Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Geração Automática de Diagramas UML-RT a partir de Especificações CSP

Patrícia Muniz Ferreira

DISSERTAÇÃO DE MESTRADO

Recife
29 de agosto de 2006

Universidade Federal de Pernambuco
Centro de Informática

Patrícia Muniz Ferreira

Geração Automática de Diagramas UML-RT a partir de Especificações CSP

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Augusto Cezar Alves Sampaio*
Co-orientador: *Alexandre Cabral Mota*

Recife
29 de agosto de 2006

À minha família, fonte inesgotável de amor e alegrias.

Agradecimentos

A todos os amigos que encontrei e reencontrei durante o mestrado.

À Motorola/Facepe, por toda credibilidade concedida ao *Brazil Test Center Research Project* quando nos são submetidos problemas reais em busca de soluções eficientes e eficazes. À Motorola, pela proposição de problemas e incentivo financeiro.

Aos pesquisadores do *Brazil Test Center Research Project*, pelas valorosas discussões e dedicação ao sucesso conjunto dos projetos de pesquisa.

Ao grupo *ForMULa*, pelas valorosas sugestões durante as apresentações.

A Joabe Jesus, Rodrigo Ramos e Manoel Messias pela avaliação crítica do trabalho, em especial a Joabe Jesus, pela importante contribuição técnica.

A meus orientadores, Augusto Sampaio e Alexandre Mota, pela inúmeras sugestões durante este mestrado, pelo acompanhamento irrestrito, e pela amizade.

A meus pais e sogros, que vibraram e torceram por mim.

A Vander, que me incentivou e fez tudo parecer mais agradável, e me inspirou em cada momento.

Resumo

A integração de linguagens formais com notações visuais semi-formais tem se tornado freqüente na comunidade de engenharia de software, acreditando-se que as duas abordagens possam ser complementares. Métodos formais, como CSP, têm semântica precisa e o poder de definir rigorosamente o comportamento do software desejado e o atendimento a determinadas propriedades, mas raramente oferecem representação gráfica intuitiva para suas especificações. Por outro lado, métodos semi-formais, como a modelagem visual, são amplamente aceitos e facilmente integrados ao processo de desenvolvimento de sistemas, embora careçam de semântica precisa e de métodos de verificação de propriedades seguros.

Sob a perspectiva de métodos formais, a vantagem da integração com a modelagem visual é a possibilidade de criar uma interface gráfica para a interpretação de especificações, mais compreensível para usuários não habituados com notações formais. A integração permite ao desenvolvedor escolher o grau de formalismo a ser empregado em um dado projeto. Tipicamente, a notação visual seria utilizada para comunicação entre os membros da equipe (e com o cliente), e o formalismo para assegurar a aderência a certas propriedades fundamentais.

A proposta da dissertação é a construção de um conjunto de regras composicionais para mapear, sistematicamente, especificações CSP em modelos UML-RT (um profile UML para modelar aplicações concorrentes e de tempo real). A tradução de especificações CSP em modelos UML-RT permite que o projeto de uma aplicação seja enriquecido com modelos diagramáticos que preservam as propriedades da especificação formal. A partir de uma especificação CSP, geramos três visões complementares em UML-RT, representadas por diagramas de classe, de estrutura e máquinas de estado. O processo foi automatizado através de uma ferramenta e um estudo de caso foi desenvolvido para ilustrar a estratégia.

Os modelos UML-RT gerados podem ser formalmente refinados, através de leis de transformação definidas em trabalhos relacionados. Assim, o projeto da aplicação torna-se incrementalmente mais concreto, com a vantagem de ter uma base formal em sua origem, com as propriedades devidamente validadas.

Escolhemos representar elementos de CSP usando UML-RT porque esta notação possui facilidades para modelar sistemas reativos e concorrentes. O conceito de processos independentes e comunicantes da álgebra de processos CSP pode ser naturalmente representado por objetos ativos e cooperantes, presentes em UML-RT. Além disto, os conceitos de UML-RT foram incorporados ao padrão UML 2.0, o que torna a abordagem passível de evolução e aplicável em diversos cenários. Questões como sincronização múltipla, causalidade entre eventos, não-determinismo, recursões mútuas entre processos, e comportamentos dinâmicos foram considerados para determinar a construção dos elementos estruturais e comportamentais de UML-RT.

Palavras-chave: CSP, estratégia sistemática, UML-RT, integração de métodos formais.

Abstract

The integration of formal languages with semi-formal visual notations has become a recurrent research topic within the software engineering community, given the belief that both approaches can be complementary.

Formal methods, like CSP, have precise semantics and enable the rigorous definition of intended software behavior as well as the compliance with specific properties, but seldom offer an intuitive graphical notation for their specifications. On the other hand, semi-formal methods, like visual modelling, are widely accepted and easily integrated into software development process, although they lack precise formal semantics and safe methods for checking system properties.

From the formal methods perspective, the advantage of the integration with visual modelling is the possibility of defining a graphical interface for the interpretation of specifications, which is more suitable for users not familiar with formalism. The integration allows the developer to choose the appropriate degree of formalism to be applied in a project. Generally, the visual notation would be used for communication between team members (as well as with the customer), and the formalism for ensuring compliance with specific fundamental properties.

This dissertation addresses the definition of a set of compositional rules for systematically mapping CSP specifications into UML-RT models (a UML profile for modelling concurrent and real-time applications). The translation of CSP specifications into UML-RT models improves the design of an application with visual models that comply with the formal specification properties. From a given CSP specification, we generate three complementary views in UML-RT, represented by class, structure, and state machine diagrams. The process has been automated with tool support, and a case study has been performed to illustrate the strategy.

The generated UML-RT models can be formally refined by means of transformation laws defined in related work. Therefore, the design of the application becomes incrementally more concrete, with the advantage of having a formal basis in its origin and with the properties ensured.

We chose to represent CSP elements using UML-RT because the latter has facilities for modelling reactive and concurrent systems. The concept of independent and communication processes from process algebra can be naturally represented by active and cooperative objects in UML-RT. Moreover, the concepts of UML-RT have been incorporated into UML 2.0, which makes the approach evolvable and useful diverse scenarios. Issues such as multiple synchronization, causality between events, non-determinism, mutual recursion between process, and dynamic behaviour have been considered for determining the construction of UML-RT's structural and behavioral elements.

Keywords: CSP, systematic strategy, UML-RT, formal method integration.

Sumário

1 Introdução **1**

2 Visão Geral de CSP **7**
2.1 Definindo Processos 9
2.1.1 Processos Primitivos 10
2.1.2 Prefixo 10
2.1.3 Recursão 11
2.1.4 Escolhas Externa e Interna 11
2.1.5 Escolha Condicional 12
2.1.6 Internalização de Eventos 12
2.1.7 Renomeação de Eventos 13
2.1.8 Composição Seqüencial 13
2.1.9 Interrupção 13
2.1.10 Paralelismo 14
2.1.11 Operadores Indexados 14
2.2 Prova de propriedades e Refinamentos 15
2.3 Ferramentas de Verificação e Animação de Modelos CSP 16

3 UML-RT **17**
3.1 Cápsulas 18
3.2 Portas 19
3.3 Protocolos 19
3.4 Modelando a estrutura 20
3.5 Modelando o comportamento 22
3.6 Comunicação 24
3.7 Outras notações visuais 26

4 Mapeamento de CSP em UML-RT **29**
4.1 Normalização de Processos 30
4.2 Mapeamento de Processos 33
4.2.1 Estratégia para Construção de Cápsulas 33
4.2.2 Estratégia para Composição de Cápsulas 39
4.2.3 Regras de Mapeamento 44
4.3 Mapeamento de Tipos de Dados 57
4.4 Representações Alternativas para Operadores CSP 59
4.5 Considerações Finais 61

5 Estudo de Caso e Automação da Estratégia **63**
5.1 Estudo de Caso 63
5.1.1 Aplicando a Estratégia 66
5.1.1.1 Normalização das Equações de Processos 66
5.1.1.2 Mapeamento dos Tipos de Dados 66
5.1.1.3 Criação da Cápsula Controladora 67
5.1.1.4 Mapeamento das Equações de Processos 68
5.1.2 Simulação do Modelo Gerado 73
5.2 Automação da Estratégia 76
5.2.1 Arquitetura da FormalDev 76
5.2.2 Características dos Modelos UML-RT Gerados 78

6 Conclusão **83**
6.1 Trabalhos Relacionados 84
6.2 Trabalhos Futuros 86

A Pseudo-código de métodos usados por *SystemController* **89**

B Regras de Mapeamento Adicionais **91**

Lista de Figuras

2.1 Equações de processos 10

3.1 Representação de cápsulas em diagramas de classes 18
3.2 Representação de protocolos em diagramas de classes 20
3.3 Máquina de estados do papel *base* de *ProtocoloComunicacao* 20
3.4 Diagrama de estrutura de cápsula 21
3.5 Cápsula *P* contendo instância ativa de *Q* 22
3.6 Máquina de estados de cápsula 23
3.7 Estados concorrentes 24
3.8 Representação de estados concorrentes como cápsulas concorrentes 25
3.9 Comunicação assíncrona entre cápsulas 25
3.10 Comunicação síncrona entre cápsulas 26

4.1 Cápsula *P* 33
4.2 Configuração entre cápsulas as *P*, *Q* e *R* 35
4.3 Estrutura do protocolo *CSPMessageProtocol* 36
4.4 Máquina de estados do protocolo *CSPMessageProtocol* 37
4.5 Máquina de estados da cápsula *Q*, atendendo ao protocolo *CSPMessageProtocol* 37
4.6 Diagrama de estrutura da cápsula *P*, contendo porta *a* 38
4.7 Cápsula *P* contendo instância da cápsula *Q* 39
4.8 Cápsula *R* contendo instâncias das cápsulas *Z* e *W* 40
4.9 Papel *base* do protocolo *CSPBehaviorProtocol* 40
4.10 Cápsula *P*, contendo a porta comportamental *b* 41
4.11 *SystemController* contendo sub-cápsulas *P* e *Q* 41
4.12 Estrutura de *SystemController* antes e depois da mudança de comportamento de *P* 42
4.13 Estrutura de *SystemController* antes e depois da mudança de comportamento de *Q* 43
4.14 Ambiente externo contendo a cápsula principal do sistema 44
4.15 Estrutura e comportamento iniciais de *SystemController* 45
4.16 Estrutura de *SystemController* antes e depois da mudança de comportamento de *P* para *Q* 47
4.17 Estrutura de *SystemController* antes e depois da mudança de comportamento de *P* para ***SKIP*** 48
4.18 Estrutura de *SystemController* antes e depois dos eventos da porta *a* 52

4.19 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para Q $[a \leftarrow c]$ 54
4.20 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para $Q \Box R$ 55
56
4.22 Estrutura de *SystemController* antes e depois da mudança de comportamento de R para ***STOP*** 57
4.23 Classes que representam tipos de dados CSP 59
4.24 Estrutura de cápsulas para representar o paralelismo 60
4.25 Mudança necessária na cápsula que representa o paralelismo 60

5.1 Classes *Valor*, *Identificador* e *Resultado* 67
5.2 Estrutura e Comportamento Iniciais da Cápsula Controladora 67
5.3 Cápsula *Maquina* 68
5.4 Cápsula *SystemController*, após mapeamento do processo *Maquina* 68
5.5 Cápsula *MSolicitador* 69
5.6 Cápsula *SystemController*, após mapeamento do processo *MSolicitador* 69
5.7 Cápsula *MSolicitador*0 70
5.8 Cápsula *MSolicitador*1 70
5.9 Cápsula *MSolicitador*2 70
5.10 Cápsula *MExecutor* 71
5.11 Cápsula *MExecutor*0 71
5.12 *SystemController* após o mapeamento dos processos 72
5.13 Configuração inicial de *SystemController* 73
5.14 Configuração de *SystemController* após mudança de comportamento de *Maquina* 74
5.15 Configuração de *SystemController* após mudança de comportamento de *MSolicitador* 75
5.16 Configuração de *SystemController* após mudança de comportamento de *MSolicitador0* 75
5.17 Arquitetura de Componentes do *FormalDev* 77
5.18 Tela prinicipal do *FormalDev* 77
5.19 Classes da biblioteca *com.rational.rosert* do *Rose RealTime* 79
5.20 Classes do pacote *datatypes* de *CSPCore* 79
5.21 Classes do pacote *expression* de *CSPCore* 80
5.22 Classes do pacote *communication* de *CSPCore* 81
5.23 Cápsula *CapsuleContainerSystem* de *CSPCore* 81
5.24 Representações da cápsula *Maquina* nas regras de mapeamento (esquerda) e no *Rose RealTime* (direita) 82
5.25 Diagrama de Classes do Estudo de Caso no *Rose RealTime* 82

CAPÍTULO 1

Introdução

À medida que os sistemas baseados em computação tornam-se progressivamente mais complexos, acentua-se a necessidade de se representar as características dos sistemas de maneira prática, concisa e estruturada. As características representáveis vão desde requisitos funcionais simples, como o padrão visual de um aplicativo, a requisitos arquiteturais complexos, como os protocolos de comunicação em um sistema distribuído e de tempo real.

O desafio para a especificação de requisitos complexos é o de como capturar precisamente propriedades estáticas, dinâmicas e de tempo real de maneira estruturada. Em particular, muitos sistemas complexos envolvem interações concorrentes entre partes distintas do sistema, enquanto cada uma destas partes encerra propriedades específicas, possivelmente tão complexas quanto as primeiras. Sistemas híbridos, por exemplo, envolvem interações entre componentes físicos e módulos de controle de software. A especificação destes sistemas requer tanto a captura da estrutura e do comportamento de cada componente (software ou hardware) separadamente, quanto das interações entre estes.

Freqüentemente, requisitos são especificados em texto, algumas vezes com gráficos. Entretanto, esta é a causa de muitos problemas no desenvolvimento de sistemas. Requisitos descritos em linguagem natural são informais e imprecisos, e tendem a causar interpretações equivocadas por parte dos clientes e dos engenheiros de software. Mais ainda, os requisitos podem estar inconsistentes ou incompletos, desde que descrições informais não permitem a realização de provas de propriedades. Estes problemas se agravam em sistemas distribuídos e concorrentes, ou que sejam críticos quanto à segurança ou ao tempo de resposta.

Para minimizar estes problemas, a comunidade de engenharia de software tem proposto e utilizado notações mais precisas para representar a arquitetura de aplicações. Estas notações oferecem abstrações para modelar sistemas complexos sob o ponto de vista estrutural e comportamental, em diversos domínios de aplicações. Notações formais são comprovadamente úteis para especificar sistemas complexos, críticos e de tempo real, por conta do seu rigor matemático e capacidade para aplicar técnicas analíticas, que facilitam a detecção de erros e a prova de propriedades desejadas ainda na fase de especificação. Algumas destas notações dispõem de ferramentas próprias (verificadores de modelos) para a aplicação destas técnicas e a análise de seus resultados.

CSP [41, 24], por exemplo, é uma linguagem formal para modelar sistemas concorrentes, onde as partes componentes do sistema são representadas como processos. CSP permite modelar diversas formas de interação entre os processos componentes do sistema, de maneira concisa e elegante. Cada primitiva, ou operador, de CSP é uma abstração para um padrão de interação específico, e o seu comportamento está implícito ao seu uso. CSP também encapsula o conceito de comunicação entre processos, que envolve passagem de mensagens síncronas. Por ser uma

álgebra de processos, CSP define leis algébricas para cada operador, assim é possível manipular expressões de processos usando raciocínio equacional. Desta forma, CSP permite provar propriedades comportamentais clássicas de sistemas, como comportamentos determinísticos e ausências de *deadlock* e *livelock*, assim como propriedades específicas da especificação através de verificadores de modelos como o FDR [19].

Entretanto, a aplicação de métodos formais ao processo de desenvolvimento de sistemas ainda enfrenta resistências. Por exemplo, em [23] é detectado que esta resistência existe em parte pela interpretação errada dos conceitos por trás das notações, e do desconhecimento da aplicabilidade de métodos formais em alguns domínios de sistemas. Já em [40], argumenta-se que mesmo um formalismo tecnicamente bem fundado, expressivo e adequado a alguma fase do processo de desenvolvimento, não é suficiente para ser aplicado ao desenvolvimento de aplicações reais se não possuir uma apresentação amigável de suas especificações, que permita aos envolvidos (clientes e engenheiros) lerem e entenderem facilmente o que está modelado. Uma deficiência relevante para muitos métodos formais é a falta de representação gráfica.

Qualquer notação gráfica estruturada oferece grandes vantagens para o desenvolvimento de sistemas: legibilidade, facilidade para entendimento, visualização e compactação. A contrapartida textual equivalente geralmente é mais difícil de compreender. A representação gráfica de alguns métodos formais, como Redes de Petri [11], SDL [9] e Statecharts [21], tem sido um dos motivos para o sucesso destes métodos na indústria e academia [40].

CSP, embora tenha demonstrado sua aplicabilidade no desenvolvimento de sistemas críticos, não possui uma visualização gráfica bem estabelecida. Mesmo as iniciativas mais recentes [26] para representar graficamente especificações CSP geram modelos pouco intuitivos. Como conseqüência, as especificações são geralmente consideradas difíceis ou custosas para incorporar ao desenvolvimento de software convencional. Mais ainda, a tarefa de associar o comportamento dinâmico de construções em CSP com elementos estruturais das fases de projeto e implementação é bastante suscetível a erros.

Por outro lado, métodos para modelagem gráfica são largamente usados na academia e indústria para estruturar, documentar e visualizar a arquitetura de sistemas. Entre estes, a UML (*Unified Modeling Language*) [34] destaca-se por ser uma compilação de várias técnicas de sucesso para modelagem visual de sistemas. Isto torna UML uma notação ampla e aplicável a diversos domínios de aplicação, sem no entanto ser dependente de plataforma. UML também permite sua adaptação a contextos específicos de aplicação, através da especialização de seus diagramas e esteriótipos. Estes padrões adaptados são chamados de *profiles*.

UML-RT [44, 31] é um *profile* conservativo de UML idealizado para modelar sistemas concorrentes e reativos. Este *profile* tem todos os elementos e diagramas de UML, somados a alguns elementos de modelagem e formalismos de ROOM (*Real-Time Object-Oriented Modeling*) [31], que é uma notação visual orientada a objetos específica para modelagem de sistemas de tempo real, distribuídos e reativos. Em UML-RT, assim como em ROOM, cada componente do sistema é representado por uma entidade independente, com estrutura e comportamento próprios, e a interação entre estes componentes se dá através de elementos específicos de modelagem.

Entretanto, as notações visuais raramente possuem fundamentos formais consolidados que permitam avaliar as propriedades de sistemas. Mesmo as notações semi-formais, como UML

e ROOM, não dispõem de recursos suficientes para tanto, embora haja, correntemente, vários esforços nesta direção.

Em geral, a aplicação de métodos formais ao desenvolvimento de aplicações assume que a especificação é seguida pelo projeto e implementação, nesta ordem. Mas esta é uma visão pouco realista do processo de desenvolvimento de software. As metodologias de desenvolvimento iterativas [5, 28], por exemplo, sugerem uma evolução gradativa da arquitetura do sistema. Na prática, o desenvolvedor deve rever os requisitos e a especificação formal a cada estágio do processo, e estes devem evoluir junto com a arquitetura, até um modelo próximo do real [23]. Entretanto, é comum as especificações formais serem mal interpretadas ou negligenciadas nas fases subseqüentes no desenvolvimento de software, porque geralmente existem diferenças conceituais entre a notação formal e os modelos usados nestas fases.

A integração de linguagens formais com notações informais (ou semi-formais) tem se tornado freqüente na comunidade de engenharia de software, acreditando-se que as duas abordagens possam ser complementares, e que esta integração possa minimizar as descontinuidades conceituais entre algumas fases do processo de desenvolvimento [32].

Do ponto de vista da engenharia de software, é possível que linguagens de modelagem visual, como UML, ganhem significado preciso através das técnicas de refinamento e prova de propriedades, além dos verificadores de modelos já bastante difundidos em métodos formais. Algumas iniciativas têm sido propostas para dar semântica formal a UML [15, 36], através da denotação dos seus diagramas e esteriótipos em alguma notação formal, como CSP, Z [46] e *Circus* [42]. Assim, modelos UML podem ser analisados através das notações usadas como modelo semântico para UML. Entretanto, estas iniciativas focam apenas em um subconjunto de UML.

Sob a perspectiva de métodos formais, a vantagem da integração com a modelagem visual é a possibilidade de especificar graficamente os requisitos comportamentais e estruturais de um sistema, criando uma interface gráfica para a interpretação de especificações, mais compreensível para usuários não habituados com notações formais [18].

De forma mais ampla, a integração permite uma unificação das notações, oferecendo flexibilidade ao desenvolvedor quanto ao grau de formalismo adequado ao seu projeto. Tipicamente, a análise de propriedades pode ser realizada com base na notação formal, e a documentação com base na notação diagramática, para fins de comunicação.

Nesta dissertação, propomos uma estratégia sistemática para o mapeamento de especificações CSP em modelos UML-RT, de forma que conceitos de processos, operadores e comunicação da notação formal sejam traduzidos em elementos e diagramas de UML-RT. O mapeamento de especificações CSP em modelos UML-RT permite que o projeto de uma aplicação seja enriquecido com modelos diagramáticos que preservam as propriedades da especificação formal. A partir de uma especificação CSP, geramos três visões complementares em UML-RT, representadas por diagramas de classe, de estrutura e máquinas de estado [13]. O processo foi automatizado através de uma ferramenta e um estudo de caso foi desenvolvido para ilustrar a estratégia.

As provas formais deste mapeamento não fazem parte do escopo desta dissertação, mas têm em trabalhos como o de Fisher [15] e Ramos [36] o embasamento inicial para sua construção. Embora as provas formais sejam sugeridas como trabalho futuro, a estratégia foi desenvolvida

com a intenção de preservar a semântica das especificações originais. As regras que sistematizam o processo devem permitir uma prova modular da tradução.

Este trabalho foi desenvolvido no contexto da parceria entre a Motorola e o Centro de Informática, como parte do BTC (*Brazil Test Center*), que intenciona desenvolver pesquisas junto ao processo de desenvolvimento e aplicação de testes aos produtos da Motorola. Como produto destas pesquisas a parceria pretende identificar possíveis pontos de melhoria ou investimentos no processo.

O projeto de pesquisa CIn/Motorola aborda a criação de modelos formais CSP não ambíguos, que representam os requisitos de aplicações da Motorola. A partir destes modelos é possível extrair, de forma sistemática e automatizada, diversas informações pertinentes aos processos de requisitos, projeto e testes.

No contexto da cooperação, a contribuição deste trabalho é a conversão da especificação formal CSP em diagramas UML-RT, para que estes sirvam de base para a implementação das *features* do sistema, aproximando potencialmente a implementação com modelo formal CSP. Os principais ganhos surgem da possibilidade da modelagem visual UML validar a consistência da especificação formal em relação aos requisitos, da facilidade de interpretação do modelo, mas principalmente dos casos de testes e da própria aplicação passarem a ter uma relação de conformidade natural com a implementação.

Escolhemos representar elementos de CSP usando UML-RT porque esta notação possui facilidades para modelar sistemas reativos e concorrentes. O conceito de processos independentes e comunicantes da álgebra de processos CSP pode ser naturalmente representado por objetos ativos e cooperantes, presentes em UML-RT. Além disto, a semântica formal herdada de ROOM possibilita a geração de código, tornando possível animar e testar especificações CSP através da tradução. Por fim, os conceitos de UML-RT foram incorporados ao padrão UML 2.0, o que torna a abordagem passível de evolução e aplicável em diversos cenários.

A estratégia de mapeamento pode ser resumida como um sistema de reescrita, composto de regras independentes e composicionais. Para isso, são usadas regras de normalização de equações CSP, seguidas de regras para a construção de modelos UML-RT. A aplicação exaustiva destas regras traduz gradual e sistematicamente uma especificação CSP em um modelo UML-RT.

Consideramos regras de mapeamento de processos e tipos de dados CSP. Os modelos UML-RT obtidos a partir destas regras oferecem uma alternativa gráfica para CSP em uma notação largamente usada na academia e indústria.

A automação da estratégia é possível porque as regras são específicas para cada padrão de expressão CSP. Desta forma, a aplicação de cada regra é não-ambígua. Como uma contribuição adicional desta dissertação, desenvolvemos uma ferramenta que automatiza a estratégia de mapeamento, gerando automaticamente modelos no *Rational Rose RealTime* [39], uma ferramenta de apoio a UML-RT.

Em particular, os modelos gerados para o *Rose RealTime* são úteis para gerar código, o que permitiria animar e testar a especificação CSP. Entretanto, os modelos gerados através da estratégia ainda são semi-executáveis, visto que as regras de mapeamento não representam funções e expressões lógicas descritas em CSP em código executável, mas métodos cujo corpo é comentado com a equação algébrica que o originou. A transformação destas equações algébricas

em código executável já foi analisada em trabalhos relacionados [17, 16], e está fora do escopo desta dissertação.

Na literatura, vários trabalhos têm abordado a integração de métodos formais com notações ou linguagens que aproximam a especificação formal de sistemas dos modelos de análise e projeto [16, 1, 2, 17, 22, 14, 37, 21, 18]. Entretanto, poucos trabalhos abordam especificamente a transformação de especificações formais em alguma notação visual estruturada. Alguns trabalhos relacionados abordam o mapeamento inverso, de UML e UML-RT para algumas notações formais relacionadas a CSP, como CSP-OZ [14] e ***OhCircus*** [37]. Abordagens alternativas [16, 1, 2, 17, 22] apresentam o mapeamento de CSP não para uma notação gráfica, mas para bibliotecas Java que implementam a semântica de CSP.

Esta dissertação está dividida em seis capítulos, além deste, e dois apêndices. O conteúdo de cada um é descrito a seguir:

- Capítulo 2: Apresentamos a álgebra de processos CSP e algumas características dos seus operadores, usados para abstrair as diversas formas de interação entre processos.
- Capítulo 3: Introduzimos os conceitos presentes em UML-RT, e as características desta notação para representar sistemas concorrentes e reativos.
- Capítulo 4: Apresentamos as regras de mapeamento de expressões CSP para elementos estruturais e diagramáticos de UML-RT. As regras são aplicadas através de uma estratégia sistemática, que traduz gradualmente tipos de dados e processos CSP em classes, cápsulas e protocolos UML-RT. São identificadas as características necessárias aos diagramas e esteriótipos de UML-RT para representarem as características de CSP.
- Capítulo 5: Aplicamos a estratégia de mapeamento a uma estudo caso, e apresentamos os detalhes da automação da estratégia.
- Capítulo 6: Discutimos alguns trabalhos relacionados e futuros, e apresentamos nossas considerações finais.
- Apêndice A: Encontra-se o pseudo-código de alguns métodos auxiliares dos modelos UML-RT gerados.
- Apêndice B: Mostramos algumas regras de mapeamento não apresentadas no Capítulo 4.

CAPÍTULO 2

Visão Geral de CSP

CSP [24, 41] (*Communicating Sequential Processes*) é uma linguagem formal projetada para modelar o comportamento de sistemas concorrentes e distribuídos. Uma forma de entender CSP é imaginar um sistema como uma composição de unidades comportamentais independentes (subsistemas, componentes ou simplesmente rotinas de processamento) que comunicam-se entre si e com o ambiente que os cerca [24]. Cada uma destas unidades independentes pode ser formada por unidades menores, combinadas por algum padrão de interação específico. Consideramos o ambiente como todo agente externo que pode interagir com o sistema, como os seus usuários ou outros sistemas.

Processos são abstrações para unidades comportamentais, e são construídos através de eventos, operadores e outros processos. Isto é, processos podem ser combinados para formar processos maiores, até que o comportamento de todo o sistema esteja especificado. A composicionalidade de processos (facilidade para compor processos complexos a partir de processos menores, sem alterar a estrutura interna das partes componentes) permite que CSP seja satisfatoriamente usado para modelar sistemas sob diversas abordagens para desenvolvimento de aplicações, como a orientada a objetos.

Eventos são abstrações de ações do mundo real. Por exemplo, o evento

$$exibir.BoasVindas$$

pode ser usado para representar a ação de exibir uma mensagem de boas vindas ao usuário do sistema. Além de eventos, que representam uma única ação, CSP permite definir *canais*, que representam coleções de eventos com características em comum. Canais também são vistos conceitualmente como meios para transmitir dados. Por exemplo, a declaração

$$channel\ e : Int$$

introduz o canal e que pode transmitir qualquer valor do tipo inteiro; o evento $e.1$ é um dos possíveis eventos que podem ocorrer a partir desta declaração. Além de tipos pré-definidos, como o *Int* usado previamente, CSP também permite definir tipos novos. Uma facilidade é a criação de tipos enumerados, através do construtor *datatype*. Por exemplo,

$$datatype\ Mensagem : BoasVindas \mid Erro \mid ConfimacaoCadastro$$
$$channel\ exibir : Mensagem$$

determina que o canal *exibir* pode transmitir qualquer valor do tipo *Mensagem*, ou seja, qualquer dos eventos $exibir.BoasVindas$, $exibir.Erro$ e $exibir.ConfimacaoCadastro$. Note que a declaração de um canal sem tipo, como em

$$channel\ e$$

define um único evento.

Como dito anteriormente, canais com tipos podem ser usados para transmitir dados desse tipo. Assim sendo, CSP também permite modelar a entrada e saída de dados em canais. Por exemplo, a expressão

$$e?x$$

determina que o canal e está pronto para receber um valor a ser atribuído à variável x. Dessa forma, a expressão $e?x$ cria um *binding* entre x e o valor recebido através de e. É possível, inclusive, restringir o conjunto de valores aceitáveis. Considere a expressão

$$e?x : \{x \leq 10\}$$

onde o canal e foi declarado sobre o tipo inteiro. Por conta da expressão de restrição $: \{x \leq 10\}$, este canal só aceitará inteiros menores ou iguais a 10.

Os exemplos anteriores consideram a possibilidade de receber algum dado através de um canal. Mas, obviamente, para que haja uma recepção de dado, deve um correspondente envio. Isso se dá através de expressões de saída, como a seguinte:

$$e!exp$$

Nesta expressão, o canal e envia o valor da expressão exp.

Desta forma, canais podem também ser abstrações para pontos de comunicação, ou a interface, de um sistema com seu ambiente. Estas ações podem ainda ser modeladas de forma integrada, como em

$$soma?x?y!(x+y)$$

A ocorrência de um evento em um processo caracteriza uma comunicação deste processo com pelo menos um participante. Geralmente o participante é um outro processo, caso contrário será o próprio ambiente em que o processo está envolvido. Para entender a ocorrência de um evento em um processo deve-se considerar que [41, 17]:

- Eventos são instantâneos: não há diferença de tempo entre o início e fim de um evento em um processo. Conceitualmente eventos podem ocorrer a qualquer momento, desde que isso seja permitido. Esta propriedade, embora permita que a especificação abstraia possíveis atrasos na comunicação, limita a linguagem a não considerar aspectos temporais. Extensões de CSP com tempo (*Timed CSP*) [29] já existem, mas não são consideradas neste trabalho.

- A comunicação entre processos é atômica e se dá através de passagem de mensagens simultâneas[1]. Estas interações não são diferenciadas seja para sincronismo simples (ou *rendezvous*, entre 2 processos) ou complexo (ou *multi-way rendezvous*, entre 3 ou mais processos). No modelo de comunicação síncrono todos os processos participantes devem estar simultaneamente prontos para executar a comunicação.

[1] Em CSP não existe compartilhamento de variáveis através de uma memória em comum.

Durante o *rendezvous*, ambos os processos emissores e receptores permanecem bloqueados no ponto de sincronização até que todos os envolvidos alcancem este ponto em seus fluxos particulares. Só então a comunicação ocorre, liberando os envolvidos. Ou seja, os atos de enviar e receber mensagens são dependentes entre si, já que nenhum é efetuado sem o outro.

A troca de mensagens em CSP é efetuada sem que necessariamente sejam definidos eventos de entrada e de saída. Isto se deve ao fato de que, na prática, toda ocorrência de $e?x$ ou $e!x$ é reduzida para um evento simples, como $e.v$, onde v é um dos valores válidos para o canal e. Assim sendo, a noção de entrada ou saída em canais é puramente conceitual, e na prática, é irrelevante do ponto de vista comportamental.

O conjunto de todos os eventos possíveis em uma especificação é dito *alfabeto*, e é representado por Σ [41]. Neste trabalho, convencionamos chamar o conjunto de eventos que podem ocorrer em um processo de *alfabeto do processo*, e para tal usamos a notação αP, onde P é um processo.

A composição de eventos para formar um processo, e o relacionamento entre diferentes processos são descritos através dos operadores algébricos de CSP. Através dos operadores é possível, por exemplo, executar dois processos em paralelo, podendo haver ou não comunicação entre eles. Outro exemplo é permitir que eventos de um processo não sejam percebidos por outros processos ou pelo ambiente externo. Isto faz CSP extremamente flexível em relação a outras notações formais, já que permite descrever tanto sistemas próximos da arquitetura, onde a estrutura de processos se assemelha à estrutura de componentes do sistema, como especificações mais abstratas, que tenham pouca relação com a estrutura, mas detalhem o comportamento desejado [41, 14, 17].

A sintaxe de CSP define a forma como eventos e processos podem ser combinados através dos operadores para formar um processo. A seguir apresentamos a sintaxe considerada nesta dissertação [41] e uma breve descrição de cada um dos elementos sintáticos mencionados.

2.1 Definindo Processos

Cada processo é na verdade uma *equação* composta por eventos, operadores algébricos e outros processos. As equações de processos são definidas de forma semelhante a funções. Ou seja, o lado esquerdo introduz o nome e os parâmetros, e o lado direito o corpo do processo.

Assumimos que o lado esquerdo da equação de um processo parametrizado P será representado como $P(s)$, onde s é o parâmetro de P. Na prática, obviamente, um processo pode ter mais de um parâmetro, porém evitamos este caso por simplicidade na representação das equações.

Os parâmetros de processo, junto com os parâmetros de entrada de eventos, representam o estado interno do processo. O escopo dos parâmetros de processo se estende por toda a definição do processo, e seus valores podem ser utilizados em qualquer evento deste processo.

Nesta seção detalhamos os operadores algébricos e os processos primitivos de CSP usados para compor a equação de processos complexos (Figura 2.1). Considere g uma expressão condicional, a um evento ($a \in \Sigma$), C um conjunto de eventos e R uma relação ($\Sigma \times \Sigma$).

$$
\begin{array}{lcll}
P(s) & ::= & \textbf{\textit{STOP}} & \\
 & | & \textbf{\textit{SKIP}} & \\
 & | & a \rightarrow P(s) & (\textit{prefixo}) \\
 & | & P(s) & (\textit{recursao}) \\
 & | & g \;\&\; P(s) & (\textit{escolha condicional}) \\
 & | & P(s) \;\Box\; P(s) & (\textit{escolha externa}) \\
 & | & P(s) \;\sqcap\; P(s) & (\textit{escolha interna}) \\
 & | & P(s) \backslash \mathrm{C} & (\textit{internalizacao}) \\
 & | & P(s)[\mathrm{R}] & (\textit{renomeacao}) \\
 & | & P(s) \;;\; P(s) & (\textit{composicao sequencial}) \\
 & | & P(s) \;\triangle\; P(s) & (\textit{interrupcao}) \\
 & | & P(s) \;\underset{C}{||}\; P(s) & (\textit{paralelismo})
\end{array}
$$

Figura 2.1 Equações de processos

2.1.1 Processos Primitivos

CSP possui dois processos especiais, considerados primitivos: ***STOP*** e ***SKIP***. O processo ***STOP*** representa um estado problemático de um sistema (o sistema quebrou), e também pode ser usado para representar um *deadlock*. Por outro lado, o processo ***SKIP*** representa o termino de uma execução com sucesso.

Embora a literatura classifique outros processos como especiais, como *DIV* e *RUN*, estes não são considerados primitivos, desde que são construídos através de outros processos.

O processo *DIV* representa um estado de *livelock*, e pode ser simulado através de um processo que executa ações internas indefinidamente. Estas ações não são percebidas por outros processos ou pelo ambiente. Sua definição é basicamente $DIV = DIV$.

O processo *RUN* representa um processo que aceita sincronizar com qualquer evento em qualquer instante de tempo, e volta a comporta-se como *RUN* novamente.

2.1.2 Prefixo

Para construir um processo através de seus eventos usamos o operador $\rightarrow$ (chamado operador de prefixo). O operador de prefixo sempre possui um evento do lado esquerdo e um processo do lado direito. O comportamento do processo

$$e \rightarrow P$$

é oferecer o evento e ao ambiente e aguardar indefinidamente até que o ambiente esteja pronto para sincronizar neste evento. Quando isso se dá, o processo passa a comporta-se como P.

Graças à composicionalidade de processos em CSP é possível criar um processo com vários prefixos em seqüência, como no exemplo a seguir. Nestes casos o escopo dos parâmetros de entrada se estende pelos eventos subseqüentes.

$$\mathit{valor}?x \rightarrow \mathit{duplica}!(x+x) \rightarrow \textbf{\textit{SKIP}}$$

2.1.3 Recursão

O operador de prefixo é usado para descrever a seqüência de eventos de qualquer processo finito, mas é inviável para processos que apresentam ciclos repetitivos. Daí a necessidade de um mecanismo para representar comportamentos infinitos. Tal mecanismo é a recursão. No exemplo

$$Clock = tick \rightarrow tack \rightarrow Clock$$

o processo *Clock* executa os eventos *tick* e *tack*, nesta ordem, e volta a comportar-se como *Clock*, indefinidamente.

A recursão é útil para definir um processo através de uma única equação, mas também é útil para definir processos que possuem recursão mútua entre si, como no exemplo abaixo.

$$Calculadora = in?x \rightarrow in?y \rightarrow Visor\,(x+y)$$
$$Visor\,(valor) = display!valor \rightarrow Calculadora$$

Se uma equação recursiva é prefixada por um evento, então é chamada *recursão guardada*. Tal classe de recursão é de grande importância em CSP, haja vista que previne a ocorrência de *livelock*, como descrito em mais detalhes na Seção 2.1.6.

Uma alternativa para definir a recursão é através do operador μ. Por exemplo, o processo *Clock* apresentado anteriormente poderia ser representado como

$$Clock = \mu X \bullet tick \rightarrow tack \rightarrow X$$

2.1.4 Escolhas Externa e Interna

Enquanto o operador de prefixo oferece uma única alternativa de execução (processar um evento e em seguida comportar-se como um processo), os operadores de escolha são usados para especificar processos que oferecem mais de uma alternativa de execução em determinado momento. Por exemplo, o processo

$$(a \rightarrow P) \Box (b \rightarrow Q)$$

aceita inicialmente comunicar qualquer um dos eventos a ou b. O evento escolhido será o primeiro que solicitado pelo ambiente, por isso este operador é chamado escolha externa ou determinística. Após a escolha feita pelo ambiente, o processo anterior comporta-se-á como o processo P ou como o processo Q, dependendo de qual evento inicial tenha sido escolhido. O operador $\Box$ é chamado de escolha externa ou determinística.

Por outro lado, o processo

$$(a \rightarrow P) \sqcap (b \rightarrow Q)$$

decide internamente qual dos eventos a ou b será disponibilizado para sincronização, independente do ambiente externo. Isto significa que o ambiente não tem influência na escolha, e qualquer um, a ou b, pode ser oferecido ou recusado para sincronização, de forma imprevisível. O operador $\sqcap$ é chamado escolha interna ou não-determinística.

Uma situação curiosa ocorre quando os eventos iniciais de uma escolha externa são os mesmos. Apesar de se tratar de um operador determinístico, tal característica introduz um comportamento não-determinístico. Por exemplo, o processo

$$(a \rightarrow P) \Box (a \rightarrow Q)$$

tem o mesmo comportamento do processo

$$(a \rightarrow P) \sqcap (a \rightarrow Q)$$

haja vista que na ocorrência do evento a, não se sabe qual foi a escolha do ambiente.

Comportamentos não-determinísticos em geral são situações indesejáveis em sistemas complexos e concorrentes, porque representam imprevisibilidade ou falhas de modelagem. Entretanto, o operador $\sqcap$ (escolha interna) pode ser usado para abstrair detalhes internos do comportamento de processos, como condições de controle não modeladas.

2.1.5 Escolha Condicional

Além das escolhas apresentadas na seção anterior, CSP possui ainda escolhas condicionais baseadas em variáveis introduzidas através de parâmetros de processos ou comunicações de entrada. Assim, estas variáveis podem ser utilizadas para determinar o comportamento dos processos, através de expressões lógicas. O construtor condicional básico é o seguinte:

$$\mathit{if}\ (g)\ \mathit{then}\ P\ \mathit{else}\ Q$$

que comporta-se como P se a expressão lógica g for verdadeira, ou como Q, se g for falsa.

Em CSP, a construção

$$\mathit{if}\ (g)\ \mathit{then}\ P\ \mathit{else}\ \boldsymbol{STOP}$$

pode ser abreviada para $g\ \&\ P$. Também é possível representar a expressão $\mathit{if}\ (g)\ \mathit{then}\ P\ \mathit{else}\ Q$ como $P\ \langle g \rangle\ Q$.

2.1.6 Internalização de Eventos

Em algumas circunstâncias é importante que ações internas de sistemas não devam ser visualizadas ou sofrer interferência do ambiente externo. CSP dispõe do operador de internalização (*hiding*) para esconder eventos de processos, e torná-los invisíveis e incontroláveis ao ambiente. Estes eventos continuam ocorrendo no processo, apenas o ambiente externo não pode vê-los. O operador recebe como parâmetros um processo e o conjunto de eventos que devem ser escondidos. Assim, o processo

$$P \setminus \{a\}$$

comporta-se exatamente como o processo P, exceto que seus eventos a não são visualizados externamente.

Através do operador de *hiding* é possível construir o processo *DIV* mencionado anteriormente. Para isto basta que um processo recursivo tenha todo o seu alfabeto escondido. No exemplo a seguir o processo Q comporta-se como a recursão infinita $P = P$, que será interpretada pelo ambiente externo como um *livelock*.

$$P = a \rightarrow P$$
$$Q = P \setminus \{a\}$$

2.1.7 Renomeação de Eventos

O operador de renomeação é usado para mudar a representação dos eventos de um processo para o ambiente, sem mudar sua representação interna no processo. O operador recebe como parâmetros um processo e uma relação do tipo $(\Sigma \times \Sigma)$. Por exemplo, o processo

$$P[a \leftarrow b, b \leftarrow a]$$

comporta-se exatamente como P, mas a relação de mapeamento garante que os eventos a e b de P serão percebidos externamente como b e a, respectivamente. A renomeação é especialmente interessante para promover reuso, porque permite criar cópias de processos com alfabetos diferentes.

2.1.8 Composição Seqüencial

O operador de composição seqüencial permite que dois processos sejam executados segundo uma ordem de precedência. O segundo processo deve ser executado apenas após o término com sucesso do primeiro processo. Por exemplo, o processo

$$Q;\ R$$

comporta-se inicialmente como o processo Q. Após o término com sucesso de Q (identificado quando este passa a comportar-se como ***SKIP***) o processo $Q;\ R$ passa a comportar-se como R.

Diferente do operador de prefixo, que permite eventos consecutivos compartilharem o escopo de uma mesma variável, o operador de composição seqüencial não permite a extensão do escopo das variáveis do primeiro processo para o segundo. Assim, na composição seqüencial

$$(a?x \rightarrow \textbf{\textit{SKIP}});\ P$$

a variável x não será percebida pelo processo P.

2.1.9 Interrupção

Similar ao operador de composição seqüencial, o operador de interrupção é útil para impor uma ordem de prioridade entre dois processos. O processo

$$Q \triangle R$$

comporta-se como Q até que o processo R o interrompa, a partir de onde o processo comporta-se como R. A interrupção ocorre quando o ambiente sincroniza com qualquer evento que R ofereça. Se Q e R oferecem os mesmos eventos ao ambiente, então ocorre uma escolha não-determinística entre eles.

2.1.10 Paralelismo

Até aqui os processos descritos têm representado ações sequenciais. Mesmo os processos que oferecem alternativas de execução (externa ou interna) determinam que apenas um fluxo de execução seja escolhido. Através do paralelismo é possível executar mais de um processo simultaneamente, possivelmente havendo comunicação entre eles.

Sejam X e Y os alfabetos de P e Q, respectivamente, então o processo

$$P \;{}_X||_Y\; Q$$

executa os processos P e Q sincronamente em todos os eventos do conjunto $X \cap Y$. Se um dos eventos de $X \cap Y$ for aceito por apenas um dos processos, ele não poderá ocorrer em qualquer dos dois. Este operador é chamado paralelismo alfabetizado, e permite que eventos fora da interseção dos alfabetos possam ser executados independentemente.

Uma outra variação do operador de paralelismo, chamada *interleaving*, permite compor processos em paralelo, sem que haja interações entre eles. Cada evento oferecido ao *interleaving* de dois processos ocorre apenas em um deles. Caso ambos estejam dispostos a aceitar um mesmo evento, a escolha entre os processos é não-determinística. Esse tipo de combinação é especificado da seguinte forma:

$$P \;|||\; Q$$

O comportamento do *interleaving* de dois processos é idêntico ao paralelismo alfabetizado de dois processos, quando a interseção dos alfabetos é um conjunto vazio.

Os dois operadores de paralelismo apresentados acima podem ser representados por um único operador, chamado paralelismo generalizado. Através deste operador basta informar o conjunto de sincronização, que contém os eventos que devem ocorrer simultaneamente nos processos participantes. Os eventos fora deste conjunto são executados como no *interleaving*. Abaixo a representação dos operadores de paralelismo alfabetizado e *interleaving* através do paralelismo generalizado.

$$\begin{aligned} P \;{}_X||_Y\; Q \quad &\equiv \quad P \;\underset{X \cap Y}{||}\; Q \\ P \;|||\; Q \quad &\equiv \quad P \;\underset{\{\}}{||}\; Q \end{aligned}$$

2.1.11 Operadores Indexados

A sintaxe de CSP permite construir processos complexos a partir de um conjunto de outros processos e de um operador indexado por este conjunto.

O processo do exemplo a seguir representa o interleaving dos processos P_1 a P_N.

$$|||_{i:1...N} \bullet P_i$$

Nesta dissertação consideramos a versão expandida de processos indexados, ou seja, a aplicação do operador em questão para cada processo envolvido, separadamente. Desta forma o exemplo acima seria representado apenas por

$$P_1 \;|||\; (P_2 \;|||\; (...P_N)...)$$

2.2 Prova de propriedades e Refinamentos

Como uma álgebra de processos, CSP possui um conjunto de leis algébricas que permitem provar equivalências semânticas (através de refinamentos) entre processos sintaticamente diferentes. Algumas destas leis podem ser usadas para reescrever a definição de processos, tornado-os mais simples ou atendendo algum padrão estrutural, sem no entanto mudar o comportamento do processo original. A leis algébricas também permitem a verificação com rigor matemático de propriedades clássicas de sistemas concorrentes e distribuídos, como o determinismo e a ausência de *deadlock* ou *livelock*. Abaixo exemplificamos duas destas leis [41]:

$$P \,\Box\, \textbf{\textit{STOP}} \equiv P$$
$$P \;|||\; \textbf{\textit{SKIP}} \equiv P$$

CSP usa modelos semânticos para estabelecer relações de refinamento entre processos, a partir de onde as leis podem ser aplicadas. Os principais modelos semânticos são *Traces*, *Falhas* e *Falhas e Divergências*.

O modelo de *Traces* é útil para encontrar sequências finitas de eventos que um processo pode executar, e verificar até que ponto um padrão comportamental é atendido. Um trace de um processo é um registro de uma sequência finita de eventos que o processo já aceitou até o momento. Por exemplo:

- $< x, y >$ - é um trace de um processo que aceita o evento x, seguido do evento y;
- $< tick >$ - é o trace de processo ***SKIP***;
- $<>$ - é o trace de um processo que ainda não aceitou evento algum. A semântica denotacional de CSP exige que todo processo possua este trace.

Entretanto, o modelo de *Traces* não é capaz de identificar os eventos que um processo não pode executar num determinado instante. Assim, segundo este modelo, o processo $P \,\Box\, Q$ é equivalente ao processo $P \sqcap Q$, visto que ambos podem, eventualmente, ter o mesmo comportamento.

O modelo de *Falhas* é mais poderoso que o de *Traces* porque permite identificar a sequência de eventos que um processo pode realizar, juntamente com o conjunto de eventos que ele pode recusar-se a executar naquele momento. Assim, este modelo permite demonstrar que o processo $P \,\Box\, Q$ não é semanticamente equivalente ao processo $P \sqcap Q$, se considerarmos o conjunto de eventos que podem não ser aceitos. Através do modelo de *Falhas* também é possível verificar se um processo é determinístico ou não. Um processo é dito determinístico se ele não se comporta diferentemente a partir da mesma situação inicial.

O modelo de *Falhas e Divergências* é ainda mais poderoso que o de *Falhas*, e conseqüentemente que o de *Traces*. Este modelo identifica todas os traces que podem levar um processo a comportar-se como uma divergência (*DIV*), ou *livelock*. Um exemplo deste tipo de comportamento é o de um processo que executa uma sequência infinita de eventos ocultos (ou não observáveis), como o processo D abaixo.

$$D = (a \rightarrow b \rightarrow D) \setminus \{a, b\}$$

Os eventos a e b ocorrerão infinitamente e sem influência nem observação do ambiente, não permitindo que o processo D comunique-se com o ambiente.

Relações de refinamento formalizam a idéia de implementação correta com respeito à especificação: se uma implementação é um refinamento de uma especificação, então é possível provar que a implementação satisfaz as propriedades (requisitos) descritas na especificação.

2.3 Ferramentas de Verificação e Animação de Modelos CSP

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (Failures and Divergences Refinament) [19]. O ProBE [3] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSP_M [19], que é um acrônimo para *machine readable CSP*. Esta linguagem tem sintaxe adaptada para representar as construções de CSP, com pequenas variações. Os canais de comunicação, por exemplo, devem ser explicitamente declarados, assim como o tipo dos dados que estes podem transmitir em seus eventos. É possível inclusive definir canais multidimensionais para transmitir mais de um dado simultaneamente.

CAPÍTULO 3

UML-RT

As abordagens tradicionais de UML usam diagramas estáticos (classes, objetos e componentes) e dinâmicos (estados, atividades e interações) para representar a estrutura e o comportamento de cada elemento do sistema ou de cenários específicos da aplicação. Apesar de serem de propósito geral, estes diagramas possuem poucos mecanismos para detalhar características internas de componentes e propriedades como concorrência, interação e distribuição.

A solução mais empregada para especializar a notação UML é a criação de profiles para contextos específicos de aplicação [32]. A utilização de novos esteriótipos ou a adaptação de diagramas facilitam a visualização de algumas propriedades de sistemas, geralmente mal representadas através dos elementos padrão de UML. UML-RT [44, 31], desenvolvido pela ObjecTime Limited e Rational Corporation, é um profile conservativo de UML 1.5 proposto para a modelagem de sistemas baseados em componentes concorrentes, reativos e possivelmente distribuídos. Embora o nome *Real-Time* sugira a presença de conceitos de tempo real, como restrições de tempo e requisitos não-funcionais, este profile é mais adequado para modelar a *arquitetura* de sistemas distribuídos, com a facilidade para representar comportamentos reativos e concorrentes [15].

UML-RT adiciona elementos de modelagem e formalismos de ROOM (*Real-Time Object-Oriented Modeling*) [31], que é uma notação visual orientada a objetos específica para modelagem de sistemas de tempo real, distribuídos e reativos. Em ROOM, cada componente (ou ator) de um sistema é modelado como uma entidade independente das demais, com estrutura e comportamento próprios. A estrutura interna de um componente pode ser composta por outros componentes e por pontos de comunicação (portas). A interface de comunicação com o ambiente externo é composta por portas públicas. O comportamento interno de um componente é definido através de máquinas de estado (ROOMCharts) baseadas no formalismo visual Statechart [21], e por isso têm potencial para serem usadas não apenas para especificação, mas também para geração do código destes componentes [43]. A comunicação entre componentes é feita através de mensagens, que trafegam pelas portas de componentes conectados e obedecem algum protocolo de comunicação.

UML-RT incorporou os conceitos de ROOM através de novos esteriótipos e da adaptação de alguns diagramas convencionais de UML. Atores são representados através do novo esteriótipo *Cápsula*, às quais pode-se associar adaptações dos diagramas de colaboração e de máquinas de estados. O novo diagrama de estados possui as características das máquinas de estados finitos de ROOM, e o diagrama de colaboração, chamado diagrama de estrutura, permite visualizar a configuração interna de uma cápsula. O esteriótipo *Protocolo* é usado para definir as regras de comunicação entre portas de cápsulas, que são nada mais que instâncias de protocolos.

Como mencionado anteriormente, UML-RT não é um padrão ideal para especificar siste-

mas críticos e de tempo real [20]. Entretanto, muitos autores concordam que a notação possui mecanismos para modelar algumas das principais características arquiteturais destes sistemas de forma satisfatória [8, 20], seja pela aplicabilidade herdada de UML ou pelo formalismo específico de contexto de ROOM, além é claro das facilidades que as ferramentas de suporte oferecem [43, 39]. UML-RT permite que propriedades como concorrência, interação e condições de controle sejam facilmente identificadas em seus modelos. A estratégia adotada para a comunicação entre cápsulas é a de passagem de mensagens, que fortalece a independência e o encapsulamento de componentes, e é adequada para representar sistemas de software ou hardware, distribuídos ou não [43].

UML-RT também foca mais em modelagem comportamental que UML, facilitando o uso da técnica desde as primeiras fases do desenvolvimento (análise) até a codificação. Isto reduz as descontinuidades conceituais nas mudanças de fases e fortalece o entendimento e a validação dos requisitos. Em [4], argumenta-se que máquinas de estado devem ser usadas inclusive para detalhar Use Cases, da mesma forma que diagramas de seqüência. Neste caso, o sistema é visto sob uma perspectiva caixa-preta, e os estados da máquina de estados correspondem a estados ou condições de todo o sistema, e não apenas a estados de um componente isolado.

3.1 Cápsulas

Cápsulas correspondem ao conceito de ator, ou componente ativo, de ROOM. Em UML-RT, cápsulas são classes ativas, que possuem seu próprio *thread* de controle lógico, e cujo fluxo de controle é sensível apenas à troca de mensagens com outras cápsulas. Estas mensagens podem apenas ser transmitidas e recebidas através das portas de uma cápsula. Cápsulas podem ainda conter instâncias internas de outras cápsulas. Estas instâncias, ou sub-cápsulas, são fortemente acopladas à cápsula que as contém, e não podem existir independentemente desta cápsula. A execução das sub-cápsulas é simultânea à da cápsula *container*.

Cápsulas também possuem métodos e atributos, como classes ordinárias, mas que são visíveis apenas ao contexto interno da cápsula. Ou seja, todos os métodos e atributos de uma cápsula são necessariamente protegidos. A representação visual de uma cápsula é similar à de classes ordinárias, mas contêm o esteriótipo *«Capsule»* e um compartimento adicional para portas (Figura 3.1).

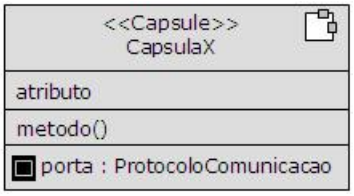


Figura 3.1 Representação de cápsulas em diagramas de classes

3.2 Portas

Portas são a única interface de comunicação das cápsulas com o ambiente externo ou suas sub-cápsulas. Portas fortalecem o encapsulamento e o reuso de cápsulas, porque permitem que estas sejam usadas apenas em função das suas interfaces, sem necessidade de conhecimento de suas características internas. Da mesma forma, as cápsulas não precisam conhecer as características do ambiente externo ou de suas sub-cápsulas.

Cada porta deve realizar algum protocolo de comunicação, que define o tipo e a sequência válida de mensagens que podem ser comunicadas pelas portas. O termo *evento* significa a recepção de uma mensagem pela porta. Portas públicas são diferenciadas do lado de fora da cápsula apenas pelo protocolo que as definem, mas internamente as portas podem ainda ser classificadas como *end* ou *relay*.

- Porta *End*: São conectadas diretamente à máquina de estados da cápsula, que pode identificar os eventos que ocorrem na porta e ler suas mensagens. As portas ***p2*** e ***p3*** da Figura 3.4 são exemplos de portas *end*.

- Porta *Relay*: São conectadas a portas de sub-cápsulas, e seus eventos não são percebidos pela máquina de estados da cápsula que a contém. O uso destas portas se assemelha ao conceito de delegação, já que seus eventos são totalmente encaminhados a outro elemento. A porta ***p1*** da Figura 3.4 é um exemplo de porta *relay*.

3.3 Protocolos

UML-RT usa o esteriótipo *«Protocol»* para identificar todas as classes que definem as propriedades de portas que as implementam. Protocolos são classes abstratas puramente comportamentais, diferente de cápsulas, que são elementos estruturais. Em um mesmo protocolo é possível definir mais de uma interface, chamada *papel*, que será implementada pela porta.

No padrão ROOM original, protocolos podem definir mais do que dois papéis. Entretanto, as ferramentas de apoio a UML-RT geralmente usam protocolos binários, envolvendo apenas dois papéis. Neste caso apenas um papel, chamado *Base*, precisa ser definido. O outro, *Conjugado*, é automaticamente derivado da inversão de sinais do papel *Base*. Portas podem realizar um único papel. A especificação de UML 2.0 [35] define que protocolos multi-papéis são permitidos, mas não deixa claras as restrições à implementação de papéis pelas portas.

Cada papel de um protocolo define os sinais de entrada e de saída que podem ser recebidos ou enviados por uma porta, respectivamente. O tipo do sinal determina exatamente o tipo das mensagens que serão transmitidas pelas portas através daquele sinal. A Figura 3.2 mostra a representação visual do protocolo binário *ProtocoloComunicacao*, onde apenas os sinais do papel *base* são informados. Neste exemplo, dois sinais são declarados: o sinal de saída *enviarRequisicao*, que carrega apenas mensagens do tipo *X*, e o sinal de entrada *receberResposta*, cujas mensagens são do tipo *Y*.

O papel também define a sequência válida de utilização de seus sinais. Esta sequência é definida através de máquinas de estado, e deve representar o padrão comportamental para cada

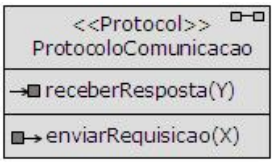


Figura 3.2 Representação de protocolos em diagramas de classes

porta que implemente aquele papel em uma cápsula. A Figura 3.3 mostra a máquina de estados do papel *base* do protocolo *ProtocoloComunicacao*, cujas portas devem enviar mensagens usando o sinal *enviarRequisicao*, e receber mensagens usando o sinal *receberResposta*, nesta ordem. As cápsulas que contiverem portas que implementam este papel devem garantir que estas, quando usadas, atendam à ordem definida anteriormente, mesmo que a ocorrência dos sinais seja intercalada com eventos de outras portas ou ações internas. Assim, máquinas de estado de protocolo são usadas para indicar às cápsulas as regras de comunicação específicas de cada porta, mas não podem pré-determinar o comportamento interno das cápsulas.

A ausência da máquina de estados do papel indica que não há restrições quanto ao uso de portas daquele papel.

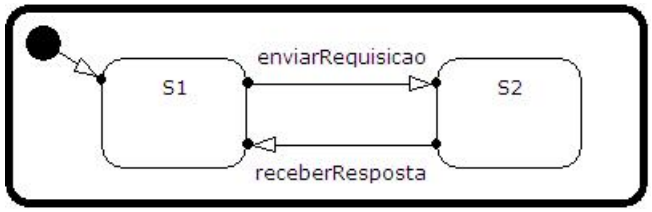


Figura 3.3 Máquina de estados do papel *base* de *ProtocoloComunicacao*

3.4 Modelando a estrutura

A estrutura de um sistema é definida através de instâncias cooperantes de seus elementos, cada uma desempenhando um papel no sistema. Cada elemento do sistema tem sua própria estrutura interna. A estrutura interna de cápsulas pode ser composta de sub-cápsulas (instâncias protegidas de outras cápsulas) e portas de comunicação, que podem ser públicas ou protegidas. Portas públicas compõem a interface de comunicação com o ambiente externo; portas protegidas são usadas para conectar sub-cápsulas e comunicar eventos internos. Em UML-RT, a estrutura interna de cápsulas é definida através de diagramas de colaboração especializados, chamados diagramas de estrutura. Estes diagramas são limitados à representação visual de portas e sub-cápsulas apenas no contexto da cápsula modelada, e não representam a colaboração geral entre as cápsulas que dão origem às sub-cápsulas. A colaboração entre cápsulas e seus atributos (que não são portas ou sub-cápsulas) deve ser modelada com diagramas de colaboração padrão.

A condição básica para que duas cápsulas se comuniquem é que exista uma *conexão* estrutural entre suas portas. Esta conexão representa o meio de propagação de mensagens entre

portas, e é representada no diagrama de estrutura por uma linha, sem orientação, entre duas portas de cápsulas. Apenas portas que possuam protocolos complementares ou simétricos podem ser conectadas. Protocolos complementares possuem sinais de saída e entrada invertidos um em relação ao outro. Protocolos simétricos possuem o mesmo conjunto de sinais, seja em tipo de dado ou orientação. Em protocolos binários, portas *base* e *conjugadas* são complementares entre si. Portas *conjugadas* são ilustradas por caixas brancas. Portas *base* são ilustradas por caixas pretas.

A Figura 3.4 mostra a cápsula ***C*** contendo duas sub-cápsulas, ***C1*** e ***C2***. A porta *relay*, pública e *conjugada* ***p1*** transmite mensagens do ambiente externo diretamente para ***C1***, e vice versa. A porta *end*, *base* e pública ***p2*** é usada para conectar a cápsula ***C*** diretamente ao ambiente externo. As mensagens transmitidas por esta porta serão diretamente identificadas pela máquina de estados de ***C***. A porta *end*, *conjugada* e protegida ***p3*** não se comunica com o ambiente externo, mas permite que a máquina de estados de ***C*** troque mensagens com a cápsula ***C2***. Adicionalmente, a conexão entre as duas portas de ***C1*** e ***C2*** garante que estas troquem mensagens entre si, sem a interferência de agentes externos.

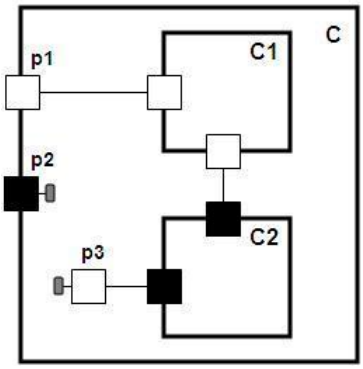


Figura 3.4 Diagrama de estrutura de cápsula

Toda cápsula é responsável pela utilização de todas as suas partes componentes. Geralmente estas partes são criadas e destruídas ao mesmo tempo em que a cápsula também é, embora sejam possíveis variações quanto ao seu uso. Cápsulas podem ser usadas como:

- Fixa: Uma instância da cápsula (ou várias instâncias, dependendo da cardinalidade) é automaticamente criada ou destruída quando a cápsula que a contém também é.
- Opcional: A cápsula é instanciada quando necessário. A ação de instanciar a cápsula é executada pela máquina de estados da cápsula que a contém.
- *Plug-in*: Uma instância já existente no sistema é dinamicamente "plugada" à estrutura de outra cápsula, e passa a agir como sua sub-cápsula, sem que seu estado interno seja alterado ou reiniciado. A ação de referenciar uma instância externa é executada pela máquina de estados da cápsula que a conterá.

Para usar instâncias opcionais (ou *plug-in*) é preciso definir uma referência opcional (ou *plug-in*) no diagrama de estrutura. Estas referências, embora sejam visualmente idênticas às

instâncias fixas, não são instâncias ativas de cápsulas, mas apontadores para as futuras instâncias que serão criadas (ou importadas) no modelo. O tipo da referência deve ser compatível com o tipo da instância que será criada, e todas as portas da referência devem ser mapeadas nas portas da instância. O número de instâncias que venham a ser criadas deve ser igual ou menor à cardinalidade da referência.

Se alguma porta estiver conectada a uma cápsula opcional (ou *plug-in*) ainda não ativa, todos os seus eventos serão perdidos.

Para facilitar a visualização da estrutura das cápsulas durante sua execução, adotamos o padrão de cápsulas tracejadas representando as instâncias ativas no momento, independente de serem fixas, opcionais ou *plug-in*. A Figura 3.5 exibe a cápsula *P* contendo uma instância ativa de *Q*.

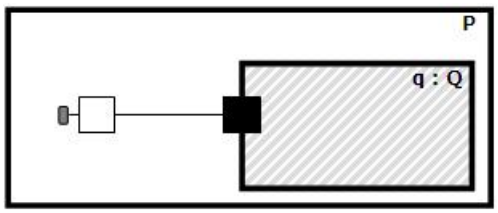


Figura 3.5 Cápsula *P* contendo instância ativa de *Q*

Portas também podem ser conectadas dinamicamente. Portas fixas (*wired*) são explicitamente conectadas no diagrama de estrutura, e não perdem sua conexão até que a cápsula que as contém seja destruída. Portas dinâmicas (*unwired*) são conectadas em tempo de execução. A conexão entre portas *wired* é estática, e por isso é visualizada diretamente no diagrama de estrutura. Já a conexão entre portas *unwired* é estabelecida dinamicamente, e por isso não é visível no mesmo diagrama. O comportamento de portas *unwired* conectadas é idêntico ao de portas *wired* conectadas.

O uso dinâmico de cápsulas e conexões entre portas permite que sistemas complexos sejam modelados de uma forma mais realista, considerando configurações dinâmicas e decisões em tempo de execução.

3.5 Modelando o comportamento

Cápsulas têm, além de seu comportamento passivo definido pelos métodos, um comportamento ativo definido por sua máquina de estados. Na verdade, este comportamento é reativo, porque depende de eventos externos percebidos pela máquina de estados. Como mencionado anteriormente, apenas os eventos que ocorrem em portas *end* são percebidos pela máquina de estados da cápsula.

Uma máquina de estados é um grafo direcionado de estados, conectados por transições. As transições são disparadas quando o evento que as definem ocorre. Um evento é definido como a ocorrência de um sinal de entrada em uma porta. Uma transição entre estados é da forma $p.s[g]/a$, onde p é a porta onde ocorre o evento, s é o sinal de entrada, g é uma expressão lógica (guarda) e a é uma ação, que deve ser executada se a guarda g for verdadeira e o evento $p.s$

ocorrer. A ausência de guarda equivale a uma guarda *True*, e implica que não há restrições para a ocorrência do evento naquela transição. Algumas transições não possuem evento associado, e por isso são ditas automáticas.

Eventos podem ainda carregar uma mensagem. Nesta dissertação, por compatibilidade com CSP, usamos a notação $p.s?x/a$ para representar a transição que aceita a mensagem x através do evento $p.s$, e em seguida executa a ação a. A ação associada a uma transição pode incluir o envio de uma mensagem através de uma porta *end* da cápsula. A notação $p.s!y$ é usada para representar o envio da mensagem y através do sinal de saída s da porta p.

A Figura 3.6 mostra a máquina de estados de uma cápsula. Convencionamos usar o nome S_P para representar a própria máquina de estados da cápsula P. Estados podem ser decompostos em sub-estados, como o estado S_3. O círculo preto S_0 representa o estado inicial da máquina de estados, e sua transição é automática (ocorre implicitamente quando a cápsula é criada). O círculo preto S_{30} representa o estado inicial do estado composto S_3, e sua transição também é automática (ocorre implicitamente quando o estado S_3 é ativado, através do disparo de uma transição que o tenha como destino). Transições iniciais são geralmente usadas para configuração interna, e podem ter uma ação associada. O círculo preto menor, na borda do estado S_3, representa um ponto de junção. Pontos de junção são pseudo-estados, usados para concatenar transições na borda de estados compostos, formando transições compostas. A transição de saída do ponto de junção é automática (o evento que a dispara é a chegada da transição de entrada). Pontos de escolha (círculo com a letra "C") seguem o mesmo princípio, mas executam uma expressão lógica (condição de escolha) para decidir qual transição de saída executar. Se a expressão lógica for verdadeira, a transição do ponto de junção T é disparada, caso contrário a transição do ponto de junção F é disparada.

Em UML-RT, máquinas de estado não possuem estados finais, porque cápsulas são consideradas elementos ativos, que nunca acabam seu processamento, a não ser que o ambiente externo decida destruí-la.

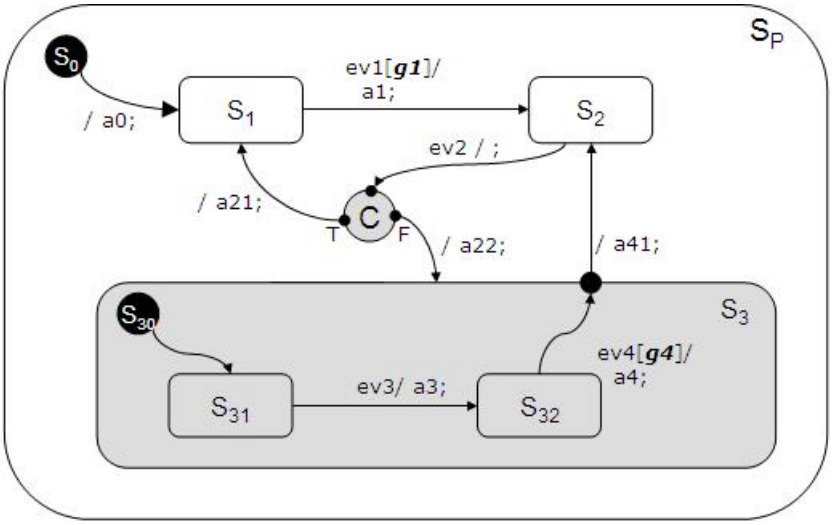


Figura 3.6 Máquina de estados de cápsula

Conceitualmente, a guarda de uma transição deve ser avaliada antes da ocorrência do evento

que dispara aquela transição, mesmo que este evento não venha a ocorrer. Se a expressão lógica da guarda for falsa, então a ocorrência do evento de disparo está bloqueada. Por conta disto, a guarda não pode conter referências a um valor comunicado pelo evento, tampouco deve ter efeitos colaterais na cápsula. Já as expressões lógicas de pontos de escolha são avaliadas se e quando o ponto de escolha é alcançado. Em outras palavras, para a execução de expressões lógicas, um ponto de escolha tem o mesmo efeito de um estado.

Estados compostos podem ainda ser divididos em sub-estados concorrentes (ou ortogonais), também chamados de regiões. As transições de uma região são independentes das demais regiões ortogonais, embora possam ser disparadas pelos mesmos eventos. A Figura 3.7 mostra um estado composto de duas regiões concorrentes. O evento *p.s* dispara simultaneamente transições nas duas regiões, se suas respectivas guardas forem verdadeiras. Se a guarda de uma das transições for falsa, a outra transição não é afetada [34]. Na verdade, a definição de máquinas de estados em UML-RT não determina a semântica exata de transições em regiões concorrentes. Não é claro, por exemplo, se é possível estabelecer dependências entre transições de regiões concorrentes.

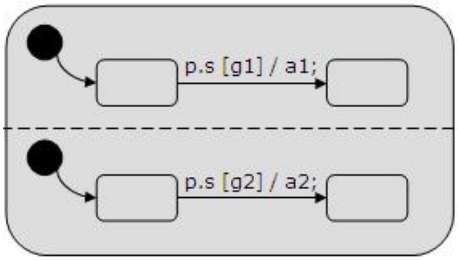


Figura 3.7 Estados concorrentes

Assim como nas máquinas de estado de ROOM, o conceito de estados concorrentes geralmente é traduzido para comunicação concorrente entre cápsulas. Isso se justifica porque a utilização de estados concorrentes pode gerar representações ambíguas ou comportamentos indesejáveis entre estados. Por outro lado, através de cápsulas concorrentes a representação de processamento paralelo e comunicação torna-se explícita [43]. A Figura 3.8 mostra dois estados ortogonais (à esquerda) e sua composição similar usando cápsulas distintas (à direita). Nesta dissertação consideramos o uso de cápsulas concorrentes ao invés de estados concorrentes.

3.6 Comunicação

Cápsulas não possuem métodos ou atributos públicos. O forte encapsulamento de cápsulas elimina a possibilidade de compartilhar variáveis ou chamar operações de outras cápsulas diretamente, mas favorece a passagem de mensagens como padrão de comunicação. A abordagem é vantajosa porque permite modelar sistemas distribuídos ou centralizados, concorrentes ou sequenciais, e software ou hardware.

Em UML-RT, as mensagens podem ser transmitidas síncrona ou assincronamente. Geralmente, sistemas de tempo-real e interativos usam as duas formas de comunicação: eventos

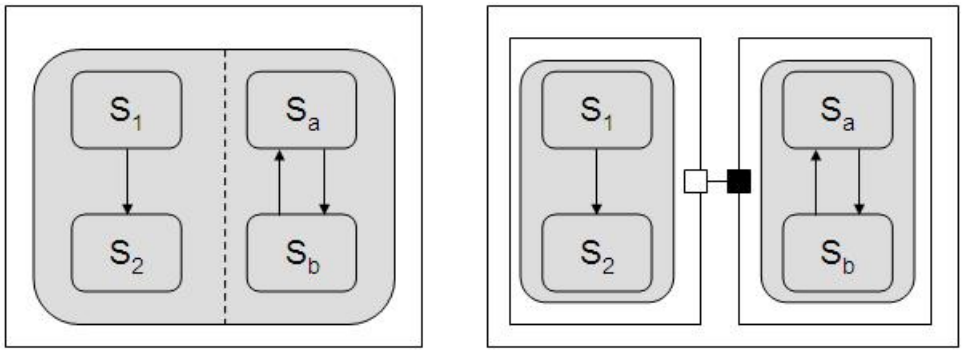


Figura 3.8 Representação de estados concorrentes como cápsulas concorrentes

externos costumam ser assíncronos, e alguns eventos internos são síncronos [6]. A comunicação assíncrona usa estruturas de dados (como *buffers*) para armazenar as mensagens comunicadas pelo emissor, até que o receptor possa processá-las. O receptor processa as mensagens uma por vez, na ordem em que foram armazenadas, sem restrições de tempo. Desta forma, o emissor permanece livre para executar outras tarefas. A ordem em que as mensagens são inseridas no *buffer* não é definida, deixando aberta a possibilidade para modelar diferentes esquemas de prioridade entre mensagens. Já no modo de comunicação síncrono, a cápsula emissora permanece bloqueada até que seu receptor confirme a recepção da mensagem comunicada. A estratégia usada para implementar a comunicação síncrona em UML-RT é a de *rendezvous*[RoseRT,UMLsuper:04].

Os diagramas de estrutura de UML-RT não diferenciam conexões síncronas de assíncronas, e cabe à máquina de estados da cápsula enviar mensagens do modo desejado. O modo de comunicação não depende do tipo das portas, mas da forma como as mensagens são enviadas. A Figura 3.9 mostra a forma como duas cápsulas se comunicam assincronamente. A cápsula ***C1*** envia a mensagem ***msg*** usando o sinal ***in*** da porta ***p***, através da primitiva ***send***. A mensagem é armazenada no *buffer* de mensagens da cápsula ***C2***, junto com as informações da porta e do sinal utilizados. Quando pronta, a cápsula ***C2*** avalia as informações no buffer. Se a guarda *g* for verdadeira, o evento *q.in*?*msg* é disparado e a mensagem removida do *buffer*. Se a guarda *g* for falsa, a mensagem é descartada. Enquanto isso, a cápsula ***C1*** já concluiu sua transição e opera independente de ***C2***.

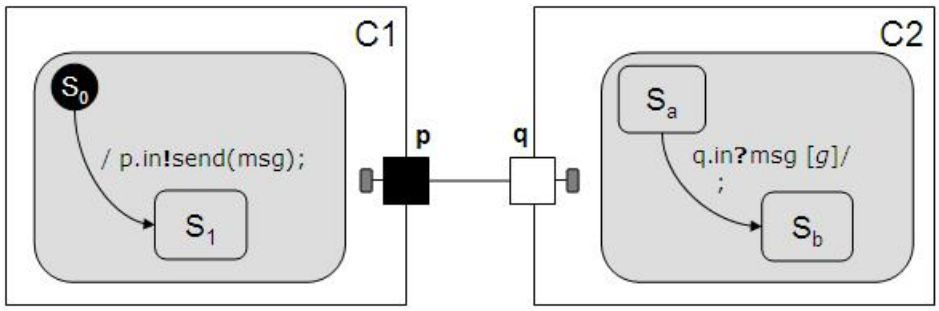


Figura 3.9 Comunicação assíncrona entre cápsulas

A Figura 3.10 mostra duas cápsulas comunicando-se sincronamente. A cápsula ***C1*** envia a mensagem ***msg*** usando o sinal ***in*** da porta ***p***, através da primitiva ***invoke***. ***C1*** permanece bloqueada até que a cápsula ***C2*** processe a mensagem comunicada, e envie uma mensagem de resposta, através da primitiva ***reply***. Esta mensagem de resposta é enviada através de um sinal de saída da mesma porta por onde a primeira mensagem foi recebida, ***q***. Na cápsula emissora, o processamento do comando *invoke* retorna a mensagem enviada pelo comando *reply* do seu receptor. Se a cápsula receptora não enviar a mensagem de resposta (porque a guarda *g* foi falsa, e a mensagem foi descartada, ou porque o comando de *reply* foi omitido na transição que consumiu a mensagem), o comando **invoke** retorna uma mensagem nula. Neste caso, o emissor considera que a mensagem síncrona foi perdida, e assume que o receptor não pôde processá-la. UML-RT não especifica a forma como o controle das mensagens de resposta é implementado, mas deixa claro que este controle não deve influenciar a construção das cápsulas.

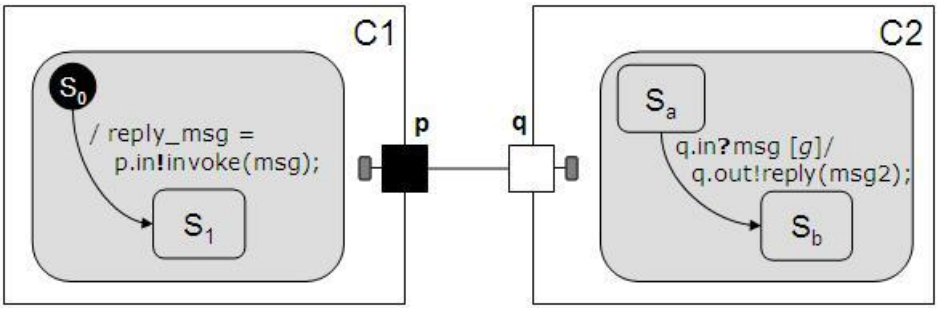


Figura 3.10 Comunicação síncrona entre cápsulas

Na prática, mensagens do tipo *invoke* e *reply* são implementadas usando comunicação assíncrona, mas têm prioridade sobre qualquer mensagem do tipo *send*. Quando a primitiva de comunicação não estiver explícita, subentende-se que o modo de comunicação é o assíncrono.

A semântica de processamento de eventos em UML-RT é fundamentada na estratégia *run-to-completion*, e significa que um evento só pode ser processado por uma cápsula se o processamento do evento anterior estiver concluído.

3.7 Outras notações visuais

Diversas outras notações visuais foram desenvolvidos com o propósito de atender a modelagem de sistemas de tempo-real, concorrentes e distribuídos, como Statecharts [43], MSC [27] e SDL. Outros profiles de UML, como *UML Profile for Schedulability, Performance, and Time Specification* e Real Time UML, são mais indicados para sistemas embarcados com restrições sérias de tempo, mas sem referência clara a componentes.

Embora a versão 2.0 de UML [35] tenha incorporado vários conceitos de UML-RT e de outras notações (que a tornariam inclusive mais completa que UML-RT), seus elementos e diagramas ainda são ambíguos e pouco divulgados. Na verdade, todos os elementos relevantes à representação dos sistemas abordados neste trabalho podem ser utilizados satisfatoriamente com UML-RT, inclusive com ferramentas de apoio sólidas [39], o que ainda não é verdade para

UML 2.0. Outra grande vantagem de UML-RT são os esteriótipos para cápsulas e protocolos, e as noções claras e intuitivas de componentes reativos e independentes, também pouco claros em UML 2.0.

CAPÍTULO 4

Mapeamento de CSP em UML-RT

Neste capítulo mostramos uma estratégia sistemática para traduzir especificações CSP em modelos UML-RT, assim como os detalhes pertinentes ao processo de mapeamento. A estratégia pode ser resumida como um sistema de reescrita, composto de regras independentes e composicionais. A aplicação exaustiva destas regras traduz gradual e sistematicamente tipos de dados e processos CSP em classes, cápsulas e protocolos UML-RT. Embora as provas formais sejam sugeridas como trabalhos futuros, as regras foram idealizadas para preservarem a semântica do modelo original.

Escolhemos representar elementos de CSP usando UML-RT porque esta notação possui facilidades para modelar sistemas reativos e concorrentes. O conceito de processos independentes e comunicantes da álgebra de processos CSP se adequa naturalmente ao de objetos ativos e cooperantes, presentes em UML-RT.

Cada regra de mapeamento se aplica a uma forma de construção em CSP. Para iniciar o processo de mapeamento, dividimos as construções de CSP em dois grupos distintos: tipos de dados e processos.

Consideramos que tipos de dados representam classes de mensagens transmitidas entre elementos do sistema, por isso são mapeados em classes UML simples. Tipos de dados compostos também devem ser mapeados como uma única classe, desde que qualquer instância desta classe possa representar qualquer combinação dos tipos envolvidos na composição. Mais detalhes sobre o mapeamento de tipos de dados são apresentados na Seção 4.3.

A estratégia de mapeamento considera que todas as equações de processos seguem um padrão de normalização. A Seção 4.1 apresenta um conjunto de regras para normalizar as equações de processos.

Cada processo CSP é mapeado em uma cápsula UML-RT de mesmo nome. Processos, assim como cápsulas, representam entidades ativas e independentes, que interagem entre si através de interfaces bem definidas (canais de comunicação) e cujos fluxos de controle são sensíveis à troca de eventos com o ambiente que os cerca. Cápsulas, assim como processos, podem ser compostos para fornecer algum comportamento cooperativo maior que a soma de todas as suas partes [7]. Finalmente, processos e cápsulas atendem aos conceitos de reuso e modularidade, já que podem ser usados para compor estruturas maiores sem que suas definições originais sejam alteradas.

A ocorrência de canais em processos dá origem a portas de mesmo nome em cápsulas, visto que ambos representam abstrações para interfaces de comunicação, através das quais ocorrem os eventos de comunicação com o ambiente externo. Mais detalhes sobre o mapeamento de processos são apresentados na Seção 4.2.

Funções e expressões lógicas descritas em CSP são mapeadas em métodos de cápsulas ou

classes auxiliares, mas a codificação destes métodos está fora do escopo desta dissertação. O corpo de cada método é comentado com a expressão CSP que o originou. Alguns trabalhos relacionados abordam a transformação destes elementos em código executável, como Java [17, 16].

4.1 Normalização de Processos

As regras de mapeamento de processos são aplicadas a padrões de equações CSP. Portanto, as equações de processos devem ser simplificadas para que tenham uma das formas a seguir:

$P([s]) = \textbf{\textit{SKIP}}$
$P([s]) = \textbf{\textit{STOP}}$
$P([s]) = N([s'])$
$P([s]) = a?x : C \rightarrow N([s'])$
$P([s]) = \mathit{if}\ (g)\ \mathit{then}\ N_1([s'_1])\ \mathit{else}\ N_2([s'_2])$
$P([s]) = N([s'])\ \textbf{\textit{uop}}\ \mathit{args}$
$P([s]) = N_1([s'_1])\ \textbf{\textit{bop}}\ N_2([s'_2])$

onde N é um nome de processo, exceto processos primitivos (***STOP*** e ***SKIP***); ***uop*** é um operador unário (*internalização* ou *renomeação*); e ***bop*** é um operador binário (*escolha externa*, *escolha interna* , *composição sequencial*, *interrupção* ou *paralelismo*). Os processos ***SKIP*** e ***SKIP***, quando ocorrerem em equações que envolvam algum operador de CSP, devem ser substituídos por nomes de processos.

Assim, o lado direito de qualquer equação de processo pode ser ***STOP***; ***SKIP***; um nome de processo; um prefixo envolvendo uma comunicação e um nome de processo; uma escolha condicional envolvendo dois nomes processos e uma expressão lógica g; um operador unário envolvendo um nome de processo e um conjunto de elementos; ou um operador binário com dois nomes de processos como argumentos.

Na realidade, os processos podem ser parametrizados, como em $P(s) = N(s')$. Nas formas normais acima, os parâmetros são opcionais ([..]) e podem ocorrer à esquerda ou à direita da equação, em nomes de processos.

Assumimos que todo processo parametrizado P é representado como $P(s)$, onde s é o parâmetro de P. Na prática, obviamente, um processo pode ter mais de um parâmetro, porém evitamos este caso por simplicidade das regras. O mapeamento de um parâmetro de processo pode ser estendido a mais de um, ou podemos assumir um parâmetro único (composto) que represente os diversos parâmetros de um processo. Nas regras a seguir, assumimos que processos são sempre parametrizados. Regras para processos sem parâmetros são casos particulares dos apresentados.

A regra a seguir é usada para simplificar equações de processos que usam o operador de prefixo. Considere *Exp* como qualquer expressão CSP que possa ser usada como lado direito da equação de um processo, exceto nomes de processos, pois neste caso a equação já estaria na forma normal.

Regra 1. *Reescrita de Prefixo*

$$P(s) = a?x : C \rightarrow Exp \quad \Longrightarrow \quad \begin{array}{l} P(s) = a?x : C \rightarrow N(s,x) \\ N(s,x) = Exp \end{array}$$

Esta regra cria um processo N, cujo termo é Exp, e troca a ocorrência de Exp em P por N. Como os parâmetros de P e os parâmetros de seus eventos, como x, fazem parte do escopo de Exp, então estes devem ser usados como parâmetros do novo processo N. Regras para outros padrões de comunicação são similares e, portanto, omitidos. □

Por simplicidade das regras de mapeamento, apresentadas na Seção 4.2.3, assumimos que todas as equações de processo do tipo $P(s) = a.s \rightarrow P(s)$ ou $P(s) = a!s \rightarrow P(s)$ são interpretadas como:

$$P(s) = a?s : \{x \mid x = s\} \rightarrow P(s)$$

Já as equações do tipo

$$P(s) = a?x : C_x?y : C_y \rightarrow P(s)$$

onde C_x e C_y são as respectivas restrições sobre x e y, são assumidas estarem representadas como:

$$P(s) = a?m : \{x.y \mid x \in C_x,\ y \in C_y\} \rightarrow P(s)$$

Finalmente, expressões como $a?x.(x+1)$ também podem ser representadas através de

$$a?m : \{x.y \mid y \in \{x+1\}\}$$

. Adotamos esta medida porque os eventos em UML-RT podem carregar um único atributo por vez, embora este possa ser de um tipo composto.

A regra a seguir é usada para simplificar equações de processos que usem algum operador unário, como o de renomeação ou o de internalização. Considere *args* como o segundo parâmetro do operador unário (um conjunto de nomes de canais no caso do operador de internalização, ou uma relação de mapeamento entre nomes de canais no caso do operador de renomeação). Considere *Exp* como qualquer expressão CSP que possa ser usada como lado direito da equação de um processo, exceto nomes de processos, pois neste caso a equação já estaria na forma normal.

Regra 2. *Reescrita de Operador Unário*

$$P(s) = Exp\ \textbf{\textit{uop}}\ args \quad \Longrightarrow \quad \begin{array}{l} P(s) = N(s)\ \textbf{\textit{uop}}\ args \\ N(s) = Exp \end{array}$$

O primeiro argumento do operador unário, que antes era uma expressão CSP, passa a ser uma referência para o processo N, cujo termo é Exp. Novamente, os parâmetros de P fazem parte do escopo de Exp, logo, estes são usados como parâmetros do novo processo N. □

A regra a seguir se aplica a equações com operadores binários.

Regra 3. *Reescrita de Operador Binário*

$$P(s) = Exp_1 \ \textbf{\textit{bop}} \ Exp_2 \quad \Longrightarrow \quad \begin{array}{l} P(s) = N_1(s) \ \textbf{\textit{bop}} \ N_2(s) \\ N_1(s) = Exp_1 \\ N_2(s) = Exp_2 \end{array}$$

Como esta equação possui duas expressões, Exp_1 e Exp_2, esta regra cria dois novos processos, N_1 e N_2, e troca as ocorrências das expressões anteriores em P. Novamente, os parâmetros de P são usados como parâmetros dos novos processos. Caso uma das expressões já seja um nome de processo, sua ocorrência em P não será substituída, e um novo processo não precisa ser criado. □

As ocorrências dos operadores de paralelismo alfabetizado ($P \ _X||_Y \ Q$) e *interleaving* (|||) são substituídas por suas representações equivalentes com o operador de paralelismo generalizado, mencionadas na Seção2.1.10.

A próxima regra é usada para simplificar equações de processos que usam escolha condicional. Por simplicidade, assumimos que toda escolha condicional é representada com a expressão *if then else*. Expressões como g & P devem ser reescritas para $(if \ (g) \ then \ P \ else \ \textbf{\textit{STOP}})$.

Regra 4. *Reescrita de Escolha Condicional*

$$P(s) = if \ (g) \ then \ Exp_1 \ else \ Exp_2 \quad \Longrightarrow \quad \begin{array}{l} P(s) = if \ (g) \ then \ N_1(s) \ else \ N_2(s) \\ N_1(s) = Exp_1 \\ N_2(s) = Exp_2 \end{array}$$

Semelhante à Regra 3, esta regra cria dois novos processos, N_1 e N_2, e troca as ocorrências das expressões anteriores em P. □

A ocorrência do processo ***SKIP***, ou do processo ***STOP***, em uma equação que envolva um operador de CSP ou um comando condicional deve ser substituída por um nome de processo que o represente. Assim o processo

$$P(s) = \textbf{\textit{SKIP}} \ ; \ \textbf{\textit{STOP}}$$

deve ser reescrito para

$$\begin{array}{l} P(s) = N_1(s) \ ; \ N_2(s) \\ N_1(s) = \textbf{\textit{SKIP}} \\ N_2(s) = \textbf{\textit{STOP}} \end{array}$$

Por simplicidade, assumimos que recursão é expressa através de equações que podem ser mutuamente recursivas. Portanto, uma recursão da forma $P = \mu X \bullet F(X)$ pode ser transformada em $P = F(P)$, onde F é uma função que representa o corpo do processo recursivo.

Processos que possuam mais de uma definição, como no exemplo:

$$\begin{array}{l} P(0) = a \rightarrow P(1) \\ P(x) = P(x+1) \ \Box \ P(x-1) \end{array}$$

devem ser reescritos usando escolhas condicionais para cada definição, como a seguir:

$$\begin{array}{l} P(x) = if \ (x = 0) \ then \ P_1(x) \ else \ P_2(x) \\ P_1(x) = a \rightarrow P(1) \\ P_2(x) = P(x+1) \ \Box \ P(x-1) \end{array}$$

Após a aplicação exaustiva das regras acima a uma especificação CSP arbitrária, toda equação de processo terá uma das formas simplificadas apresentadas anteriormente.

4.2 Mapeamento de Processos

Nesta seção analisamos os detalhes pertinentes à representação de processos CSP através de cápsulas UML-RT. Este mapeamento provê uma representação gráfica dos processos CSP através dos diagramas de classes, de estrutura e de máquina de estados das cápsulas geradas. Os diagramas de classes são úteis para identificar os atributos e métodos das cápsulas, e os relacionamentos com outros elementos do modelo. As máquinas de estados são construídas para representarem a semântica comportamental do processo que lhe deu origem. Os diagramas de estrutura identificam a configuração estrutural de cada cápsula.

Como condição para aplicar as regras de mapeamento, todos os processos devem estar em uma das formas normais descritas na Seção 4.1. As regras para o mapeamento de processos apresentadas aqui foram idealizadas para serem independentes e composicionais, e para preservarem a semântica dos processos CSP originais.

Cada equação de processo é contemplada por uma regra de mapeamento. O lado esquerdo da equação dá origem a uma cápsula de mesmo nome do processo. Cada parâmetro do processo é mapeado em um atributo da cápsula. Pela própria definição de cápsulas, todo atributo é protegido, e sua visibilidade está restrita ao escopo da cápsula. A Figura 4.1 mostra a forma geral de uma cápsula P gerada a partir de um processo parametrizado $P(s)$.

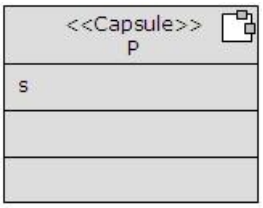


Figura 4.1 Cápsula P

A ocorrência de um canal no lado direito da equação implica na criação de uma porta, de mesmo nome do canal, na estrutura da cápsula. Por fim, a ocorrência de eventos no processo determina os eventos processados na máquina de estados da cápsula, através das suas transições. Analisamos primeiramente o impacto na estrutura e no comportamento das cápsulas geradas, na Seção 4.2.1. Em seguida, na Seção 4.2.2, analisamos a composição entre cápsulas para representar os diferentes relacionamentos entre processos, através dos operadores de CSP.

4.2.1 Estratégia para Construção de Cápsulas

Nesta seção analisamos o impacto na estrutura e no comportamento das cápsulas geradas para representarem as equações que lhes deram origem. Consideramos inadequado dissociar a construção dos diagramas de estrutura da construção das máquinas de estado, porque o comportamento das cápsulas está diretamente ligado aos elementos que a compõem, como portas e suas

regras de comunicação, definidas nos protocolos.

Em parte, isto se justifica porque muitos elementos sintáticos de CSP, que são mapeados em elementos estruturais das cápsulas, possuem um comportamento implícito associado. Este comportamento implícito não faz parte explicitamente da equação de um processo, mas precisa ser mapeado na cápsula que representa aquele processo. Um exemplo claro são os canais de comunicação, cujo uso envolve regras de sincronização complexas.

Como mencionado anteriormente, a ocorrência de um canal no lado direito da equação de um processo implica na criação de uma porta, de mesmo nome do canal, na estrutura da cápsula. Ambos, canais e portas, representam abstrações para interfaces de comunicação. Adicionalmente, a ocorrência de eventos no processo determina as transições na máquina de estados da cápsula.

Sob o ponto de vista dos classificadores de UML-RT envolvidos, a tradução de processos CSP envolve cápsulas e protocolos, porque os eventos de comunicação entre cápsulas ocorrem através de portas, que realizam protocolos.

Em CSP, a interação ente dois processos é chamada de *sincronização*, ou *rendezvous*. A interação simultânea entre mais de dois processos é chamada de *sincronização múltipla*, ou *multi-way rendezvous*. Se dois ou mais processos devem comunicar-se simultaneamente em um conjunto de eventos, então eles possuem uma relação de *causalidade* entre estes eventos [41, 32].

Inicialmente, o modo de comunicação síncrono de UML-RT parece ser a alternativa mais adequada para representar a comunicação em CSP, que também é síncrona. De fato, uma única mensagem síncrona entre duas cápsulas pode representar o comportamento de dois processos comunicando-se em um determinado evento, ou seja, uma sincronização simples. Mas o mesmo não pode ser dito se existir sincronização múltipla ou comportamentos não-determinísticos entre cápsulas.

No exemplo a seguir, os processos P, Q e R sincronizam no evento a. A impossibilidade de um dos processos para sincronizar em a inibe a sincronização dos demais, mas este controle é intrínseco aos canais e operadores de CSP.

$$
\begin{aligned}
&P = a!1 \rightarrow \textbf{\textit{STOP}} \\
&Q = a?x : \{1,2\} \rightarrow \textbf{\textit{STOP}} \\
&R = a?x : \{2,3\} \rightarrow \textbf{\textit{STOP}} \\
&P \underset{\{a\}}{\|} (Q \underset{\{a\}}{\|} R)
\end{aligned}
$$

A Figura 4.2 mostra uma possível configuração para as cápsulas P, Q e R do exemplo acima. O envio da mensagem $a.1$ da cápsula P para a cápsula Q resultará em uma sincronização com sucesso, porque Q está preparada para receber este evento. Entretanto, o envio simultâneo da mesma mensagem para a cápsula R não será concluído, visto que esta cápsula não está apta a receber esta mensagem. Assim, a relação de causalidade entre os três processos não foi atendida, já que a cápsula Q estabeleceu uma sincronização com a cápsula P, mas a cápsula R não.

O uso de canais e operadores permite que CSP abstraia os detalhes de concorrência e comunicação específicos do projeto ou da plataforma de desenvolvimento, e foque no comportamento e na estrutura do sistema. Assim, é possível representar *multithreading* sem se

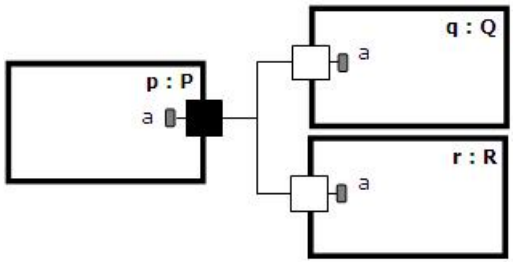


Figura 4.2 Configuração entre cápsulas as *P*, *Q* e *R*

preocupar com a administração de *threads*. Por outro lado, UML e UML-RT não possuem mecanismos naturais para representar a causalidade de eventos como os operadores de CSP. UML determina que eventos sejam executados um por vez (de acordo com a semântica *run-to-completion*) [34, 35, 39], tornando complexo especificar o comportamento de um sistema quando vários eventos ocorrem concorrentemente, e principalmente criar uma relação de dependência entre mensagens. Mesmo mensagens síncronas não são suficientes para representar a causalidade de eventos quando mais de duas cápsulas estão envolvidas.

Existem diversas abordagens para representar a causalidade em UML [32]. Uma delas é a criação de um elemento de modelagem adicional a UML, que represente ou contenha o conceito de causalidade implicitamente. Este elemento poderia ser uma especialização dos protocolos ou dos sinais de comunicação atuais de UML-RT, ou um novo esteriótipo que possa ser incorporado à estrutura das cápsulas; esta alternativa, embora prática e elegante, está fora do escopo desta dissertação, que pretende explorar os elementos atuais de UML-RT, mas pode ser abordada em trabalhos futuros. Outras abordagens [45] sugerem o uso de índices e vetores para estabelecer uma ordem parcial entre eventos, que geralmente complicam o modelo final. Por último, é possível ignorar a causalidade, abstraindo limitações externas na ocorrência de eventos em cápsulas; esta alternativa é descartada, já que o propósito deste trabalho é a geração de modelos UML-RT que preservem a semântica comportamental de CSP.

Isto sugere que o controle da causalidade entre eventos é uma decisão arquitetural, e que não pode ser representado diretamente em UML e UML-RT apenas com os seus esteriótipos atuais. UML, e a versão recente de UML2.0 [35], ainda possuem ambiguidades e limitações semânticas (especialmente a semântica de eventos em máquinas de estado) que reforçam a idéia de que ainda há muito que fazer para adequar estas notações a certos padrões arquiteturais [32].

A solução que propomos é representar explicitamente o controle da causalidade no modelo UML-RT gerado. Embora este controle seja intrínseco a CSP, entendemos que a modelagem comportamental de um sistema em UML-RT está diretamente relacionada a esta propriedade, que representa uma regra de comunicação ou um requisito do sistema. Por se tratar de uma regra de comunicação, podemos especificá-la através de protocolos de comunicação de UML-RT. Também consideramos este trabalho como um precursor para a construção de esteriótipos que representem o controle da causalidade, visto que identificamos as características associadas a este esteriótipo.

Como exposto no Capítulo 3, a escolha do protocolo é relevante para determinar o *tipo* e *ordem* dos sinais transmitidos pelas portas, mas não afeta o *modo* de comunicação entre as

cápsulas, que depende da forma como as mensagens são enviadas. A *ordem* dos sinais em máquinas de estado de protocolos deve indicar às cápsulas as regras de comunicação específicas de cada porta, como algoritmos de comunicação dividida em fases. Esta ordem, embora restrinja a sequência de eventos nas cápsulas, não deve limitar o comportamento interno das mesmas. É importante diferenciar o modo de comunicação, que pode ser síncrono ou assíncrono, das regras de comunicação, que envolvem sequências específicas de eventos para atender algum requisito do sistema. Logo, se abstrairmos o tipo e a ordem dos sinais, a escolha do protocolo é meramente uma questão estrutural do ponto de vista das cápsulas, porque todas as portas terão as mesmas características.

Se todos os protocolos envolvidos na tradução atenderem às mesmas regras genéricas de comunicação de CSP, e todos os seus sinais aceitarem apenas tipos definidos no sistema, então as portas criadas pelas regras de mapeamento não ferem a definição dos canais que as originaram.

Nos casos de sincronização múltipla é preciso saber previamente o status das cápsulas receptoras, para só então concluir a comunicação. Nossa estratégia usa uma variação do protocolo de requisição em duas fases (*Two-phase Commit Protocol*) [33] para controlar a comunicação entre cápsulas. A Figura 4.3 mostra os sinais usados pelo papel *base* de um protocolo *CSPMessageProtocol*, usado para construir portas que representam canais CSP.

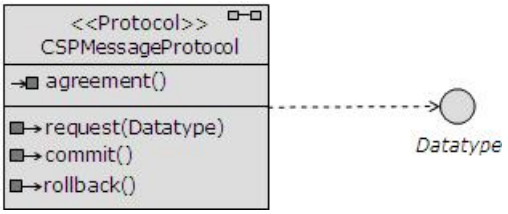


Figura 4.3 Estrutura do protocolo *CSPMessageProtocol*

Este protocolo assume que o emissor da mensagem deve comunicar-se com os receptores em duas fases:

- Na primeira fase o emissor envia mensagens síncronas aos receptores, através do sinal *request*, e aguarda que os receptores respondam se a mensagem foi recebida. Cada receptor que estiver apto a processar aquela mensagem deve respondê-la, através do sinal *agreement*, e mudar o seu estado interno para bloqueado. Se não estiver apto para receber a mensagem, esta é descartada e nenhuma mensagem de retorno é enviada para o emissor. UML-RT assume que existe um controle interno que verifica a resposta de mensagens síncronas. Se o emissor não recebe a resposta de uma mensagem síncrona, então esta mensagem é considerada perdida e assume-se que o receptor não pôde respondê-la.

- Na segunda fase, o emissor processa a resposta dos receptores e decide quais devem efetivar ou não a operação, de acordo com o comportamento esperado. Caso algum receptor deva concluir o processo de sincronização iniciado na primeira fase, é enviada uma mensagem para este, através do sinal *commit*. Caso contrário, é enviada uma mensagem através do sinal *rollback*. Cada receptor que recebe uma das duas mensagens muda seu estado interno e torna-se desbloqueado. Nesta fase, o modo de comunicação é irrelevante, e as mensagens podem ser assíncronas.

A Figura 4.4 mostra a ordem dos sinais do protocolo *CSPMessageProtocol* para implementar o algoritmo descrito acima.

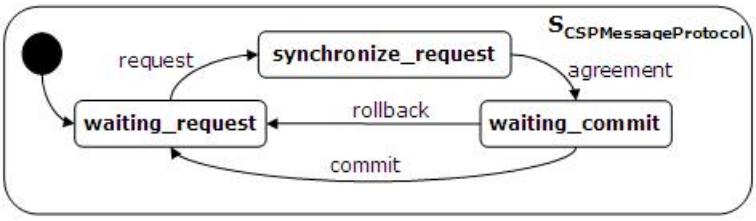


Figura 4.4 Máquina de estados do protocolo *CSPMessageProtocol*

Através desta regra de comunicação, a cápsula *P* da Figura 4.2 apenas concluiria seu processo de multi-sincronização com *Q* e *R* se ambas enviassem suas respostas ao sinal de *request*. Se apenas uma cápsula responder ao sinal de *request*, a cápsula *P* envia a esta um sinal de *rollback*. A Figura 4.5 mostra um exemplo de máquina de estados para a cápsula *Q*, atendendo a ordem de sinais determinada pelo protocolo *CSPMessageProtocol*. Se o valor comunicado através do sinal *request* estiver no conjunto $\{1,2\}$, o sinal de resposta é enviado, e a cápsula *Q* muda para o estado *locked*, onde aguarda a conclusão da comunicação (através do sinal *commit*), ou a desistência (através do sinal *rollback*). Se o valor comunicado através do sinal *request* não estiver no conjunto $\{1,2\}$, nenhuma resposta é enviada, e *Q* volta para o estado anterior ao primeiro evento.

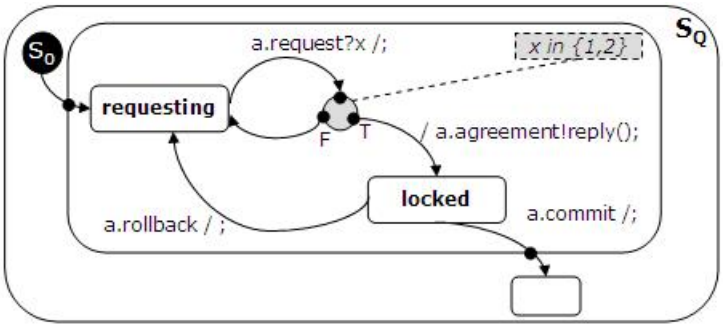


Figura 4.5 Máquina de estados da cápsula *Q*, atendendo ao protocolo *CSPMessageProtocol*

A grande desvantagem desta abordagem é o fato das cápsulas participantes permanecerem bloqueadas até que as duas fases sejam concluídas. Isto significa que outros eventos de comunicação devem aguardar até que as cápsulas bloqueadas sejam liberadas. Por outro lado, esta abordagem apenas sugere o controle da causalidade em modelos UML-RT, abstraindo os detalhes próprios do projeto ou da plataforma de desenvolvimento adotada, como políticas de acesso ou compartilhamento de recursos. É importante distinguir o conceito de concorrência no nível de design do conceito de concorrência no nível de implementação [20].

Uma possível estratégia de mapeamento seria criar um protocolo para cada possível conexão entre cápsulas, ou seja, para cada ocorrência de canal em um processo, mas consideramos esta abordagem dispendiosa e pouco prática, devido à grande quantidade de protocolos gerados. Outra alternativa seria criar um protocolo para cada canal definido na especificação, e toda

ocorrência daquele canal em um processo implicaria na criação de uma porta que atendesse àquele protocolo. A mesma abordagem poderia ser aplicada para criar um protocolo para cada tipo de dado, e o uso de qualquer canal daquele tipo de dado daria origem a uma porta que atendesse ao protocolo.

Por simplicidade, usamos um único protocolo para dar origem a portas que representam canais, considerando que os sinais deste protocolo aceitam qualquer tipo de dado definido na especificação, e que sua máquina de estados representa as regras de sincronização entre eventos CSP. Para isso, convencionamos que todas as classes que representam tipos de dados definidos na especificação devem implementar uma mesma interface, chamada *Datatype*, cujo uso é detalhado na Seção 4.3. Os sinais deste protocolo devem aceitar qualquer instância do tipo *Datatype*. Denotamos este protocolo de *CSPMessageProtocol*.

Assim, toda ocorrência de um canal de comunicação em um processo é mapeada em uma porta pública do tipo *CSPMessageProtocol*, de mesmo nome do canal, na cápsula correspondente. Os sinais deste protocolo, quando usados por esta porta, devem transportar apenas objetos que correspondam exatamente ao tipo de dado do canal. Os eventos associados a este canal devem determinar as transições disparadas através daquela porta na máquina de estados da cápsula. A Figura 4.6 mostra um possível diagrama de estrutura da cápsula originada a partir do mapeamento do processo $P(s) = a \rightarrow P(s)$, cuja porta a realiza o protocolo *CSPMessageProtocol*.

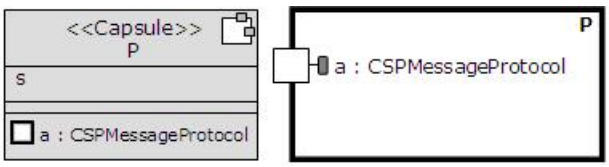


Figura 4.6 Diagrama de estrutura da cápsula *P*, contendo porta *a*

Reforçamos que a decisão de usar um único protocolo foi tomada para simplificar o modelo UML-RT gerado, e não altera a semântica do modelo CSP original, visto que é possível duplicar protocolos e especializar o tipo de seus sinais, sem que o comportamento das cápsulas seja afetado, apenas mudando o tipo de suas portas. Esta estratégia também não afeta o modo de comunicação entre cápsulas, e tem a vantagem de concentrar possíveis regras de comunicação em uma mesma máquina de estado de protocolo, e aplicá-las a todos os eventos que ocorram em suas portas.

Assim, nossas regras para mapeamento de processos consideram apenas a construção de cápsulas, tendo em vista que o protocolo é fixo.

Finalmente, cápsulas UML-RT podem ter portas *base* e *conjugadas* para enviar e receber mensagens, respeitando a orientação dos sinais de cada papel. Entretanto, as cápsulas geradas pelas regras de mapeamento foram simplificadas para que contenham apenas portas *conjugadas*, representando canais de comunicação, exceto em casos especiais, apresentados posteriormente. Esta decisão também foi tomada para simplificar o modelo UML-RT gerado, e não muda a semântica da especificação CSP original, desde que é possível representar eventos CSP sem orientação ou replicar os sinais de entrada e saída nos papéis de protocolos. Esta última

alternativa criaria portas que se comunicariam em todas as direções, independente de serem *conjugadas* ou *base*.

4.2.2 Estratégia para Composição de Cápsulas

Na seção anterior, analisamos a representação do modo de comunicação de CSP em cápsulas. Esta representação está diretamente ligada à estrutura das cápsulas (porque a comunicação ocorre através de portas) e à máquina de estados (porque o comportamento das cápsulas deve atender aos requisitos impostos pela multi-sincronização de CSP). Nesta seção analisamos os casos em que processos passam a comportar-se como outros processos, e a representação deste comportamento nas cápsulas. Estas situações ocorrem quando um processo passa a usar a definição de outro processo, através dos operadores CSP.

Ainda com relação à estrutura das cápsulas geradas pela estratégia, quando um processo P passa a comportar-se como um processo Q (como em $P = a \rightarrow Q$, por exemplo), a cápsula P deve reutilizar a definição da cápsula Q. Intuitivamente, a cápsula P poderia conter uma instância da cápsula Q, e todas as portas públicas de Q seriam replicadas como portas públicas de P. Desta forma, a cápsula P assimilaria o alfabeto da sub-cápsula Q. P repassaria todos os eventos que ocorressem nas portas replicadas para as portas correspondentes de Q, e vice-versa. Desta forma, P encapsularia o comportamento de Q em relação ao ambiente externo. A Figura 4.7 mostra o diagrama de estrutura de P para esta alternativa de tradução. Considere que o alfabeto de Q também é $\{a\}$.

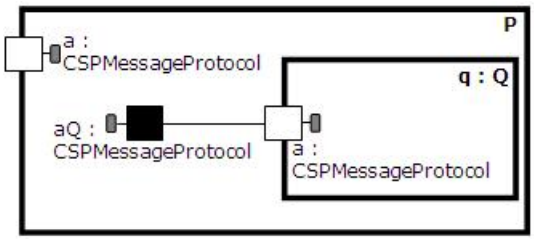


Figura 4.7 Cápsula P contendo instância da cápsula Q

A mesma abordagem se aplicaria a processos que representam composições de outros dois processos (como em $R = W \underset{\{\}}{||} Z$, por exemplo). Intuitivamente, a composição deve ser mapeada em uma cápsula que coordene as cápsulas de seus processos atributos, segundo a semântica do operador CSP em consideração. A Figura 4.8 mostra um diagrama de estrutura de R para esta alternativa de tradução. Considere que os alfabetos de W e Z são iguais a $\{a\}$.

Entretanto, esta abordagem não permite capturar referências mútuas entre os processos (por exemplo, quando $P = a \rightarrow Q$ e $Q = a \rightarrow P$), porque UML-RT não permite que exista inclusão cíclica entre cápsulas. Logo, no exemplo da Figura 4.7, se a cápsula P contiver uma sub-cápsula Q, Q não poderá conter uma sub-cápsula P ou qualquer outra sub-cápsula que contenha P.

No exemplo da Figura 4.8, se a cápsula W possui o comportamento da equação $W = a \rightarrow R$, o problema da recursão cíclica se repete.

Em UML-RT, a impossibilidade de criar inclusões cíclicas entre cápsulas é importante para

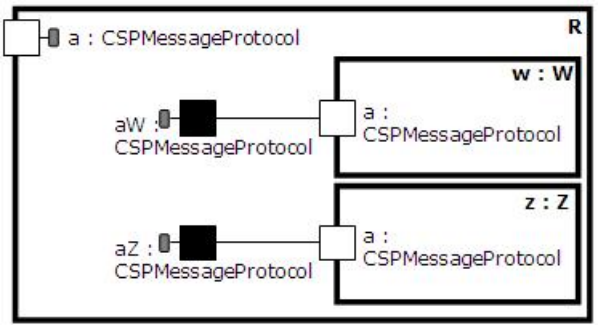


Figura 4.8 Cápsula *R* contendo instâncias das cápsulas *Z* e *W*

evitar recursão infinita na criação das cápsulas, e se aplica inclusive para cápsulas criadas dinamicamente (opcionais ou *plug-in*).

Como solução, propomos que cápsulas geradas através das regras não contenham instâncias de outras cápsulas, mas possuam uma porta *base* pública específica para informar o ambiente externo sobre possíveis mudanças comportamentais que referenciem outras cápsulas. Esta porta especial implementa o protocolo *CSPBehaviorProtocol* (ver Figura 4.9), e deve transmitir mensagens que representem as novas equações comportamentais dos processos, através do sinal *term*. Denotamos esta porta por *porta comportamental*. A Figura 4.9 mostra a estrutura do papel *base* do protocolo *CSPBehaviorProtocol*.

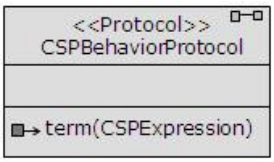


Figura 4.9 Papel *base* do protocolo *CSPBehaviorProtocol*

A Figura 4.10 mostra a cápsula originada a partir do processo $P(s) = a \rightarrow Q(s')$, onde a porta *b* realiza o protocolo *CSPBehaviorProtocol*. O estado *Sa* concentra as transições da porta *a*. A expressão $b.term!Q(s')$ representa o envio da expressão $Q(s')$ ao ambiente externo. No Capítulo 5, mostramos que a operacionalização desta estratégia usa uma estrutura de classes para representar as equações. Aqui, por legibilidade, usamos as próprias expressões CSP como argumentos para o sinal *term*.

Todas as cápsulas geradas devem ainda ser usadas como sub-cápsulas opcionais da cápsula principal do modelo, que deve controlar a configuração dinâmica das demais. Assim, se uma cápsula informar que o seu comportamento passou a ser uma composição paralela de duas outras cápsulas, a cápsula principal deve remover a instância opcional da primeira cápsula e coordenar a composição paralela das novas instâncias.

A decisão de usar cápsulas opcionais se justifica pelas características dinâmicas dos modelos CSP. Cápsulas opcionais podem ser criadas e usadas até que uma nova configuração seja necessária, quando elas são removidas e novas instâncias opcionais são criadas. Assim, o sistema pode ser configurado dinamicamente. É possível, inclusive, coordenar várias instâncias

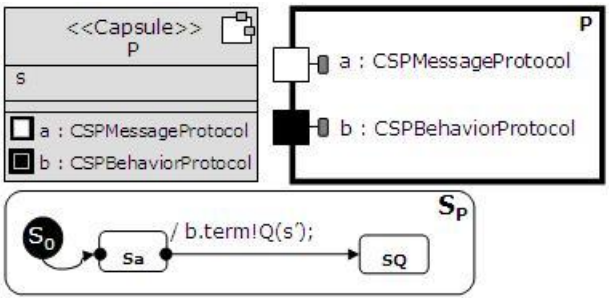


Figura 4.10 Cápsula *P*, contendo a porta comportamental *b*

de uma mesma sub-cápsula opcional, desde que o número de instâncias seja menor ou igual à cardinalidade da sub-cápsula. Neste caso, é preciso considerar os índices destas instâncias e das portas conectadas a elas. Por fim, cápsulas opcionais podem ser criadas com parâmetros, diferentemente de cápsulas fixas, facilitando a representação de processos parametrizados.

Esta cápsula principal, nomeada *SystemController*, é construída incrementalmente, junto com as cápsulas de processos. *SystemController* replica todas as portas públicas das suas sub-cápsulas, à medida que estas são criadas. As portas comportamentais (do tipo *CSPBehaviorProtocol*) não são replicadas pela cápsula *SystemController*, porque são elementos de controle interno; o ambiente externo não tem conhecimento da configuração interna das cápsulas, apenas da interface e do comportamento final do sistema. Ao fim do mapeamento de todos os processos da especificação, o "alfabeto" de *SystemController* corresponde à soma dos alfabetos de todas as cápsulas geradas pela estratégia.

SystemController coordena a comunicação com as sub-cápsulas através de portas *end* e protegidas, conectadas às portas das sub-cápsulas. Estas portas são complementares àquelas as quais estão conectadas. Considere os processos $P = a \rightarrow Q$ e $Q = P \underset{\{\}}{||} P$. A Figura 4.11 exemplifica a estrutura estática de *SystemController* contendo as sub-cápsulas opcionais *P* e *Q*. A porta *end*, protegida e base *aP* é usada para repassar mensagens de dados para a sub-cápsula *P*. As portas *end*, protegidas e conjugadas *bP* e *bQ* são usadas para receber as mensagens de mudança de comportamento vindas das sub-cápsulas *P* e *Q*, respectivamente.

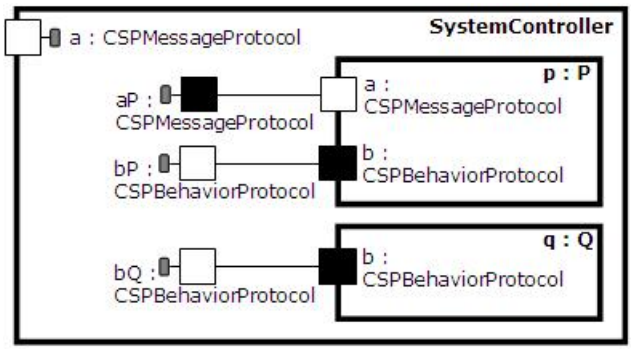


Figura 4.11 *SystemController* contendo sub-cápsulas *P* e *Q*

SystemController também centraliza a equação expandida do sistema em cada momento. A equação expandida contém referências para as cápsulas ativas no momento, combinadas com os operadores de CSP. Cada referência contém o nome e o índice de uma cápsula. Se uma cápsula informa um novo comportamento, sua referência na equação expandida deve ser substituída por novas referências, combinadas conforme a equação CSP informada. Todas as ações de *SystemController* devem considerar esta equação.

A Figura 4.12 mostra a estrutura dinâmica de *SystemController* apenas com as sub-cápsulas ativas (cápsulas tracejadas), antes (esquerda) e depois (direita) da mudança de comportamento da instância da cápsula *P*. No primeiro caso a equação expandida de *SystemController* possui uma referência à cápsula *P*. Após a mudança de comportamento de *P*, a equação expandida passa a referenciar a cápsula *Q*.

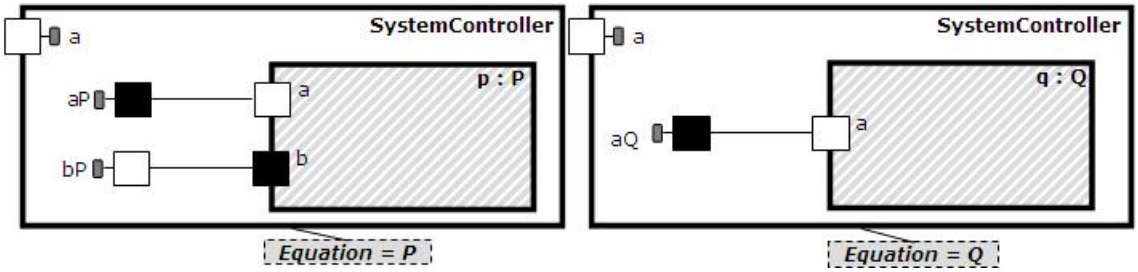


Figura 4.12 Estrutura de *SystemController* antes e depois da mudança de comportamento de *P*

SystemController pode, eventualmente, controlar várias sub-cápsulas ativas simultaneamente, e não apenas uma instância de *P* ou *Q*. Quando em execução, a configuração de *SystemController* é dinâmica, e as sub-cápsulas ativas podem estar associadas através de operadores aninhados.

A estrutura da equação expandida é similar à de uma árvore, onde as folhas são as referências das cápsulas ativas, e os nós são os operadores de CSP. A raiz da árvore é o operador mais externo da equação expandida. A semântica dos operadores está concentrada na interpretação da equação expandida de maneira *top-down*. A cada evento que ocorre no sistema, a equação expandida é avaliada da raiz para as folhas, sempre verificando a semântica do operador em questão, até que todas as cápsulas que devam ser influenciadas por este evento sejam identificadas. A cada mensagem de comportamento comunicada por uma cápsula ativa, sua referência na equação expandida é substituída pela referência à nova cápsula, ou pelo nó que representa o operador em questão, cujas folhas são as referências das cápsulas envolvidas.

Entretanto, a semântica dos operadores está concentrada na interpretação da equação expandida de *SystemController*. A cada mudança de comportamento de uma cápsula, sua referência é substituída na equação expandida e o controle da semântica daquele operador é dado pela nova referência.

A estrutura da equação expandida é similar à de uma árvore, onde as folhas são as referências das cápsulas ativas, e os nós são os operadores de CSP. A raiz da árvore é o operador mais externo da equação expandida. A semântica dos operadores está concentrada na interpretação da equação expandida de maneira *top-down*. A cada evento que ocorre no sistema, a equação expandida é avaliada da raiz para as folhas, sempre verificando a semântica do operador em

questão, até que todas as cápsulas que devam ser influenciadas por este evento sejam identificadas. A cada mensagem de comportamento comunicada por uma cápsula ativa, sua referência na equação expandida é substituída pela referência à nova cápsula, ou pelo nó que representa o operador em questão, cujas folhas são as referências das cápsulas envolvidas.

A Figura 4.13 mostra a estrutura dinâmica de *SystemController* antes (esquerda) e depois (direita) da mudança de comportamento da instância da cápsula Q. No primeiro caso a equação expandida de *SystemController* possui uma referência à cápsula Q. Após a mudança de comportamento de Q, a equação expandida passa a referenciar a composição paralela $P \underset{\{\}}{||} P$.

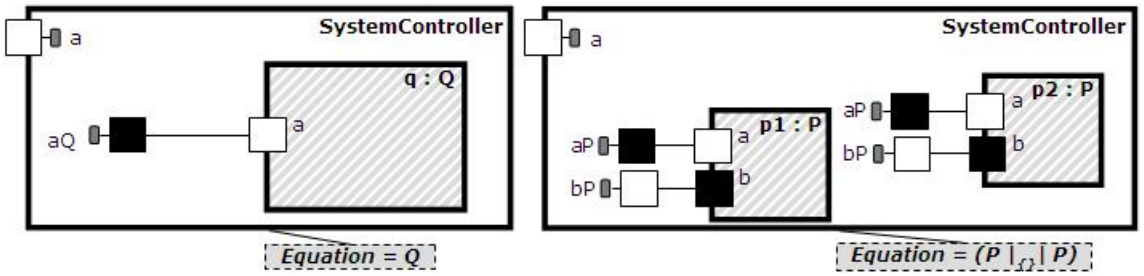


Figura 4.13 Estrutura de *SystemController* antes e depois da mudança de comportamento de Q

Portanto, em cada instante, *SystemController* implementa o comportamento dos operadores de CSP envolvidos na equação em consideração. Uma alternativa para esta abordagem seria a utilização de estados concorrentes (ortogonais) em *SystemController*. Entretanto, como exposto no Capítulo 3, a utilização destes estados pode gerar representações ambíguas ou comportamentos indesejáveis. Isto se deve principalmente porque a semântica de estados concorrentes em UML-RT é mais limitada que em UML1.5. Não é possível, por exemplo, estabelecer correlações entre transições de estados concorrentes, o que dificulta ou inviabiliza a representação de paralelismo ou não-determinismo entre cápsulas. Na Seção 4.4, apresentamos uma discussão mais detalhada sobre a representação de operadores de CSP através de cápsulas específicas, e expomos suas limitações.

Resumindo, as funções de *SystemController* são:

- Receber as mensagens de dados enviadas pelo ambiente externo e repassá-las para as sub-cápsulas ativas. A decisão de quais sub-cápsulas devem receber a mensagem é tomada em função da equação expandida. Por exemplo, se a equação expandida for um paralelismo total de cápsulas, todas as mensagens devem ser repassadas para todas as sub-cápsulas ativas. Se a equação expandida for uma escolha interna entre cápsulas, deve-se fazer uma escolha aleatória entre as cápsulas antes que a mensagem seja repassada.

- Receber as mensagens de mudança de comportamento enviadas pelas sub-cápsulas ativas, e mudar sua configuração interna. A equação expandida é atualizada e, dependendo da expressão comunicada, novas sub-cápsulas são criadas.

Por fim, *SystemController* é conectada diretamente ao ambiente externo, simulando a interface de todo o sistema. O ambiente externo, que é implícito em CSP, é tornado explícito

no modelo UML-RT através de uma cápsula que contém a cápsula *SystemController*. Desta forma, apenas o ambiente externo, que é conectado a esta cápsula através de portas *base*, pode enviar mensagens usando os sinais de saída, definidos para o papel *base* do protocolo. A Figura 4.14 mostra a cápsula *Environment*, que representa o ambiente externo, contendo a cápsula *SystemController*, que representa a cápsula principal de um sistema cujo alfabeto é $\{a\}$.

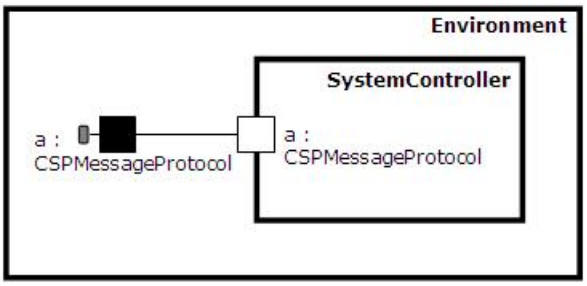


Figura 4.14 Ambiente externo contendo a cápsula principal do sistema

4.2.3 Regras de Mapeamento

Como descrito anteriormente, as regras para o mapeamento de processos consideram apenas a construção de cápsulas, tendo em vista que o protocolo usado para criar portas é fixo. A decisão de usar um único protocolo foi tomada para simplificar o modelo UML-RT gerado, e não altera a semântica da especificação CSP original. As regras de tradução de processos consideram que:

- toda equação de processo deve estar em uma das formas normais apresentadas na Seção 4.1;
- o lado esquerdo da equação de um processo dá origem a uma cápsula UML-RT de mesmo nome do processo;
- cada parâmetro do processo é mapeado em um atributo, de mesmo nome do parâmetro, na cápsula;
- cada canal presente no lado esquerdo da equação do processo dá origem a uma porta pública, de mesmo nome do canal, na cápsula. Esta porta é *conjugada* e atende ao protocolo *CSPMessageProtocol*;
- toda cápsula que represente um processo deve possuir uma porta comportamental, do tipo *CSPBehaviorProtocol*, para informar ao seu ambiente externo possíveis novos comportamentos;
- toda cápsula é usada como sub-cápsula opcional de *SystemController*, que é a cápsula principal do modelo.

A estratégia de mapeamento considera que *SystemController* foi previamente criada. A construção de *SystemController* inclui a definição da equação inicial do sistema, através da indicação do nome e dos parâmetros do processo que representa o sistema.

A Figura 4.15 mostra a estrutura estática e a máquina de estados de *SystemController* imediatamente antes da aplicação das regras de mapeamento. Considere o processo $P(s)$ como processo do sistema. A transição inicial contém os comandos necessários para criar uma instância da cápsula *P*, com o parâmetro *s*, e inicializar a equação expandida com a referência a esta instância. Os métodos *createCapsule* e *setEquation* encapsulam estes procedimentos. A estrutura inicial de *SystemController* é vazia, visto que os processos da especificação ainda não foram traduzidos em cápsulas.

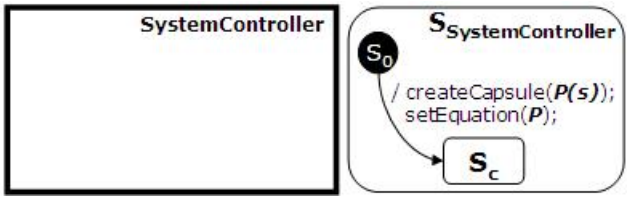


Figura 4.15 Estrutura e comportamento iniciais de *SystemController*

A estratégia de mapeamento considera cada equação de processo individualmente, na seqüência em que foram definidas. Em cada regra de mapeamento, o lado esquerdo representa uma forma normal de equação de processo; o lado direito apresenta as alterações no modelo UML-RT. Mostramos como a cápsula que representa o processo em questão é construída, e como *SystemController* é atualizada para controlar o uso da primeira. Embora *SystemController* centralize o fluxo de controle, sua construção também é composicional.

Nas regras a seguir, considere que os processos CSP estão em alguma das formas normais apresentadas na Seção 4.1, e que os nomes das portas que implementam o protocolo *CSPBehaviorProtocol* não fazem parte do alfabeto da especificação. As sub-cápsulas de *SystemController* são, na verdade, apontadores para cápsulas opcionais, isto é, ainda não são cápsulas ativas. Considere também que existe no máximo uma instância ativa de cada sub-cápsula em *SystemController*. Esta última restrição objetiva simplificar as regras de mapeamento, abstraindo o controle de índices de várias instâncias de uma mesma sub-cápsula e de suas conexões.

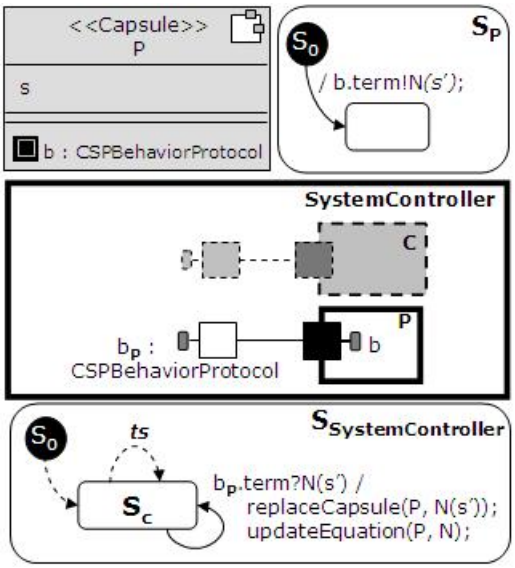
Por fim, *SystemController* deve ter uma transição para cada possível equação que possa ser transmitida pelas portas comportamentais de suas sub-cápsulas, porque não é possível saber previamente a expressão que cada cápsula informará. Entretanto, para facilitar a visualização das regras, exibimos apenas as transições relevantes para a regra em questão.

O estado S_C de *SystemController* é o estado atual da sua máquina de estados. Considere a sub-cápsula *C* como qualquer sub-cápsula já existente na estrutura de *SystemController*, e *ts* como o conjunto de transições definidas na máquina de estados de *SystemController* antes da aplicação da regra em questão.

A regra a seguir é usada para mapear equações de processos cujo lado direito é um nome de processo. Considere *N* como o parâmetro que representa o nome de um processo.

Regra 5. *Mapeamento de Nome de Processo*

$P(s) = N(s') \quad \Longrightarrow$



O processo que assimila o comportamento de outro processo (podendo inclusive ser ele próprio) é mapeado na cápsula que indica a *SystemController* que seu processamento deve ser substituído pelo de outra cápsula. O processo $P(s)$ é mapeando na cápsula de mesmo nome, contendo o atributo s e a porta *base* b, do tipo *CSPBehaviorProtocol*. A máquina de estados de P, S_P, modela sua mudança de comportamento para $N(s')$. O comando $b.term!N(s')$, executado na transição inicial da cápsula, informa esta mudança comportamental ao ambiente externo.

O diagrama de estrutura da cápsula *SystemController* é atualizado para que contenha uma sub-cápsula opcional de P e uma porta *end* protegida conectada a esta, chamada b_P. As sub-cápsulas e portas existentes antes da aplicação da regra (representadas por C) não são afetadas.

A máquina de estados de *SystemController* é atualizada com uma nova transição para cada equação comunicada à porta b_P. A transição disparada pelo evento $b_P.term?N(s')$ coordena a substituição da cápsula P pela cápsula cujo nome é representado por N. O método *replaceCapsule*, executado na ação desta transição, remove a instância ativa da cápsula P, e cria uma instância ativa da cápsula N, usando o parâmetro s'. Em seguida, o método *updateEquation* atualiza a equação expandida corrente de *SystemController*, trocando a referência de P pela de N. As transições existentes antes da aplicação da regra (representadas por *ts*) não são afetadas. □

A regra de mapeamento apresentada, assim como as demais a seguir, detalham a construção da máquina de estados e da estrutura estática das cápsulas envolvidas. Entretanto, estas cápsulas ainda não estão em execução. Quando em execução, a cápsula *SystemController* e as sub-cápsulas ativas simulam o comportamento dos respectivos processos.

Suponha esta regra aplicada à equação $P = Q$. A Figura 4.16 mostra uma simulação da estrutura dinâmica de *SystemController*, apenas com suas sub-cápsulas ativas (cápsulas tracejadas), antes (esquerda) e depois (direita) da mudança de comportamento da instância da cápsula P.

A regra a seguir trata do mapeamento do processo ***SKIP***.

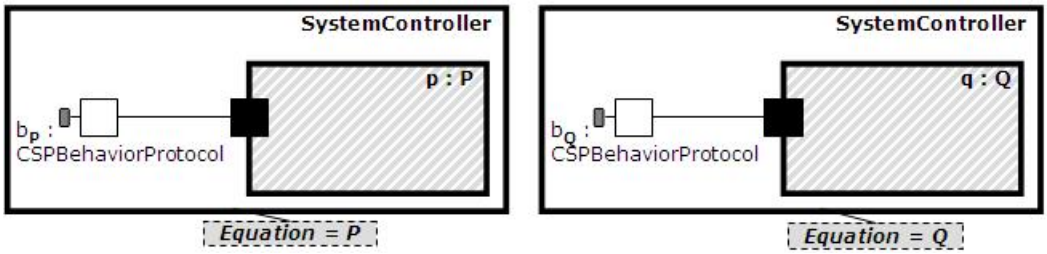
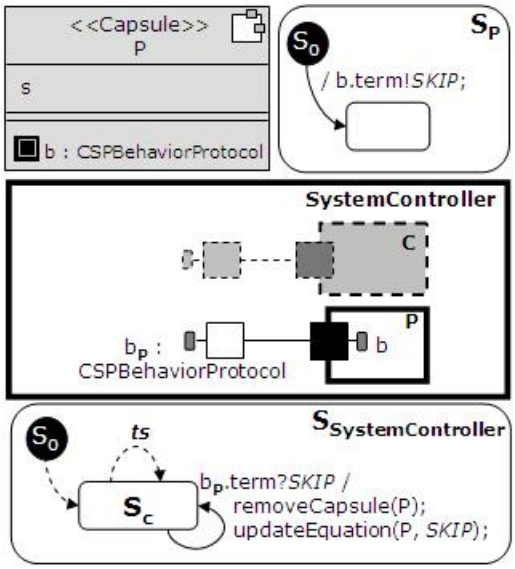


Figura 4.16 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para Q

Regra 6. *Mapeamento de Skip*

$P(s) = \textbf{\textit{SKIP}} \implies$



A máquina de estados de P, S_P, modela a mudança de comportamento para ***SKIP***. O comando $b.term!\textbf{\textit{SKIP}}$, executado na transição inicial da cápsula, informa esta mudança comportamental ao ambiente externo.

O diagrama de estrutura da cápsula *SystemController* é atualizado para que contenha uma sub-cápsula opcional de P e uma porta *end* protegida conectada a esta, chamada b_P. As sub-cápsulas e portas existentes antes da aplicação da regra (representadas por C) não são afetadas.

A máquina de estados de *SystemController* é atualizada com uma nova transição para cada equação comunicada à porta b_P. Quando a transição do evento $b_P.term?\textbf{\textit{SKIP}}$ é disparada, o método *removeCapsule* é executado, removendo a instância ativa de P. Em seguida, o método *updateEquation* é executado, trocando a referência da instância de P por ***SKIP***, na equação expandida de *SystemController*. Neste caso, uma nova cápsula não é criada porque a mudança de comportamento para ***SKIP*** representa uma terminação com sucesso. Apenas a equação expandida de *SystemController* possui a informação de que aquele comportamento é ***SKIP***. As transições existentes antes da aplicação da regra (representadas por *ts*) não são afetadas. □

Suponha esta regra aplicada à equação $P = \textbf{\textit{SKIP}}$. A Figura 4.17 mostra uma simulação da estrutura dinâmica de *SystemController*, apenas com suas sub-cápsulas ativas (cápsulas tracejadas), antes (esquerda) e depois (direita) da mudança de comportamento da instância da cápsula

P para ***SKIP***.

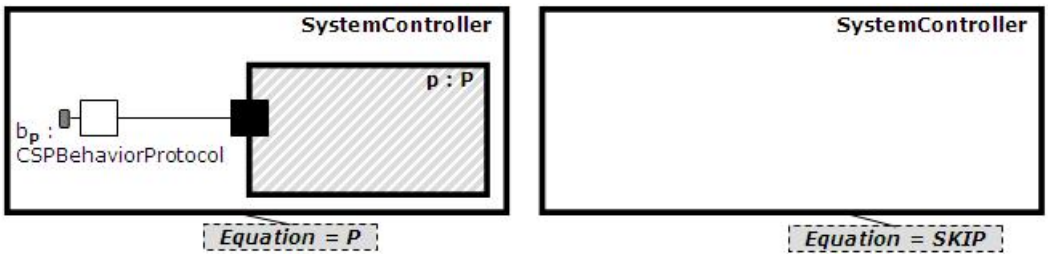


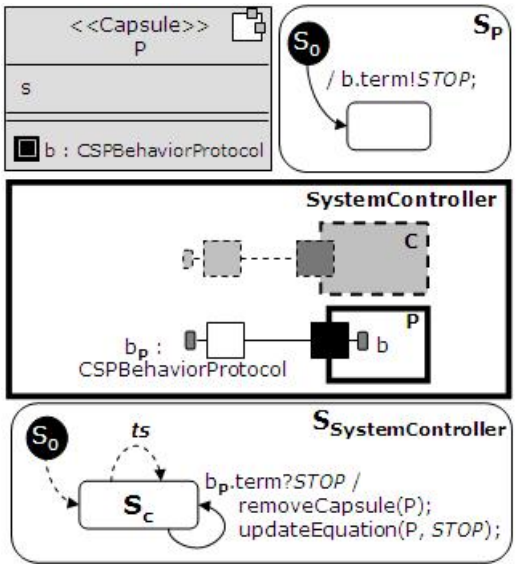
Figura 4.17 Estrutura de *SystemController* antes e depois da mudança de comportamento de *P* para ***SKIP***

A terminação com sucesso de *P* poderia ser representada por uma transição automática que levasse a um estado final. Entretanto, máquinas de estado em UML-RT não possuem estado final, sugerindo que cápsulas são objetos ativos que nunca terminam por conta própria, embora possam ser criadas e destruídas pelo ambiente externo. Outra alternativa seria representar a terminação com sucesso com um evento especial (um *tick*, por exemplo). Decidimos representar esta regra desta forma por uniformidade com as demais.

A próxima regra trata do mapeamento do processo ***STOP***.

Regra 7. *Mapeamento de Stop*

$P(s) = \textbf{\textit{STOP}} \quad \Longrightarrow$



Similar à Regra 6, a mudança de comportamento do processo $P(s)$ para ***STOP*** é mapeada na transição inicial da cápsula *P*, cuja ação é o comando *b.term*!***STOP***. Uma nova transição é criada no estado atual S_C de *SystemController* para coordenar a mudança de comportamento desta cápsula. Quando o evento b_P.*term*?***STOP*** ocorre, a instância ativa de *P* é removida e a referência de *P* na equação expandida é substituída por ***STOP***. □

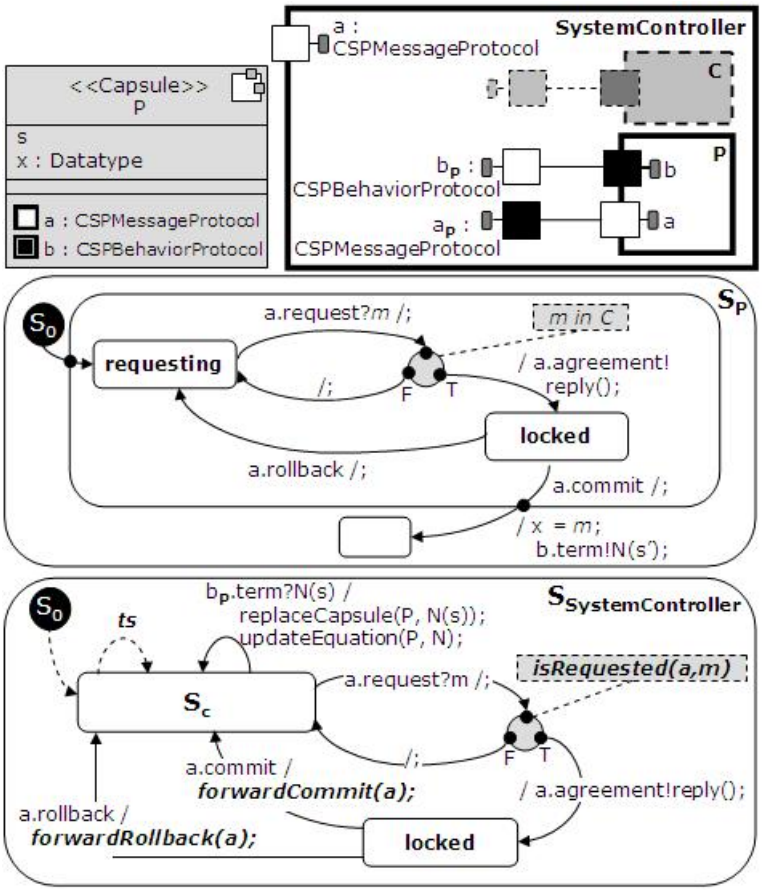
O comportamento do processo ***STOP*** poderia ser representado por um estado que não possui qualquer transição de saída, assim todas as mensagens enviadas para uma cápsula neste

estado não receberiam reposta. Novamente, decidimos por esta representação por uniformidade com as demais regras.

A regra a seguir é usada para mapear processos que envolvem um prefixo da forma $P(s) = a?x : C \rightarrow N(s')$. Considere que $a?x : C$ é uma comunicação cujos valores aceitáveis são listados no conjunto C. Se o conjunto C não é informado, então não há restrições quanto ao valor de x, e o evento pode ser resumido para $a?x$. Assuma que x não ocorre na lista de parâmetros de P.

Regra 8. *Mapeamento de Prefixo*

$$P(s) = \quad a?x : C \quad \rightarrow \quad N(s') \qquad \Longrightarrow$$



O processo $P(s)$ é mapeando na cápsula de mesmo nome, contendo o atributo s e a porta *base* b, do tipo *CSPBehaviorProtocol*. A ocorrência do canal a, no lado direito da equação de $P(s)$, é mapeada na porta *conjugada* a, do tipo *CSPMessageProtocol*, na cápsula P. A variável x, presente na comunicação CSP, é mapeada no atributo de mesmo nome na cápsula P. O tipo deste atributo é *Datatype*, visto que todas as portas do tipo *CSPMessageProtocol* transportam mensagens deste tipo.

O comportamento da comunicação $a?x : C$ é mapeado nas transições da máquina de estados da cápsula P, atendendo à sequência de sinais determinada pelo protocolo *CSPMessageProtocol*. Se o evento síncrono $a.request?m$ ocorrer, sua confirmação só deve ser enviada se o valor da mensagem m estiver contida no conjunto C.

Conceitualmente, a guarda de uma transição não tem acesso às mensagens transmitidas pelos eventos que disparam a transição, porque são avaliadas antes que os eventos sejam percebidos. Conseqüentemente, a primeira transição deve ser direcionada a um ponto de escolha, que verifica a restrição de valor da mensagem *m*. Se a restrição é atendida, a transição do ponto de junção *T* é encaminhada para o estado *locked*, e a confirmação do evento anterior é enviada, através do comando $a.agreement!reply()$. Se a restrição não é atendida, a transição do ponto de junção *F* retorna para o estado *requesting*, e a confirmação do evento não é enviada. Neste caso, o emissor da mensagem entende que a cápsula *P* não está pronta para sincronizar na mensagem enviada. Se não houver restrições quanto ao valor de *m*, então a expressão lógica no ponto de escolha é sempre *True*.

No estado *locked*, a cápsula *P* está pronta para receber os eventos *a.rollback* ou *a.commit*. Se receber o primeiro, a cápsula volta para o estado onde estava antes do primeiro evento, e a mensagem *m* é descartada. Se receber o segundo, todo o processo de sincronização do evento CSP $a?x : C$ estará concluído, e o valor de *m* é atribuído a *x*. Em seguida, a cápsula *P* informa seu novo comportamento, através do comando $b.term!N(s')$.

O diagrama de estrutura de *SystemController* é atualizado para que contenha uma sub-cápsula opcional de *P* e portas *end* protegidas conectadas a esta, chamadas a_P e b_P. A porta pública *a* de *P* é replicada em *SystemController*, exceto, é claro, que *SystremController* já possua um porta com este nome. É importante salientar que *SystemController* pode, eventualmente, ter várias sub-cápsulas ativas que possuam uma porta *a*. A ocorrência de um evento na porta *a* de *SystemController* pode estar relacionada a mais de uma destas instâncias, e não apenas a *P*, se esta estiver ativa.

A porta *end base* a_P é usada por *SystemController* para encaminhar as mensagens que chegam em sua porta *a* para a porta *a* de *P*. As sub-cápsulas e portas existentes antes da aplicação da regra (representadas por *C*) não são afetadas.

A máquina de estados de *SystemController* é atualizada para que as mensagens comunicadas à sua porta *a* sejam encaminhadas para as sub-cápsulas ativas, de acordo com a equação expandida corrente. Novamente, a confirmação destes eventos depende de condições de controle. Neste caso, a condição é que pelo menos uma sub-cápsula ativa possa receber a mensagem encaminhada, sem ferir a semântica da equação expandida.

Quando *SystemController* recebe o evento *a.request*?*m*, a transição disparada é encaminhada para um ponto de escolha, onde é executado o método *isRequested*. Neste método a mensagem recebida é encaminha para as sub-cápsulas ativas, de acordo com equação expandida. Por exemplo, se a equação expandida for um paralelismo total ou uma escolha externa entre duas cápsulas, então a mensagem é encaminhada para as duas. Também é considerada a presença de uma porta *a* nas instâncias ativas, o que significa que os eventos da porta *a* fazem parte do "alfabeto" da instância em questão. Cada instância que confirma o recebimento da mensagem encaminhada tem sua referência marcada como *locked*.

Após a confirmação das sub-cápsulas, ainda é verificado o comportamento da equação expandida para os eventos confirmados. Neste passo é verificado, por exemplo, se existe uma sincronização múltipla obrigatória entre cápsulas, e se todas estas fazem parte do conjunto de cápsulas que confirmaram o recebimento da mensagem. Também é verificada a necessidade de escolhas não-determinísticas entre cápsulas, como no caso do *interleaving*. Um sinal de

rollback é enviado para as cápsulas que não devem fazer parte do processo de sincronização, e suas referências são marcadas como *free*.

Se pelo menos uma cápsula deve continuar o processo de sincronização, o método *isRequested* retorna *True*, e o estado do *SystemController* muda para *locked*. Neste ponto, pelo menos uma sub-cápsula ativa de *SystemController* está bloqueada. O pseudo-código do método *isRequested* encontra-se no Apêndice A.

No estado *locked*, *SystemController* está pronta para receber os eventos *a.rollback* ou *a.commit*. Se o evento recebido for *a.rollback*, então todo o processo de sincronização é abortado. Se o evento recebido for *a.commit*, então todo o processo de sincronização deve ser concluído.

No caso do evento *a.rollback*, *SystemController* encaminha o evento recebido para todas as sub-cápsulas cujas referências estejam marcadas como *locked*, desbloqueando-as. O método *forwardRollback* encapsula a ação de encaminhar o evento para as sub-cápsulas marcadas. Em seguida, *SystemController* volta para o estado S_C, onde estava antes do primeiro evento.

No caso do evento *a.commit*, *SystemController* encaminha o evento recebido para todas as sub-cápsulas cujas referências estejam marcadas como *locked*. Entretanto, se todo o processo de sincronização foi concluído, então possivelmente a equação expandida precisa ser atualizada. Suponha que a equação expandida seja $P \square Q$. A conclusão do processo de sincronização com a cápsula Q implica que a cápsula P não será mais necessária. O método *forwardCommit*, executado na transição do evento *a.commit*, encapsula a ação de encaminhar o evento para as sub-cápsulas marcadas, remover as cápsulas inutilizadas e atualizar a equação expandida. Em seguida, *SystemController* volta para o estado S_C, onde pode processar qualquer evento de mudança de comportamento. Neste caso, a mudança de comportamento de todas as cápsulas que receberam o evento *a.commit*. O pseudo-código dos métodos *forwardRollback* e *forwardCommit* encontra-se no Apêndice A.

Finalmente, a transição que gerencia a mudança de comportamento de P é adicionada ao estado S_C de *SystemController*. □

Em contextos onde esta regra é aplicada à equação de um processo Q, que também possua um canal a, então apenas a última transição (de mudança de comportamento de Q) será adicionada à máquina de estados de *SystemController*, visto que as transições que controlam os eventos da porta a já foram criadas. Na verdade, estas transições são criadas para a porta a de *SystemController*, que é única, independente da quantidade de sub-cápsulas que possuam uma porta de mesmo nome.

A complexidade desta regra ocorre porque *SystemController* controla, eventualmente, várias sub-cápsulas ativas simultaneamente, e não apenas a instância de P. Quando em execução, a configuração de *SystemController* é dinâmica, e as sub-cápsulas ativas podem estar associadas através de operadores aninhados. Por isto, as transições de *SystemController* precisam de métodos auxiliares, que implementam a semântica de vários operadores simultaneamente. Entretanto, o comportamento final de todo o sistema é decidido em função de cada operador envolvido na equação expandida, de maneira composicional.

Considere a equação expandida de *SystemController* na Figura 4.18. À direita, a configuração dinâmica de *SystemController* é composta de uma instância P, em paralelo com uma escolha externa entre instâncias de Q e R. Quando o evento *a.request*?*m* ocorrer, *SystemCon-*

troller verifica se q e r estão prontos para sincronizar, e escolhe um deles, segundo a semântica do operador de escolha externa. Em seguida, *SystemController* verifica se p e a instância escolhida podem sincronizar juntas, segundo a semântica do paralelismo. À esquerda, a nova configuração de *SystemController*, supondo que as instâncias de P e Q concluíram a sincronização. A instância de R foi descartada.

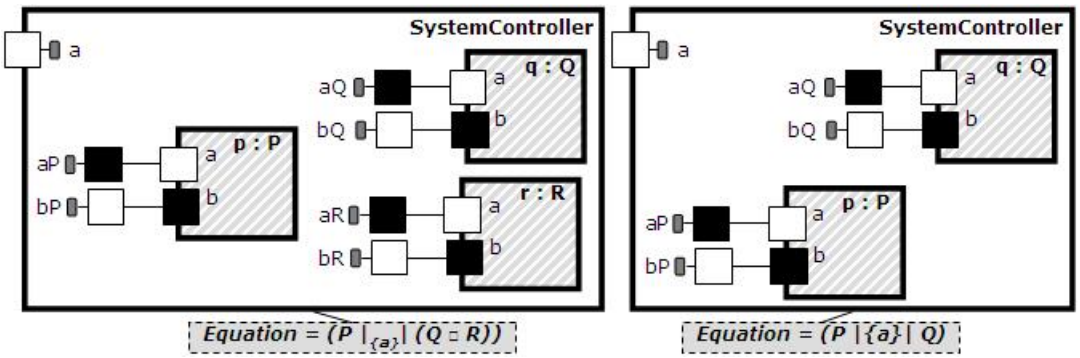
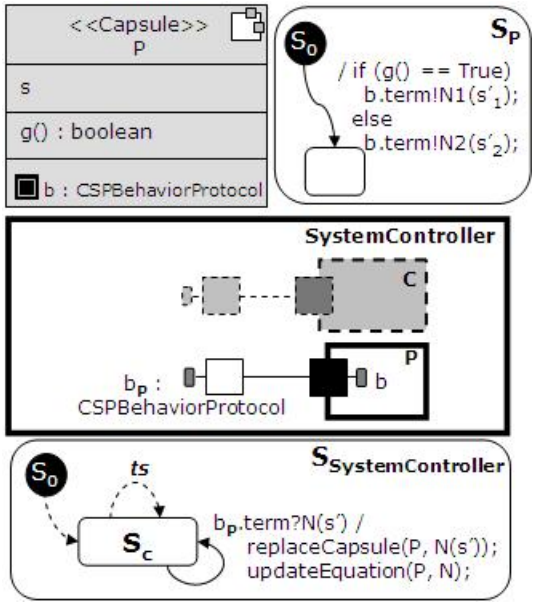


Figura 4.18 Estrutura de *SystemController* antes e depois dos eventos da porta a

A regra a seguir gera uma cápsula que assimila o comportamento de uma escolha condicional.

Regra 9. *Mapeamento de Escolha Condicional*

$$P(s) = \begin{array}{l} \textit{if } (g) \\ \textit{then } N1(s_1') \\ \textit{else } N2(s_2') \end{array} \quad \Longrightarrow$$



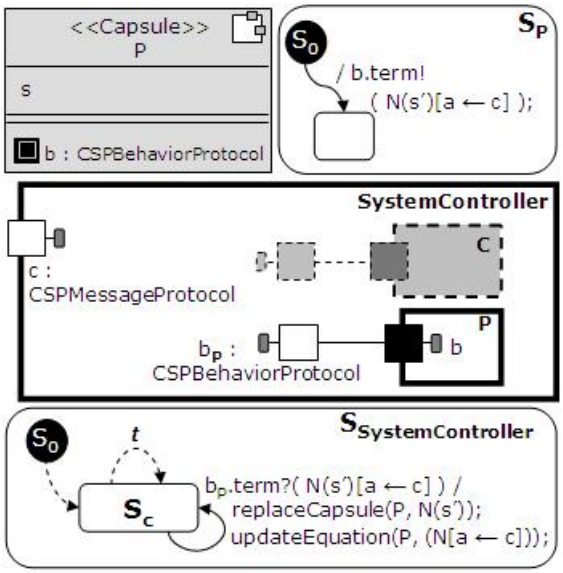
O processo $P(s)$ deve assimilar o comportamento de $N1(s_1')$ se a condição g resultar em *True*; caso contrário, deve assimilar o comportamento de $N2(s_2')$. A cápsula P obtida através desta regra contém um método $g()$ que representa a expressão lógica g em CSP. Um comando condicional é utilizado para avaliar o método $g()$ e decidir o comportamento de P. Assim, a mensagem enviada a *SystemController* pode conter uma referência à cápsula $N1$ ou à cápsula $N2$, depedendo do resultado do método $g()$. A mesma abordagem poderia ser aplicada através de

um ponto de escolha após a transição inicial da cápsula P. A estrutura e a máquina de estados de *SystemController* são atualizas como descrito na Regra 5. □

A próxima regra avalia equações de processos que usam o operador de renomeação.

Regra 10. *Mapeamento de Renomeação*

$$P(s) = N(s')\ [a \leftarrow c] \quad \Longrightarrow$$



O processo $P(s)$ comporta-se exatamente como $N(s')$, exceto pelos eventos presentes no domínio da relação $a \leftarrow c$, que são renomeados. A regra de mapeamento para equações que envolvem renomeação gera uma cápsula P, que informa ao ambiente externo o nome e os parâmetros da nova cápsula (representado por $N(s')$) e a relação de mapeamento entre nomes de portas (representada por $a \leftarrow c$), combinados pelo operador de renomeação. O comando $b.term!(N(s')\ [a \leftarrow c])$, executado na transição inicial da cápsula P, informa esta novo comportamento ao ambiente externo.

A estrutura de *SystemController* é atualizada para que contenha uma sub-cápsula opcional de P. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada pelo evento $b_P.term?(N(s)\ [a \leftarrow c])$ remove a instância de P e cria uma instância de N, usando o parâmetro s'. Em seguida, a equação expandida é atualizada, substituindo a referência de P pela expressão $(N\ [a \leftarrow c])$, contendo a referência à nova cápsula e a relação de renomeação.

A imagem da relação R, $\{c\}$, dá origem a portas públicas *conjugadas* em *SystemController*, do tipo *CSPMessageProtocol*. Futuramente, qualquer evento que ocorra na porta c será encaminhado para à porta a da instância de N, se esta contiver uma. Portanto, a estratégia para modelar renomeação é reduzida ao mapeamento entre as portas de *SystemController* e as portas das instâncias influenciadas pela relação de renomeação. □

Suponha $P = Q\ [a \leftarrow c]$. A Figura 4.19 mostra uma simulação da estrutura dinâmica de *SystemController*, apenas com suas sub-cápsulas ativas (cápsulas tracejadas), antes (esquerda) e depois (direita) da mudança de comportamento da instância da cápsula P para $Q\ [a \leftarrow c]$. As mensagens comunicadas à porta c de *SystemController* serão encaminhadas à porta a de Q, devido a relação de renomeação, presente na equação expandida. Se a cápsula Q informar

uma mudança de comportamento, sua referência na equação expandida será substituída, mas a relação de renomeação será preservada. Assim, as mensagens comunicadas à porta c de *SystemController* serão encaminhadas à porta a da nova cápsula (ou cápsulas), se esta contiver uma.

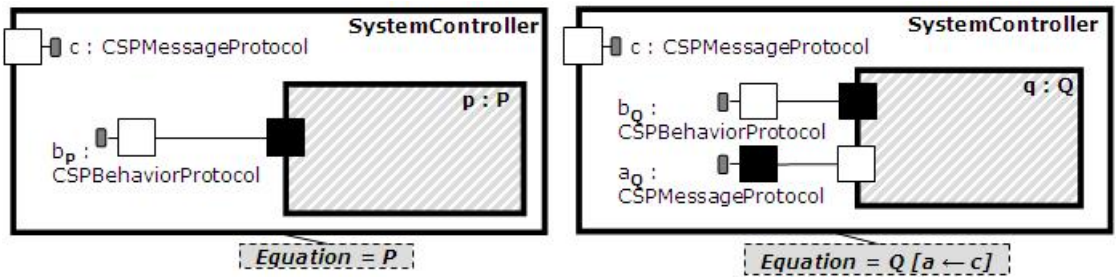
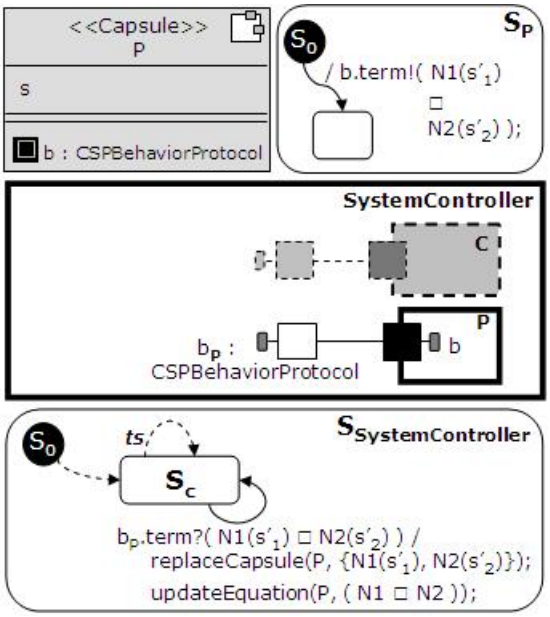


Figura 4.19 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para $Q\ [a \leftarrow c]$

A próxima regra avalia os processos que se comportam como uma escolha externa.

Regra 11. *Mapeamento de Escolha Externa*

$$P(s) = \begin{array}{l} N1(s'_1) \\ \Box \\ N2(s'_2) \end{array} \Longrightarrow$$



A cápsula P gerada através desta regra informa o nome e os parâmetros de duas cápsulas, combinados pelo operador de escolha externa.

A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada pelo evento $b_P.term?(N1(s'_1) \Box N2(s'_2))$ remove a instância de P e cria instâncias de $N1$ e $N2$, usando seus respectivos parâmetros. A equação expandida é atualizada substituindo a referência de P pela expressão $(N1 \Box N2)$, contendo as referências de $N1$ e $N2$. □

A Figura 4.20 mostra as sub-cápsulas ativas de *SystemController* antes (esquerda) e depois (direita) da mudança de comportamento de P para uma composição binária de Q e R. Embora

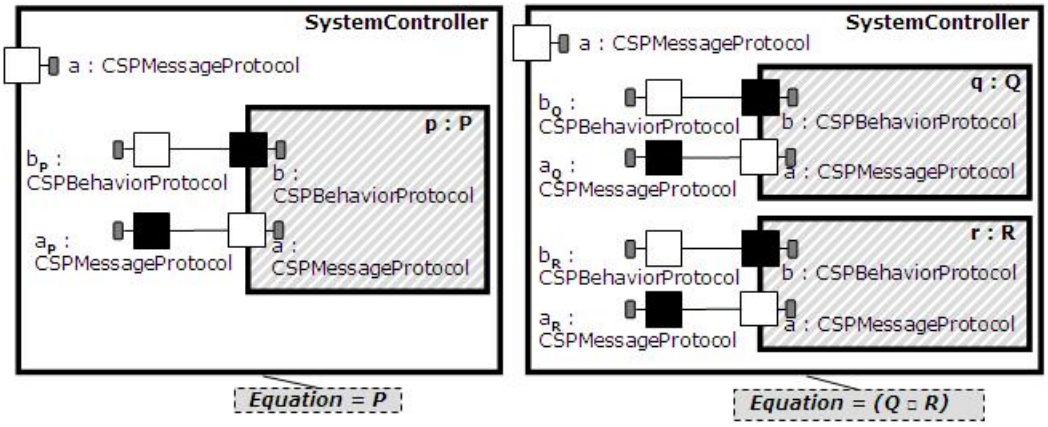


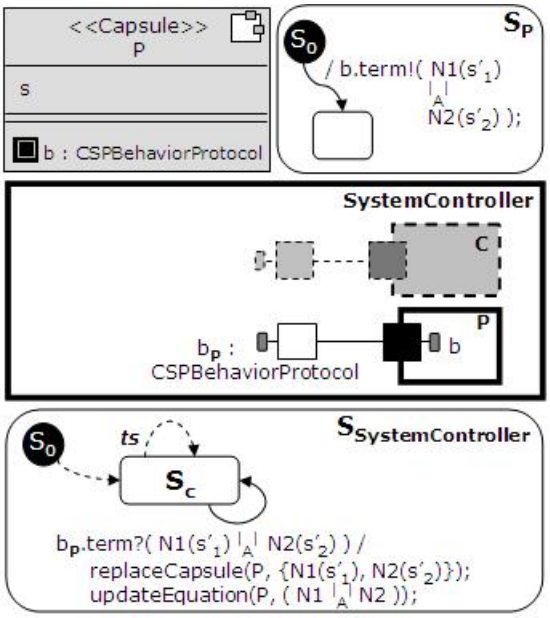
Figura 4.20 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para $Q \square R$

as duas cápsulas sejam instanciadas, apenas uma poderá permanecer ativa no futuro. A primeira que completar o processo de sincronização na porta a, de acordo com o padrão de comunicação do protocolo *CSPMessageProtocol*, será a escolhida. A cápsula preterida será removida, assim como sua referência na equação expandida. Se ambas as cápsulas estiverem simultaneamente aptas para sincronizar, então será feita uma escolha não-determinística entre elas. Se uma das cápsulas informar uma mudança de comportamento antes de qualquer evento que force a escolha entre elas, sua referência será substituída na equação, mas a semântica do operador de escolha externa ainda será aplicada, quando o evento ocorrer.

A próxima regra avalia os processos que se comportam como uma composição paralela generalizada. Através deste operador, representamos outras formas de paralelismo (alfabetizado e *interleaving*).

Regra 12. *Mapeamento de Paralelismo*

$$P(s) = \begin{array}{c} N1(s'_1) \\ || \\ A \\ N2(s'_2) \end{array} \Longrightarrow$$



A cápsula P obtida da aplicação desta regra informa o comportamento de uma composição paralela de duas outras cápsulas. Além do nome e dos parâmetros das novas cápsulas, é informado um conjunto de nomes de portas A, representando o alfabeto de sincronização. Assim, podemos representar outras formas de paralelismo, como o *interleaving*, que possui um conjunto de sincronização vazio.

A estrutura de *SystemController* é atualizada para que contenha uma sub-cápsula opcional de P. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada pelo evento $b_P.term?(N1(s_1) \underset{A}{||} N2(s_2))$ remove a instância de P, cria as novas instâncias de $N1$ e $N2$, com seus respectivos parâmetros, e troca a referência de P pela expressão $N_1 \underset{A}{||} N_2$ na equação expandida.

O comportamento associado ao paralelismo de sub-cápsulas está definida na expressão da equação expandida de *SystemController*. Cada evento que ocorrer nas portas de *SystemController* deve ser repassado às instâncias ativas que façam parte do paralelismo. Se o nome da porta estiver no conjunto de sincronização, todas as instâncias participantes do paralelismo devem confirmar o evento encaminhado. Se o nome da porta não estiver no conjunto de sincronização, uma escolha não-determinística é feita entre as instâncias que confirmarem os eventos enviados para suas portas. □

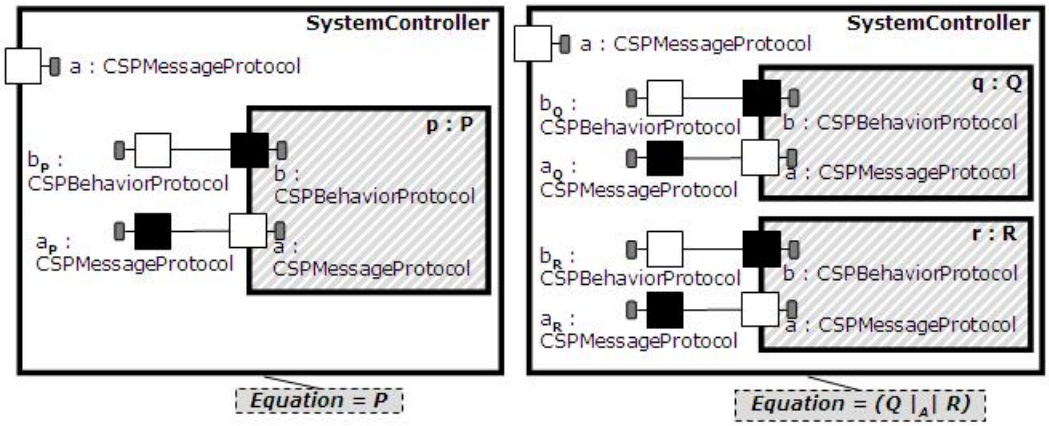


Figura 4.21 Estrutura de *SystemController* antes e depois da mudança de comportamento de P para $Q \underset{\{a\}}{||} R$

A Figura 4.21 mostra as sub-cápsulas ativas de *SystemController* antes (esquerda) e depois (direita) de P assumir o comportamento de uma composição paralela de Q e R. Os eventos ocorridos na porta a de *SystemController* devem ser encaminhados às instâncias de Q e R. Para que o processo de comunicação seja concluído, Q e R devem aceitar os eventos conjuntamente.

Suponha, agora, que a instância de R assuma o comportamento de ***STOP***. A Figura 4.22 mostra a nova configuração de *SystemController*. Nesta situação, a composição paralela não permite que a instância de Q sincronize nos eventos da porta a, visto que o segundo elemento da composição jamais aceitará sincronizar em qualquer evento. O comportamento da composição

paralela é igual a ***STOP***, exatamente como a semântica de paralelismo em CSP determina, para este caso.

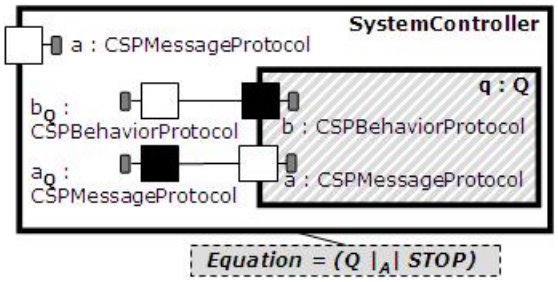


Figura 4.22 Estrutura de *SystemController* antes e depois da mudança de comportamento de *R* para ***STOP***

As demais regras de mapeamento, que contemplam internalização, escolha interna, composição sequencial e interrupção, estão descritas no Apêndice B.

As regras apresentadas têm o propósito de fazer um mapeamento sintático das equações de processos de CSP. A composição das cápsulas geradas e da cápsula *SystemController* simula o comportamento da especificação em relação ao ambiente externo. Exemplos do comportamento dinâmico de *SystemController* e suas sub-cápsulas são apresentados no Capítulo 5.

4.3 Mapeamento de Tipos de Dados

CSP usa tipos de dados para parametrizar os eventos que podem ocorrer através de um canal. Tipos de dados podem ser definidos através do conjunto de seus valores ou através de tipos já definidos. É possível, inclusive, definir tipos recursivos, que geralmente dão origem a conjuntos infinitos de valores. Segundo Roscoe [41], a sintaxe simplificada para a definição de tipos de dados é:

$$\begin{array}{lcll} \textit{datatype } T & ::= & v_0 \mid .. \mid v_n & (\textit{tipo simples}) \\ & \mid & T.T & (\textit{tipo composto}) \end{array}$$

onde $v_0..v_n$ são os valores constantes do tipo.

A sintaxe simplificada para a definição de canais é:

channel e
channel a : *T*

onde *e* é um canal vazio, representando um evento de sincronização que não transmite valores, e *a* é um canal que pode transmitir qualquer dos valores do tipo de dado *T*. Por simplicidade, consideramos que toda declaração de canal usa no máximo um tipo de dado. Assim, expressões do tipo

channel position : *Int.Int.Int*

devem ser reescritas para

$$datatype\ XYZ = Int.Int.Int$$
$$channel\ position : XYZ$$

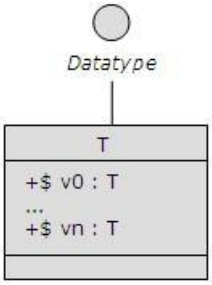
Tipos de dados representam classes de mensagens transmitidas entre elementos do sistema, ou classes básicas da camada de negócio, por isso são mapeados em classes UML simples. Tipos de dados compostos também devem ser mapeados em uma única classe, considerando-se que qualquer instância desta classe possa representar qualquer dos valores compostos deste tipo.

Na estratégia de mapeamento apresentada aqui, todas as classes que representam tipos de dados devem implementar uma mesma interface, denotada *Datatype*. A necessidade desta interface se dá porque as portas que representam canais CSP implementam um mesmo protocolo, *CSPMessageProtocol*, cujos sinais aceitam apenas objetos que implementam esta interface.

A regra a seguir é usada para mapear tipos de dados simples, cujos valores constantes são expressamente listados.

Regra 13. *Mapeamento de Tipo de Dado Simples*

$datatype\ T = v_0 \mid .. \mid v_n \quad \Longrightarrow$

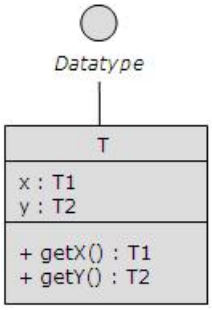


O tipo de dado T é mapeado em uma classe de mesmo nome, que implementa a interface *Datatype*. Os valores constantes do tipo, $v0, .., vn$, são mapeados em atributos públicos e estáticos da classe T. Assim, cada valor é usado como um identificador único da classe. □

A próxima regra trata do mapeamento de tipos de dados compostos, que usam a definição de outros tipos.

Regra 14. *Mapeamento de Tipo de Dado Composto*

$datatype\ T = T1.T2 \quad \Longrightarrow$



O tipo de dado T, composto da combinação dos tipos $T1$ e $T2$, é mapeado em uma classe de mesmo nome, que implementa a interface *Datatype*. Cada elemento da composição do tipo T é

mapeado em um atributo da classe T. O tipo do atributo é o tipo daquele elemento. Os métodos públicos $getX()$ e $getY()$ encapsulam o acesso a estes atributos. □

Por similaridade com a representação de mensagens transmitidas através do protocolo *CSP-MessageProtocol*, usamos uma classe específica para representar o tipo de dado dos canais vazios, como e. A Figura 4.23 mostra a interface *Datatype*, e a classe *EmptyType*, usada para representar o tipo de dado dos canais vazios.

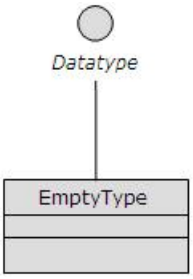


Figura 4.23 Classes que representam tipos de dados CSP

Uma alternativa para a representação de tipos enumerados é a utilização de classes enumeradas, presentes em UML [35] e UML 2.0 [35]. Classes enumeradas representam tipos enumerados do sistema, cujos valores são representados como identificadores da classe. Classes enumeradas são representadas da mesma forma que classes comuns, mas são explicitamente marcadas com o esteriótipo *«enumeration»*.

A representação de classes enumeradas em Java usa atributos públicos e estáticos para representar os valores da classe. A versão de Java 1.5 já possui um tipo de classe específico para esta representação, chamada *Enum*.

Ressaltamos que o mapeamento de tipos de dados apresentado aqui tem o propósito de associá-los a elementos concretos de modelagem, que encerram propriedades específicas ou conjuntos de valores relacionados. A implementação destas classes está fora do escopo desta dissertação. Mais detalhes sobre a tradução de tipos de dados CSP para classes podem ser encontrados em [17, 14, 16] e no Capítulo 5.

4.4 Representações Alternativas para Operadores CSP

A estratégia apresentada até aqui tem apontando para a representação de expressões CSP através de elementos diagramáticos de UML-RT, como máquinas de estado e diagramas de estrutura. Canais, por exemplo, são mapeados em portas, e seu comportamento implícito associado é representado através de máquinas de estado de protocolos e de cápsulas. Intuitivamente, os operadores de CSP deveriam ser mapeados da mesma forma. Entretanto, através da estratégia, a representação do comportamento dos operadores está concentrada na equação expandida de *SystemController*.

Inicialmente, a idéia era que existisse um elemento estrutural para representar a semântica de cada operador de CSP, como uma cápsula de controle. Assim, se uma composição paralela entre duas cápsulas fosse necessária, uma cápsula que representasse o operador de paralelismo

controlaria as cápsulas dos processos envolvidos. A Figura 4.24 exemplifica uma possível estrutura de cápsulas para $P \underset{\{\}}{||} Q$, considerando esta abordagem.

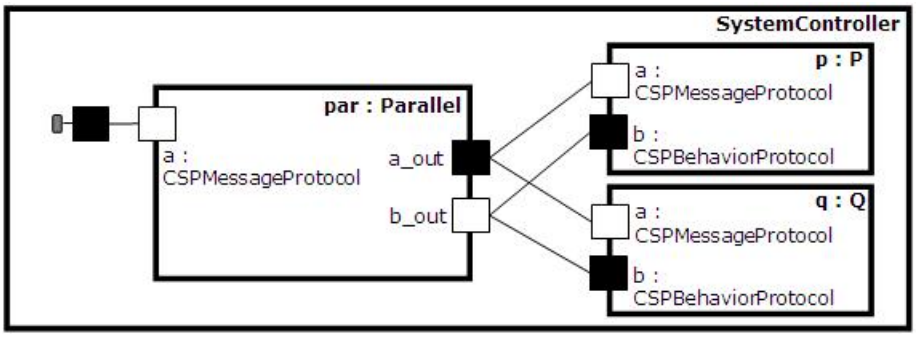


Figura 4.24 Estrutura de cápsulas para representar o paralelismo

Entretanto, pela natureza dinâmica dos processos em CSP, é impossível prever o comportamento futuro das cápsulas. Se uma das cápsulas presentes na composição paralela informar o comportamento de uma estrutura cujo conjunto de portas é diferente do original, a cápsula que representa o paralelismo deveria criar mais portas para conectar-se à nova estrutura. A Figura 4.25 mostra as mudanças necessárias, na estrutura do exemplo anterior, para se adequar à substituição da instância de *P* por *R*. A presença da porta *c* na estrutura da cápsula *R* implica na necessidade de novas portas (representadas por uma linha tracejada) na estrutura da cápsula *Parallel*. Entretanto, cápsulas não podem criar portas dinamicamente. Assim, torna-se impossível prever o "alfabeto" de uma cápsula que representa um paralelismo.

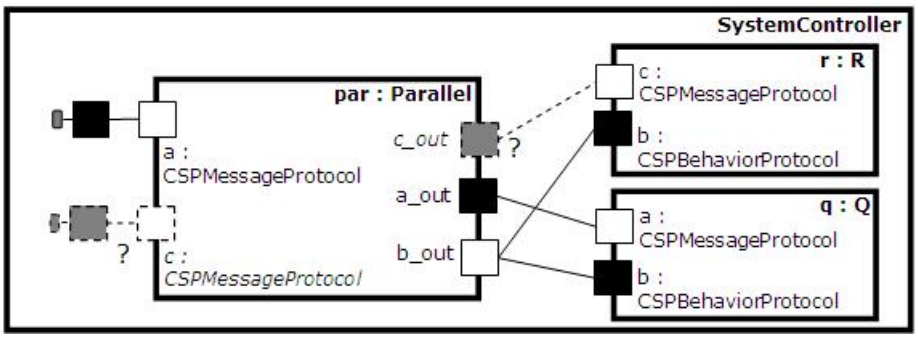


Figura 4.25 Mudança necessária na cápsula que representa o paralelismo

Esta abordagem seria válida se cada cápsula de processo, conectada à cápsula do operador, fosse auto-contida, como a cápsula *SystemController*, e não requeresse sua substituição por outras cápsulas dinamicamente. Seria possível, inclusive, usar cápsulas de processos como sub-cápsulas das cápsulas de operadores, visto que o problema da recursão mútua, mencionado na Seção 4.2.2, não ocorreria no contexto do operador. As regras e implicações práticas para a construção das cápsulas de operadores envolvem refatoramento de cápsulas, e são indicadas como trabalho futuro.

Protocolos de UML-RT também não podem definir as regras dos operadores de CSP, porque protocolos são usados para definir regras de comunicação entre portas (como as regras de comunicação em duas fases). Dizer que estas regras incluem as diversas formas de composição entre cápsulas (como composições paralelas ou escolhas não-determinísticas) seria o mesmo que pré-estabelecer a forma como uma cápsula deve ser composta com outras, ferindo os conceitos de encapsulamento e reuso. Por exemplo, uma porta que realiza um protocolo de comunicação paralela poder ser usada apenas para este propósito, impossibilitando que a cápsula que a contém seja usada de outra forma.

4.5 Considerações Finais

Neste capítulo, foram analisadas as implicações práticas para representar a estrutura e o comportamento de especificações CSP através de UML-RT. Questões como sincronização múltipla, não-determinismo, recursões mútuas entre processos, e comportamentos dinâmicos foram consideradas para determinar a construção de cápsulas e protocolos.

UML-RT não possui elementos diagramáticos para representar estas propriedades de forma tão concisa e elegante quanto CSP, o que torna os modelos extensos e complexos. Além disto, as regras geram um número excessivo de cápsulas, dificultando mais ainda a interpretação do modelo. Regras de refatoramento podem minimizar estes problemas, e aproximar o modelo UML-RT de uma estrutura mais amigável. Um exemplo é a transformação das cápsulas que representam eventos consecutivos em uma única cápsula. Assim, um processo como

$$P = a1 \rightarrow a2 \rightarrow a3 \rightarrow P$$

é reescrito como

$$P = a1 \rightarrow P1$$
$$P1 = a2 \rightarrow P2$$
$$P2 = a3 \rightarrow P$$

e dá origem a três cápsulas. Através do refatoramento, este processo P poderia ser representado por uma única cápsula, cuja máquina de estados aceita os três eventos consecutivamente. A construção de regras de refatoramento para cápsulas é sugerida como trabalho futuro.

As regras de mapeamento apresentadas neste capítulo foram propostas para serem composicionais e preservarem a semântica dos operadores de CSP. A prova formal destas propriedades também é indicada como trabalho futuro. A estratégia para o mapeamento de uma especificação inteira usa regras para a reescrita de expressões CSP (para simplificar e padronizar as expressões), seguidas de regras para a construção de elementos estruturais de UML-RT. Um estudo de caso da aplicação da estratégia é apresentado no capítulo 5.

CAPÍTULO 5

Estudo de Caso e Automação da Estratégia

Como a estratégia descrita no Capítulo 4 é sistemática, temos algumas fases bem definidas que nos permitem partir de um modelo em CSP e chegar a um resultado final descrito em UML-RT. Estas fases são listadas a seguir, na seqüência em que são executadas:

- 1 - Normalização das equações de processos.
- 2 - Mapeamento dos tipos de dados CSP em classes UML.
- 3 - Criação da cápsula principal do modelo (e controladora das cápsulas de processos).
- 4 - Mapeamento das equações de processos em cápsulas UML-RT, e definição progressiva do comportamento e da estrutura da cápsula controladora para gerenciá-las.

As fases 1, 3 e 4 abordam basicamente equações de processos, e devem ser executadas nesta ordem, como mencionado na Seção 4.2. Atualmente, a fase 2 (Mapeamento dos tipos de dados CSP em classes UML) não tem dependência com as demais, e pode ser executada em qualquer ordem. Decidimos por esta seqüência com o intuito de facilitar o entendimento da estratégia em relação à forma como especificações CSP costumam ser construídas.

A estratégia de mapeamento é ilustrada na Seção 5.1, através de um estudo de caso. O objetivo é ilustrar na prática as regras de transformação apresentadas no Capítulo 4, bem como as fases listadas acima.

Além disso, como uma contribuição adicional desta dissertação, desenvolvemos uma ferramenta que automatiza a estratégia de mapeamento, gerando automaticamente modelos UML-RT no *Rational Rose RealTime* [39] a partir de especificações descritas em CSP_M [19] (*machine readable CSP*). A Seção 5.2 apresenta a arquitetura da ferramenta e algumas características dos modelos UML-RT gerados por esta.

5.1 Estudo de Caso

Nosso estudo de caso é uma especificação simplificada de um *Sistema de Computação em Grade* descrita em CSP, seguindo o padrão adotado nesta dissertação (ver detalhes nos Capítulos 2 e 4). Um sistema deste tipo pode ser entendido como uma plataforma de computadores geograficamente distribuídos, que compartilham recursos e serviços sem a necessidade de conhecimento onde os mesmos estão localizados. Quando uma máquina necessita de algum processamento remoto, esta o solicita a uma interface única (gerenciador), que escolhe entre as

máquinas disponíveis qual delas pode executar a tarefa solicitada. Por fim, o resultado é comunicado à máquina solicitante. Vale salientar que qualquer máquina pode solicitar ou executar um processamento.

Iniciamos a apresentação de nosso estudo de caso através das declarações de canais. Assumimos que tais declarações usam apenas um tipo de dado.

Eventos do tipo $a?x : C_x?y : C_y$, onde C_x e C_y são as respectivas restrições sobre x e y, são assumidos estarem representados como $a?m : \{x.y \mid x \in C_x,\ y \in C_y\}$. Adotamos esta representação porque os eventos em UML-RT podem carregar um único atributo por vez, embora este possa ser de um tipo composto.

Para declarar os canais, introduzimos os tipos de dados usados pelo estudo de caso. São três os tipos de dados:

$$\begin{array}{l} \mathit{datatype}\ \mathit{Valor} = S \\ \mathit{datatype}\ \mathit{Identificador} = M0 \mid M1 \mid M2 \\ \mathit{datatype}\ \mathit{Resultado} = \mathit{Identificador}.\mathit{Valor} \end{array}$$

O tipo de dado *Valor*, por simplicidade, possui um único valor constante, *S*, que representa o resultado de algum processamento. O tipo de dado *Identificador* possui três valores constantes, usados como identificadores das máquinas do sistema. O tipo composto *Resultado* é a combinação dos identificadores de máquinas e do resultado de algum processamento.

A partir destes tipos de dados, declaramos os canais em si. Em nosso estudo de caso, empregamos duas categorias de canais. A primeira está associada a cada máquina e por isso usa o tipo de dado *Identificador*. A segunda categoria é usada para lidar com resultados enviados ou recebidos por uma máquina (tipo de dado *Resultado*). Os canais são:

$$\begin{array}{l} \mathit{channel}\ \mathit{solicitarApp},\ \mathit{executarApp},\ \mathit{falhaApp} :\ \mathit{Identificador} \\ \mathit{channel}\ \mathit{resultadoIn},\ \mathit{resultadoOut} :\ \mathit{Resultado} \end{array}$$

Finalmente, os processos:

- $\mathit{Maquina}(id)$: processo que representa uma máquina do sistema distribuído, cujo identificador é dado pelo índice id.

  $$\mathit{Maquina}(id) = \mathit{MSolicitador}(id) \ |||\ \mathit{MExecutor}(id)$$

  Cada máquina é composta de dois módulos que se comportam de forma independente:

  - $\mathit{MSolicitador}(id)$: processo que representa o módulo solicitador de uma máquina. Sua função é enviar uma requisição de processamento remoto ao gerenciador do sistema (através do canal *solicitarApp*), e aguardar sua resposta, que pode ser o resultado do processamento (*resultadoIn*) ou uma mensagem de falha de alocação de recursos (*falhaApp*).

    $$\begin{array}{l} \mathit{MSolicitador}(id) = \\ \quad \mathit{solicitarApp}.id \rightarrow \\ \qquad \mathit{resultadoIn}?v : \{x.y \mid x \in \{id\}\} \rightarrow \mathit{MSolicitador}(id) \\ \qquad \Box \\ \qquad \mathit{falhaApp}.id \rightarrow \mathit{MSolicitador}(id) \end{array}$$

– *MExecutor*(id): processo que representa o módulo de processamento de uma máquina. Sua função é receber as requisições vindas do gerenciador do sistema (através do canal *executarApp*), e enviar o resultado do processamento solicitado (através do canal *resultadoOut*). Esta última ação envolve a produção do resultado, através da função *produzirResultado*().

$$produzirResultado() = S$$

$$\begin{aligned}&MExecutor(id) = \\ &\quad executarApp.id \rightarrow \\ &\quad resultadoOut?v : \{x.y \mid x \in \{id\}, y \in \{produzirResultado()\}\} \rightarrow \\ &\quad MExecutor(id)\end{aligned}$$

- *Gerenciador*$(maqLivres, tabela)$: representa a máquina responsável pelo gerenciamento das demais. As funções deste processo são alocar um recurso remoto e repassar resultado de um processamento remoto para a máquina solicitante. O gerenciador deve dispor de um cadastro de máquinas livres no momento (parâmetro *maqLivres*) e de uma tabela de máquinas em uso no momento (parâmetro *tabela*). Cada elemento da tabela deve conter o identificador da máquina que solicitou o recurso e o identificador da máquina alocada para processar aquele recurso.

$$\begin{aligned}&consultarSolicitador(tabela, idExecutor) = \\ &\quad \{solicitador \mid (solicitador, executor) \leftarrow tabela, executor == idExecutor\}\end{aligned}$$

$$\begin{aligned}&Gerenciador(maqLivres, tabela) = \\ &\quad (solicitarApp?idSolicitador \rightarrow \\ &\quad\ if\,(maqLivres\ != \{\}) \\ &\quad\quad then\ executarApp?idExecutor : maqLivres \rightarrow \\ &\quad\quad\quad Gerenciador(diff\,(maqLivres, \{idSolicitador, idExecutor\}), \\ &\quad\quad\quad\quad union(tabela, \{(idSolicitador, idExecutor)\})) \\ &\quad\quad else\ falhaApp.idSolicitador \rightarrow Gerenciador(maqLivres, tabela)\ ) \\ &\quad \Box \\ &\quad (resultadoOut?resExecutor : \{x.y\} \rightarrow \\ &\quad\ resultadoIn?resSolicitador : \{x.y \mid \\ &\quad\quad\quad\quad x \in \{consultarSolicitador(tabela, resExecutor.x)\}, \\ &\quad\quad\quad\quad y \in \{resExecutor.y\}\} \rightarrow \\ &\quad\ Gerenciador(union(maqLivres, \{idSolicitador, idExecutor\}), \\ &\quad\quad\quad diff\,(tabela, \{(idSolicitador, idExecutor)\}))\ )\end{aligned}$$

- *Maquinas*: representa a composição paralela das máquinas do sistema, que são independentes entre si.

$$Maquinas = Maquina(M0) \mid\mid\mid Maquina(M1) \mid\mid\mid Maquina(M2)$$

- *Sistema*: processo que representa todo o sistema de computação em grade, onde o gerenciador deve comunicar-se com todas as outras máquinas.

$$Sistema = Gerenciador(\{M0, M1, M2\}, \{\}) \underset{Maquinas}{\underset{||}{\{solicitarApp, executarApp, falhaApp, resultadoIn, resultadoOut\}}}$$

A partir da especificação apresentada anteriormente, ilustramos nossa estratégia de mapeamento de especificações CSP para modelos UML-RT na próxima seção.

5.1.1 Aplicando a Estratégia

Com o intuito de simplificar a apresentação desta seção, decidimos selecionar um sub-conjunto da especificação do estudo de caso para demonstrar a aplicação das regras. Para tal, escolhemos os processos *Maquina*, *MSolicitador* e *MExecutor*, que são os mais representativos, enquanto os processos *Gerenciador*, *Maquinas* e *Sistema* não são considerados durante a transformação.

5.1.1.1 Normalização das Equações de Processos

Como esforço inicial para aplicar as regras de mapeamento, todos os processos devem estar em uma das formas normais descritas na Seção 4.1. A aplicação das regras de normalização aos processos *MSolicitador* e *MExecutor* cria quatro novos processos ($MSolicitador0(id)$, $MSolicitador1(id)$, $MSolicitador2(id)$ e $MExecutor0(id)$). A ocorrência do operador de *interleaving*($|||$) é substituída por $\underset{\{\}}{||}$. O resultado desta fase de normalização é o seguinte trecho de especificação:

$$Maquina(id) = MSolicitador(id) \underset{\{\}}{||} MExecutor(id)$$
$$MSolicitador(id) = solicitarApp?id : \{x \mid x \in \{id\}\} \rightarrow MSolicitador0(id)$$
$$MSolicitador0(id) = MSolicitador1(id) \,\square\, MSolicitador2(id)$$
$$MSolicitador1(id) = resultadoIn?v : \{x.y \mid x \in \{id\}\} \rightarrow MSolicitador(id)$$
$$MSolicitador2(id) = falhaApp?id : \{x \mid x \in \{id\}\} \rightarrow MSolicitador(id)$$

$$MExecutor(id) = executarApp?id : \{x \mid x \in \{id\}\} \rightarrow MExecutor0(id)$$
$$MExecutor0(id) = resultadoOut?v : \{x.y \mid x \in \{id\}, y \in \{produzirResultado()\}\} \rightarrow MExecutor(id)$$

5.1.1.2 Mapeamento dos Tipos de Dados

A próxima fase de nossa estratégia consiste em transformar os tipos de dados definidos na especificação CSP em tipos válidos no contexto de UML-RT. Em nosso estudo de caso temos basicamente três tipos de dados: os tipos de dados simples, *Valor* e *Identificador*, mapeados

em classes UML através da Regra 13, e o tipo de dado composto *Resultado*, mapeado através da Regra 14. A Figura 5.1 exibe as classes UML geradas a partir da aplicação das regras mencionadas. Os valores contantes dos tipos simples *Valor* e *Identificador* são mapeados em identificadores estáticos das respectivas classes. Cada elemento do tipo composto *Resultado* é mapeado em um atributo da classe *Resultado*. Os métodos *getIdentificador*() e *getValor*() encapsulam o acesso a estes atributos.

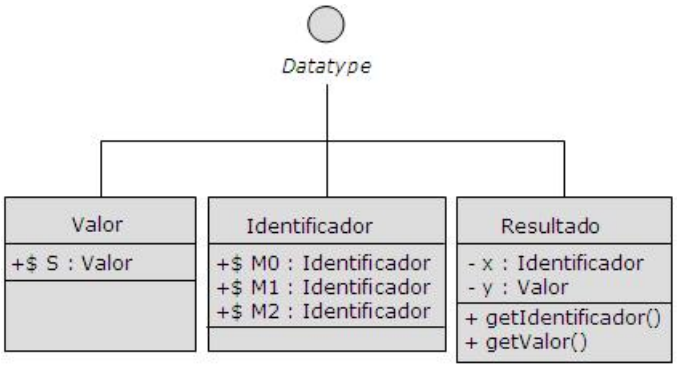


Figura 5.1 Classes *Valor*, *Identificador* e *Resultado*

5.1.1.3 Criação da Cápsula Controladora

Após transformar os tipos de dados CSP em classes UML-RT, iniciamos a construção da cápsula *SystemController*, que irá controlar o comportamento deste subconjunto do sistema que estamos usando como exemplo. O processo de construção desta cápsula inclui a definição da equação inicial do sistema, através da indicação do nome e dos parâmetros do processo que inicializa o comportamento do o sistema. Em nosso estudo de caso, selecionamos o processo *Maquina(M0)* como o processo inicial do sistema.

A Figura 5.2 mostra a estrutura e o comportamento iniciais de *SystemController*. A estrutura inicial de *SystemController* é vazia, visto que nenhum processo foi traduzido em cápsula até o momento. A máquina de estados inicial de *SystemController* possui apenas uma transição, partindo do estado inicial (S_0) e chegando no estado (S_C). Esta transição contém os comandos necessários para criar a instância da cápsula *Maquina*, cujo parâmetro de instanciação é *M*0, bem como inicializar a equação expandida com a referência a esta instância.

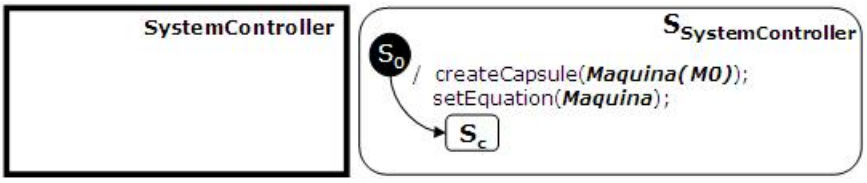


Figura 5.2 Estrutura e Comportamento Iniciais da Cápsula Controladora

5.1.1.4 Mapeamento das Equações de Processos

Finalmente, a última fase refere-se à transformação das equações de processos CSP para cápsulas UML-RT. Equações de processos são avaliadas em seqüência, inclusive aquelas originadas da normalização de outros processos. A Figura 5.3 mostra a cápsula *Maquina*, resultante da aplicação da Regra 12 (mapeamento de paralelismo) à equação do processo $Maquina(id)$. O parâmetro *id* do processo é mapeado no atributo de mesmo nome, na cápsula. O lado direito da equação do processo dá origem à porta comportamental *b* na estrutura da cápsula, e à transição inicial da máquina de estados da cápsula. O comando

$$b.term!(MSolicitador(id) \underset{\{\}}{||} MExecutor(id))$$

é usado para informar ao ambiente externo que a cápsula vai passar a se comportar como uma composição paralela entre $MSolicitador(id)$ e $MExecutor(id)$.

Figura 5.3 Cápsula *Maquina*

A cápsula *SystemController* também é influenciada pela aplicação da regra de mapeamento ao processo $Maquina(id)$. A Figura 5.4 mostra a estrutura e a máquina de estados de *SystemController* imediatamente após esta aplicação. Uma sub-cápsula opcional, do tipo *Maquina*, é adicionada à sua estrutura. Adicionalmente, a máquina de estados de *SystemController* é atualizada com transições disparadas pelas mensagens enviadas por instâncias desta sub-cápsula. A ação associada à transição inicial de *SystemController* foi omitida para facilitar a visualização.

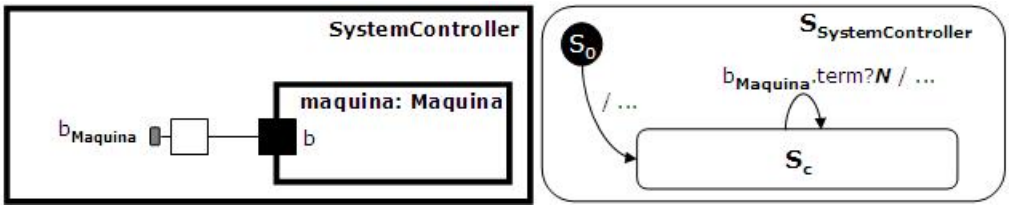


Figura 5.4 Cápsula *SystemController*, após mapeamento do processo *Maquina*

O próximo mapeamento é aplicado ao processo $MSolicitador(id)$. A Figura 5.5 mostra a cápsula *MSolicitador*, resultante da aplicação da Regra 8 (mapeamento de prefixo) à equação deste processo. O parâmetro *id* do processo é mapeado no atributo de mesmo nome, na cápsula. O lado direito da equação do processo dá origem à porta *solicitarApp* (representando o canal de comunicação de mesmo nome), e à porta comportamental *b* na estrutura da cápsula. As transições da máquina de estados de *MSolicitador* atendem ao padrão de comunicação estabelecido pelo protocolo *CSPMessageProtocol*. A seqüência de eventos *solicitarApp.request*?*m*

e *solicitarApp.commit* representa o comportamento de um evento síncrono em CSP, e leva à execução do comando

$$b.term!MSolicitador0(id)$$

que representa o comportamento esperado após a comunicação anterior.

A estrutura da cápsula *SystemController* é alterada para controlar o uso das instâncias da cápsula *MSolicitador*, o que inclui a replicação da porta *solicitarApp* de *MSolicitador* (Figura 5.6). A máquina de estados de *SystemController* também é alterada, para repassar as mensagens comunicadas por sua porta *solicitarApp* às sub-cápsulas ativas que possuam uma porta deste nome. No caso, as instâncias da cápsula *MSolicitador*.

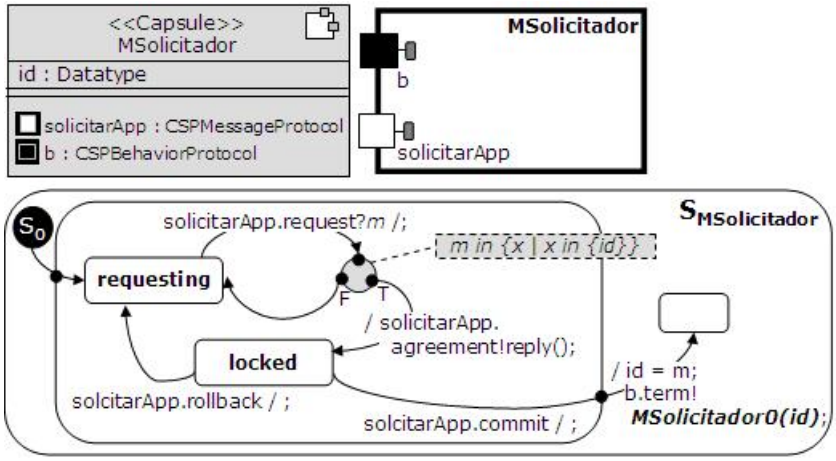


Figura 5.5 Cápsula *MSolicitador*

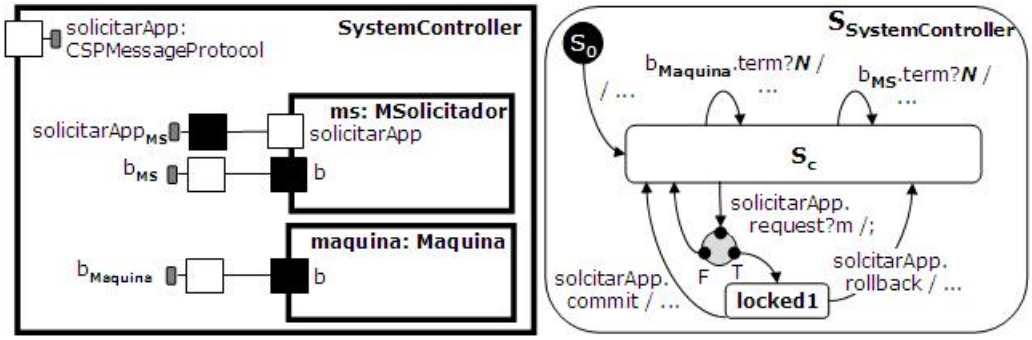


Figura 5.6 Cápsula *SystemController*, após mapeamento do processo *MSolicitador*

A Figura 5.7 mostra a cápsula *MSolicitador0*, resultante da aplicação da Regra 11 (mapeamento de escolha externa) à equação do processo $MSolicitador0(id)$. Da mesma forma que nos mapeamentos anteriores, a cápsula *SystemController* é alterada para controlar o uso das instâncias da cápsula *MSolicitador0*. A partir deste ponto, omitimos estas alterações por serem similares às anteriores.

Figura 5.7 Cápsula *MSolicitador*0

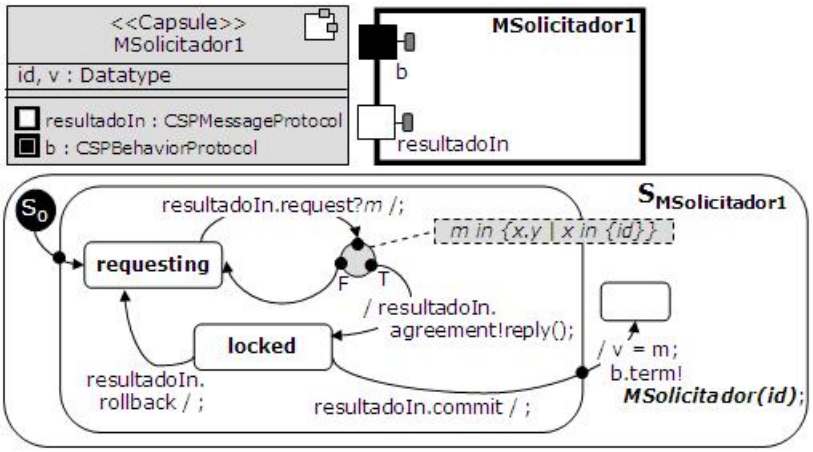


Figura 5.8 Cápsula *MSolicitador*1

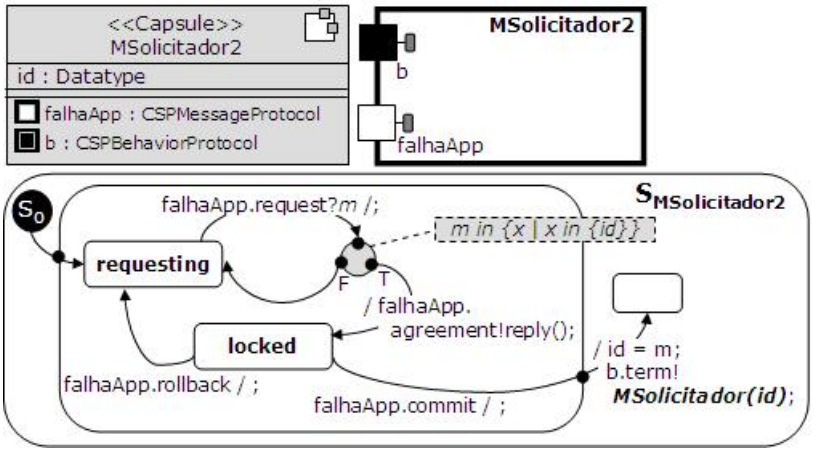


Figura 5.9 Cápsula *MSolicitador*2

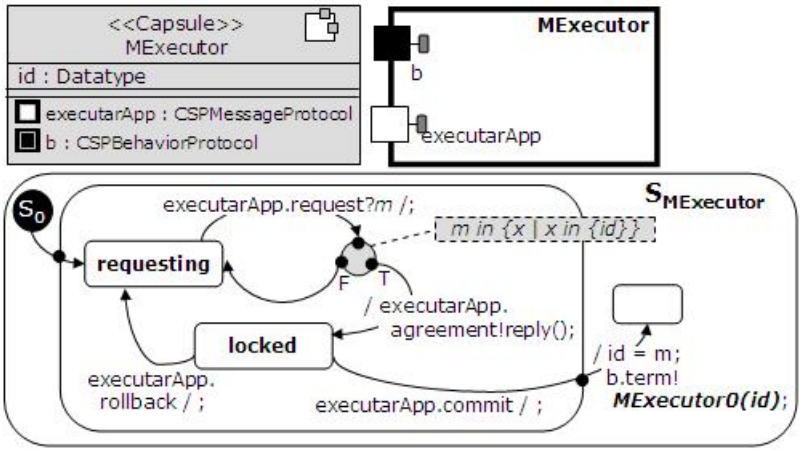


Figura 5.10 Cápsula *MExecutor*

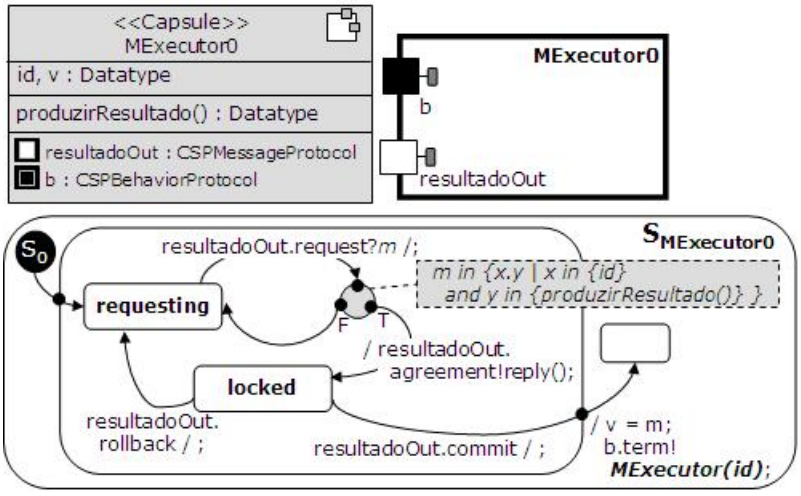


Figura 5.11 Cápsula *MExecutor*0

As Figuras 5.8, 5.9, 5.10 e 5.11 mostram as cápsulas *MSolicitador*1, *MSolicitador*2, *MExecutor* e *MExecutor*0 resultantes da aplicação da Regra 8(mapeamento de prefxo) às equações dos processos com os respectivos nomes.

Finalmente, a Figura 5.12 mostra a estrutura e a máquina de estados de *SystemController* resultantes da criação das cápsulas de processos. As sub-cápsulas de *SystemController* são na verdade apontadores para cápsulas opcionais ainda não ativas. A máquina de estados de *SystemController* foi simplificada para facilitar sua visualização, e as transições tracejadas representam todas aquelas que foram omitidas na Figura 5.12, para facilitar a visualização. Considere que toda porta protegida e conjugada de *SystemController* é do tipo *CSPBehaviorProtocol*, e toda porta protegida e base de *SystemController* é do tipo *CSPMessageProtocol*.

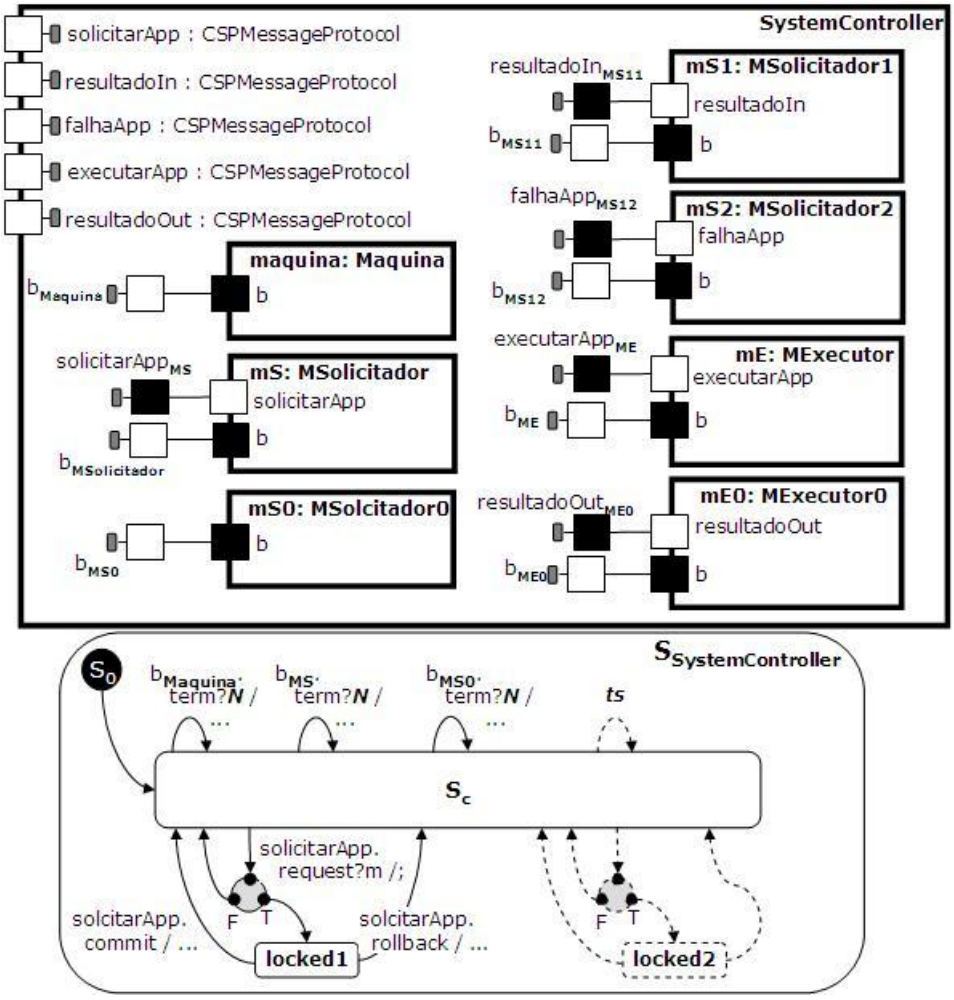


Figura 5.12 *SystemController* após o mapeamento dos processos

5.1.2 Simulação do Modelo Gerado

Esta seção apresenta, para um cenário específico de execução, o comportamento de *SystemController* para cada transição (ou seqüência de transições) que implique em mudança na equação expandida, e conseqüentemente no conjunto de sub-cápsulas ativas (cápsulas tracejadas). Com o intuito de simplificar a apresentação, consideramos nesta seção apenas as transições e sub-cápsulas relevantes para cada situação em questão.

O comportamento da cápsula *SystemController* começa pela sua inicialização. Durante a inicialização, a transição inicial cria uma instância ativa da sub-cápsula *Maquina*, e atualiza a equação expandida com a referência a esta instância (ver Figura 5.2). A transição inicial é responsável por iniciar o sistema com o processo escolhido para representá-lo (no caso, $Maquina(M0)$). A Figura 5.13 mostra a estrutura e o comportamento *SystemController* imediatamente após a criação da sub-cápsula *Maquina*.

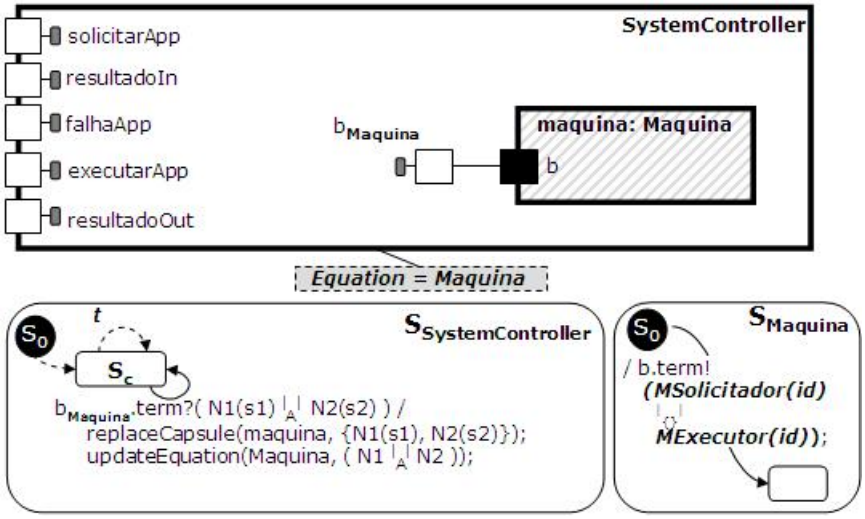


Figura 5.13 Configuração inicial de *SystemController*

Assim que inicializada, a instância da sub-cápsula *Maquina* informa automaticamente uma mudança de comportamento, através do comando

$$b.term!(MSolicitador(id) \underset{\{\}}{||} MExecutor(id))$$

que acarreta no disparo de uma transição em *SystemController*. Esta transição remove a instância inicial (*maquina*) e cria instâncias das sub-cápsulas *MSolicitador* e *MExecutor*. O parâmetro *id*, cujo valor é $M0$, é usado para a instanciação das sub-cápsulas. Em seguida, a equação expandida é atualizada com as referências a estas novas instâncias.

A Figura 5.14 mostra a estrutura de *SystemController* após a transição mencionada anteriormente. O comportamento de *SystemController* agora é definido pela seqüência de eventos *solicitarApp.request*?*m* e *solcitarApp.commit*, ambos recebidos do ambiente externo. As demais transições (setas tracejadas) não são disparadas por esta seqüência de eventos.

O primeiro evento, *solicitarApp.request*?*m*, é encaminhado à instância *mS* de *MSolicitador*, uma vez que esta é a única sub-cápsula ativa que possui uma porta de nome *solicitarApp*. Se a mensagem *m* for igual ao atributo *id* de *mS*, então haverá a confirmação da mensagem síncrona enviada a esta instância.

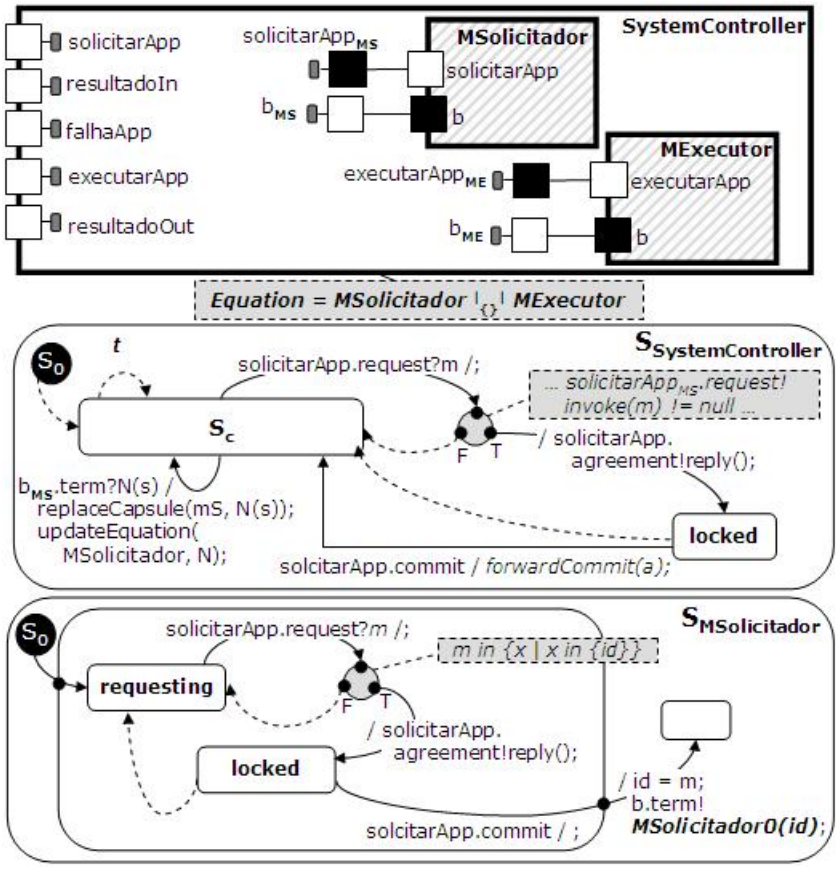


Figura 5.14 Configuração de *SystemController* após mudança de comportamento de *Maquina*

A chegada do segundo evento, *solicitarApp.commit*, representa a conclusão do processo de sincronização em duas fases, iniciado pelo evento anterior. O evento *solicitarApp.commit* é encaminhado à instância *mS*, que informa sua mudança de comportamento imediatamente após o recebimento do evento. As duas transições de *MSolicitador*, disparadas pelos dois eventos descritos anteriormente, são necessárias para preservar a semântica do evento *solicitarApp.id* na equação $solicitarApp.id \rightarrow MSolicitador0(id)$ (ver Página 66).

Em seguida, a instância ativa de *MSolicitador*, *mS*, comunica a mensagem

$$b.term?MSolicitador0(id)$$

que dispara a transição em *SystemController* que remove a instância *mS*, cria uma instância de *MSolicitador0*, e atualiza a equação expandida com a referência a esta instância. A Figura 5.15 mostra a estrutura de *SystemController* após o disparo desta transição.

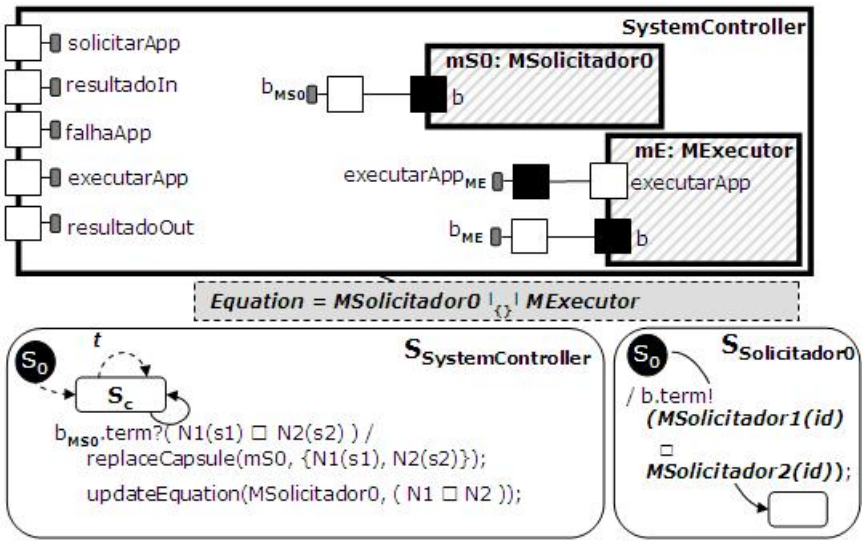


Figura 5.15 Configuração de *SystemController* após mudança de comportamento de *MSolicitador*

Assim que inicializada, a instância de *MSolicitador0*, *mS*0, executa o comando

$$b.term?(MSolicitador1(id) \square MSolicitador2(id))$$

que dispara a transição de *SystemController* que remove a instância *mS*0, cria instâncias de *MSolicitador1* e *MSolicitador2*, e atualiza a equação expandida. A Figura 5.16 mostra a estrutura de *SystemController* após o disparo desta transição. Neste momento, a configuração de *SystemController* é composta de uma escolha externa entre as instâncias *mS*1 e *mS*2, ambas em paralelo como a instância *mE*.

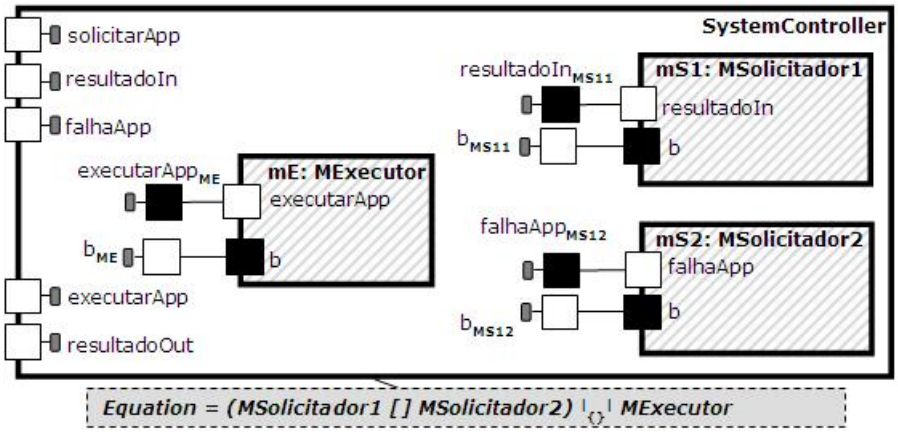


Figura 5.16 Configuração de *SystemController* após mudança de comportamento de *MSolicitador0*

5.2 Automação da Estratégia

A automação da estratégia de mapeamento envolve a interpretação das expressões CSP (através da identificação de tipos de dados e processos), a aplicação das regras de mapeamento a cada construção identificada, e a construção do modelo UML-RT incrementalmente.

O *IBM Rational Rose RealTime* [39] é atualmente o ambiente de desenvolvimento visual para UML-RT mais difundido no mercado e academia. Este ambiente permite a construção de modelos UML-RT através da sua interface gráfica (*GUI*) ou através de um mecanismo de extensibilidade [10] que simula a interação do usuário com o ambiente.

O mecanismo de extensibilidade do *Rose RealTime*, que é uma API (*Application Programming Interface*), permite criar uma cápsula, seus métodos, atributos, e diagramas de estrutura e máquina de estados da mesma forma que um usuário faria usando diretamente a interface gráfica do *Rose RealTime*. Esta *API* pode ser utilizada por outros aplicativos para mecanizar a construção de modelos UML-RT no *Rose RealTime*.

A ferramenta *FormalDev* (acrônimo para *Formal Developer*) foi desenvolvida para reconhecer especificações CSP, descritas em CSP_M (*machine readable CSP*), inicializar o *Rose RealTime* e, usando o mecanismo de extensibilidade deste aplicativo, construir gradativamente o modelos UML-RT enquanto aplica as regras de mapeamento descritas no Capítulo 4.

A Seção 5.2.1 apresenta uma breve descrição da arquitetura da *FormalDev*. A Seção 5.2.2 detalha algumas características operacionais dos modelos UML-RT gerados para o *Rose RealTime*, em comparação com os modelos abstratos apresentados nas regras de mapeamento.

5.2.1 Arquitetura da FormalDev

A ferramenta *FormalDev* consiste basicamente de um módulo controlador, responsável pela sistematização das regras, e de três componentes de apoio, todos coordenados pelo primeiro. Estes componentes podem ser vistos na Figura 5.17 e estão detalhados a seguir:

- CSPParser.dll - Biblioteca capaz de realizar a análise textual de uma especificação CSP_M e criar uma árvore sintática abstrata (*AST - abstract syntax tree*) que a represente.

- CSPTypeChecker.dll - Componente responsável por realizar a análise semântica da especificação CSP. Este componente usa a *AST* gerada pelo *CSPParser* e faz a sua verificação de tipos e a análise contextual. A *AST* original é "decorada" com informações necessárias para identificar declarações de tipos de dados, funções, e equações de processos. Através deste componente é possível ainda aplicar as regras de normalização de processos, alterando a *AST*.

- Rrtres.dll - Como mencionado, a *FormalDev* faz uso do mecanismo de extensibilidade do *Rose RealTime*. Este componente permite não apenas simular a interação do usuário com a interface gráfica do *Rose RealTime*, mas também controlar o aplicativo como uma aplicação embarcada, executando em plano de fundo. Assim, é possível criar e atualizar elementos de um modelo UML-RT através de ações coordenadas pelo módulo controlador da *FormalDev*.

- FormalDev.exe - Módulo controlador do aplicativo, responsável pela coordenação dos demais componentes e sistematização da estratégia de mapeamento.

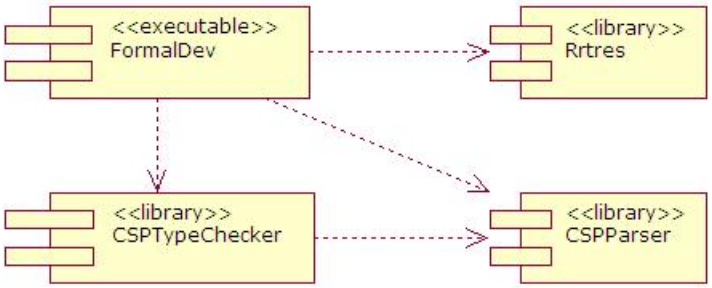


Figura 5.17 Arquitetura de Componentes do *FormalDev*

A Figura 5.18 exibe a tela principal da *FormalDev* antes da aplicação da estratégia de mapeamento ao estudo de caso descrito na seção anterior. A aba *CSPM* exibe o código da especificação CSP, carregada na ferramenta através da opção de menu *File*. Após carregar uma especificação CSP_M na *FormalDev*, a árvore sintática (*AST*) é gerada e exibida na aba *CSP Specification Tree*. Neste momento, as verificações sintática e semântica da especificação já foram processadas. O operador de *interleaving*(|||), usado no processo *Maquina*, é representado na árvore sintática como o operador de paralelismo generalizado, onde o conjunto de sincronização é vazio ($\underset{\{\}}{||}$).

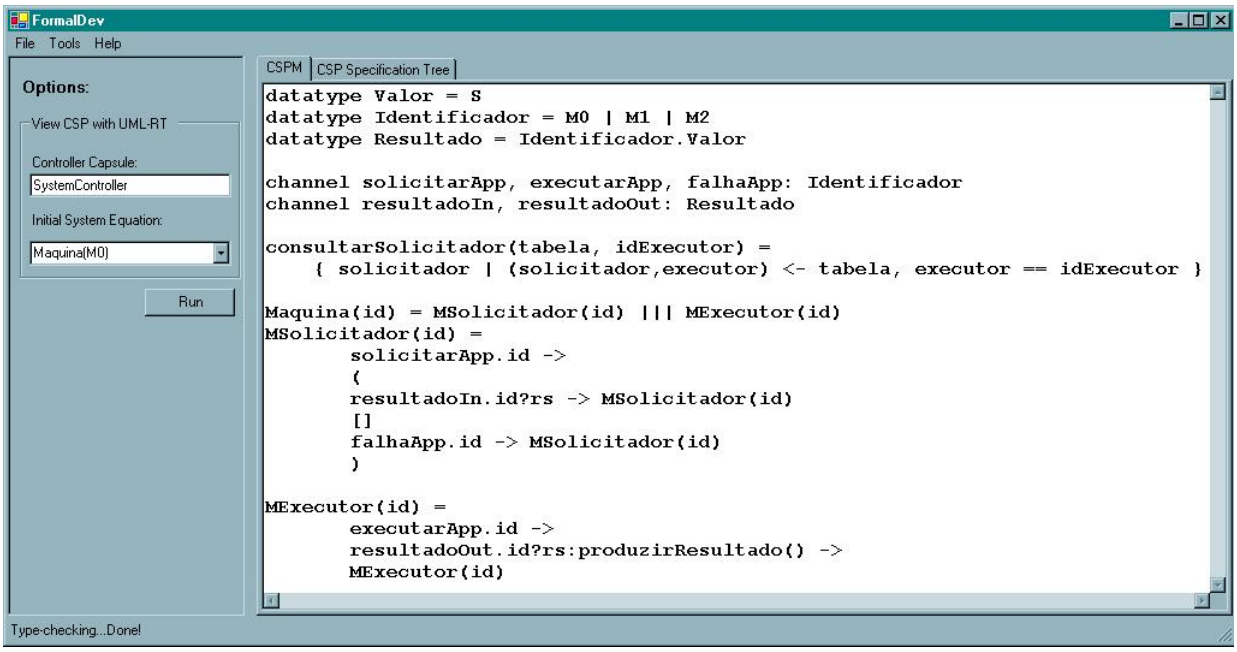


Figura 5.18 Tela prinicipal do *FormalDev*

5.2.2 Características dos Modelos UML-RT Gerados

As ferramentas de apoio a UML-RT disponíveis no mercado, como o *Rational Rose RealTime*, foram idealizadas para incorporar os conceitos de UML-RT a modelos UML reais, através de novos esteriótipos e da adaptação de diagramas. Os modelos resultantes reforçam os conceitos de cooperação entre objetos ativos, comunicação baseada em eventos, e estruturas *plug-and-play*.

Entretanto, nem todos os conceitos de UML-RT são aplicados aos modelos no *Rose RealTime*. Existem algumas limitações práticas de modelagem, bem como imprecisões na especificação original de UML-RT. Por exemplo, o conceito de estados concorrentes de UML-RT não tem semântica clara[1], e é melhor representado através de cápsulas concorrentes [43], como exposto no Capítulo 3. As regras de utilização de protocolos multi-papéis também não são precisamente definidas. Não é claro, por exemplo, se portas podem atender a mais de um papel, o que implicaria em herança múltipla. Por isso a ferramenta limita-se a um subconjunto da especificação de UML-RT.

Felizmente, as cápsulas e protocolos obtidas das regras de mapeamento têm o padrão estrutural e comportamental totalmente adaptável aos modelos do *Rational Rose*. Em parte, isto se justifica porque as regras de mapeamento foram idealizadas considerando muitos dos aspectos práticos e limitações de UML-RT, como os apresentados no parágrafo anterior.

No *Rose RealTime*, os modelos também servem como base para geração de código executável em Java ou C++. O gerador de código usa as propriedades dos elementos do modelo para produzir código fonte, e dispõem de bibliotecas de classes auxiliares que dão suporte ao código gerado. Por exemplo, se um modelo UML-RT é usado para extrair código Java, então todas as cápsulas do modelo serão traduzidas em classes que generalizam a classe *com.rational.rosert.Capsule*, da biblioteca de classes auxiliares em Java. Esta classe encerra propriedades comuns a todas as cápsulas, como o *thread* de controle lógico e *buffers* para armazenar as mensagens recebidas em suas portas. Já os apontadores para sub-cápsulas são instâncias da classe *com.rational.rosert.CapsuleRole*, que encerra propriedades como a cardinalidade, o tipo (fixa, opcional ou *plug-in*), e a referência à cápsula que será instanciada quando aquele apontador estiver ativo. Papéis de protocolos, por sua vez, são generalizações da classe *com.rational.rosert.ProtocolRole*, que implementa as primitivas de comunicação síncrona (***invoke*** e ***reply***) e assíncrona (***send***). A Figura 5.19 mostra as classes mencionadas, presentes na biblioteca Java do *Rose RealTime*.

O *Rose RealTime* permite ainda que métodos, ações e condições de controle (guardas e pontos de escolha) sejam codificados na linguagem desejada. Assim, todos os detalhes de implementação, necessários para construir o código executável, ou estão contidos na biblioteca de classes auxiliares ou são extraídos do modelo automaticamente.

Para facilitar a execução dos modelos gerados no *Rose RealTime*, as ações das máquinas de estado das cápsulas, descritas em pseudo-código nas regras de mapeamento, são traduzidas em código Java. Entretanto, os modelos gerados são semi-executáveis, visto que as regras de mapeamento traduzem funções e expressões lógicas descritas em CSP em métodos de cápsu-

[1] A especificação de UML não determina uma semântica exata para estados concorrentes por considerar que esta é uma decisão específica da arquitetura do sistema [34].

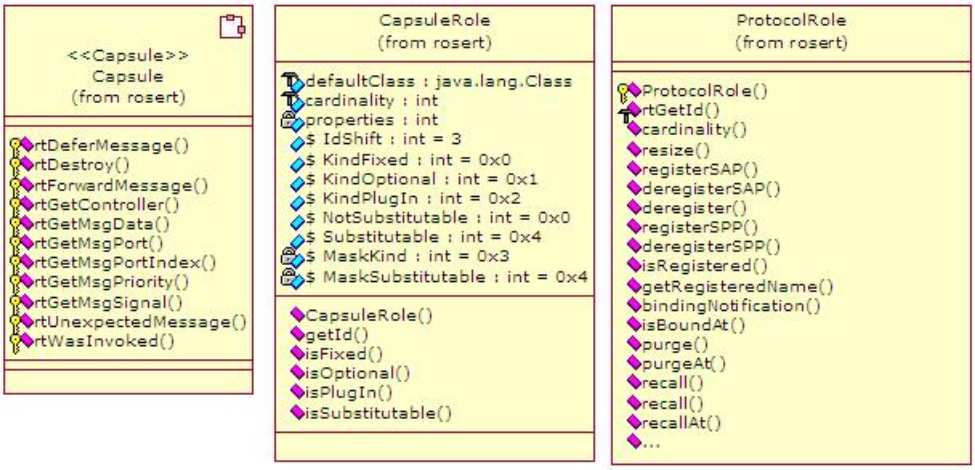


Figura 5.19 Classes da biblioteca *com.rational.rosert* do *Rose RealTime*

las, mas o corpo destes métodos é apenas comentado com a equação algébrica que o originou. A transformação destas equações algébricas em código executável está fora do escopo desta dissertação, entretanto apontamos esta tarefa como uma melhoria futura da ferramenta *Formal-Dev*, visto que alguns trabalhos já abordam a transformação de equações algébricas em CSP para Java [17, 16].

Desenvolvemos uma biblioteca de classes Java, chamada *CSPCore*, para auxiliar a execução do modelo UML-RT. Esta biblioteca é importada a todo modelo UML-RT gerado, e é composta dos seguintes pacotes principais:

- *datatypes* - contém as classes e interfaces usadas para criar classes que representam tipos de dados. A Figura 5.20 mostra a estrutura de classes deste pacote.

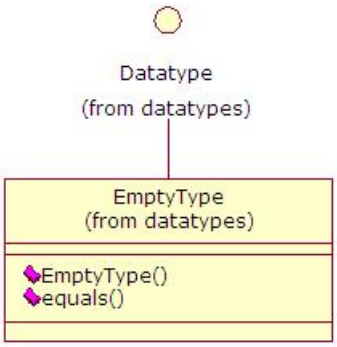


Figura 5.20 Classes do pacote *datatypes* de *CSPCore*

- *expression* - As mensagens transmitidas através das portas do tipo *CSPBehaviorProtocol*, representadas nas regras de mapeamento com a própria expressão CSP, são traduzidas em classes Java que representam a sintaxe das expressões. A Figura 5.21 mostra o diagrama de classes deste pacote. Estas mesmas classes são usadas para compor a equação expandida da cápsula controladora.

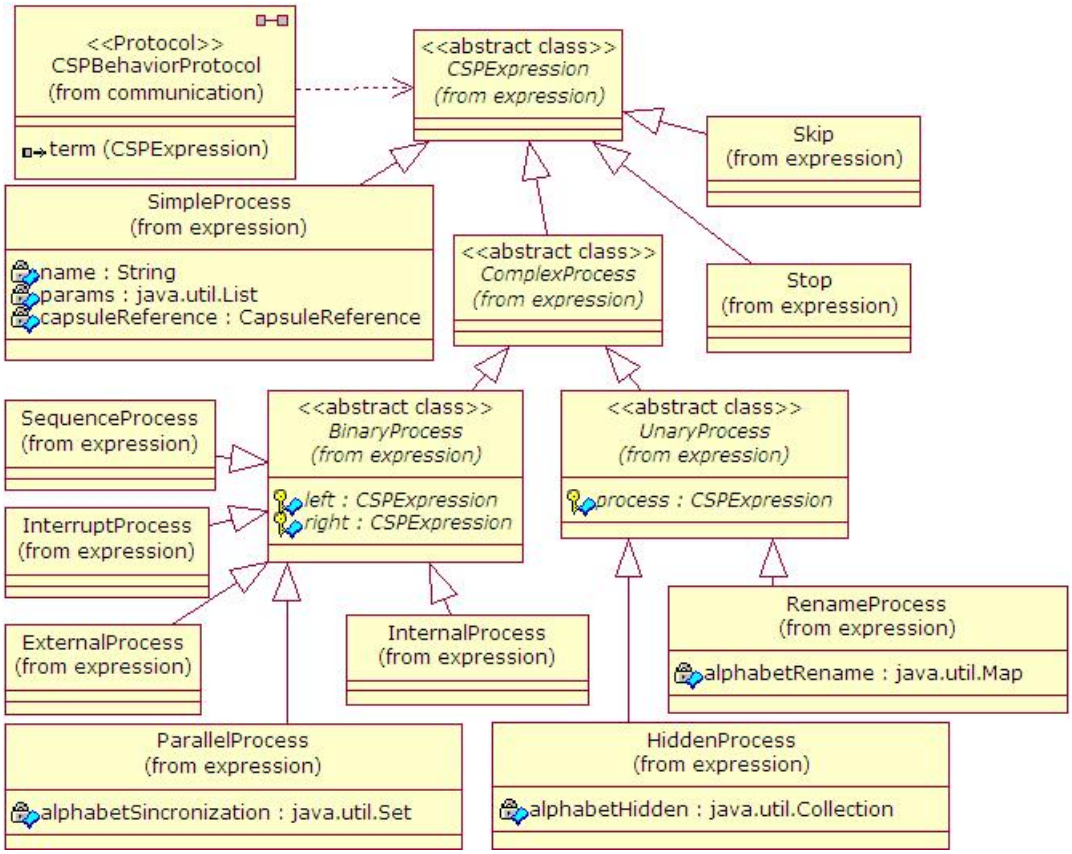


Figura 5.21 Classes do pacote *expression* de *CSPCore*

- *communication* - contém os protocolos *CSPMessageProtocol* e *CSPBehaviorProtocol*. A Figura 5.22 mostra as classes deste pacote. Veja que o sinal *request*, do protocolo *CSPMessageProtocol*, aceita qualquer mensagem do tipo *Datatype*, que é a interface comum a todas as classes usadas para representar tipos de dados. Veja também que o sinal *term*, do protocolo *CSPBehaviorProtocol*, aceita qualquer mensagem do tipo *CSPExpression*, que é a interface comum a todas as classes usadas para representar termos de processos.

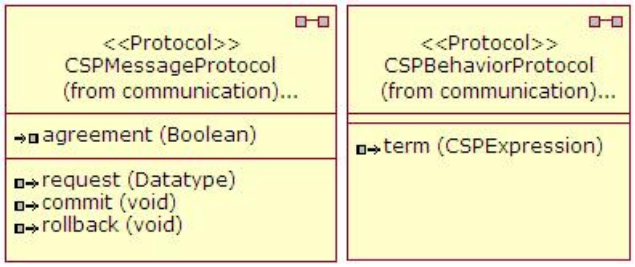


Figura 5.22 Classes do pacote *communication* de *CSPCore*

- *systemcontroller* - contém a cápsula *CapsuleContainerSystem*, usada como super-classe da cápsula controladora do modelo. Os métodos de *SystemController*, mencionados nas regras de mapeamento, estão codificados em Java na cápsula *CapsuleContainerSystem*, e são transmitidos a *SystemController* por herança. *CapsuleContainerSystem* é usada apenas por uma facilidade de implementação, e sua estrutura e máquina de estados são vazios. A Figura 5.23 mostra a cápsula *CapsuleContainerSystem*.

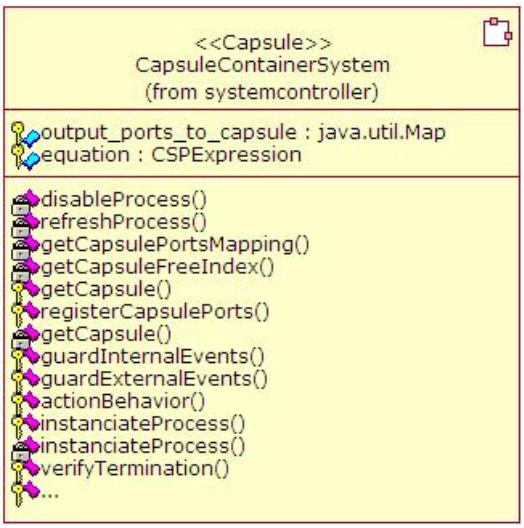


Figura 5.23 Cápsula *CapsuleContainerSystem* de *CSPCore*

A Figura 5.24 mostra a máquina de estados da cápsula *Maquina* no padrão usado nas regras de mapeamento (esquerda), e sua representação equivalente no *Rational Rose RealTime* (direita), usando as classes auxiliares. O construtor da classe *SimpleProcess* recebe dois parâmetros, usados para informar o nome e os parâmetros de uma cápsula. por sua vez, o contrutor

de *ParallelProcess* recebe os parâmetros que representam o processo da esquerda, o conjunto de sincronização, e o processo da direita, respectivamente.

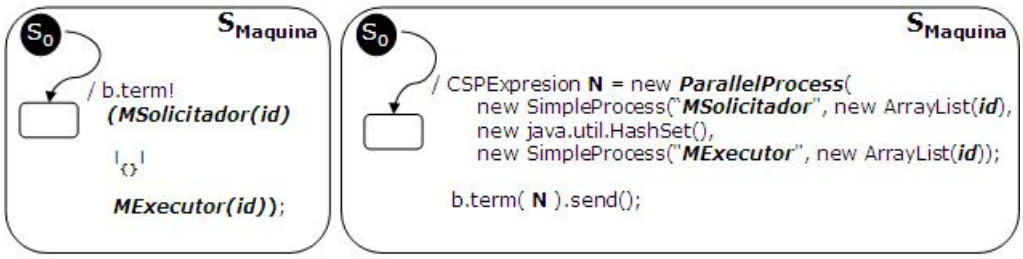


Figura 5.24 Representações da cápsula *Maquina* nas regras de mapeamento (esquerda) e no *Rose RealTime* (direita)

Por fim, a utilização de sub-cápsulas opcionais permite que várias instâncias de uma mesma cápsula sejam criadas por demanda. O número de instâncias simultâneas deve ser igual ou menor à cardinalidade da referência. Esta cardinalidade é fixa, e não pode ser alterada em tempo de execução. Decidimos por usar um número padrão para determinar a cardinalidade das sub-cápsulas de *SystemController*, assumindo que este número representa a quantidade máxima de instâncias de uma cápsula que podem existir simultaneamente no sistema. A Figura 5.24 exibe o diagrama de classes simplificado das cápsulas do estudo de caso, descrito na seção anterior.

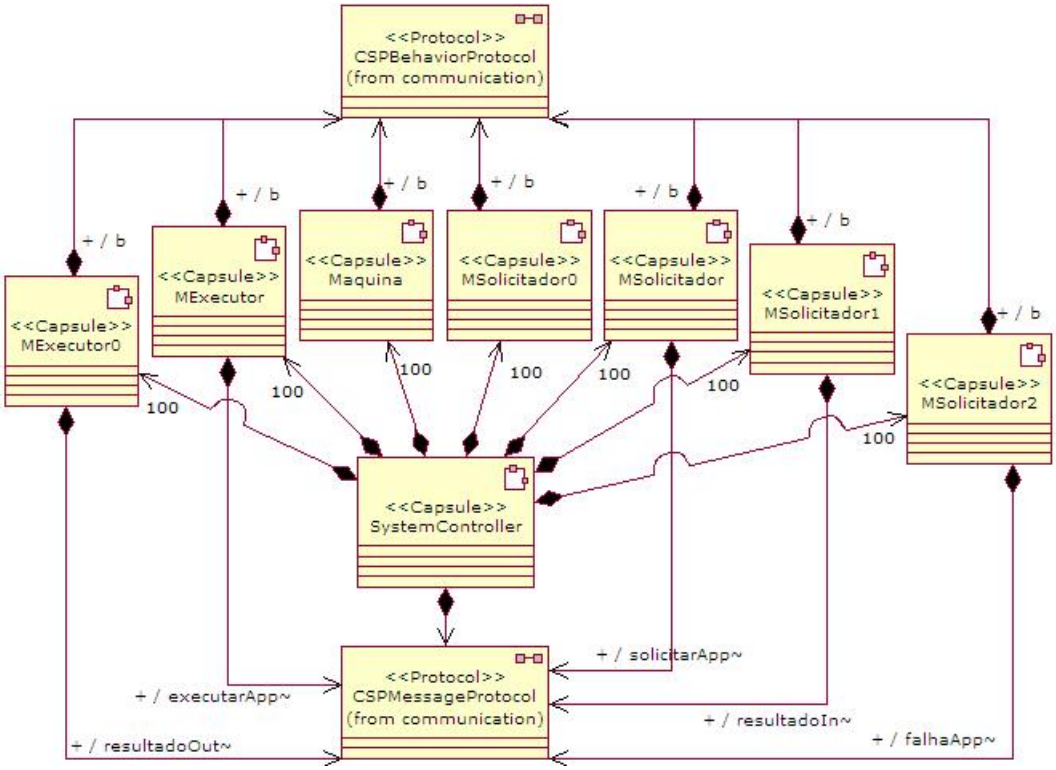


Figura 5.25 Diagrama de Classes do Estudo de Caso no *Rose RealTime*

CAPÍTULO 6

Conclusão

Neste trabalho, apresentamos uma estratégia sistemática para o mapeamento de processos e tipos de dados CSP em cápsulas, protocolos e classes UML-RT. O mapeamento foi motivado pelas semelhanças entre tipos de dados e classes, e entre processos e cápsulas.

A estratégia permite que modelos UML-RT sejam construídos levando em consideração as propriedades comportamentais de especificações CSP. Uma contribuição é a possibilidade de representar estas propriedades através de uma notação amplamente divulgada e empregada na academia e indústria para a documentação e projeto de aplicações.

A estratégia de mapeamento é composta de um conjunto de regras independentes e composicionais, aplicáveis a padrões de expressões CSP. A aplicação exaustiva das regras reescreve gradualmente uma especificação CSP, até um modelo UML-RT completo. A aplicação das regras inclui a construção dos diagramas de classe, estrutura e máquina de estados das cápsulas geradas [13]. Os diagramas de classes são úteis para identificar os atributos e métodos das cápsulas, e os relacionamentos com outros elementos do modelo. As máquinas de estados são construídas para representar a semântica comportamental do processo que lhe deu origem. Os diagramas de estrutura identificam a configuração estrutural de cada cápsula. Embora as provas formais da tradução sejam sugeridas como trabalho futuro, as regras foram idealizadas para preservarem o padrão comportamental das equações CSP a que se aplicam.

As regras de mapeamento de tipos de dados têm o propósito de associá-los a elementos concretos de modelagem, que encerram propriedades específicas ou conjuntos de valores relacionados, como as classes básicas de negócio. As regras para o mapeamento de processos se aplicam a padrões únicos de equações, permitindo que cada aplicação seja única e não-ambígua. Como condição para aplicar as regras de mapeamento de processos, todas as equações devem estar em uma das formas normais descritas na Seção 4.1.

A representação dos canais CSP através das portas das cápsulas levou em consideração o padrão de comunicação síncrono de CSP, que implica no controle da causalidade entre os eventos que ocorrem nas cápsulas. Este controle, que é intrínseco a CSP, não pode ser representado diretamente em modelos UML-RT, por limitações da sua semântica de processamento de eventos, fundamentada na estratégia *run-to-completion*. Por outro lado, entendemos que esta propriedade representa uma regra de comunicação do sistema, e que a modelagem comportamental em UML-RT deve identificar explicitamente este controle. A estratégia usa uma variação do protocolo de requisição em duas fases (*Two-phase Commit Protocol*) [33] para controlar a comunicação entre cápsulas. A adequação das máquinas de estados das cápsulas a este protocolo permite que a causalidade entre eventos seja capturada. Consideramos este trabalho como um precursor para a construção de esteriótipos que representem o controle da causalidade implicitamente, visto que identificamos as características associadas a esta propriedade

em modelos UML-RT.

É importante salientar que em CSP, regras de negócio que governam a comunicação entre processos, para um cenário específico, são usualmente representadas por processos também, que pela estratégia de mapeamento são traduzidos em cápsulas. Os protocolos UML-RT gerados a partir da nossa estratégia de mapeamento não contêm regras de negócio, mas padrões de comunicação ponto a ponto entre portas. Adicionalmente, protocolos que representam as mesmas regras de comunicação podem ser agrupados por similaridade.

As regras de mapeamento também consideram os casos em que um processo assume o comportamento de um outro processo (ou de uma composição de processos), e situações específicas, onde ocorrem recursões mútuas entre processos. A adoção de portas comportamentais nas cápsulas de processos transfere o controle deste comportamento à cápsula principal do modelo, responsável por ler as mensagens comunicadas através destas portas, e substituir a ocorrência de uma cápsula pela de outra cápsula (ou cápsulas). Para o usuário final, o sistema simplesmente assume um novo comportamento.

A sistematização da estratégia e a composicionalidade das regras permitiu a construção da ferramenta *FormalDev*, que automatiza a aplicação das regras e a construção dos modelos UML-RT no *Rational Rose RealTime* [39]. O uso da *FormalDev* previne o usuário de construir os modelos UML-RT manualmente, uma tarefa cansativa e suscetível a erros. O *Rose RealTime* permite gerar código a partir de seus modelos, tornando possível animar e testar os modelos gerados através da estratégia.

No contexto do projeto de pesquisa CIn/Motorola, a ferramenta de mapeamento também gera diagramas de seqüência. As especificações CSP desenvolvidas no projeto de pesquisa representam modelos de uso de celulares. Nestes modelos, cada componente de um celular é representado por um processo CSP. Através da estratégia de mapeamento, cada componente é mapeado em uma cápsula. Os diagramas gerados de seqüência identificam os pontos de interação entre estes componentes em cenários específicos de aplicação. Para tanto, as especificações CSP são anotadas com comentários que facilitam a construção dos diagramas.

Por fim, as regras apresentadas aqui podem ser adaptadas à versão 2 de UML [35], com possíveis melhorias na representação dos diagramas, visto que esta notação incorporou vários conceitos de UML-RT e de outras notações (que a tornariam inclusive mais completa que UML-RT). A utilização de UML 2.0 pode, inclusive, facilitar a criação de esteriótipos que suportem as regras de mapeamento de maneira mais concisa e elegante. Entretanto, os novos elementos e diagramas de UML 2.0, em relação a UML-RT, ainda são ambíguos e pouco divulgados. A vantagem de UML-RT são os esteriótipos para cápsulas e protocolos, e as noções claras e intuitivas de componentes reativos e independentes, pouco claros em UML 2.0. UML-RT também dispõe de ferramentas de apoio sólidas [39], o que ainda não é verdade para UML 2.0.

6.1 Trabalhos Relacionados

Na literatura, poucos trabalhos têm abordado a integração de métodos formais com notações ou linguagens que aproximam a especificação formal de sistemas dos modelos de análise e projeto. Jovanovic *et al* [26] apresenta uma biblioteca gráfica para construção de especificações CSP. A representação textual destes modelos gráficos é passível de análise pelo FDR. Entretanto, estes

modelos são pouco intuitivos, e não se relacionam como os modelos de UML ou UML-RT, tampouco sugerem a evolução para modelos de análise e projeto.

Outros trabalhos abordam a adaptação de UML ou UML-RT para representar padrões arquiteturais complexos entre componentes de sistemas [8, 30]. Alguns destes padrões têm forte relação com as características concorrentes de CSP.

Cheng *et al* [8] aborda o mapeamento de ACME, uma linguagem de descrição arquitetural, para UML-RT. Embora o trabalho não aborde o mapeamento de uma notação formal propriamente dita, identifica algumas limitações de UML-RT e UML para representar conceitos arquiteturais. Por exemplo, os diagramas de classe de UML-RT não identificam conectores como entidades de primeira classe. Conectores são apenas associações entre cápsulas. Isto limita a representação de comportamentos específicos associados à conexão entre componentes dos sistemas.

Lavazza *et al* [30] propõe uma extensão de UML, denotada UML+, para representar as principais características de sistemas concorrentes, reativos e de tempo real. Neste *profile*, por exemplo, transições podem ser definidas através de um conjunto de eventos simultâneos. Isto poderia solucionar o problema da multi-sincronização de uma única cápsula com outras, mas não resolve o problema da causalidade entre eventos de cápsulas distintas.

Abordagens alternativas [16, 1, 2, 17, 22] apresentam o mapeamento de CSP não para uma notação gráfica, mas para bibliotecas Java que implementam alguns operadores de CSP. Entre estas bibliotecas destacam-se CTJ [2], JCSP [1, 22] e Jack [17]. Entretanto, devido o tamanho de sistemas concorrentes reais, estas abordagens em geral dão origem a implementações problemáticas e com padrões de comunicação geralmente complexos. Mais ainda, a visualização da estrutura é geralmente tão ou mais difícil do que a própria especificação CSP. Nossa abordagem tem a vantagem da representação diagramática, somada à geração de código através do *Rose RealTime*, como discutido anteriormente.

Em JCSP, por exemplo, os canais são representados através de instâncias da classe *Channel*. Dois processos comunicantes compartilham uma mesma instância de um canal. Os eventos para troca de mensagens nestes canais são representados através dos métodos *write* e *read*. Entretanto, não há controle da causalidade entre estes eventos. Desta forma, o processo emissor pode enviar um evento (através do método *write*), e continuar seu processamento, mesmo que o processo receptor jamais receba aquele evento (através do método *read*).

Outra grande deficiência de JCSP, reforçada pela ausência de controle da causalidade entre eventos, é a ocorrência de eventos com restrições de valores. O processo receptor pode ignorar entradas de dados cujos valores não são aceitos pela restrição, mas o processo emissor não tem como decidir se houve aceitação por parte do receptor. Chamamos esse comportamento de falso sincronismo.

Por fim, o trabalho apresentado nesta dissertação também poderia se chamar "Uma Biblioteca UML-RT para CSP", em uma analogia com os trabalhos para JCSP e CTJ, sendo que estas consideram apenas um pequeno subconjunto de CSP, enquanto este trabalho considera todos os operadores, e conceitos como multi-sincronização e causalidade.

Outros trabalhos relacionados abordam o mapeamento inverso, de UML-RT para outras notações formais relacionadas a CSP, como CSP-OZ [14] e ***OhCircus*** [37]. Analogamente, cápsulas são mapeadas em processos. Estes trabalhos avaliam a estrutura e o comportamento

das cápsulas separadamente, aproveitando-se do fato de que o comportamento e a estrutura de processos também podem ser vistos isoladamente em CSP-OZ e ***OhCircus***. Entretanto, este trabalhos abordam apenas um subconjunto de UML, especificamente os elementos característicos de UML-RT, e têm um propósito distinto do nosso: formalizar a semântica de um subconjunto de UML-RT.

6.2 Trabalhos Futuros

Apesar de termos proposto um conjunto abrangente de regras de mapeamento, e discutido a estratégia para representar o comportamento das especificações CSP, ainda são necessárias provas formais que validem a consistência das regras. Para a construção destas provas é necessária a formalização dos elementos de UML-RT utilizados nesta dissertação. Trabalhos como o de Fisher [15] e Ramos [36, 37] abordam esta formalização, e fornecem o embasamento inicial para as provas mencionadas.

A estratégia para o mapeamento de processos usa regras de normalização de equações CSP, seguidas de regras para a construção de cápsulas UML-RT. Entretanto, a normalização das equações de processos gera processos adicionais que, quando mapeados em cápsulas, tornam o modelo UML-RT extenso e complexo, prejudicando sua legibilidade e a associação entre as cápsulas e a especificação original. Propomos a utilização de regras de refatoramento de cápsulas [36] para minimizar estes problemas, considerando conceitos de modelagem em UML-RT e preservando as propriedades da especificação original. Estes refatoramentos tornariam o modelo incrementalmente mais concreto, com a vantagem de ter uma base formal em sua origem, e reforçariam as semelhanças entre as cápsulas do modelo e os processos da especificação. Um exemplo de refatoramento é a transformação de cápsulas que representam eventos consecutivos em uma única cápsula. A aplicação destas regras seria feita automaticamente pela ferramenta *FormalDev*, que identifica previamente os pontos de aplicação através das equações originais dos processos.

Além disto, as regras de mapeamento consideram uma única cápsula controladora, denotada *SystemController*, que centraliza a equação expandida do sistema. Embora o comportamento dos operadores de CSP esteja distribuído aos elementos da equação expandida, a centralização deste controle também dificulta a interpretação do modelo. Como melhoria, propomos mais regras de refatoramento, que identifiquem cápsulas relacionadas e transfiram o seu controle a uma cápsula controladora específica. Cada grupo de cápsulas, controlado por sua própria cápsula controladora, seria uma estrutura auto-contida, que não necessita ser substituída por outras cápsulas no ambiente externo. Esta abordagem favorece o uso de cápsulas específicas para representar operadores CSP.

Para facilitar a execução dos modelos gerados no *Rose RealTime*, as ações das máquinas de estado das cápsulas, descritas em pseudo-código nas regras de mapeamento, são traduzidas em código Java. Entretanto, os modelos gerados são semi-executáveis, visto que as regras de mapeamento traduzem funções e expressões lógicas descritas em CSP em métodos de cápsulas, mas o corpo destes métodos é apenas comentado com a equação algébrica que o originou. A transformação destas equações algébricas em código executável é apontada como melhoria futura da ferramenta *FormalDev*, considerando alguns trabalhos relacionados, que já abordam

a transformação de equações algébricas em CSP para Java [17, 16].

Finalmente, as regras foram simplificadas para que os canais de comunicação sejam todos mapeados em portas *conjugadas*. Esta decisão foi tomada para simplificar o modelo UML-RT gerado, e não muda a semântica da especificação CSP original, desde que é possível representar eventos CSP sem orientação ou replicar os sinais de entrada e saída nos papéis de protocolos. As regras podem ser adaptadas para originarem portas *base* e *conjugadas*, sem prejuízo à construção dos modelos ou à definição das próprias regras.

APÊNDICE A

Pseudo-código de métodos usados por *SystemController*

O método *isRequested* verifica se pelo menos uma sub-cápsula ativa de *SystemController* respondeu ao evento *a.request*?*m*.

```
boolean isRequested(Port port, Datatype x) {
  boolean r = False;

  List capsule_refs = getCapsuleReferencesToRequest(port);
  foreach (reference in capsule_refs) {
     Port mirror = getConnectedPort(reference, port);
     if (mirror.request.invoke(x) != null) {
       reference.status = lock;
     }
  }

  boolean verify = verifySemanticsOfParallelism(capsule_refs);
  if (verify == True) {
   r = True;
  } else {
    foreach (reference in capsule_refs) {
     Port mirror = getConnectedPort(reference, port);
     mirror.rollback.send();
     reference.status = free;
    }
    r = thereIsAtLeastOneLockedCapsule();
  }
  return r;
}
```

O método *forwardRollback* envia o evento *a.rollback* as todas sub-cápsulas ativas que estejam bloqueadas.

```
void forwardRollback(Port port) {
  List capsule_refs = getCapsuleReferencesToRollback(port);
  foreach (reference in capsule_refs) {
     Port mirror = getConnectedPort(capsule_ref, port);
     mirror.rollback.send();
     reference.status = free;
```

```
  }
}
```

O método *forwardCommit* envia o evento *a.commit* as todas sub-cápsulas ativas que estejam bloqueadas.

```
void forwardCommit(Port port) {
  List capsule_refs = getCapsuleReferencesToCommit(port);
  foreach (reference in capsule_refs) {
     Port mirror = getConnectedPort(reference, port);
     if (mirror.commit.invoke() != null) {
       reference.status = free;
     }
  }

  List capsule_refs = getCapsuleReferencesToRemove();
  foreach (reference in capsule_refs) {
     removeCapsule(reference);
  }
  refreshEquation();
}
```

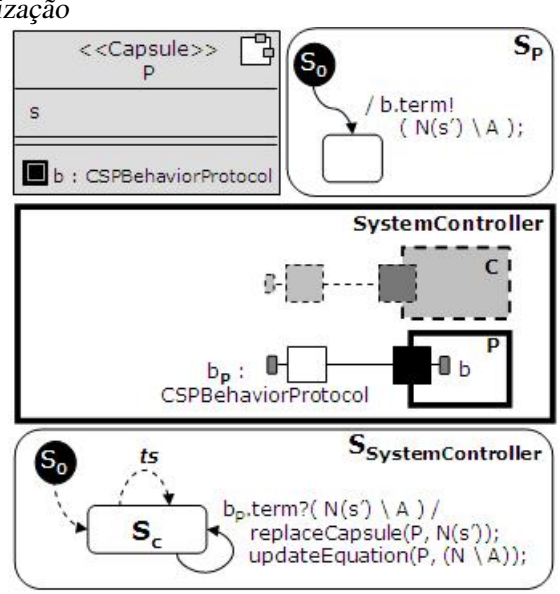
APÊNDICE B

Regras de Mapeamento Adicionais

A regra a seguir mapeia processos que usam o operador de internalização.

Regra 15. *Mapeamento de Internalização*

$P(s) = N(s') \setminus A \quad \Longrightarrow$



Nesta regra, a cápsula P gerada informa para o ambiente externo o nome e os parâmetros de um processo (representado por $N(s')$) e um conjunto de nomes de portas (representado pelo conjunto A), combinados pelo operador de internalização.

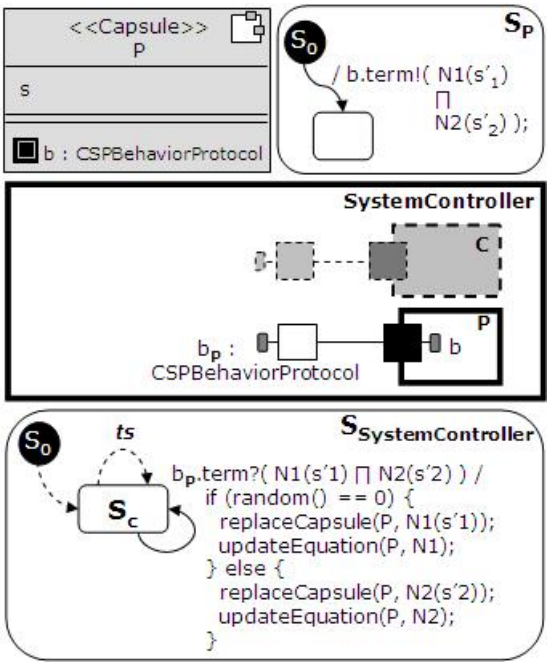
A estrutura de *SystemController* é atualizada para que contenha uma sub-cápsula opcional de P. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada em *SystemController* pelo evento $b_P.term?(N(s'))$, remove a instância de P e cria uma instância de N, usando o parâmetro s'. A equação expandida é atualizada substituindo a referência de P pela expressão $(N \setminus A)$, contendo a referência de N e o conjunto de nomes de portas A. □

Futuramente, mensagens enviadas pelo ambiente externo através de portas cujos nomes estejam em A, não serão repassadas para esta instância de N. Portanto, a estratégia para modelar internalização é reduzida ao mapeamento entre as portas de *SystemController* e as portas das instâncias influenciadas pela relação de internalização, exceto para as portas cujos nomes estejam no conjunto A.

A próxima regra avalia os processos que se comportam como uma escolha interna.

Regra 16. *Mapeamento de Escolha Interna*

$$P(s) = \begin{array}{l} N1(s'_1) \\ \sqcap \\ N2(s'_2) \end{array} \quad \Longrightarrow$$

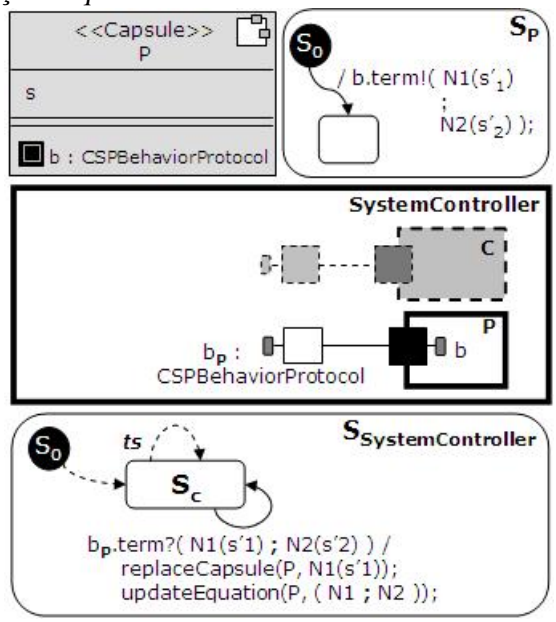


A cápsula P gerada informa o nome e os parâmetros das cápsulas combinados pelo operador de escolha interna. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada em *SystemController* pelo evento $b_P.term?(N1(s_1) \sqcap N2(s_2))$ remove a instância de P. Entretanto, neste caso, *SystemController* cria uma instância para apenas um dos parâmetros da equação. Uma escolha aleatória é feita entre $N1$ e $N2$. Se $N1$ é escolhido, *SystemController* cria uma instância de $N1$, e substitui a referência de P pela referência de $N1$. O mesmo processo é feito para $N2$, se este é escolhido. □

A próxima regra avalia os processos que se comportam como uma composição sequencial.

Regra 17. *Mapeamento de Composição Sequencial*

$$P(s) = \begin{array}{l} N1(s'_1) \\ ; \\ N2(s'_2) \end{array} \quad \Longrightarrow$$



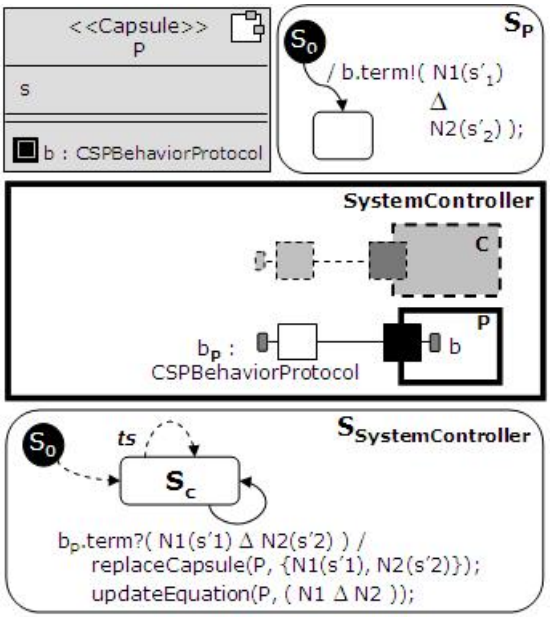
A cápsula P informa o nome e os parâmetros das cápsulas $N1$ e $N2$, combinados pelo operador de composição sequencial. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada pelo evento $b_P.term?(N1(s'_1)\ ;\ N2(s'_2))$ processa a remoção de P, e cria uma instância do primeiro parâmetro da equação, $N1$. A referência de P na equação expandida é substituída pela expressão $(N1\ ;\ N2)$. □

Futuramente, se a cápsula $N1$ assumir o comportamento de ***SKIP***, sua instância é removida e uma instância de $N2$ é criada. Também a expressão $(N1\ ;\ N2)$ será substituída por $N2$ na equação expandida. A criação da instância de $N2$, após a terminação com sucesso da cápsula de $N1$, é necessária para evitar uma recursão infinita não prevista na especificação. Por exemplo, quando a equação de P é algo como $P = Q; P$.

A próxima regra avalia os processos que se comportam como uma interrupção.

Regra 18. *Mapeamento de Interrupção*

$$P(s) = \begin{array}{l} N1(s'_1) \\ \triangle \\ N2(s'_2) \end{array} \quad \Longrightarrow$$



A cápsula P obtida através desta regra informa o nome e os parâmetros das cápsulas $N1$ e $N2$, combinados pelo operador de interrupção. A máquina de estados de *SystemController* é atualizada para interpretar os eventos da porta b_P. A transição disparada pelo evento $b_P.term?(N1(s_1) \triangle N2(s_2))$ processa a remoção de P e a criação de $N1$ e $N2$, usando suas listas de parâmetros. A referência de P na equação expandida é substituída pela expressão $(N1 \triangle N2)$, contendo as referências de $N1$ e $N2$. □

Se alguma mensagem for repassada para a instância de $N2$, então a instância de $N1$ é removida e a expressão $N1 \triangle N2$ será substituída por $N2$ na equação expandida. Isto encapsula o comportamento da interrupção da primeira cápsula pela segunda.

Referências Bibliográficas

[1] Communicating Sequential Processes for Java (JCSP). `http://www.cs.kent.ac.uk/jcsp`.

[2] Communicating Threads in Java (CTJ). `http://www.ce.utwente.nl/`.

[3] PROBE Users Manual. `http://www.fsel.com/software.html`.

[4] M. Antonsson and P. Hansson. Modeling of Real-Time Systems in UML with Rational Rose and Rose Real-Time based on RUP. Master's thesis, Ericsson, 2001.

[5] D. Avison and A. Wood-Harper. Information systems development research: An exploration of ideas in practice.

[6] L. Baresi and M. Pezzè. Formal interpreters for diagram notations. *ACM Trans. Softw. Eng. Methodol.*, 14(1):42–84, 2005.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[8] S. Cheng and D. Garlan. Mapping Architectural Concepts to UML-RT. In *PDPTA '2001: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, EUA, 2001. Springer-Verlag.

[9] R. Clark and A. Moreira. Sdl in rigorus object-oriented analysis (short paper). In *FMOODS*, 1999.

[10] Rational Software Corporation. Extensibility Interface Reference, Rational Rose RealTime, VERSION: 2002.05.20.

[11] S. Donatelli and P. S. Thiagarajan, editors. *Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings*, volume 4024 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] H. Eriksson, M. Penker, and D. Fado. *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[13] P. Ferreira, A. Sampaio, and A. Mota. Viewing CSP specifications with UML-RT diagrams (to appear). In *9th Brazilian Symposium on Formal Methods (SBMF)*, Natal, Brazil, september 2006.

[14] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[15] C. Fischer, E. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029, pages 91–108. Springer, 2001.

[16] A. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In *Formal Methods (FM 2006)*, 2006. To appear.

[17] L. Freitas. JACK: A process algebra implementation in Java. Master's thesis, Centro de Informatica, Universidade Federal de Pernambuco, April 2002. http://www.cin.ufpe.br/ lf25.

[18] M. Gogolla and M. Richters. On combining semi-formal and formal object specification techniques. In Francesco Parisi-Presice, editor, *Recent trends in algebraic development techniques: 12th international workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997: selected papers*, volume 1376. Springer, 1998.

[19] M. Goldsmith. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, August 2001.

[20] Z. Gu and K. G. Shin. Synthesis of Real-Time Implementation from UML-RT Models. Technical report, Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, 2003.

[21] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[22] G. Hilderink, A. Bakkers, and J. Broenink. A Distributed Real-Time Java System Based on CSP, 2000.

[23] M. Hinchey and J. Bowen. Seven more myths of formal methods: Dispelling industrial prejudices. In M. Bertran, M. Naftalin, and T. Denvir, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 105–117. Springer-Verlag, 1994.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[25] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng. Ptolemy ii heterogeneous concurrent modeling and design in java, volume 3: Ptolemy ii domains, chapter 6, july 2003. Technical Report UCB/ERL M03/29, EECS Department, University of California, Berkeley, 2003.

[26] D. Jovanovic, G.K. Liet, and J.F. Broenink. gCSP: a graphical tool for designing CSP systems. In *Communicating Process Architectures 2004*, pages 233 – 251, Oxford, UK, 2004. ACM Press.

[27] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[28] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[29] L.M. Lai and P. Watson. A case study in Timed CSP: the railroad crossing problem. In O. Maler, editor, *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART 97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 69–74. Springer Verlag, January 1997.

[30] L. Lavazza, G. Quaroni, and M. Venturelli. Combining UML and Formal Notations for Modeling Real-Time Systems. In *Proc Joint 8th ESEC and 9th ACM SIGSOFT FSE[C]*, pages 196–206. Springer-Verlag, 2001.

[31] A. Lyons. UML for Real-Time Overview. 1998.

[32] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[33] E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, Cambridge, MA, USA, 1981.

[34] OMG. *OMG Unified Modeling Language Specification, version 1.5, OMG document formal/03-03-01*. Object Management Group, 2003.

[35] OMG. *UML 2.0 superstructure specification, version 2.0, documents ptc/03-08-02 and ptc/04-10-02*. Object Management Group, 2004.

[36] R. Ramos, A. Sampaio, and A. Mota. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *FMOODS*, pages 99–114, 2005.

[37] Rodrigo Teixeira Ramos. Desenvolvimento Rigoroso com UML-RT. Master's thesis, Centro de Informatica, Universidade Federal de Pernambuco, April 2004.

[38] Rational/IBM. IBM Rational Software Modeler.

[39] Rational/IBM. Rose Real-time Development Environment.

[40] G. Reggio and M. Larosa. A graphic notation for formal specifications of dynamic systems. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, pages 40–61, London, UK, 1997. Springer-Verlag.

[41] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[42] A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in *Circus*. In L Eriksson and PA Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, unknown 2002.

[43] B. Selic. An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 321–330, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[44] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[45] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[46] M. Spivey. *The Z Notation: A Reference Manual. second edition.*, 1992.