
 vars QI F : Qid .
 var  QIL : QidList .
 var  TL : TermList .
 var  DB : Database .

--- procUnit3 do FM chama procUnit4 com duas copias do modulo, uma
--- para conter declaracoes com bubbles e outra sem, e um conjunto de
--- variaveis. Embora a procCbUnit4 seja declarada apenas com uma unit
--- como parametro pode ser conveniente no futuro usar o parametro de
--- conj de variaveis da procUnit4 original para renomear as variaveis
--- dos modulos. No entanto, em cbabel nao ha necessidade de lidar com
--- bubbles

*** op procCbUnit3 : Term Term Term Unit Database -> Database .
 op procCbUnit4 : Term Term Unit Database -> Database .

 eq procUnit2(T, 'module_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyModule, DB) .
 eq procUnit2(T, 'connector_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyConnector, DB) .
 eq procUnit2(T, 'application_'{_'}[T', T''], DB)
   = procUnit3(T, T', T'', emptyApplication, DB) .

***  ceq procCbUnit3(T, 'cbtoken[T'], T'', U, DB)
***     = procCbUnit4(T, T'', setName(U, QI), DB)
***   if QI := downQid(T') .

 ceq procUnit3(T, 'cbtoken[T'], T'', U, DB)
     = procCbUnit4(T, T'', setName(U, QI), DB)
  if QI := downQid(T') .

 ceq procCbUnit4(T, '__[T', T''], PU, DB)
```

```
    = procCbUnit4(T, T'', preUnit(PDR), DB)
  if PDR := parseDecl(T', PU, PU, none) .

 ceq procCbUnit4(T, F[TL], PU, DB)
    = evalCbUnit(preUnit(PDR), insertTermUnit(getName(PU), T, DB))
  if F =/= '__  /\
     PDR := parseDecl(F[TL], PU, PU, none) .
endfm

mod EXT-DATABASE-HANDLING is
 inc DATABASE-HANDLING .
 inc INITIAL-DB .
 pr CBABEL-PROCESSING .

 var  F : Qid .
 vars QIL QIL' QIL'' : QidList .
 vars T T' T'' T''' : Term .
 var TL : TermList .
 vars DB DB' : Database .
 var MN : ModName .
 var Atts : AttributeSet .
 var X@Database : DatabaseClass .
 var O : Oid .

 eq parseModName('cbtoken[T]) = downQid(T) .

 crl [module] :
   < O : X@Database | db : DB,  input : (F[T, T']),
         output : nil, default : MN, Atts >
   =>
   < O : X@Database | db : procUnit(F[T, T'], DB), input : nilTermList,
         output :
          ('Introduced 'CBabel 'Component modNameToQid(parseModName(T)) '\n),
         default : parseModName(T), Atts >
   if (F == 'module_`{_`}) or-else
      (F == 'connector_`{_`}) or-else
      (F == 'application_`{_`}) .
endm


mod CBABEL-TOOL is
  pr EXT-DATABASE-HANDLING .
  pr EXT-META-FULL-MAUDE-SIGN .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .

  op o : -> Oid .
  op init : -> System .

  var  Atts : AttributeSet .
  var  X@Database : DatabaseClass .
  var  O : Oid .
```

```
  var  DB : Database .
  var  MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var  TL : TermList .
  var  N : Nat .
  vars RP RP' : ResultPair .


  rl [init] : init =>
    [nil,
     < o : Database | db : initial-db, input : nilTermList, output : nil,
                      default : 'CONVERSION >,
     ('\n '\t '\s '\s '\s '\s '\s '\s '\s
      'Cbabel 'Tool '2.4 '\s '`( 'February '14th '`, '\s '2005 '`) '\n)] .

  crl [in] :
    [QIL, < O : X@Database | input : nilTermList, Atts >, QIL'] =>
    [nil, < O : X@Database |
                input : getTerm(metaParse(CB-GRAMMAR, QIL, 'Input)), Atts >,
     QIL']
   if QIL =/= nil
      /\ metaParse(CB-GRAMMAR, QIL, 'Input) : ResultPair .


  crl [in] :
    [QIL, < O : X@Database | Atts, output : nil >, QIL'] =>
    [nil,
          < O : X@Database | Atts,
           output : ('\r 'Warning:
                     printSyntaxError(metaParse(CB-GRAMMAR, QIL, 'Input), QIL)
                     '\n
                     '\r 'Error: '\o 'No 'parse 'for 'input. '\n) >,
     QIL']
    if QIL =/= nil
       /\ not metaParse(CB-GRAMMAR, QIL, 'Input) :: ResultPair .


  crl [out] :
    [QIL, < O : X@Database | output : QIL', Atts >, QIL'']
    =>
    [QIL, < O : X@Database | output : nil, Atts >, (QIL' QIL'')]
   if QIL' =/= nil .
endm

loop init .

in cbabel-configuration.maude

trace exclude CBABEL-TOOL .
set show loop stats on .
set show loop timing on .
```

APÊNDICE B – Módulos orientados a objetos dos estudos de caso

Apresentamos neste Apêndice os módulos orientados a objetos produzidos por Maude CBabel tool para cada um dos componentes das arquiteturas CBabel apresentadas nos estudos de caso do Capítulo 4. Em Maude CBabel tool, para exibirmos o módulo orientado a objetos produzido a partir da carga de um componente CBabel, podemos usar o comando `(show [nome-do-componente] .)`.

B.1 Máquina de Venda

Ao carregarmos os componentes da arquitetura `VENDING-MACHINE` (Figura 5) em Maude CBabel tool, os seguintes módulos orientados a objetos de Full Maude são gerados por Maude CBabel tool e importados no banco de dados de módulos de Full Maude:

Módulo BUY-APPLE

```
omod BUY-APPLE is
  including CBABEL-CONFIGURATION .

  class BUY-APPLE .
  op buy-apple : -> PortOutId [ctor] .

  eq instantiate(O:Oid,BUY-APPLE) = < O:Oid : BUY-APPLE | none > .

  rl < O:Oid : BUY-APPLE | > ack([O:Oid,buy-apple]) => < O:Oid : BUY-APPLE | >
   [label BUY-APPLE-receivingAck-buy-apple] .

  rl < O:Oid : BUY-APPLE | > do(O:Oid,buy-apple,none) =>
     < O:Oid : BUY-APPLE | > send(O:Oid,buy-apple,[O:Oid,buy-apple])
   [label BUY-APPLE-sending-buy-apple] .
endom
```

Módulo BUY-CAKE

```
omod BUY-CAKE is
  including CBABEL-CONFIGURATION .
```

```
3
4    class BUY-CAKE .
5    op buy-cake : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,BUY-CAKE) = < O:Oid : BUY-CAKE | none > .
8
9    rl < O:Oid : BUY-CAKE | > ack([O:Oid,buy-cake]) =>
10      < O:Oid : BUY-CAKE | >
11    [label BUY-CAKE-receivingAck-buy-cake] .
12
13   rl < O:Oid : BUY-CAKE | > do(O:Oid,buy-cake,none) =>
14      < O:Oid : BUY-CAKE | > send(O:Oid,buy-cake,[O:Oid,buy-cake])
15    [label BUY-CAKE-sending-buy-cake] .
16 endom
```

Módulo ADD-DOLLAR

```
1  omod ADD-DOLLAR is
2    including CBABEL-CONFIGURATION .
3
4    class ADD-DOLLAR .
5    op add-$ : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,ADD-DOLLAR) = < O:Oid : ADD-DOLLAR | none > .
8
9    rl < O:Oid : ADD-DOLLAR | > ack([O:Oid,add-$]) =>
10      < O:Oid : ADD-DOLLAR | >
11    [label ADD-DOLLAR-receivingAck-add-$] .
12   rl < O:Oid : ADD-DOLLAR | > do(O:Oid,add-$,none) =>
13      < O:Oid : ADD-DOLLAR | > send(O:Oid,add-$,[O:Oid,add-$])
14    [label ADD-DOLLAR-sending-add-$] .
15 endom
```

Módulo ADD-QUARTER

```
1  omod ADD-QUARTER is
2    including CBABEL-CONFIGURATION .
3
4    class ADD-QUARTER .
5    op add-q : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,ADD-QUARTER) = < O:Oid : ADD-QUARTER | none > .
8
9    rl < O:Oid : ADD-QUARTER | > ack([O:Oid,add-q]) =>
10      < O:Oid : ADD-QUARTER | >
11    [label ADD-QUARTER-receivingAck-add-q] .
12   rl < O:Oid : ADD-QUARTER | > do(O:Oid,add-q,none) =>
13      < O:Oid : ADD-QUARTER | > send(O:Oid,add-q,[O:Oid,add-q])
14    [label ADD-QUARTER-sending-add-q] .
15 endom
```

Módulo CHANGE

```
1  omod CHANGE is
2    including CBABEL-CONFIGURATION .
3
```

```
4    class CHANGE .
5    op change : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,CHANGE) = < O:Oid : CHANGE | none > .
8
9    rl < O:Oid : CHANGE | > ack([O:Oid,change]) =>
10      < O:Oid : CHANGE | >
11    [label CHANGE-receivingAck-change] .
12   rl < O:Oid : CHANGE | > do(O:Oid,change,none) =>
13      < O:Oid : CHANGE | > send(O:Oid,change,[O:Oid,change])
14    [label CHANGE-sending-change] .
15 endom
```

Módulo SLOT

```
1  omod SLOT is
2    including CBABEL-CONFIGURATION .
3
4    class SLOT | apples : Int, cakes : Int .
5    op get-apples : Object -> Int .
6    op get-cakes : Object -> Int .
7    op set-apples : Object Int -> Object .
8    op set-cakes : Object Int -> Object .
9    op put-apple : -> PortInId [ctor] .
10   op put-cake : -> PortInId [ctor] .
11
12   eq get-apples(< O:Oid : SLOT | apples : V:Int >) = V:Int .
13   eq get-cakes(< O:Oid : SLOT | cakes : V:Int >) = V:Int .
14   eq set-apples(< O:Oid : SLOT | apples : V:Int >,V':Int) =
15      < O:Oid : SLOT | apples : V':Int > .
16   eq set-cakes(< O:Oid : SLOT | cakes : V:Int >,V':Int) =
17      < O:Oid : SLOT | cakes : V':Int > .
18   eq instantiate(O:Oid,SLOT) = < O:Oid : SLOT |(none,apples : 0),cakes : 0 > .
19
20   rl < O:Oid : SLOT | > done(O:Oid,put-apple,IT:Interaction) =>
21      < O:Oid : SLOT | >
22    [label SLOT-done-put-apple] .
23   rl < O:Oid : SLOT | > done(O:Oid,put-cake,IT:Interaction) =>
24      < O:Oid : SLOT | >
25    [label SLOT-done-put-cake] .
26   rl < O:Oid : SLOT | > send(O:Oid,put-apple,IT:Interaction) =>
27      < O:Oid : SLOT | > do(O:Oid,put-apple,IT:Interaction)
28    [label SLOT-receivingAndDo-put-apple] .
29   rl < O:Oid : SLOT | > send(O:Oid,put-cake,IT:Interaction) =>
30      < O:Oid : SLOT | > do(O:Oid,put-cake,IT:Interaction)
31    [label SLOT-receivingAndDo-put-cake] .
32 endom
```

Conector MAKE-CHANGE

```
1  omod MAKE-CHANGE is
2    including CBABEL-CONFIGURATION .
3
4    class MAKE-CHANGE | ch@dollars : StateRequired, ch@quarters : StateRequired .
5    op get-ch@dollars : Object -> Int .
```

```
6    op get-ch@quarters : Object -> Int .
7    op set-ch@dollars : Object Int -> Object .
8    op set-ch@quarters : Object Int -> Object .
9    op change-in : -> PortInId [ctor] .
10
11   eq get-ch@dollars(< O:Oid : MAKE-CHANGE | ch@dollars : st(V:Int,S:Status)>)
12      = V:Int .
13   eq get-ch@quarters(< O:Oid : MAKE-CHANGE | ch@quarters : st(V:Int,S:Status)>)
14      = V:Int .
15   eq instantiate(O:Oid,MAKE-CHANGE) =
16      < O:Oid : MAKE-CHANGE |(none,ch@dollars : st(0,unchanged)),ch@quarters : st(0,
17     unchanged)> .
18   eq set-ch@dollars(< O:Oid : MAKE-CHANGE | ch@dollars : st(V:Int,S:Status)>,V':Int)
19      = < O:Oid : MAKE-CHANGE | ch@dollars : st(V':Int,changed)> .
20   eq set-ch@quarters(< O:Oid : MAKE-CHANGE | ch@quarters : st(V:Int,S:Status)>,V':Int) =
21      < O:Oid : MAKE-CHANGE | ch@quarters : st(V':Int,changed)> .
22
23   ceq after(OBJ:Object) = OBJ:Object
24     if class(OBJ:Object)= MAKE-CHANGE .
25
26   ceq before(OBJ:Object) =
27       set-ch@quarters(set-ch@dollars(OBJ:Object,get-ch@dollars(OBJ:Object)+ 1),
28        get-ch@quarters(set-ch@dollars(OBJ:Object,get-ch@dollars(OBJ:Object)+ 1))- 4)
29    if class(OBJ:Object)= MAKE-CHANGE .
30
31   ceq open?(OBJ:Object) =
32       get-ch@quarters(OBJ:Object)> 3
33     if class(OBJ:Object)= MAKE-CHANGE .
34
35   rl < O:Oid : MAKE-CHANGE | > ack([O:Oid,ground]:: IT:Interaction)
36     => after(< O:Oid : MAKE-CHANGE | >) ack(IT:Interaction)
37    [label MAKE-CHANGE-acking-ground] .
38   crl < O:Oid : MAKE-CHANGE | > send(O:Oid,change-in,IT:Interaction)
39     => before(< O:Oid : MAKE-CHANGE | >) send(O:Oid,ground,[O:Oid,ground]:: IT:Interaction)
40     if open?(< O:Oid : MAKE-CHANGE | >)= true
41    [label MAKE-CHANGE-sending-change-in] .
42 endom
```

Conector COUNT-DOLLAR

```
1  omod COUNT-DOLLAR is
2    including CBABEL-CONFIGURATION .
3
4    class COUNT-DOLLAR | dollars : Int .
5    op get-dollars : Object -> Int .
6    op set-dollars : Object Int -> Object .
7    op inc-$ : -> PortInId [ctor] .
8
9    eq get-dollars(< O:Oid : COUNT-DOLLAR | dollars : V:Int >) = V:Int .
10   eq instantiate(O:Oid,COUNT-DOLLAR) =
11      < O:Oid : COUNT-DOLLAR | none,dollars : 0 > .
12   eq set-dollars(< O:Oid : COUNT-DOLLAR | dollars : V:Int >,V':Int)
13      = < O:Oid : COUNT-DOLLAR | dollars : V':Int > .
14
15   ceq after(OBJ:Object) = OBJ:Object
```

```
16      if class(OBJ:Object)= COUNT-DOLLAR .
17
18    ceq before(OBJ:Object)
19      = set-dollars(OBJ:Object,get-dollars(OBJ:Object)+ 1)
20      if class(OBJ:Object)= COUNT-DOLLAR .
21
22    ceq open?(OBJ:Object)
23      = true
24      if class(OBJ:Object)= COUNT-DOLLAR .
25
26    rl < O:Oid : COUNT-DOLLAR | > ack([O:Oid,ground]:: IT:Interaction)
27      => after(< O:Oid : COUNT-DOLLAR | >) ack(IT:Interaction)
28     [label COUNT-DOLLAR-acking-ground] .
29    crl < O:Oid : COUNT-DOLLAR | > send(O:Oid,inc-$,IT:Interaction)
30      => before(< O:Oid : COUNT-DOLLAR | >) send(O:Oid,ground,[O:Oid,ground] :: IT:Interaction)
31      if open?(< O:Oid : COUNT-DOLLAR | >)= true
32     [label COUNT-DOLLAR-sending-inc-$] .
33  endom
```

Conector COUNT-QUARTER

```
1   omod COUNT-QUARTER is
2     including CBABEL-CONFIGURATION .
3
4     class COUNT-QUARTER | quarters : Int .
5     op get-quarters : Object -> Int .
6     op set-quarters : Object Int -> Object .
7     op inc-q : -> PortInId [ctor] .
8
9     eq get-quarters(< O:Oid : COUNT-QUARTER | quarters : V:Int >) = V:Int .
10    eq instantiate(O:Oid,COUNT-QUARTER)
11      = < O:Oid : COUNT-QUARTER | none,quarters : 0 > .
12    eq set-quarters(< O:Oid : COUNT-QUARTER | quarters : V:Int >,V':Int)
13      = < O:Oid : COUNT-QUARTER | quarters : V':Int > .
14
15    ceq after(OBJ:Object) = OBJ:Object
16      if class(OBJ:Object)= COUNT-QUARTER .
17    ceq before(OBJ:Object)
18      = set-quarters(OBJ:Object,get-quarters(OBJ:Object)+ 1)
19      if class(OBJ:Object)= COUNT-QUARTER .
20    ceq open?(OBJ:Object) = true
21      if class(OBJ:Object)= COUNT-QUARTER .
22
23    rl < O:Oid : COUNT-QUARTER | > ack([O:Oid,ground]:: IT:Interaction)
24      => after(< O:Oid : COUNT-QUARTER | >) ack(IT:Interaction)
25     [label COUNT-QUARTER-acking-ground] .
26    crl < O:Oid : COUNT-QUARTER | > send(O:Oid,inc-q,IT:Interaction)
27      => before(< O:Oid : COUNT-QUARTER | >) send(O:Oid,ground,[O:Oid,ground]:: IT:Interaction)
28      if open?(< O:Oid : COUNT-QUARTER | >)= true
29     [label COUNT-QUARTER-sending-inc-q] .
30  endom
```

Conector SOLD-APPLE

```
1   omod SOLD-APPLE is
2     including CBABEL-CONFIGURATION .
```

```
3
4    class SOLD-APPLE | sa@dollars : StateRequired, sa@quarters : StateRequired .
5    op get-sa@dollars : Object -> Int .
6    op get-sa@quarters : Object -> Int .
7    op set-sa@dollars : Object Int -> Object .
8    op set-sa@quarters : Object Int -> Object .
9    op ack-apple : -> PortInId  [ctor] .
10   op give-apple : -> PortOutId [ctor] .
11
12   eq get-sa@dollars(< O:Oid : SOLD-APPLE | sa@dollars : st(V:Int,S:Status)>) = V:Int .
13   eq get-sa@quarters(< O:Oid : SOLD-APPLE | sa@quarters : st(V:Int,S:Status)>) = V:Int .
14   eq set-sa@dollars(< O:Oid : SOLD-APPLE | sa@dollars : st(V:Int,S:Status)>,V':Int)
15     = < O:Oid : SOLD-APPLE | sa@dollars : st(V':Int,changed)> .
16   eq set-sa@quarters(< O:Oid : SOLD-APPLE | sa@quarters : st(V:Int,S:Status)>,V':Int)
17     = < O:Oid : SOLD-APPLE | sa@quarters : st(V':Int,changed)> .
18   eq instantiate(O:Oid,SOLD-APPLE)
19     = < O:Oid : SOLD-APPLE |(none,sa@dollars : st(0,unchanged)),sa@quarters : st(0,
20     unchanged)> .
21
22   ceq after(OBJ:Object) = OBJ:Object
23     if class(OBJ:Object)= SOLD-APPLE .
24
25   ceq before(OBJ:Object)
26     = set-sa@quarters(set-sa@dollars(OBJ:Object,get-sa@dollars(OBJ:Object)- 1),
27     get-sa@quarters(set-sa@dollars(OBJ:Object,get-sa@dollars(OBJ:Object)- 1))+ 1)
28     if class(OBJ:Object)= SOLD-APPLE .
29
30   ceq open?(OBJ:Object) = get-sa@dollars(OBJ:Object)> 0
31     if class(OBJ:Object)= SOLD-APPLE .
32
33   rl < O:Oid : SOLD-APPLE | > ack([O:Oid,give-apple]:: IT:Interaction)
34     => after(< O:Oid : SOLD-APPLE | >) ack(IT:Interaction)
35    [label SOLD-APPLE-acking-give-apple] .
36   crl < O:Oid : SOLD-APPLE | > send(O:Oid,ack-apple,IT:Interaction)
37     => before(< O:Oid : SOLD-APPLE | >) send(O:Oid,give-apple,[O:Oid,give-apple]:: IT:Interaction)
38     if open?(< O:Oid : SOLD-APPLE | >)= true
39    [label SOLD-APPLE-sending-ack-apple] .
40 endom
```

Conector SOLD-CAKE

```
1  omod SOLD-CAKE is
2    including CBABEL-CONFIGURATION .
3
4    class SOLD-CAKE | sc@dollars : StateRequired .
5    op get-sc@dollars : Object -> Int .
6    op set-sc@dollars : Object Int -> Object .
7    op ack-cake : -> PortInId  [ctor] .
8    op give-cake : -> PortOutId [ctor] .
9
10   eq get-sc@dollars(< O:Oid : SOLD-CAKE | sc@dollars : st(V:Int,S:Status)>)
11     = V:Int .
12   eq set-sc@dollars(< O:Oid : SOLD-CAKE | sc@dollars : st(V:Int,S:Status)>,V':Int)
13     = < O:Oid : SOLD-CAKE | sc@dollars : st(V':Int,changed)> .
14   eq instantiate(O:Oid,SOLD-CAKE)
```

```
15      = < O:Oid : SOLD-CAKE | none,sc@dollars : st(0,unchanged)> .
16
17    ceq after(OBJ:Object) = OBJ:Object
18      if class(OBJ:Object)= SOLD-CAKE .
19    ceq before(OBJ:Object)
20      = set-sc@dollars(OBJ:Object,get-sc@dollars(OBJ:Object)- 1)
21      if class(OBJ:Object)= SOLD-CAKE .
22    ceq open?(OBJ:Object)
23      = get-sc@dollars(OBJ:Object)> 0
24      if class(OBJ:Object)= SOLD-CAKE .
25
26    rl < O:Oid : SOLD-CAKE | > ack([O:Oid,give-cake]:: IT:Interaction)
27      => after(< O:Oid : SOLD-CAKE | >) ack(IT:Interaction)
28     [label SOLD-CAKE-acking-give-cake] .
29    crl < O:Oid : SOLD-CAKE | > send(O:Oid,ack-cake,IT:Interaction)
30      => before(< O:Oid : SOLD-CAKE | >) send(O:Oid,give-cake,[O:Oid,give-cake]:: IT:Interaction)
31      if open?(< O:Oid : SOLD-CAKE | >)= true
32     [label SOLD-CAKE-sending-ack-cake] .
33  endom
```

Módulo aplicação da arquitetura VENDING-MACHINE

```
1   omod VENDING-MACHINE is
2     including CBABEL-CONFIGURATION .
3     including ADD-DOLLAR .
4     including ADD-QUARTER .
5     including BUY-APPLE .
6     including BUY-CAKE .
7     including CHANGE .
8     including COUNT-DOLLAR .
9     including MAKE-CHANGE .
10    including COUNT-QUARTER .
11    including SOLD-APPLE .
12    including SOLD-CAKE .
13    including SLOT .
14
15    op bt-ad : -> Oid .
16    op bt-aq : -> Oid .
17    op bt-ba : -> Oid .
18    op bt-bc : -> Oid .
19    op bt-change : -> Oid .
20    op con-cd : -> Oid .
21    op con-change : -> Oid .
22    op con-cq : -> Oid .
23    op con-sa : -> Oid .
24    op con-sc : -> Oid .
25    op slot : -> Oid .
26    op topology : -> Configuration .
27
28    eq topology =
29       instantiate(bt-ad,ADD-DOLLAR) instantiate(bt-aq,ADD-QUARTER)
30       instantiate(bt-ba,BUY-APPLE) instantiate(bt-bc,BUY-CAKE)
31       instantiate(bt-change,CHANGE) instantiate(con-cd,COUNT-DOLLAR)
32       instantiate(con-change,MAKE-CHANGE) instantiate(con-cq,COUNT-QUARTER)
33       instantiate(con-sa,SOLD-APPLE) instantiate(con-sc,SOLD-CAKE)
```

```
34      instantiate(slot,SLOT) none .
35
36    eq < con-change : MAKE-CHANGE | ch@dollars : st(V1:Int,changed)>
37       < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
38       < con-change : MAKE-CHANGE | ch@dollars : st(V1:Int,unchanged)>
39       < con-cd : COUNT-DOLLAR | dollars : V1:Int > .
40
41    eq < con-change : MAKE-CHANGE | ch@quarters : st(V1:Int,changed)>
42       < con-cq : COUNT-QUARTER | quarters : V2:Int > =
43       < con-change : MAKE-CHANGE | ch@quarters : st(V1:Int,unchanged)>
44       < con-cq : COUNT-QUARTER | quarters : V1:Int > .
45
46    eq < con-sa : SOLD-APPLE | sa@dollars : st(V1:Int,changed)>
47       < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
48       < con-sa : SOLD-APPLE | sa@dollars : st(V1:Int,unchanged)>
49       < con-cd : COUNT-DOLLAR | dollars : V1:Int > .
50
51    eq < con-sa : SOLD-APPLE | sa@quarters : st(V1:Int,changed)>
52       < con-cq : COUNT-QUARTER | quarters : V2:Int > =
53       < con-sa : SOLD-APPLE | sa@quarters : st(V1:Int,unchanged)>
54       < con-cq : COUNT-QUARTER | quarters : V1:Int > .
55
56    eq < con-sc : SOLD-CAKE | sc@dollars : st(V1:Int,changed)>
57       < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
58       < con-sc : SOLD-CAKE | sc@dollars : st(V1:Int,unchanged)>
59       < con-cd : COUNT-DOLLAR | dollars : V1:Int > .
60
61    ceq < con-change : MAKE-CHANGE | ch@dollars : st(V1:Int,unchanged)>
62        < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
63        < con-change : MAKE-CHANGE | ch@dollars : st(V2:Int,unchanged)>
64        < con-cd : COUNT-DOLLAR | dollars : V2:Int >
65      if V1:Int =/= V2:Int = true .
66
67    ceq < con-change : MAKE-CHANGE | ch@quarters : st(V1:Int,unchanged)>
68        < con-cq : COUNT-QUARTER | quarters : V2:Int > =
69        < con-change : MAKE-CHANGE | ch@quarters : st(V2:Int,unchanged)>
70        < con-cq : COUNT-QUARTER | quarters : V2:Int >
71      if V1:Int =/= V2:Int = true .
72
73    ceq < con-sa : SOLD-APPLE | sa@dollars : st(V1:Int,unchanged)>
74        < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
75        < con-sa : SOLD-APPLE | sa@dollars : st(V2:Int,unchanged)>
76        < con-cd : COUNT-DOLLAR | dollars : V2:Int >
77      if V1:Int =/= V2:Int = true .
78
79    ceq < con-sa : SOLD-APPLE | sa@quarters : st(V1:Int,unchanged)>
80        < con-cq : COUNT-QUARTER | quarters : V2:Int > =
81        < con-sa : SOLD-APPLE | sa@quarters : st(V2:Int,unchanged)>
82        < con-cq : COUNT-QUARTER | quarters : V2:Int >
83      if V1:Int =/= V2:Int = true .
84
85    ceq < con-sc : SOLD-CAKE | sc@dollars : st(V1:Int,unchanged)>
86        < con-cd : COUNT-DOLLAR | dollars : V2:Int > =
87        < con-sc : SOLD-CAKE | sc@dollars : st(V2:Int,unchanged)>
```

```
88        < con-cd : COUNT-DOLLAR | dollars : V2:Int >
89      if V1:Int =/= V2:Int = true .
90
91    eq send(bt-ad,add-$,IT:Interaction) = send(con-cd,inc-$,IT:Interaction)
92     [label add-$-linking-inc-$] .
93    eq send(bt-aq,add-q,IT:Interaction) = send(con-cq,inc-q,IT:Interaction)
94     [label add-q-linking-inc-q] .
95    eq send(bt-ba,buy-apple,IT:Interaction) = send(con-sa,ack-apple,IT:Interaction)
96     [label buy-apple-linking-ack-apple] .
97    eq send(bt-bc,buy-cake,IT:Interaction)  = send(con-sc,ack-cake,IT:Interaction)
98     [label buy-cake-linking-ack-cake] .
99    eq send(bt-change,change,IT:Interaction)  = send(con-change,change-in,IT:Interaction)
100    [label change-linking-change-in] .
101
102   eq send(con-sa,give-apple,IT:Interaction) = send(slot,put-apple,IT:Interaction)
103    [label give-apple-linking-put-apple] .
104   eq send(con-sc,give-cake,IT:Interaction)  = send(slot,put-cake,IT:Interaction)
105    [label give-cake-linking-put-cake] .
106 endom
```

B.2 Produtores e Consumidores

Os seguintes módulos orientados a objetos são produzidos quando os componentes das arquiteturas `PC-DEFAULT`, `PC-MUTEX` e `PC-GUARDS-MUTEX`, das Figuras 8, 9 e 10, respectivamente, são importados em Maude CBabel tool.

Módulo PRODUCER

```
1  omod PRODUCER is
2    including CBABEL-CONFIGURATION .
3
4    class PRODUCER | producer@put-status : PortStatus .
5    op producer@put : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,PRODUCER)
8      = < O:Oid : PRODUCER | none,producer@put-status : unlocked > .
9
10   rl ack([O:Oid,producer@put]) < O:Oid : PRODUCER | producer@put-status : locked >
11     => done(O:Oid,producer@put,none) < O:Oid : PRODUCER | producer@put-status : unlocked >
12    [label PRODUCER-receivingAck-producer@put] .
13   rl do(O:Oid,producer@put,none) < O:Oid : PRODUCER | producer@put-status : unlocked >
14     => send(O:Oid,producer@put,[O:Oid,producer@put]) < O:Oid : PRODUCER | producer@put-status : locked >
15    [label PRODUCER-sending-producer@put] .
16 endom
```

Módulo CONSUMER

```
1  omod CONSUMER is
2    including CBABEL-CONFIGURATION .
3
4    class CONSUMER | consumer@get-status : PortStatus .
```

```
5    op consumer@get : -> PortOutId [ctor] .
6
7    eq instantiate(O:Oid,CONSUMER)
8      = < O:Oid : CONSUMER | none,consumer@get-status : unlocked > .
9
10   rl ack([O:Oid,consumer@get]) < O:Oid : CONSUMER | consumer@get-status : locked >
11     => done(O:Oid,consumer@get,none) < O:Oid : CONSUMER | consumer@get-status : unlocked >
12    [label CONSUMER-receivingAck-consumer@get] .
13   rl do(O:Oid,consumer@get,none) < O:Oid : CONSUMER | consumer@get-status : unlocked >
14     => send(O:Oid,consumer@get,[O:Oid,consumer@get]) < O:Oid : CONSUMER | consumer@get-status : locked >
15    [label CONSUMER-sending-consumer@get] .
16 endom
```

Módulo BUFFER

```
1  omod BUFFER is
2    including CBABEL-CONFIGURATION .
3
4    class BUFFER | items : Int, maxitems : Int .
5    op get-items : Object -> Int .
6    op get-maxitems : Object -> Int .
7    op set-items : Object Int -> Object .
8    op set-maxitems : Object Int -> Object .
9    op buffer@get : -> PortInId [ctor] .
10   op buffer@put : -> PortInId [ctor] .
11
12   eq get-items(< O:Oid : BUFFER | items : V:Int >) = V:Int .
13   eq get-maxitems(< O:Oid : BUFFER | maxitems : V:Int >) = V:Int .
14   eq set-items(< O:Oid : BUFFER | items : V:Int >,V':Int)
15     = < O:Oid : BUFFER | items : V':Int > .
16   eq set-maxitems(< O:Oid : BUFFER | maxitems : V:Int >,V':Int)
17     = < O:Oid : BUFFER | maxitems : V':Int > .
18   eq instantiate(O:Oid,BUFFER)
19     = < O:Oid : BUFFER |(none,items : 0),maxitems : 2 > .
20
21   rl < O:Oid : BUFFER | > done(O:Oid,buffer@get,IT:Interaction)
22     => < O:Oid : BUFFER | > ack(IT:Interaction)
23    [label BUFFER-doneAndAcking-buffer@get] .
24   rl < O:Oid : BUFFER | > done(O:Oid,buffer@put,IT:Interaction)
25     => < O:Oid : BUFFER | > ack(IT:Interaction)
26    [label BUFFER-doneAndAcking-buffer@put] .
27   rl < O:Oid : BUFFER | > send(O:Oid,buffer@get,IT:Interaction)
28     => < O:Oid : BUFFER | > do(O:Oid,buffer@get,IT:Interaction)
29    [label BUFFER-recevingAndDo-buffer@get] .
30   rl < O:Oid : BUFFER | > send(O:Oid,buffer@put,IT:Interaction)
31     => < O:Oid : BUFFER | > do(O:Oid,buffer@put,IT:Interaction)
32    [label BUFFER-recevingAndDo-buffer@put] .
33 endom
```

Conector DEFAULT

```
1  omod DEFAULT is
2    including CBABEL-CONFIGURATION .
3
4    class DEFAULT .
```

```
  op default@in : -> PortInId   [ctor] .
  op default@out : -> PortOutId [ctor] .

  eq instantiate(O:Oid,DEFAULT)
    = < O:Oid : DEFAULT | none > .

  rl ack([O:Oid,default@out]:: V#0:Interaction) < O:Oid : DEFAULT | >
    => < O:Oid : DEFAULT | > ack(V#0:Interaction) [label DEFAULT-acking] .
  rl send(O:Oid,default@in,V#0:Interaction) < O:Oid : DEFAULT | >
    => < O:Oid : DEFAULT | > send(O:Oid,default@out,[O:Oid,default@out]:: V#0:Interaction)
   [label DEFAULT-sending] .
endom
```

Conector MUTEX

```
omod MUTEX is
  including CBABEL-CONFIGURATION .

  class MUTEX | status : PortStatus .
  op mutex@in1 : -> PortInId  [ctor] .
  op mutex@in2 : -> PortInId  [ctor] .
  op mutex@out1 : -> PortOutId [ctor] .
  op mutex@out2 : -> PortOutId [ctor] .

  eq instantiate(O:Oid,MUTEX)
    = < O:Oid : MUTEX | none,status : unlocked > .

  rl < O:Oid : MUTEX | status : locked > ack([O:Oid,mutex@out1]:: IT:Interaction)
    => < O:Oid : MUTEX | status : unlocked > ack(IT:Interaction)
   [label MUTEX-acking-mutex@out1] .
  rl < O:Oid : MUTEX | status : locked > ack([O:Oid,mutex@out2]:: IT:Interaction)
    => < O:Oid : MUTEX | status : unlocked > ack(IT:Interaction)
   [label MUTEX-acking-mutex@out2] .
  rl < O:Oid : MUTEX | status : unlocked > send(O:Oid,mutex@in1,IT:Interaction)
    => < O:Oid : MUTEX | status : locked > send(O:Oid,mutex@out1,[O:Oid,mutex@out1]:: IT:Interaction)
   [label MUTEX-sending-mutex@in1] .
  rl < O:Oid : MUTEX | status : unlocked > send(O:Oid,mutex@in2,IT:Interaction)
    => < O:Oid : MUTEX | status : locked > send(O:Oid,mutex@out2,[O:Oid,mutex@out2]:: IT:Interaction)
   [label MUTEX-sending-mutex@in2] .
endom
```

Conector GUARD-GET

```
omod GUARD-GET is
  including CBABEL-CONFIGURATION .

  class GUARD-GET | empty : Int, gg@full : StateRequired .
  op get-empty : Object -> Int .
  op get-gg@full : Object -> Int .
  op set-empty : Object Int -> Object .
  op set-gg@full : Object Int -> Object .
  op gg@in : -> PortInId   [ctor] .
  op gg@out : -> PortOutId [ctor] .

  eq get-empty(< O:Oid : GUARD-GET | empty : V:Int >)
```

```
13      = V:Int .
14    eq get-gg@full(< O:Oid : GUARD-GET | gg@full : st(V:Int,S:Status)>)
15      = V:Int .
16    eq set-empty(< O:Oid : GUARD-GET | empty : V:Int >,V':Int)
17      = < O:Oid : GUARD-GET | empty : V':Int > .
18    eq set-gg@full(< O:Oid : GUARD-GET | gg@full : st(V:Int,S:Status)>,V':Int)
19      = < O:Oid : GUARD-GET | gg@full : st(V':Int,changed)> .
20    eq instantiate(O:Oid,GUARD-GET)
21      = < O:Oid : GUARD-GET |(none,empty : 2),gg@full : st(0,unchanged)> .
22
23    ceq after(OBJ:Object)
24      = set-empty(OBJ:Object,get-empty(OBJ:Object)+ 1)
25      if class(OBJ:Object)= GUARD-GET .
26    ceq before(OBJ:Object)
27      = set-gg@full(OBJ:Object,get-gg@full(OBJ:Object)- 1)
28      if class(OBJ:Object)= GUARD-GET .
29    ceq open?(OBJ:Object)
30      = get-gg@full(OBJ:Object)> 0
31      if class(OBJ:Object)= GUARD-GET .
32
33    rl < O:Oid : GUARD-GET | > ack([O:Oid,gg@out]:: IT:Interaction)
34      => after(< O:Oid : GUARD-GET | >) ack(IT:Interaction)
35     [label GUARD-GET-acking-gg@out] .
36    crl < O:Oid : GUARD-GET | > send(O:Oid,gg@in,IT:Interaction)
37      => before(< O:Oid : GUARD-GET | >) send(O:Oid,gg@out,[O:Oid,gg@out]:: IT:Interaction)
38      if open?(< O:Oid : GUARD-GET | >)= true
39     [label GUARD-GET-sending-gg@in] .
40  endom
```

Conector GUARD-PUT

```
1   omod GUARD-PUT is
2     including CBABEL-CONFIGURATION .
3
4     class GUARD-PUT | full : Int, gp@empty : StateRequired .
5     op get-full : Object -> Int .
6     op get-gp@empty : Object -> Int .
7     op set-full : Object Int -> Object .
8     op set-gp@empty : Object Int -> Object .
9     op gp@in : -> PortInId   [ctor] .
10    op gp@out : -> PortOutId [ctor] .
11
12    eq get-full(< O:Oid : GUARD-PUT | full : V:Int >)
13      = V:Int .
14    eq get-gp@empty(< O:Oid : GUARD-PUT | gp@empty : st(V:Int,S:Status)>)
15      = V:Int .
16    eq set-full(< O:Oid : GUARD-PUT | full : V:Int >,V':Int)
17      = < O:Oid : GUARD-PUT | full : V':Int > .
18    eq set-gp@empty(< O:Oid : GUARD-PUT | gp@empty : st(V:Int,S:Status)>,V':Int)
19      = < O:Oid : GUARD-PUT | gp@empty : st(V':Int,changed)> .
20    eq instantiate(O:Oid,GUARD-PUT)
21      = < O:Oid : GUARD-PUT |(none,full : 0),gp@empty : st(0,unchanged)> .
22
23    ceq after(OBJ:Object)
24      = set-full(OBJ:Object,get-full(OBJ:Object)+ 1)
```

```
25      if class(OBJ:Object)= GUARD-PUT .
26    ceq before(OBJ:Object)
27      = set-gp@empty(OBJ:Object,get-gp@empty(OBJ:Object)- 1)
28      if class(OBJ:Object)= GUARD-PUT .
29    ceq open?(OBJ:Object)
30      = get-gp@empty(OBJ:Object)> 0
31      if class(OBJ:Object)= GUARD-PUT .
32
33    rl < O:Oid : GUARD-PUT | > ack([O:Oid,gp@out]:: IT:Interaction)
34      => after(< O:Oid : GUARD-PUT | >) ack(IT:Interaction)
35     [label GUARD-PUT-acking-gp@out] .
36    crl < O:Oid : GUARD-PUT | > send(O:Oid,gp@in,IT:Interaction)
37      => before(< O:Oid : GUARD-PUT | >) send(O:Oid,gp@out,[O:Oid,gp@out]:: IT:Interaction)
38      if open?(< O:Oid : GUARD-PUT | >)= true
39     [label GUARD-PUT-sending-gp@in] .
40  endom
```

Módulo aplicação da arquitetura PC-DEFAULT

```
1   omod PC-DEFAULT is
2     including CBABEL-CONFIGURATION .
3     including BUFFER .
4     including CONSUMER .
5     including DEFAULT .
6     including PRODUCER .
7     op buff : -> Oid .
8     op cons1 : -> Oid .
9     op cons2 : -> Oid .
10    op default1 : -> Oid .
11    op default2 : -> Oid .
12    op prod1 : -> Oid .
13    op prod2 : -> Oid .
14    op topology : -> Configuration .
15    eq topology =
16       instantiate(buff,BUFFER) instantiate(cons1,CONSUMER)
17       instantiate(cons2,CONSUMER) instantiate(default1,DEFAULT)
18       instantiate(default2,DEFAULT) instantiate(prod1,PRODUCER)
19       instantiate(prod2,PRODUCER) .
20
21    eq send(cons1,consumer@get,IT:Interaction) = send(default2,default@in,IT:Interaction)
22     [label consumer@get-linking-default@in] .
23    eq send(cons2,consumer@get,IT:Interaction) = send(default2,default@in,IT:Interaction)
24     [label consumer@get-linking-default@in] .
25    eq send(default1,default@out,IT:Interaction) = send(buff,buffer@put,IT:Interaction)
26     [label default@out-linking-buffer@put] .
27    eq send(default2,default@out,IT:Interaction) = send(buff,buffer@get,IT:Interaction)
28     [label default@out-linking-buffer@get] .
29    eq send(prod1,producer@put,IT:Interaction) = send(default1,default@in,IT:Interaction)
30     [label producer@put-linking-default@in] .
31    eq send(prod2,producer@put,IT:Interaction) = send(default1,default@in,IT:Interaction)
32     [label producer@put-linking-default@in] .
33  endom
```

Módulo aplicação da arquitetura PC-MUTEX

```
omod PC-MUTEX is
  including CBABEL-CONFIGURATION .
  including BUFFER .
  including CONSUMER .
  including MUTEX .
  including PRODUCER .

  op buff : -> Oid .
  op cons1 : -> Oid .
  op cons2 : -> Oid .
  op mutx : -> Oid .
  op prod1 : -> Oid .
  op prod2 : -> Oid .
  op topology : -> Configuration .
  eq topology =
     instantiate(buff,BUFFER) instantiate(cons1,CONSUMER)
     instantiate(cons2,CONSUMER) instantiate(mutx,MUTEX)
     instantiate(prod1,PRODUCER) instantiate(prod2,PRODUCER) .

  eq send(cons1,consumer@get,IT:Interaction) = send(mutx,mutex@in2,IT:Interaction)
   [label consumer@get-linking-mutex@in2] .
  eq send(cons2,consumer@get,IT:Interaction) = send(mutx,mutex@in2,IT:Interaction)
   [label consumer@get-linking-mutex@in2] .
  eq send(mutx,mutex@out1,IT:Interaction) = send(buff,buffer@put,IT:Interaction)
   [label mutex@out1-linking-buffer@put] .
  eq send(mutx,mutex@out2,IT:Interaction) = send(buff,buffer@get,IT:Interaction)
   [label mutex@out2-linking-buffer@get] .
  eq send(prod1,producer@put,IT:Interaction) = send(mutx,mutex@in1,IT:Interaction)
   [label producer@put-linking-mutex@in1] .
  eq send(prod2,producer@put,IT:Interaction) = send(mutx,mutex@in1,IT:Interaction)
   [label producer@put-linking-mutex@in1] .
endom
```

Módulo aplicação da arquitetura PC-GUARDS-MUTEX

```
omod PC-GUARDS-MUTEX is
  including CBABEL-CONFIGURATION .
  including BUFFER .
  including CONSUMER .
  including GUARD-GET .
  including GUARD-PUT .
  including MUTEX .
  including PRODUCER .
  op buff : -> Oid .
  op cons1 : -> Oid .
  op cons2 : -> Oid .
  op gget : -> Oid .
  op gput : -> Oid .
  op mutx : -> Oid .
  op prod1 : -> Oid .
  op prod2 : -> Oid .
  op topology : -> Configuration .
  eq topology =
     instantiate(buff,BUFFER) instantiate(cons1,CONSUMER)
     instantiate(cons2,CONSUMER) instantiate(gget,GUARD-GET)
```

```
21      instantiate(gput,GUARD-PUT) instantiate(mutx,MUTEX)
22      instantiate(prod1,PRODUCER) instantiate(prod2,PRODUCER) .
23
24    eq < gget : GUARD-GET | gg@full : st(V1:Int,changed)>
25       < gput : GUARD-PUT | full : V2:Int > =
26       < gget : GUARD-GET | gg@full : st(V1:Int,unchanged)>
27       < gput : GUARD-PUT | full : V1:Int > .
28
29    eq < gput : GUARD-PUT | gp@empty : st(V1:Int,changed)>
30       < gget : GUARD-GET | empty : V2:Int > =
31       < gput : GUARD-PUT | gp@empty : st(V1:Int,unchanged)>
32       < gget : GUARD-GET | empty : V1:Int > .
33
34    ceq < gget : GUARD-GET | gg@full : st(V1:Int,unchanged)>
35        < gput : GUARD-PUT | full : V2:Int > =
36        < gget : GUARD-GET | gg@full : st(V2:Int,unchanged)>
37        < gput : GUARD-PUT | full : V2:Int >
38     if V1:Int =/= V2:Int = true .
39
40    ceq < gput : GUARD-PUT | gp@empty : st(V1:Int,unchanged)>
41        < gget : GUARD-GET | empty : V2:Int > =
42        < gput : GUARD-PUT | gp@empty : st(V2:Int,unchanged)>
43        < gget : GUARD-GET | empty : V2:Int >
44      if V1:Int =/= V2:Int = true .
45
46    eq send(cons1,consumer@get,IT:Interaction) = send(gget,gg@in,IT:Interaction)
47     [label consumer@get-linking-gg@in] .
48    eq send(cons2,consumer@get,IT:Interaction) = send(gget,gg@in,IT:Interaction)
49     [label consumer@get-linking-gg@in] .
50    eq send(gget,gg@out,IT:Interaction)  = send(mutx,mutex@in2,IT:Interaction)
51     [label gg@out-linking-mutex@in2] .
52    eq send(gput,gp@out,IT:Interaction)  = send(mutx,mutex@in1,IT:Interaction)
53     [label gp@out-linking-mutex@in1] .
54    eq send(mutx,mutex@out1,IT:Interaction) = send(buff,buffer@put,IT:Interaction)
55     [label mutex@out1-linking-buffer@put] .
56    eq send(mutx,mutex@out2,IT:Interaction) = send(buff,buffer@get,IT:Interaction)
57     [label mutex@out2-linking-buffer@get] .
58    eq send(prod1,producer@put,IT:Interaction) = send(gput,gp@in,IT:Interaction)
59     [label producer@put-linking-gp@in] .
60    eq send(prod2,producer@put,IT:Interaction) = send(gput,gp@in,IT:Interaction)
61     [label producer@put-linking-gp@in] .
62  endom
```

B.3 Leitores e Escritores

Os componentes da arquitetura READERS-WRITERS (Figura 13), quando carregados em Maude CBabel tool, produzem os seguintes módulos orientados a objetos:

Módulo READER

```
1  omod READER is
2    including CBABEL-CONFIGURATION .
```

```
3
4     class READER | r@read-status : PortStatus .
5     op r@read : -> PortOutId [ctor] .
6     eq instantiate(O:Oid,READER) = < O:Oid : READER | none,r@read-status : unlocked > .
7
8     rl ack([O:Oid,r@read]) < O:Oid : READER | r@read-status : locked >
9       => done(O:Oid,r@read,none) < O:Oid : READER | r@read-status : unlocked >
10     [label READER-receivingAck-r@read] .
11    rl do(O:Oid,r@read,none) < O:Oid : READER | r@read-status : unlocked >
12      => send(O:Oid,r@read,[O:Oid,r@read]) < O:Oid : READER | r@read-status : locked >
13     [label READER-sending-r@read] .
14  endom
```

Módulo WRITER

```
1   omod WRITER is
2     including CBABEL-CONFIGURATION .
3
4     class WRITER | w@write-status : PortStatus .
5     op w@write : -> PortOutId [ctor] .
6     eq instantiate(O:Oid,WRITER) = < O:Oid : WRITER | none,w@write-status : unlocked > .
7
8     rl ack([O:Oid,w@write]) < O:Oid : WRITER | w@write-status : locked >
9       => done(O:Oid,w@write,none) < O:Oid : WRITER | w@write-status : unlocked >
10     [label WRITER-receivingAck-w@write] .
11    rl do(O:Oid,w@write,none) < O:Oid : WRITER | w@write-status : unlocked >
12      => send(O:Oid,w@write,[O:Oid,w@write]) < O:Oid : WRITER | w@write-status : locked >
13     [label WRITER-sending-w@write] .
14  endom
```

Módulo BUFFER

```
1   omod BUFFER is
2     including CBABEL-CONFIGURATION .
3
4     class BUFFER .
5     op buffer@read : -> PortInId  [ctor] .
6     op buffer@write : -> PortInId [ctor] .
7
8     eq instantiate(O:Oid,BUFFER) = < O:Oid : BUFFER | none > .
9
10    rl < O:Oid : BUFFER | > done(O:Oid,buffer@read,IT:Interaction)
11      => < O:Oid : BUFFER | > ack(IT:Interaction)
12     [label BUFFER-doneAndAcking-buffer@read] .
13    rl < O:Oid : BUFFER | > done(O:Oid,buffer@write,IT:Interaction)
14      => < O:Oid : BUFFER | > ack(IT:Interaction)
15     [label BUFFER-doneAndAcking-buffer@write] .
16    rl < O:Oid : BUFFER | > send(O:Oid,buffer@read,IT:Interaction)
17      => < O:Oid : BUFFER | > do(O:Oid,buffer@read,IT:Interaction)
18     [label BUFFER-recevingAndDo-buffer@read] .
19    rl < O:Oid : BUFFER | > send(O:Oid,buffer@write,IT:Interaction)
20      => < O:Oid : BUFFER | > do(O:Oid,buffer@write,IT:Interaction)
21     [label BUFFER-recevingAndDo-buffer@write] .
22  endom
```

Conector WANT-WRITE

```
omod WANT-WRITE is
  including CBABEL-CONFIGURATION .

  class WANT-WRITE | want-write : Int .
  op get-want-write : Object -> Int .
  op set-want-write : Object Int -> Object .
  op in-want-write : -> PortInId   [ctor] .
  op out-want-write : -> PortOutId [ctor] .

  eq get-want-write(< O:Oid : WANT-WRITE | want-write : V:Int >) = V:Int .
  eq set-want-write(< O:Oid : WANT-WRITE | want-write : V:Int >,V':Int)
    = < O:Oid : WANT-WRITE | want-write : V':Int > .
  eq instantiate(O:Oid,WANT-WRITE)
    = < O:Oid : WANT-WRITE | none,want-write : 0 > .

  ceq after(OBJ:Object)
    = OBJ:Object
    if class(OBJ:Object)= WANT-WRITE .
  ceq before(OBJ:Object)
    = set-want-write(OBJ:Object,get-want-write(OBJ:Object)+ 1)
    if class(OBJ:Object)= WANT-WRITE .
  ceq open?(OBJ:Object)
    = true
    if class(OBJ:Object)= WANT-WRITE .

  rl < O:Oid : WANT-WRITE | > ack([O:Oid,out-want-write]:: IT:Interaction)
    => after(< O:Oid : WANT-WRITE | >) ack(IT:Interaction)
   [label WANT-WRITE-acking-out-want-write] .
  crl < O:Oid : WANT-WRITE | > send(O:Oid,in-want-write,IT:Interaction)
    => before(< O:Oid : WANT-WRITE | >) send(O:Oid,out-want-write,[O:Oid,out-want-write]:: IT:Interaction)
    if open?(< O:Oid : WANT-WRITE | >)= true
   [label WANT-WRITE-sending-in-want-write] .
endom
```

Conector WANT-READ

```
omod WANT-READ is
  including CBABEL-CONFIGURATION .

  class WANT-READ | want-read : Int .
  op get-want-read : Object -> Int .
  op set-want-read : Object Int -> Object .
  op in-want-read : -> PortInId    [ctor] .
  op out-want-read : -> PortOutId  [ctor] .

  eq get-want-read(< O:Oid : WANT-READ | want-read : V:Int >) = V:Int .
  eq set-want-read(< O:Oid : WANT-READ | want-read : V:Int >,V':Int)
    = < O:Oid : WANT-READ | want-read : V':Int > .
  eq instantiate(O:Oid,WANT-READ)
    = < O:Oid : WANT-READ | none,want-read : 0 > .

  ceq after(OBJ:Object)
    = OBJ:Object
    if class(OBJ:Object)= WANT-READ .
```

```
19    ceq before(OBJ:Object)
20      = set-want-read(OBJ:Object,get-want-read(OBJ:Object)+ 1)
21      if class(OBJ:Object)= WANT-READ .
22    ceq open?(OBJ:Object)
23      = true
24      if class(OBJ:Object)= WANT-READ .
25
26    rl < O:Oid : WANT-READ | > ack([O:Oid,out-want-read]:: IT:Interaction)
27      => after(< O:Oid : WANT-READ | >) ack(IT:Interaction)
28     [label WANT-READ-acking-out-want-read] .
29    crl < O:Oid : WANT-READ | > send(O:Oid,in-want-read,IT:Interaction)
30      => before(< O:Oid : WANT-READ | >) send(O:Oid,out-want-read,[O:Oid,out-want-read]:: IT:Interaction)
31      if open?(< O:Oid : WANT-READ | >)= true
32     [label WANT-READ-sending-in-want-read] .
33  endom
```

Conector COUNT-WRITE

```
1   omod COUNT-WRITE is
2     including CBABEL-CONFIGURATION .
3
4     class COUNT-WRITE | cw@readers : StateRequired, cw@turn : StateRequired,
5                         cw@want-read : StateRequired, cw@want-write : StateRequired,
6                         writing : Bool .
7
8     op get-cw@readers : Object -> Int .
9     op get-cw@turn : Object -> Int .
10    op get-cw@want-read : Object -> Int .
11    op get-cw@want-write : Object -> Int .
12    op get-writing : Object -> Bool .
13    op set-cw@readers : Object Int -> Object .
14    op set-cw@turn : Object Int -> Object .
15    op set-cw@want-read : Object Int -> Object .
16    op set-cw@want-write : Object Int -> Object .
17    op set-writing : Object Bool -> Object .
18
19    op in-count-write : -> PortInId   [ctor] .
20    op out-count-write : -> PortOutId [ctor] .
21
22    eq get-cw@readers(< O:Oid : COUNT-WRITE | cw@readers : st(V:Int,S:Status)>)
23      = V:Int .
24    eq get-cw@turn(< O:Oid : COUNT-WRITE | cw@turn : st(V:Int,S:Status)>)
25      = V:Int .
26    eq get-cw@want-read(< O:Oid : COUNT-WRITE | cw@want-read : st(V:Int,S:Status)>)
27      = V:Int .
28    eq get-cw@want-write(< O:Oid : COUNT-WRITE | cw@want-write : st(V:Int,S:Status)>)
29      = V:Int .
30    eq get-writing(< O:Oid : COUNT-WRITE | writing : V:Bool >) = V:Bool .
31    eq set-cw@readers(< O:Oid : COUNT-WRITE | cw@readers : st(V:Int,S:Status)>,V':Int)
32      = < O:Oid : COUNT-WRITE | cw@readers : st(V':Int,changed)> .
33    eq set-cw@turn(< O:Oid : COUNT-WRITE | cw@turn : st(V:Int,S:Status)>,V':Int)
34      = < O:Oid : COUNT-WRITE | cw@turn : st(V':Int,changed)> .
35    eq set-cw@want-read(< O:Oid : COUNT-WRITE | cw@want-read : st(V:Int,
36      S:Status)>,V':Int)
37      = < O:Oid : COUNT-WRITE | cw@want-read : st(V':Int,changed)> .
```

```
38    eq set-cw@want-write(< O:Oid : COUNT-WRITE | cw@want-write : st(V:Int,
39      S:Status)>,V':Int)
40      = < O:Oid : COUNT-WRITE | cw@want-write : st(V':Int,changed)> .
41    eq set-writing(< O:Oid : COUNT-WRITE | writing : V:Bool >,V':Bool)
42      = < O:Oid : COUNT-WRITE | writing : V':Bool > .
43
44    eq instantiate(O:Oid,COUNT-WRITE)
45      = < O:Oid : COUNT-WRITE | writing : false ,
46          cw@readers : st(0,unchanged) , cw@turn : st(0,unchanged) ,
47          cw@want-read : st(0,unchanged) , cw@want-write : st(0,unchanged) > .
48
49    ceq after(OBJ:Object)
50      = set-cw@turn(set-writing(OBJ:Object,false),0)
51      if class(OBJ:Object)= COUNT-WRITE .
52    ceq before(OBJ:Object)
53      = set-writing(set-cw@want-write(OBJ:Object,
54         get-cw@want-write(OBJ:Object)- 1),true)
55      if class(OBJ:Object)= COUNT-WRITE .
56    ceq open?(OBJ:Object)
57      = (get-cw@readers(OBJ:Object)== 0 and get-writing(OBJ:Object)== false) and
58        (get-cw@want-read(OBJ:Object)== 0 or get-cw@turn(OBJ:Object)== 1)
59      if class(OBJ:Object)= COUNT-WRITE .
60
61    rl < O:Oid : COUNT-WRITE | > ack([O:Oid,out-count-write]:: IT:Interaction)
62      => after(< O:Oid : COUNT-WRITE | >) ack(IT:Interaction)
63     [label COUNT-WRITE-acking-out-count-write] .
64    crl < O:Oid : COUNT-WRITE | > send(O:Oid,in-count-write,IT:Interaction)
65      => before(< O:Oid : COUNT-WRITE | >)
66         send(O:Oid,out-count-write,[O:Oid,out-count-write]:: IT:Interaction)
67      if open?(< O:Oid : COUNT-WRITE | >)= true
68     [label COUNT-WRITE-sending-in-count-write] .
69  endom
```

Conector COUNT-READ

```
1   omod COUNT-READ is
2     including CBABEL-CONFIGURATION .
3     class COUNT-READ | cr@want-read : StateRequired,
4                        cr@want-write : StateRequired, cr@writing : StateRequired,
5                        readers : Int, turn : Int .
6
7     op get-cr@want-read : Object -> Int .
8     op get-cr@want-write : Object -> Int .
9     op get-cr@writing : Object -> Bool .
10    op get-readers : Object -> Int .
11    op get-turn : Object -> Int .
12    op set-cr@want-read : Object Int -> Object .
13    op set-cr@want-write : Object Int -> Object .
14    op set-cr@writing : Object Bool -> Object .
15    op set-readers : Object Int -> Object .
16    op set-turn : Object Int -> Object .
17
18    op in-count-read : -> PortInId   [ctor] .
19    op out-count-read : -> PortOutId [ctor] .
20
```

```
21    eq get-cr@want-read(< O:Oid : COUNT-READ | cr@want-read : st(V:Int,S:Status)>)
22      = V:Int .
23    eq get-cr@want-write(< O:Oid : COUNT-READ | cr@want-write : st(V:Int,S:Status)>)
24      = V:Int .
25    eq get-cr@writing(< O:Oid : COUNT-READ | cr@writing : st(V:Bool,S:Status)>)
26      = V:Bool .
27    eq get-readers(< O:Oid : COUNT-READ | readers : V:Int >)
28      = V:Int .
29    eq get-turn(< O:Oid : COUNT-READ | turn : V:Int >)
30      = V:Int .
31    eq set-cr@want-read(< O:Oid : COUNT-READ | cr@want-read : st(V:Int,S:Status)>,V':Int)
32      = < O:Oid : COUNT-READ | cr@want-read : st(V':Int,changed)> .
33    eq set-cr@want-write(< O:Oid : COUNT-READ | cr@want-write : st(V:Int,S:Status)>,V':Int)
34      = < O:Oid : COUNT-READ | cr@want-write : st(V':Int,changed)> .
35    eq set-cr@writing(< O:Oid : COUNT-READ | cr@writing : st(V:Bool,S:Status)>,V':Bool)
36      = < O:Oid : COUNT-READ | cr@writing : st(V':Bool,changed)> .
37    eq set-readers(< O:Oid : COUNT-READ | readers : V:Int >,V':Int)
38      = < O:Oid : COUNT-READ | readers : V':Int > .
39    eq set-turn(< O:Oid : COUNT-READ | turn : V:Int >,V':Int)
40      = < O:Oid : COUNT-READ | turn : V':Int > .
41    eq instantiate(O:Oid,COUNT-READ)
42      = < O:Oid : COUNT-READ | readers : 0 , turn : 0 ,
43                               cr@want-read : st(0,unchanged) ,
44                               cr@want-write : st(0,unchanged) ,
45                               cr@writing : st(true,unchanged) > .
46
47    ceq after(OBJ:Object)
48      = set-readers(OBJ:Object,get-readers(OBJ:Object)- 1)
49      if class(OBJ:Object)= COUNT-READ .
50    ceq before(OBJ:Object)
51      = if get-cr@want-read(set-readers(set-cr@want-read(OBJ:Object,
52            get-cr@want-read(OBJ:Object)- 1),get-readers(set-cr@want-read(OBJ:Object,
53             get-cr@want-read(OBJ:Object)- 1))+ 1))== 0
54      then
55       set-turn(set-readers(set-cr@want-read(OBJ:Object,
56        get-cr@want-read(OBJ:Object) - 1),get-readers(
57         set-cr@want-read(OBJ:Object,get-cr@want-read(OBJ:Object)- 1)) + 1),1)
58      else
59       set-readers(set-cr@want-read(OBJ:Object,get-cr@want-read(OBJ:Object)- 1),
60        get-readers(set-cr@want-read(OBJ:Object,get-cr@want-read(OBJ:Object)- 1))+1)
61      fi
62      if class(OBJ:Object)= COUNT-READ .
63    ceq open?(OBJ:Object)
64      = get-cr@writing(OBJ:Object)== false and(get-cr@want-write(OBJ:Object)== 0
65        or get-turn(OBJ:Object)== 0)
66     if class(OBJ:Object)= COUNT-READ .
67
68    rl < O:Oid : COUNT-READ | > ack([O:Oid,out-count-read]:: IT:Interaction)
69      => after(< O:Oid : COUNT-READ | >) ack(IT:Interaction)
70     [label COUNT-READ-acking-out-count-read] .
71    crl < O:Oid : COUNT-READ | > send(O:Oid,in-count-read,IT:Interaction)
72      => before(< O:Oid : COUNT-READ | >)
73         send(O:Oid,out-count-read,[O:Oid,out-count-read]:: IT:Interaction)
74      if open?(< O:Oid : COUNT-READ | >)= true
```

```
75     [label COUNT-READ-sending-in-count-read] .
76 endom
```

Módulo de aplicação da arquitetura READERS-WRITERS

```
1  omod READERS-WRITERS is
2    including CBABEL-CONFIGURATION .
3    including BUFFER .
4    including COUNT-READ .
5    including COUNT-WRITE .
6    including READER .
7    including WANT-READ .
8    including WRITER .
9    including WANT-WRITE .
10   op buff : -> Oid .
11   op cr : -> Oid .
12   op cw : -> Oid .
13   op reader1 : -> Oid .
14   op reader2 : -> Oid .
15   op topology : -> Configuration .
16   op wr : -> Oid .
17   op writer1 : -> Oid .
18   op writer2 : -> Oid .
19   op ww : -> Oid .
20   eq topology
21     = instantiate(buff,BUFFER) instantiate(cr,COUNT-READ)
22       instantiate(cw,COUNT-WRITE) instantiate(reader1,READER)
23       instantiate(reader2,READER) instantiate(wr,WANT-READ)
24       instantiate(writer1,WRITER) instantiate(writer2,WRITER)
25       instantiate(ww,WANT-WRITE) .
26
27   eq < cr : COUNT-READ | cr@want-read : st(V1:Int,changed)>
28      < wr : WANT-READ | want-read : V2:Int > =
29      < cr : COUNT-READ | cr@want-read : st(V1:Int,unchanged)>
30      < wr : WANT-READ | want-read : V1:Int > .
31
32   eq < cr : COUNT-READ | cr@want-write : st(V1:Int,changed)>
33      < ww : WANT-WRITE | want-write : V2:Int > =
34      < cr : COUNT-READ | cr@want-write : st(V1:Int,unchanged)>
35      < ww : WANT-WRITE | want-write : V1:Int > .
36
37   eq < cr : COUNT-READ | cr@writing : st(V1:Bool,changed)>
38      < cw : COUNT-WRITE | writing : V2:Bool > =
39      < cr : COUNT-READ | cr@writing : st(V1:Bool,unchanged)>
40      < cw : COUNT-WRITE | writing : V1:Bool > .
41
42   eq < cw : COUNT-WRITE | cw@readers : st(V1:Int,changed)>
43      < cr : COUNT-READ | readers : V2:Int > =
44      < cw : COUNT-WRITE | cw@readers : st(V1:Int,unchanged)>
45      < cr : COUNT-READ | readers : V1:Int > .
46
47   eq < cw : COUNT-WRITE | cw@turn : st(V1:Int,changed)>
48      < cr : COUNT-READ | turn : V2:Int > =
49      < cw : COUNT-WRITE | cw@turn : st(V1:Int,unchanged)>
50      < cr : COUNT-READ | turn : V1:Int > .
```

```
51
52    eq < cw : COUNT-WRITE | cw@want-read : st(V1:Int,changed)>
53       < wr : WANT-READ | want-read : V2:Int > =
54       < cw : COUNT-WRITE | cw@want-read : st(V1:Int,unchanged)>
55       < wr : WANT-READ | want-read : V1:Int > .
56
57    eq < cw : COUNT-WRITE | cw@want-write : st(V1:Int,changed)>
58       < ww : WANT-WRITE | want-write : V2:Int > =
59       < cw : COUNT-WRITE | cw@want-write : st(V1:Int,unchanged)>
60       < ww : WANT-WRITE | want-write : V1:Int > .
61
62    ceq < cr : COUNT-READ | cr@want-read : st(V1:Int,unchanged)>
63        < wr : WANT-READ | want-read : V2:Int > =
64        < cr : COUNT-READ | cr@want-read : st(V2:Int,unchanged)>
65        < wr : WANT-READ | want-read : V2:Int >
66      if V1:Int =/= V2:Int = true .
67
68    ceq < cr : COUNT-READ | cr@want-write : st(V1:Int,unchanged)>
69        < ww : WANT-WRITE | want-write : V2:Int > =
70        < cr : COUNT-READ | cr@want-write : st(V2:Int,unchanged)>
71        < ww : WANT-WRITE | want-write : V2:Int >
72      if V1:Int =/= V2:Int = true .
73
74    ceq < cr : COUNT-READ | cr@writing : st(V1:Bool,unchanged)>
75        < cw : COUNT-WRITE | writing : V2:Bool > =
76        < cr : COUNT-READ | cr@writing : st(V2:Bool,unchanged)>
77        < cw : COUNT-WRITE | writing : V2:Bool >
78      if V1:Bool =/= V2:Bool = true .
79
80    ceq < cw : COUNT-WRITE | cw@readers : st(V1:Int,unchanged)>
81        < cr : COUNT-READ | readers : V2:Int > =
82        < cw : COUNT-WRITE | cw@readers : st(V2:Int,unchanged)>
83        < cr : COUNT-READ | readers : V2:Int >
84      if V1:Int =/= V2:Int = true .
85
86    ceq < cw : COUNT-WRITE | cw@turn : st(V1:Int,unchanged)>
87        < cr : COUNT-READ | turn : V2:Int > =
88        < cw : COUNT-WRITE | cw@turn : st(V2:Int,unchanged)>
89        < cr : COUNT-READ | turn : V2:Int >
90      if V1:Int =/= V2:Int = true .
91
92    ceq < cw : COUNT-WRITE | cw@want-read : st(V1:Int,unchanged)>
93        < wr : WANT-READ | want-read : V2:Int > =
94        < cw : COUNT-WRITE | cw@want-read : st(V2:Int,unchanged)>
95        < wr : WANT-READ | want-read : V2:Int >
96      if V1:Int =/= V2:Int = true .
97
98    ceq < cw : COUNT-WRITE | cw@want-write : st(V1:Int,unchanged)>
99        < ww : WANT-WRITE | want-write : V2:Int > =
100       < cw : COUNT-WRITE | cw@want-write : st(V2:Int,unchanged)>
101       < ww : WANT-WRITE | want-write : V2:Int >
102     if V1:Int =/= V2:Int = true .
103
104   eq send(cr,out-count-read,IT:Interaction)  = send(buff,buffer@read,IT:Interaction)
```

```
105     [label out-count-read-linking-buffer@read] .
106   eq send(cw,out-count-write,IT:Interaction) = send(buff,buffer@write,IT:Interaction)
107     [label out-count-write-linking-buffer@write] .
108   eq send(reader1,r@read,IT:Interaction) = send(wr,in-want-read,IT:Interaction)
109     [label r@read-linking-in-want-read]
110      .
111   eq send(reader2,r@read,IT:Interaction) = send(wr,in-want-read,IT:Interaction)
112     [label r@read-linking-in-want-read]
113      .
114   eq send(wr,out-want-read,IT:Interaction) = send(cr,in-count-read,IT:Interaction)
115     [label out-want-read-linking-in-count-read] .
116   eq send(writer1,w@write,IT:Interaction) = send(ww,in-want-write,IT:Interaction)
117     [label w@write-linking-in-want-write] .
118   eq send(writer2,w@write,IT:Interaction) = send(ww,in-want-write,IT:Interaction)
119     [label w@write-linking-in-want-write] .
120   eq send(ww,out-want-write,IT:Interaction) = send(cw,in-count-write,IT:Interaction)
121     [label out-want-write-linking-in-count-write] .
122 endom
```

B.4 Ceia de Filósofos

Os componentes CBabel da arquitetura `4-PHILOSOPHERS` (Figura 16), quando carregados em Maude CBabel tool, produzem os seguintes módulos orientados a objetos de Full Maude:

Módulo TABLE

```
1  omod TABLE is
2    including CBABEL-CONFIGURATION .
3
4    class TABLE | places : Int .
5    op get-places : Object -> Int .
6    op set-places : Object Int -> Object .
7
8    eq get-places(< O:Oid : TABLE | places : V:Int >) = V:Int .
9    eq set-places(< O:Oid : TABLE | places : V:Int >,V':Int)
10     = < O:Oid : TABLE | places : V':Int > .
11
12   eq instantiate(O:Oid,TABLE)
13     = < O:Oid : TABLE | none,places : 3 > .
14 endom
```

Módulo FORK

```
1  omod FORK is
2    including CBABEL-CONFIGURATION .
3
4    class FORK | available : Bool .
5    op get-available : Object -> Bool .
6    op set-available : Object Bool -> Object .
7
```

```
8     eq get-available(< O:Oid : FORK | available : V:Bool >)
9       = V:Bool .
10    eq set-available(< O:Oid : FORK | available : V:Bool >,V':Bool)
11      = < O:Oid : FORK | available : V':Bool > .
12
13    eq instantiate(O:Oid,FORK)
14      = < O:Oid : FORK | none,available : true > .
15  endom
```

Módulo PHILOSOPHER

```
1   omod PHILOSOPHER is
2     including CBABEL-CONFIGURATION .
3
4     class PHILOSOPHER | phi@eat-status : PortStatus .
5     op phi@eat : -> PortOutId  [ctor] .
6     eq instantiate(O:Oid,PHILOSOPHER)
7       = < O:Oid : PHILOSOPHER | none,phi@eat-status : unlocked > .
8
9     rl ack([O:Oid,phi@eat]) < O:Oid : PHILOSOPHER | phi@eat-status : locked >
10      => done(O:Oid,phi@eat,none) < O:Oid : PHILOSOPHER | phi@eat-status : unlocked >
11     [label PHILOSOPHER-receivingAck-phi@eat] .
12    rl do(O:Oid,phi@eat,none) < O:Oid : PHILOSOPHER | phi@eat-status : unlocked >
13      => send(O:Oid,phi@eat,[O:Oid,phi@eat]) < O:Oid : PHILOSOPHER | phi@eat-status : locked >
14     [label PHILOSOPHER-sending-phi@eat] .
15  endom
```

Módulo FOOD

```
1   omod FOOD is
2     including CBABEL-CONFIGURATION .
3
4     class FOOD .
5     op food@eat : -> PortInId  [ctor] .
6     eq instantiate(O:Oid,FOOD) = < O:Oid : FOOD | none > .
7
8     rl < O:Oid : FOOD | > done(O:Oid,food@eat,IT:Interaction)
9       => < O:Oid : FOOD | > ack(IT:Interaction)
10     [label FOOD-doneAndAcking-food@eat] .
11    rl < O:Oid : FOOD | > send(O:Oid,food@eat,IT:Interaction)
12      => < O:Oid : FOOD | > do(O:Oid,food@eat,IT:Interaction)
13     [label FOOD-recevingAndDo-food@eat] .
14  endom
```

Conector GET-TABLE

```
1   omod GET-TABLE is
2     including CBABEL-CONFIGURATION .
3
4     class GET-TABLE | get-table@places : StateRequired .
5     op get-get-table@places : Object -> Int .
6     op set-get-table@places : Object Int -> Object .
7     op get-table@in : -> PortInId    [ctor] .
8     op get-table@out : -> PortOutId  [ctor] .
9
10    eq get-get-table@places(< O:Oid : GET-TABLE | get-table@places : st(V:Int,S:Status)>)
```

```
    = V:Int .
  eq set-get-table@places(< O:Oid : GET-TABLE | get-table@places : st(V:Int,S:Status)>,V':Int)
    = < O:Oid : GET-TABLE | get-table@places : st(V':Int,changed)> .

  eq instantiate(O:Oid,GET-TABLE)
    = < O:Oid : GET-TABLE | none,get-table@places : st(0,unchanged)> .

  ceq after(OBJ:Object)
    = set-get-table@places(OBJ:Object,get-get-table@places(OBJ:Object)+ 1)
    if class(OBJ:Object)= GET-TABLE .
  ceq before(OBJ:Object)
    = set-get-table@places(OBJ:Object,get-get-table@places(OBJ:Object)- 1)
    if class(OBJ:Object)= GET-TABLE .
  ceq open?(OBJ:Object)
    = get-get-table@places(OBJ:Object)> 0
    if class(OBJ:Object)= GET-TABLE .

  rl < O:Oid : GET-TABLE | > ack([O:Oid,get-table@out]:: IT:Interaction)
    => after(< O:Oid : GET-TABLE | >) ack(IT:Interaction)
   [label GET-TABLE-acking-get-table@out] .
  crl < O:Oid : GET-TABLE | > send(O:Oid,get-table@in,IT:Interaction)
    => before(< O:Oid : GET-TABLE | >)
       send(O:Oid,get-table@out,[O:Oid,get-table@out]:: IT:Interaction)
    if open?(< O:Oid : GET-TABLE | >)= true
   [label GET-TABLE-sending-get-table@in] .
endom
```

Conector GET-FORK

```
omod GET-FORK is
  including CBABEL-CONFIGURATION .

  class GET-FORK | get-fork@available : StateRequired .
  op get-get-fork@available : Object -> Bool .
  op set-get-fork@available : Object Bool -> Object .
  op get-fork@in : -> PortInId    [ctor] .
  op get-fork@out : -> PortOutId  [ctor] .

  eq get-get-fork@available(< O:Oid : GET-FORK | get-fork@available : st(V:Bool,S:Status)>)
    = V:Bool .
  eq set-get-fork@available(< O:Oid : GET-FORK | get-fork@available : st(V:Bool,S:Status)>,V':Bool)
    = < O:Oid : GET-FORK | get-fork@available : st(V':Bool,changed)> .

  eq instantiate(O:Oid,GET-FORK)
    = < O:Oid : GET-FORK | none,get-fork@available : st(true,unchanged)> .

  ceq after(OBJ:Object)
    = set-get-fork@available(OBJ:Object,true)
    if class(OBJ:Object)= GET-FORK .
  ceq before(OBJ:Object)
    = set-get-fork@available(OBJ:Object,false)
    if class(OBJ:Object)= GET-FORK .
  ceq open?(OBJ:Object)
    = get-get-fork@available(OBJ:Object)== true
    if class(OBJ:Object)= GET-FORK .
```

```
27
28    rl < O:Oid : GET-FORK | > ack([O:Oid,get-fork@out]:: IT:Interaction)
29      => after(< O:Oid : GET-FORK | >) ack(IT:Interaction)
30     [label GET-FORK-acking-get-fork@out] .
31
32    crl < O:Oid : GET-FORK | > send(O:Oid,get-fork@in,IT:Interaction)
33      => before(< O:Oid : GET-FORK | >)
34         send(O:Oid,get-fork@out,[O:Oid,get-fork@out]:: IT:Interaction)
35      if open?(< O:Oid : GET-FORK | >)= true
36     [label GET-FORK-sending-get-fork@in]  .
37  endom
```

Módulo de aplicação da arquitetura 4-PHILOSOPHERS

```
1   omod 4-PHILOSOPHERS is
2     including CBABEL-CONFIGURATION .
3     including FOOD .
4     including FORK .
5     including GET-TABLE .
6     including GET-FORK .
7     including PHILOSOPHER .
8     including GET-FORK .
9     including TABLE .
10
11    op food : -> Oid
12    op table : -> Oid .
13    ops fork1 fork2 fork3 fork4 : -> Oid .
14    ops gtable1 gtable2 gtable3 gtable4 : -> Oid .
15    ops lfork1 lfork2 lfork3 lfork4 : -> Oid .
16    ops rfork1 rfork2 rfork3 rfork4 : -> Oid .
17    ops phi1 phi2 phi3 phi4 : -> Oid .
18
19    op topology : -> Configuration .
20    eq topology
21      = instantiate(food,FOOD) instantiate(fork1,FORK)
22        instantiate(fork2,FORK) instantiate(fork3,FORK)
23        instantiate(fork4,FORK) instantiate(gtable1,GET-TABLE)
24        instantiate(gtable2,GET-TABLE) instantiate(gtable3,GET-TABLE)
25        instantiate(gtable4,GET-TABLE) instantiate(lfork1,GET-FORK)
26        instantiate(lfork2,GET-FORK) instantiate(lfork3,GET-FORK)
27        instantiate(lfork4,GET-FORK) instantiate(phi1,PHILOSOPHER)
28        instantiate(phi2,PHILOSOPHER) instantiate(phi3,PHILOSOPHER)
29        instantiate(phi4,PHILOSOPHER) instantiate(rfork1,GET-FORK)
30        instantiate(rfork2,GET-FORK) instantiate(rfork3,GET-FORK)
31        instantiate(rfork4,GET-FORK) instantiate(table,TABLE) .
32
33    eq < gtable1 : GET-TABLE | get-table@places : st(V1:Int,changed)>
34       < table : TABLE | places : V2:Int >
35     = < gtable1 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
36       < table : TABLE | places : V1:Int > .
37
38    eq < gtable2 : GET-TABLE | get-table@places : st(V1:Int,changed)>
39       < table : TABLE | places : V2:Int >
40     = < gtable2 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
41       < table : TABLE | places : V1:Int > .
```

```
42
43    eq < gtable3 : GET-TABLE | get-table@places : st(V1:Int,changed)>
44       < table : TABLE | places : V2:Int >
45     = < gtable3 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
46       < table : TABLE | places : V1:Int > .
47
48    eq < gtable4 : GET-TABLE | get-table@places : st(V1:Int,changed)>
49       < table : TABLE | places : V2:Int >
50     = < gtable4 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
51       < table : TABLE | places : V1:Int > .
52
53    eq < lfork1 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
54       < fork1 : FORK | available : V2:Bool >
55     = < lfork1 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
56       < fork1 : FORK | available : V1:Bool > .
57
58    eq < lfork2 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
59       < fork2 : FORK | available : V2:Bool >
60     = < lfork2 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
61       < fork2 : FORK | available : V1:Bool > .
62
63    eq < lfork3 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
64       < fork3 : FORK | available : V2:Bool >
65     = < lfork3 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
66       < fork3 : FORK | available : V1:Bool > .
67
68    eq < lfork4 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
69       < fork4 : FORK | available : V2:Bool >
70     = < lfork4 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
71       < fork4 : FORK | available : V1:Bool > .
72
73    eq < rfork1 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
74       < fork4 : FORK | available : V2:Bool >
75     = < rfork1 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
76       < fork4 : FORK | available : V1:Bool > .
77
78    eq < rfork2 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
79       < fork1 : FORK | available : V2:Bool >
80     = < rfork2 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
81       < fork1 : FORK | available : V1:Bool > .
82
83    eq < rfork3 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
84       < fork2 : FORK | available : V2:Bool >
85     = < rfork3 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
86       < fork2 : FORK | available : V1:Bool > .
87
88    eq < rfork4 : GET-FORK | get-fork@available : st(V1:Bool,changed)>
89       < fork3 : FORK | available : V2:Bool >
90     = < rfork4 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
91       < fork3 : FORK | available : V1:Bool > .
92
93    ceq < gtable1 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
94        < table : TABLE | places : V2:Int >
95      = < gtable1 : GET-TABLE | get-table@places : st(V2:Int,unchanged)>
```

```
96        < table : TABLE | places : V2:Int >
97      if V1:Int =/= V2:Int = true .
98
99    ceq < gtable2 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
100       < table : TABLE | places : V2:Int >
101     = < gtable2 : GET-TABLE | get-table@places : st(V2:Int,unchanged)>
102       < table : TABLE | places : V2:Int >
103     if V1:Int =/= V2:Int = true .
104
105   ceq < gtable3 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
106       < table : TABLE | places : V2:Int >
107     = < gtable3 : GET-TABLE | get-table@places : st(V2:Int,unchanged)>
108       < table : TABLE | places : V2:Int >
109     if V1:Int =/= V2:Int = true .
110
111   ceq < gtable4 : GET-TABLE | get-table@places : st(V1:Int,unchanged)>
112       < table : TABLE | places : V2:Int >
113     = < gtable4 : GET-TABLE | get-table@places : st(V2:Int,unchanged)>
114       < table : TABLE | places : V2:Int >
115     if V1:Int =/= V2:Int = true .
116
117   ceq < lfork1 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
118       < fork1 : FORK | available : V2:Bool >
119     = < lfork1 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
120       < fork1 : FORK | available : V2:Bool >
121     if V1:Bool =/= V2:Bool = true .
122
123   ceq < lfork2 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
124       < fork2 : FORK | available : V2:Bool >
125     = < lfork2 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
126       < fork2 : FORK | available : V2:Bool >
127     if V1:Bool =/= V2:Bool = true .
128
129   ceq < lfork3 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
130       < fork3 : FORK | available : V2:Bool >
131     = < lfork3 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
132       < fork3 : FORK | available : V2:Bool >
133     if V1:Bool =/= V2:Bool = true .
134
135   ceq < lfork4 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
136       < fork4 : FORK | available : V2:Bool >
137     = < lfork4 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
138       < fork4 : FORK | available : V2:Bool >
139     if V1:Bool =/= V2:Bool = true .
140
141   ceq < rfork1 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
142       < fork4 : FORK | available : V2:Bool >
143     = < rfork1 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
144       < fork4 : FORK | available : V2:Bool >
145     if V1:Bool =/= V2:Bool = true .
146
147   ceq < rfork2 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
148       < fork1 : FORK | available : V2:Bool >
149     = < rfork2 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
```

```
150       < fork1 : FORK | available : V2:Bool >
151     if V1:Bool =/= V2:Bool = true .
152
153   ceq < rfork3 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
154       < fork2 : FORK | available : V2:Bool >
155     = < rfork3 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
156       < fork2 : FORK | available : V2:Bool >
157     if V1:Bool =/= V2:Bool = true .
158
159   ceq < rfork4 : GET-FORK | get-fork@available : st(V1:Bool,unchanged)>
160       < fork3 : FORK | available : V2:Bool >
161     = < rfork4 : GET-FORK | get-fork@available : st(V2:Bool,unchanged)>
162       < fork3 : FORK | available : V2:Bool >
163     if V1:Bool =/= V2:Bool = true .
164
165   eq send(gtable1,get-table@out,IT:Interaction) = send(lfork1,get-fork@in,IT:Interaction)
166    [label get-table@out-linking-get-fork@in] .
167   eq send(gtable2,get-table@out,IT:Interaction) = send(lfork2,get-fork@in,IT:Interaction)
168    [label get-table@out-linking-get-fork@in] .
169   eq send(gtable3,get-table@out,IT:Interaction) = send(lfork3,get-fork@in,IT:Interaction)
170    [label get-table@out-linking-get-fork@in] .
171   eq send(gtable4,get-table@out,IT:Interaction) = send(lfork4,get-fork@in,IT:Interaction)
172    [label get-table@out-linking-get-fork@in] .
173   eq send(lfork1,get-fork@out,IT:Interaction) = send(rfork1,get-fork@in,IT:Interaction)
174    [label get-fork@out-linking-get-fork@in] .
175   eq send(lfork2,get-fork@out,IT:Interaction) = send(rfork2,get-fork@in,IT:Interaction)
176    [label get-fork@out-linking-get-fork@in] .
177   eq send(lfork3,get-fork@out,IT:Interaction) = send(rfork3,get-fork@in,IT:Interaction)
178    [label get-fork@out-linking-get-fork@in] .
179   eq send(lfork4,get-fork@out,IT:Interaction) = send(rfork4,get-fork@in,IT:Interaction)
180    [label get-fork@out-linking-get-fork@in] .
181   eq send(phi1,phi@eat,IT:Interaction) = send(gtable1,get-table@in,IT:Interaction)
182    [label phi@eat-linking-get-table@in] .
183   eq send(phi2,phi@eat,IT:Interaction) = send(gtable2,get-table@in,IT:Interaction)
184    [label phi@eat-linking-get-table@in] .
185   eq send(phi3,phi@eat,IT:Interaction) = send(gtable3,get-table@in,IT:Interaction)
186    [label phi@eat-linking-get-table@in] .
187   eq send(phi4,phi@eat,IT:Interaction) = send(gtable4,get-table@in,IT:Interaction)
188    [label phi@eat-linking-get-table@in] .
189   eq send(rfork1,get-fork@out,IT:Interaction) = send(food,food@eat,IT:Interaction)
190    [label get-fork@out-linking-food@eat] .
191   eq send(rfork2,get-fork@out,IT:Interaction) = send(food,food@eat,IT:Interaction)
192    [label get-fork@out-linking-food@eat] .
193   eq send(rfork3,get-fork@out,IT:Interaction) = send(food,food@eat,IT:Interaction)
194    [label get-fork@out-linking-food@eat] .
195   eq send(rfork4,get-fork@out,IT:Interaction)
196     = send(food,food@eat,IT:Interaction)
197    [label get-fork@out-linking-food@eat] .
198 endom
```