Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

DESENVOLVIMENTO RIGOROSO COM UML-RT

Rodrigo Teixeira Ramos

DISSERTAÇÃO DE MESTRADO

Recife
Abril/2005

Universidade Federal de Pernambuco
Centro de Informática

Rodrigo Teixeira Ramos

DESENVOLVIMENTO RIGOROSO COM UML-RT

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 Cesar Alves Sampaio*
Co-orientador: *Alexandre Cabral Mota*

Recife
Abril/2005

Aos meus pais pelo exemplo de caráter, fortaleza e bondade.

AGRADECIMENTOS

... eu poderia suportar, embora não sem dor, que tivessem morrido todos os meus amores, mas enlouqueceria se morressem todos os meus amigos! Até mesmo aqueles que não percebem o quanto são meus amigos e o quanto minha vida depende de suas existências ... A alguns deles não procuro, basta-me saber que eles existem. Esta mera condição me encoraja a seguir em frente pela vida ... mas é delicioso que eu saiba e sinta que os adoro, embora não declare e não os procure ...

—VINÍCIUS DE MORAES (Para você, com carinho)

Terminada mais esta etapa da grande jornada da vida, sinto-me grato à diversas pessoas que contribuíram direta ou indiretamente para o meu aprendizado e fortalecimento durante estes dois anos de mestrado. A estas, manifesto, agora, meus profundos e sinceros agradecimentos.

Sou grato a Deus por estar sempre presente em minha vida e por ter me honrado com o afeto e o companheirismo de tantos amigos. Dentre eles, ressalto: os meus grandes amigos Leonardo Cole e Danielly Cruz, que tanto me ajudaram em todas as horas com sua amizade fraterna e sincera; Márcio Dahia, que com seu bom grande humor me ajudou a enfrentar alguns tediosos dias no Cin; e Fernando Trinta e Thiago Santos, que sempre tomei como exemplos de disciplina e obstinação na busca de meus ideais. Agradeço também aos amigos Márcio Cornélio, Rafael Borges, Rohit Gheyi, Thiago Massoni pelas enriquecedoras discussões.

Expresso meus agradecimentos aos professores Augusto Sampaio e Alexandre Mota pela inúmeras ajudas e sugestões durante o meu mestrado. Mais do que orientadores, eles foram bons e préstimos amigos durante todo este período. Principalmente, sou grato à Augusto pela preocupação com a minha formação e pelos duradouros ensinamentos, que levarei pelo restante da minha vida.

Por fim, e sobretudo, gostaria de manifestar minha especial gratidão aos meus pais, a quem devo não apenas esta conquista mas tudo o que sou, e meus irmãos pelo amor e pela confiança inabaláveis.

RESUMO

Como outros métodos visuais orientados a objetos, UML tem influenciado tremendamente a prática de modelagem na engenharia de software com ricos mecanismos de estruturação. Porém, apesar de suas vantagens e adoção em larga escala, na prática, a falta de uma semântica formal tem dificultado o desenvolvimento rigoroso baseado em modelos de aplicações não triviais (aplicações que por sua natureza necessitam de ênfase na especificação e na verificação de seus componentes). A razão para isto é que transformações de modelos podem não preservar a semântica e, como conseqüência, o comportamento do modelo. Este problema é ainda mais sério em transformações que envolvem diferentes visões do modelo.

Limitações similares podem ser encontradas durante o desenvolvimento com UML-RT. Esta linguagem é uma extensão conservativa de UML que provê a noção de objetos ativos (objetos com um comportamento próprio, independente do fluxo de execução do restante do sistema) para descrever aplicações concorrentes e distribuídas. Neste tipo de desenvolvimento, transformações devem lidar simultaneamente com as diferentes visões estáticas e dinâmicas do modelo, representadas por seus diagramas e propriedades.

Por estes motivos, este trabalho propõe uma semântica para UML-RT, mapeando suas construções em *OhCircus*, uma linguagem formal, orientada a objetos, que combina CSP e Z, e que suporta o cálculo de refinamentos de Morgan. A partir desta semântica, bem como das noções e leis de refinamentos de *OhCircus*, é possível propor leis de transformação de modelos passíveis de demonstração e que preservam o comportamento do sistema.

Estas leis de transformação são propostas em duas categorias: a primeira delas é um conjunto abrangente de leis básicas que expressam pequenas mudanças nas principais visões do modelo, como a declaração ou remoção de elementos do modelo; já a segunda representa leis de transformação de maior granularidade, derivadas a partir da composição de leis básicas, como a decomposição de uma cápsulas em cápsulas operando em paralelo. Tais transformações derivadas podem ser vistas como refatoramentos (*refactorings*) corretos sobre o modelo, facilmente aplicáveis durante um processo de desenvolvimento rigoroso, sem que o desenvolvedor tenha conhecimento do formalismo que o suporta.

Finalmente, a abrangência deste conjunto de leis é discutida particularmente através dos principais passos de uma estratégia de redução de modelos UML-RT a um modelo UML estendido com um único objeto ativo, responsável por todas as interações com o ambiente e por conservar o comportamento dinâmico do sistema modelado. Este modelo UML estendido pode ser visto como uma *forma normal*, e, portanto, nossa estratégia pode ser vista como uma contribuição para uma estratégia mais global de completude capturada por redução a esta *forma normal*.

Palavras-chave: UML-RT, transformação de modelos, *OhCircus*, integração de métodos formais

ABSTRACT

As other object-oriented visual methods, UML has tremendously influenced the software engineering modeling practice with rich structuring mechanisms. Despite its strengths, the rigorous development of non-trivial applications (those applications that, due their complexity, need to emphasise the specification and verification of their components) do not seem feasible without a formal semantics. The reason is that well-known model transformations might not preserve behaviour. This problem is even more serious in a model driven development, where transformations are as important as models, and involve different model views.

Similar limitations can be found during the development with UML-RT. This language is a conservative UML profile that includes active objects (objects with an execution flow independent of the rest of the system) to describe concurrent and distributed applications. In this kind of development, transformations have to simultaneously handle both static and dynamic model views, represented by the diagrams and properties of the model.

For these reasons, this work proposes a semantics for UML-RT via mapping into *OhCircus*, a formal object oriented language that combines CSP, Z and specification statements, and also support Morgan's refinement calculus. From this semantics and the laws of *OhCircus*, we are able to propose and prove model transformation laws that preserve the system behaviour.

Two groups of laws are proposed: the first one embodies a comprehensive set of laws that govern small changes in the main model views, like introducing or removing a model element; the second group presents more elaborated laws derived from the composition of these basic laws, like decomposing a capsule into parallel component capsules. The derived laws can be taken as precise model refactorings that are easily applied in a rigourous development, without the developer directly dealing with the formalism that supports them.

Finally, the comprehensiveness of the set of laws is particularly discussed through the main steps of a reduction strategy of UML-RT models into a UML model extended with a single capsule responsible for all the interactions with the environment; This capsule is also responsible for maintaining the active behaviour of the modeled system. This extended UML model can be regarded as a *normal form*, and, therefore, our strategy can be regarded as a contribution to a completeness strategy captured by *normal form* reduction.

Keywords: UML-RT, model transformations, *OhCircus*, formal method integration

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

A noção de Engenharia de Software como a conhecemos, originou-se durante a década de 60, quando a comunidade da ciência da computação começou a despertar interesse [13] na necessidade pela produção de software baseado em fundamentos teóricos e disciplinas práticas, como as encontradas em ramos tradicionais e estabilizados da engenharia [62]. Preocupados com a recém instalada *crise do software* [11, 39], desejava-se um aumento da qualidade de software, através de processos mais produtivos durante o seu desenvolvimento.

Hoje, esta produção de software tem caminhado para uma abordagem de desenvolvimento dirigido a modelos (MDE - Model Driven Engineering) [87, 51], onde os artefatos principais durante o desenvolvimento são modelos, ao invés do código em uma linguagem de programação. Diferentemente da abordagem tradicional de que modelos são utilizados somente para fins de documentação de aspectos interessantes durante a construção de programas, implantadas indiretamente por diversos processos da engenharia de software, o objetivo principal no desenvolvimento dirigido a modelos é que o processo de desenvolvimento é dirigido pelas atividades de modelagem de software.

Durante o ciclo de vida deste processo, é possível aplicar diversos tipos de transformação para se especializar ou reestruturar o modelo. Neste contexto, transformações são artefatos tão importantes quanto os próprios modelos, e têm finalidades semelhantes a transformações tradicionais de código, como *refactorings* [33, 77] e refinamentos [67]. De fato, estes tipos de transformação também são aplicados para resolver problemas relacionados à evolução e manutenção [57] existentes em sistemas baseados em modelos [51]. Enquanto *refactorings* reestruturam o sistema para adicionar aspectos de qualidade (por exemplo, através da aplicação de padrões de projeto), preservando o seu comportamento, refinamentos são utilizados para concretizar o modelo (por exemplo, especializando-o para uma plataforma específica).

Em esforços recentes, a OMG (*Object Management Group*) [71] tem atestado o valor de abordagens baseadas em modelos para o desenvolvimento de software, através da padronização de notações e interoperabilidade de ferramentas. Uma evidência é a especificação de MDA (*Model-Driven Architecture*) [65] e UML [81]. Enquanto UML tem se tornado uma notação padrão para a modelagem de sistemas, MDA têm contribuído, entre outros benefícios, com padrões de interoperabilidade e portabilidade ao suporte metodológico do desenvolvimento dirigido a modelos, através de uma arquitetura de modelos. MDA define uma arquitetura que separa a especificação das funcionalidades do sistema da especificação de como elas são implementadas. A especificação inicial é estruturada como um modelo de plataforma independente (PIM), que pode ser refinado em um ou mais modelos específicos de plataforma (PSM).

Apesar de não possuir uma definição precisa, a noção geral de arquitetura na en-

genharia de software é um pouco diferente da encontrada em MDA. Um consenso em engenharia de software parece ser que uma arquitetura descreve um sistema em partes com um alto nível de abstração e através do relacionamentos entre estas partes. Em MDA, a arquitetura do sistema é vista como um conjunto de modelos durante o desenvolvimento do sistema, que descrevem níveis de abstração distintos da modelagem, e o relacionamento destes modelos é realizado através de transformações. Atribuiremos neste trabalho o conceito de arquitetura a um único nível de abstração do software, que pode ser vista em termos da decomposição do sistema em componentes, suas propriedades, e suas relações [6, 20], enquanto transformações são utilizadas para relacionar o modelo dos diferentes níveis abstrações desta arquitetura; desta maneira, modelos e transformações podem ainda assim representar os papeis sugeridos em MDA.

Neste sentido, arquiteturas podem ser representadas por modelos, e evoluir durante um processo de desenvolvimento baseado em modelos, através da transformações. Porém, a criação e utilização de artefatos de software (modelos e transformações) não devem ser baseadas no empirismo [45] para a produção de software de qualidade. Modelos sem uma semântica formal podem ser ambíguos e, conseqüentemente, não expressarem corretamente suas propriedades. Por outro lado, transformações sem bases sólidas podem não garantir a preservação destas propriedades [26]. Isto se torna mais sério no desenvolvimento de aplicações não triviais (por exemplo, concorrentes ou distribuídas), onde a verificação e especificação destas propriedades se torna ainda mais difícil.

Normalmente, modelos são expressos através de notações semi-formais como UML e suas extensões, úteis para especificar aspectos de um domínio especifico, ainda que independentes de plataforma. Em particular, enfatizamos o uso de UML-RT [90] (UML para tempo real), uma extensão conservativa de UML que provê a noção de objetos ativos [56] (objetos com um comportamento próprio, independente do fluxo de execução do restante do sistema) para descrever aplicações concorrentes e distribuídas.

Na literatura, diversos esforços [30, 12, 69] atacam este problema através da integração de notações informais (ou semi-formais), utilizadas para descrever estes modelos, com linguagens formais, esperando que conceitos tradicionais da teoria da programação conduzam a um desenvolvimento de sistemas consistente, sem que, durante a sua evolução, propriedades, anteriormente definidas e verificadas, não sejam preservadas.

Estes trabalhos atribuem uma semântica, definida pela linguagem formal, à notação informal. Assim, modelos podem ser mapeados para a linguagem formal sempre que se deseje verificar sua consistência. Um incômodo, porém, é que, a cada nova transformação do modelo, este mapeamento deve ser realizado e propriedades devem ser provadas para se assegurar a consistência do modelo.

Uma prática que pode assistir na solução deste problema é a criação de leis de transformação de modelos, seguindo as mesmas diretrizes das leis da programação [46]. As leis da programação são leis que governam as propriedades básicas da programação, e que por este motivo sua aplicação pode ser utilizada para garantir a preservação de transformações de programa; como as leis que governam a aritmética descrevem as operações sobre números, leis de programação descrevem as propriedades dos construtores de programas. Uma lei da aritmética, por exemplo, é: $0 + x = x$, que denota que o número 0 é o elemento neutro (ou a identidade) para a operação de soma. Através da composição

de leis básicas, é possível demonstrar propriedades mais complexas. Por exemplo, na aritmética, a lei $2a - a = a$, pode ser reescrita, passo-a-passo, através de leis básicas para se provar sua corretude; em particular, ela pode ser derivada das básicas leis da álgebra através dos seguintes passos: $(a + a) - a = a$; $a + (a - a) = a$; $a + 0 = a$; e $a = a$. Na realidade, leis da programação podem também definir uma semântica axiomática [98], definindo propriedades sobre os construtores da linguagem através de axiomas e regras de inferência, da lógica simbólica, sendo também úteis para provar propriedades de programas [67].

Diversos paradigmas podem ser beneficiados através das leis de programação. Por exemplo leis para programas que utilizam linguagens orientadas a objeto [9], imperativas [46] e concorrente [80] podem ser úteis para estabelecer uma base formal para práticas informais de desenvolvimento de software já existentes, como *refactorings*. Desta maneira, desenvolvedores podem realizar suas tarefas sem explicitamente lidarem com formalismos, mas baseados neles. Transformações complexas de código podem ser explicadas através da composição de leis básicas de programação, utilizando uma interface já conhecida pelo programador e livre de formalismos.

Apesar desta ser uma área ainda incipiente, leis de modelagem podem trazer benefícios similares aos das leis da programação como, por exemplo, derivando-se *refactorings* para modelos a partir de leis mais básicas [37]. Este é um dos objetivos fundamentais deste trabalho: propor um desenvolvimento rigoroso baseado em modelos UML-RT. Este desenvolvimento é rigoroso por garantir a consistência e preservação de propriedades do sistema durante a fase de modelagem. Para garantir esta preservação de propriedades, contribuímos neste trabalho para o desenvolvimento rigoroso em UML da seguinte maneira:

- Atribuição de uma semântica formal a UML-RT, através do seu mapeamento sintático para a linguagem formal *OhCircus* [16];
- Proposição de um conjunto abrangente de leis básicas de transformação para UML-RT para auxiliar a criação de tarefas de modelagem consistentes, que preservam o comportamento do modelo;
- Derivação de leis de projeto para UML-RT a partir das leis básicas, com a finalidade de aumentar sua aplicabilidade;
- Demonstração da abrangência das leis propostas através dos principais passos de uma estratégia de normalização;
- Aplicação destas leis durante o projeto de um estudo de caso para demonstrar sua integração com um processo de desenvolvimento prático;

Propomos uma semântica para os elementos de UML-RT através de seu mapeamento para a linguagem formal *OhCircus* [16], uma linguagem orientada a objetos que combina CSP [79], Z [95] e o cálculo de refinamentos de Morgan [67]; este mapeamento é baseado em resultados previamente obtidos [74]. Tanto em [74] como neste trabalho, focamos no mapeamento dos novos elementos (classes ativas e outros construtores relacionados)

que UML-RT adiciona a UML. Consideramos conjuntamente as visões comportamentais e estruturais de um modelo UML-RT.

Uma razão para a escolha de *OhCircus* é que sua semântica é definida com base na *Unified Theory of Programming* (UTP) [47], cujo modelo relacional tem se mostrado conveniente para formalizar diversos paradigmas da programação. Outra vantagem é que *OhCircus* inclui os principais conceitos encontrados em UML-RT. Ao invés de, por exemplo, CSP-OZ [31], *OhCircus* desacopla a ocorrência de eventos de operações sobre o estado, e é projetada para suportar um cálculo de refinamento [67]. Leis de *OhCircus* têm inspirado a aplicação de diversas leis para UML-RT [83], e a prova de leis de transformação para UML-RT através de nosso mapeamento semântico [74]. Diferentemente de outros trabalhos relacionados à formalização de UML [12, 69] e UML-RT [32, 25], este trabalho considera conjuntamente as visões estrutural e comportamental do modelo.

Utilizando como base a semântica atribuída aos elementos de UML-RT, através de seu mapeamento para *OhCircus*, propomos leis de transformação que preservam o comportamento do sistema. Focamos na proposição de leis básicas, porque estas são mais facilmente provadas e permitem um desenvolvimento sistemático, onde leis básicas podem ser utilizadas para justificar leis mais complexas.

As leis propostas capturam pequenos passos consistentes que preservam tanto aspectos estáticos quanto dinâmicos do modelo (levando em conta conjuntamente propriedades estruturais e comportamentais). Através da combinação de leis básicas introduzimos novas leis mais elaboradas. Tais leis são capazes de capturar e formalizar algumas das práticas utilizadas em atividades de projeto, como no Processo Unificado da Rational (RUP) [52], sendo esta uma outra contribuição importante deste trabalho. Através de um estudo de caso, mostramos como um modelo concreto de projeto pode ser gerado a partir de um modelo inicial e abstrato de análise.

Como exemplo de lei básica de transformação podemos citar a inserção, remoção ou alteração de um elemento do modelo. Usualmente, estas leis alteram apenas uma das visões do modelo, porém impondo condições para a sua aplicação com base com seu relacionamento com as visões. Por lidarmos com leis que preservam o comportamento do modelo, criamos leis de equivalência entre modelos, que podem ser aplicadas em qualquer direção (da esquerda para a direita, ou da direita para a esquerda); na realidade cada lei pode ser vista como a definição conjunta de duas regras de transformação de modelo.

As leis propostas aqui estão baseadas nas idéias inicialmente reportadas em [83], que apresenta algumas poucas leis de transformação de grande granularidade para UML-RT, refletindo a importância de transformações na prática de desenvolvimento.

Como não definimos leis para os elementos de UML, nosso trabalho pode ser considerado suplementar a outros trabalhos que focam especificamente nestes elementos [41, 30, 38], ou a trabalhos que descrevem sucintamente leis para UML-RT [26, 85], sem expressar os padrões e as condições necessários para sua aplicação. Transformações consistentes entre classes no diagrama são livremente permitidas, desde que estas não interfiram com a interface esperada por cápsulas que invoquem métodos destas classes. Isto acontece porque a comunicação entre objetos ativos em UML-RT e classes é através da chamada de métodos e porque cápsulas não compartilham classes entre si. Assim do ponto de vista destas classes, cápsulas são simplesmente elementos externos que invocam seus serviços.

Além da avaliação das leis sob o ponto de vista de sua corretude e aplicabilidade, a abrangência do conjunto de leis proposto é discutida particularmente através dos principais passos de uma estratégia de redução de modelos em UML-RT a um modelo UML estendido com um único objeto ativo (Capítulo 5), responsável por todas as interações com o ambiente e por conservar o comportamento dinâmico do sistema modelado. Este modelo UML estendido pode ser visto como uma *forma normal*, e, portanto, nossa estratégia pode ser vista como uma contribuição para uma estratégia de completude capturada por uma redução a esta *forma normal*, semelhante à estratégia apresentada em [9] para uma linguagem de programação orientada a objetos. Nosso foco é em um conjunto de leis para diagramas de classe e estrutura de modelos; uma estratégia completa de normalização necessita de um grande número de leis para capturar transformações nos diagramas de estados, que está fora do escopo deste trabalho.

Apesar de lidarmos, neste trabalho, com um subconjunto da linguagem UML-RT e desta linguagem não ter sido criada por organizações de padronização (como a OMG), sendo mantida pelas empresas que suportam suas ferramentas [19], vários conceitos desta linguagem estão diretamente presentes em outras linguagens para a descrição de arquiteturas ou de componentes. Assim, todas as contribuições que possam ser incorporadas ao desenvolvimento usando UML-RT podem, a principio, ser adaptadas para o desenvolvimento nessas linguagens.

Esta dissertação esta dividida em cinco capítulos, além deste, e três apêndices. O conteúdo de cada um é descrito a seguir:

- **Capítulo 2:** Neste capítulo, introduzimos os conceitos presentes na linguagem UML-RT; além disto contextualizamos esta linguagem em relação a algumas outras linguagens de modelagem, indicando suas vantagens e desvantagens. Um exemplo sobre como esta linguagem será utilizada na modelagem de um sistema rigoroso também será apresentadao neste capítulo.

- **Capítulo 3:** Neste capítulo, introduzimos a linguagem formal *OhCircus* e apresentamos os mapeamentos semânticos de um subconjunto expressivo das visões estruturais e comportamentais de UML para esta linguagem formal.

- **Capítulo 4:** Neste capítulo, apresentamos, de uma maneira incremental, um conjunto abrangente de leis básicas de transformação para UML-RT e algumas leis derivadas a partir deste conjunto. Além disto, provamos algumas destas leis utilizando a semântica proposta no Capítulo 3.

- **Capítulo 5:** Neste capítulo, aplicamos leis de transformação de modelos em UML-RT a um estudo de caso: um sistema de automação industrial. As aplicações das leis são utilizadas em uma estratégia de normalização de um ponto intermediário do desenvolvimento deste estudo de caso, e no desenvolvimento deste ponto intermediário até um modelo de concreto, próximo à implementação. Durante este desenvolvimento, as leis são contextualizadas segundo as atividades de projeto, utilizando no RUP.

- **Capítulo 6:** Neste capítulo, discutimos alguns trabalhos relacionados e futuros, e apresentamos nossas considerações finais
- **Apêndice A:** Neste apêndice, mostramos a gramática completa da linguagem *Oh-Circus*.
- **Apêndice B:** Neste apêndice, mostramos algumas leis de CSP utilizadas para a prova das leis de UML-RT.
- **Apêndice C:** Neste apêndice, mostramos algumas leis restantes de transformação para UML-RT não apresentadas no Capítulo 4.

CAPÍTULO 2

LINGUAGENS DE MODELAGEM

Neste capítulo, introduziremos alguns conceitos gerais de UML [81], e indicaremos os motivos pelos quais ela é inapropriada para o desenvolvimentos de componentes reativos para sistemas software, cuja arquiteturas incluem aspectos concorrentes e distribuídos. Apesar da proposta para uma versão mais nova de UML (UML 2.0) possuir várias das características desejáveis para a modelagem destas arquiteturas, escolhemos a linguagem UML-RT neste trabalho, por a considerarmos uma linguagem mais consolidada e que conta (no momento) com um suporte mais amplo de ferramentas comerciais.

O projeto de arquitetura de tais sistemas distribuídos e reativos tem como elemento central a interação de seus componentes durante a execução de algumas tarefas [54]. Como veremos a seguir, a versão atual da UML não possui várias características desejáveis para a especificação de componente de *software* [48], tampouco define explicitamente a interação entre as interfaces destes componente.

Para se obter um contexto amplo de onde UML-RT esta inserida, explicaremos neste capítulo varias linguagens de modelagens relacionadas, como descrito a seguir, a UML-RT. Apesar de discutimos elas superficialmente, conseguimos traçar que influências elas despertam umas nas outras. Na Seção 2.1 exibiremos alguns problemas relacionados à versão de UML 1.5. Na Seção 2.2 mostraremos algumas linguagens idealizadas para a descrição de arquiteturas utilizando componentes. Na Seção 2.3, introduziremos a sintaxe da linguagem UML-RT, e como ela foi inspirada em UML e em tais linguagens para descrição de arquiteturas. Por fim, na Seção 2.4, discutiremos sobre a nova versão de UML, reformulada com base em várias características encontradas em UML-RT.

2.1 UML 1.X

Desde a padronização da Linguagem de Modelagem Unificada (UML) pelo OMG (*Object Management Group*) em 1997, ela tem se tornado para a comunidade um padrão *de facto* para a especificação e modelagem de *software*.

UML provê uma diversividade de técnicas para descrever aspectos estruturais e comportamentais de um sistema modelado. Diagramas de classe, objeto, componente e implantação focam principalmente na estrutura do sistema, enquanto diagramas de estados enfatizam no comportamento, tipicamente de componentes individuais. Diagramas de atividades, seqüência, colaboração e casos de uso também focam no comportamento, porém adicionalmente referenciam elementos estruturais.

Diagramas de seqüência e colaboração são utilizados para descrever interações entre componentes, expressando a ordem de ocorrência de mensagens trocadas entre eles. Devido à sua expressividade limitada (por exemplo, eles não conseguem expressar alternativas ou repetições), eles são utilizados normalmente apenas para mostrar cenários de utilização do sistema.

Devido a popularidade de UML, métodos e práticas de modelagem tem se desenvolvido enormemente. Diversas necessidades têm requerido mudanças e extensões da atual versão de UML, versão 1.5. Tais extensões têm gerado novos *profiles*, ou culminado na reestruturação da linguagem, UML versão 2.0. Podemos destacar alguns problemas na atual versão de UML que tem gerado tais mudanças.

- Falta de uma semântica formal
- Tamanho excessivo e multiplicidade de modelos.
- Suporte inadequado para o desenvolvimento baseado com componentes.

Semântica Formal: UML é conhecida como uma linguagem semi-formal, cuja necessidade de noções mais formais é reconhecida em diversos trabalhos [30, 4, 58]. A semântica de seus elementos sintáticos é imprecisa, o que gera problemas de comunicação entre os membros do projeto de *software*. Além disto, sem uma semântica formal precisa não é possível automatizar análises rigorosas de um modelo, tampouco executá-lo diretamente para testar seu comportamento; a única maneira de se testar um sistema modelado é o codificando.

Através da atribuição de uma noção formal a seus elementos, pode-se prover suporte à verificação de propriedades do modelo, atribuir noções de equivalência entre modelos, e, conseqüentemente, realizar transformações seguras de seus elementos; reestruturando o modelo sem alterar suas propriedades.

Tamanho Excessivo a Multiplicidade de Modelos: A versão atual de UML tem sido vista por muitos como muito grande e complexa [22, 44, 7]. Seu tamanho excessivo transforma seu aprendizado, aplicação e implementação difícil. Além disto, a multiplicidade de seus modelos resulta numa quantidade excessiva de diagramas e símbolos para expressar os aspectos estruturais e comportamentais do modelo. Tais diagramas não conseguem expressar isoladamente estes aspectos, resultando em problemas de produtividade e consistência durante sua elaboração [22].

Suporte ao desenvolvimento baseado em componentes: Apesar de suas inúmeras vantagens, UML tem diversas desvantagens quando comparado com linguagens para descrição de componentes [91]. Isto se deve ao fato de que diagramas de componentes em UML não tem por finalidade representar a decomposição lógica de um sistema, durante seu projeto, em subsistemas composicionais e reusáveis. Em UML, um componente é uma unidade física da implementação [81], sendo utilizado somente na fase de implantação. Além disto, UML não oferece o conceito de conexões como objetos de primeira ordem, visto com um híbrido entre associação (associação entre classes) e uma dependência entre uma classe e a interface de outra classe. Tais interfaces não podem ser diretamente utilizadas para descrever, em um nível de detalhes suficiente, as múltiplas interações entre componentes de *software* [20].

Várias abordagens utilizam UML no Desenvolvimento Baseado em Componentes (CBD), porém esta linguagem não é especializada para este tipo de desenvolvimento, e certas extensões ao seu padrão são necessárias [20, 48]. Devidos aos problemas já descritos, abordagens naturais utilizando classes, interfaces, e subsistemas não contemplam

adequadamente a especificação do comportamento de um componente [48]. Tais problemas estimularam a criação de extensões de UML, a sua unificação com Linguagens de Descrição de arquitetura (ADLs) (Seção 2.2.5), ou mesmo a reformulação da linguagem (Seção 2.4)

2.2 ADLS

Componentes provém um nível de abstração mais alto que objetos. Usualmente, eles podem ser vistos como caixas pretas que fornecem ou requerem um conjunto de serviços (representados por interface). Componentes que interagem podem assim ser compostos para formar um componente com um maior nível de abstração. A estrutura destas composições são o foco da arquitetura de *software*, que é definida em [91], como: *"A arquitetura de um sistema de* software *define o sistema em termos de componentes computacionais e de suas interações"*.

Para a especificação de arquitetura de *software*, um grande número de Linguagens de Descrição de Arquitetura (ADLs) têm sido propostas [64]. Esta especificação é utilizada para descrever a interação entre componentes, através de seu comportamento individual, interfaces de comunicação internas e externas, além da influência mútua entre estas interfaces.

Em [64], ADLs têm como características essenciais a definição explícita de componentes, conexões e configurações de arquitetura e a possibilidade de modelagem de interfaces de componentes; definidos com uma semântica clara. Aspectos adicionais, relacionados a estas noções podem ser desejáveis, porém não essenciais. Seus benefícios serão reconhecidos, e demonstrados, em contextos de problema específico (por exemplo, aspectos não funcionais de tempo ou desempenho), porém sua ausência não significa que a linguagem não representa uma ADL.

Várias linguagens possuem uma semântica formal para descrever o comportamento de seus componentes e conexões, por exemplo, através de algebra de processos [3] ou máquinas de estados finitas. Este tipo de semântica reforça o uso da declaração de propriedades arquiteturais, e assegura o mapeamento de arquiteturas de um nível de abstração para outro. Adicionalmente, é desejável que ADLs possibilitem refinamentos consistentes e corretos de arquiteturas de sistemas abstratas em outras exeqüíveis. Outros aspectos desejáveis são a possibilidade de evolução e de dinamismo da configuração de arquiteturas de *software*. Enquanto o primeiro está relacionado à adição e manutenção de funcionalidade na arquitetura, o segundo aspecto se refere à modificação da configuração do sistema durante a sua execução.

Vários ADLs são considerados em respeitos a diferentes aspectos do desenvolvimento baseado em componentes e arquitetura de *software*. A seguir ilustramos alguns ADLs que tem uma suporte formal de sua descrição ou influenciaram de alguma maneira mudanças na versão de UML 2.0.

2.2.1 ROOM

A Linguagem de Modelagem de Tempo Real Orientada a Objetos (ROOM) [89] utiliza um variante do conceito de "componente-porta-conector"para a arquitetura de componentes, descrita em [91, 23]. Tais conceitos de componentes, portas e conectores, são semelhantes a outros ADLs, e são descritos como:

- Componentes: Um componente (ou atores): é qualquer elemento que realiza algum tipo de computação. A declaração deste descreve sua interface externa, seu comportamento, e uma estrutura interna, utilizada para a composição de novos componentes.
- Portas: um conjunto de portas definem as interfaces externas de um ator. Portas são os únicos locais por onde um ator oferece ou requisita serviços; sendo portanto bidirecional.
- Protocolos: Um protocolo define o conjunto de sinais permitidos na transmissão ou recepção de mensagem entre componentes conectados. Qualquer porta associada a um protocolo deve enviar os sinais de saída e receber os de entrada deste protocolo.
- Portas Conjugadas: Portas podem ser conjugadas com relação ao seu protocolo associado, recebe os sinais de saída de seu protocolo e envia seus sinais de entrada.
- Conectores: Um conector estabelece uma conexão entre duas portas. Ele liga uma porta de um componente a uma porta conjugada de outro componente com um protocolo associado compatível (na maioria das vezes, o mesmo).

ROOM descreve principalmente dois tipos de diagramas. Diagramas de atores em ROOM descrevem a decomposição hierárquica de um sistema de software em seus componentes, como também as conexões entre estes componentes; estes diagramas são utilizados para descrever a estrutura interna de componentes. ROOM Charts, por outro lado, são utilizados para descrever o comportamento de componentes e protocolos, sendo derivados de StateCharts [43].

2.2.2 Wright

Wright [3] é um ADL com suporte à descrição de componentes, conexões e composição de componentes. Ela foca na especificação comportamental de componentes e conexões através de uma notação formal baseada em CSP. As interfaces de eventos de cada componente misturam os eventos transmitidos e recebidos pelo componente, não distinguindo serviços fornecidos e requeridos.

Componentes podem ser descritos individualmente através de uma especificação explícita, ou através da composição de outros componentes utilizando operadores de CSP, de uma maneira *bottom-up*; caracterizando descrição comportamental caixa branca. Através deste formalismo, é possível realizar verificações de consistência e integridade da especificação; verificando, por exemplo, se está livre de *deadlocks*.

2.2.3 ACME

O propósito de ACME [36] é prover um formato para a troca de informação entre ferramentas e ambientes de diferentes ADLs. Ele é baseado em abstrações comuns existentes em ADLs, suportando diretamente componentes, conexões e sistemas (arquiteturas). Para suportar qualquer ADL, ACME inclui uma sintaxe aberta para característica baseada em propriedades definidas pelo usuário, com uma semântica fora do escopo de ACME. Através de extensões da linguagem, ACME é capaz de descrever comunicação de componentes através de eventos, podendo descrever seqüências de eventos (*traces*), e utilizar a notação de XML para descrever arquiteturas, focando apenas aspectos estruturais do sistema.

2.2.4 SDL

A Linguagem de Especificação e Descrição (SDL) [40] é um padrão da ITU-T, e aceito pela ISO. SDL é uma linguagem formal e orientada a objetos, com suporte a descrição de componentes ativos (processos) e passivos (dados), eventos de comunicação e composição de componentes, além de outras características inerentes a ADLs. Por ter uma base formal utilizando máquinas de estados abstratas (ASM) [40], ela provê uma especificação não ambígua de sistemas.

Apesar de, em SDL, canais de comunicação serem definidos explicitamente, interconexões entre componentes são especificados implicitamente no comportamento dos componentes (através do uso de canais comuns). Por esta razão, linguagens como esta são chamadas em [64] como linguagems de configuração *in-line* e estariam mais próximas de serem classificadas como linguagens de interconexão de módulos (MIL) [73] do que propriamente como ADLs.

2.2.5 ADLs e UML

Em [35], é argumentado que UML não é adequado para modelar a estrutura de sistemas tipicamente utilizados em linguagens de descrição de arquiteturas. O maior problema é que nem todos os elementos encontrados em ADLs possuem um mapeamento direto para os conceitos de modelagem encontrados em UML. Apesar disto, diversos trabalhos [18, 82, 76] mapeiam os conceitos arquiteturais de ADLs em UML, geralmente alterando o significados dos elementos de UML através de estereótipos; o "como" este elementos são modificados não é especificado, utilizando geralmente a semântica associadas a estes em seus contra-elementos em ADL. Estes trabalhos contribuem para a melhoria de um aspecto desejável de compreensão da arquitetura, através do mapeamento de ADLs em linguagens mais populares e disseminadas na comunidade.

Em [18], os conceitos de ACME são mapeados em UML através de estereótipos. Devido a algumas especifidades da linguagem, nem todos os conceitos de ACME puderam ser representados em UML. De forma similar, em [76], estes mecanismos de extensão (estereótipos) são utilizados para mapear os ADLs: Wright, Darwin [61], and Rapide [59].

Em [82, 18], o conceitos arquiteturais de ROOM são mapeados em UML, através do uso de estereótipos. Em [82] o papel de cada porta é definido explicitamente por uma

classe, que implementa o comportamento associado ao seu protocolo em ROOM e realiza uma interface com os serviços associados a este este protocolo; portas conjugadas realizam interfaces de protocolos conjugados. Este modelo facilita o uso de portas como elementos de primeira classe, porém dificultam a herança de comportamento dos protocolos e restrigem os serviços dos protocolos à simples chamadas de métodos. Em [18] um mapeamento menos detalhado é realizado, indicando somente quais os conceitos arquiteturais de ROOM poderiam ser representados em UML, da mesma forma que em [82] sinais de protocolos são restringidos a métodos de classes em UML.

Em [90], a integração entre UML e ROOM é realizada adicionando-se a UML elementos com uma nova semânticas. Sendo o resultado desta integração uma linguagem mais aceita pela comunidade, popularmente chamada de UML-RT. Tal linguagem cria novos elementos (cápsulas) para representar componententes de software em UML, reservando componentes passivos para serem representados como classes ordinárias. Os diagrams de atores são representados em extensão do diagrama de colaboração, enquanto ROOM Charts são representados por diagramas de estados específicos para cápsulas e protocolos. Apesar de conter em sua definição elementos para representar os papeis realizados pelas portas (ordinárias ou conjugadas), ela sugere uma notação mais compacta para a representação destes papeis através de associações e sufixos no nome destas portas, como a que é empregada em suas ferramentas [19] e assumida por nós na Seção 2.3.

Como vimos a integração de ADLs e UML podem gerar novas extensões da UML (profile de UML-RT), ou mesmo contribuem para à reformulação de novas versões de UML. Em [88], é indicado que várias das características de modelagem arquitetural de UML 2.0 são derivadas de três linguagem de de descrição arquitetural, UML-RT [90] [61], ACME [36] e SDL [40].

2.3 UML-RT

A especificação e projeto de sistemas distribuídos é uma tarefa complexa que envolve a especificação de dados, comportamento, comunicação e de aspectos arquiteturais do sistema. Com a finalidade de atender estes requisitos, UML e ROOM foram combinadas na linguagem UML for Real-Time (UML-RT) [90]. Apesar de seu nome, UML-RT não possui um suporte adequado para qualquer aplicação que envolva tempo real, na realidade ela é focada na modelagem de sistemas baseados em componentes e sistemas embarcados qualificados como soft-real time (onde o tempo apesar de fazer parte da modelagem, não é um fator crítico). Sendo o enfoque deste trabalho a modelagem de componentes ativos, sem lidar aspectos de tempo.

UML-RT é considerada um extensão conservativa de UML, introduzindo apenas novos elementos e diagramas (baseados nos conceitos de ROOM) que se inter-relacionam com os elementos já existentes em UML. Quatro novos construtores são introduzidos: cápsulas, protocolos, portas e conectores.

Cápsulas descrevem componentes ativos, potencialmente concorrentes, do sistema, que possuem uma estrutura interna e uma comportamento associado. Diferentes de atores em ROOM, cápsulas são utilizadas primordialmente para descrever elementos ativos, com um fluxo de dados independente do restante do sistema. Enquanto elementos passivos

são descritos por classes em UML.

Assim, uma cápsula tem além de seu comportamento passivo definidos em seus métodos, o seu comportamento ativo definido por uma máquina de estados. Na realidade o comportamento das cápsulas são reativas, e dependem de estímulos externos para que alguma ação associada seja executada. Tais estímulos são originados pelo ambiente, e só podem ser transmitidos ou recebidos a uma cápsula através de suas portas, objetos definem um ponto de interação com a cápsula. Portas realizam protocolos, que definem o conjunto de sinais de entrada e saída de uma cápsulas, que representam os serviços fornecidos e requeridos pela cápsula. Um protocolo também define um fluxo válido destes sinais através das portas. Conectores agem como canais de comunicação físicos entre portas.

Um modelo em UML-RT é formado por um conjunto de diagrama e propriedades, tal qual em UML. Para que os elementos de ROOM fossem incorporados em UML-RT, alguns diagramas foram estendidos: diagramas de classe, estado e estrutura (extensão de diagramas de colaboração). Este conjunto de diagramas consegue representar inteiramente as seguintes visões arquiteturais: dados, comportamento, configuração da arquitetura. A fim de simplificar a especificação do modelo, iremos focar nosso trabalho somente nesta visões; outros diagramas que indicam somente cenários de aplicação do sistema, como diagramas de seqüência não serão considerados. Expressaremos propriedades do sistema diretamente na linguagem *Circus* 3; Eles poderiam ser alternativamente expressados em OCL, porém um mapeamento entre OCL e *Circus* é fora de nosso escopo. Neste trabalho, não consideraremos herança, devido a semântica de herança de cápsulas em UML-RT ainda não ser bem-definida.

Os diagramas de classes de UML definem adicionalmente a declaração de cápsulas e protocolos. Cápsulas podem ter associação para classes, cápsulas, ou protocolos do modelo. Uma associação com uma classe permite que a cápsula possa utilizar métodos e atributos da classe; gerando como conseqüência um atributo do tipo da classe na cápsula. Enquanto uma associação entre cápsulas é utilizada para permitir que parte do comportamento de uma cápsula seja explicado a partir da composição de instâncias de outras cápsulas. Por fim, uma associação a um protocolo gera, como conseqüência, uma porta na cápsula. Estes relacionamentos são vistos como agregações, onde o elemento de destino da associação é parte da cápsula, e é criado apenas após a criação da cápsula; referências cíclicas entre cápsulas não são permitidas. Associações para cápsulas, oriundas de classes e protocolos também não são permitidas.

Apesar de serem associadas como tipos da linguagem, cápsulas não são elementos de primeira classe e por definição não podem ser atribuídas a nenhuma variável de ambiente. Sendo suas instâncias localizadas unicamente nos diagramas de estrutura que as contém; a visão hierárquica representada por este diagrama descreve a visão extensional das cápsulas do sistema. Por o foco deste trabalho ser a modelagem de cápsulas, e por não haver sentido o compartilhamento de instâncias de cápsulas em diferentes estrutura, assumiremos uma semântica de cópia na passagem de parâmetros de uma mensagem; classes não podem ser compartilhadas entre cápsulas. Como a principal aplicações deste trabalho é no desenvolvimento de sistemas distribuídos, onde cápsulas representam elementos passíveis de distribuição, consideramos que o compartilhamento de recursos entre cápsulas deve ser

feitos explicitamente através da troca de mensagem entre elas utilizando uma semântica de cópia. Esta é uma abordagem comum em tecnologias para sistemas distribuídos, como RMI ou CORBA.

Para ilustrarmos a notação de UML-RT através de um exemplo, um estudo de caso, utilizado também nos capítulos seguintes deste trabalho, de um sistema de automação industrial é utilizado. Este sistema é utilizado para o processamento de peças industriais. Tais peças são inseridas no sistema por um operador e após algum tempo são recuperadas do sistema por ele. Como pode ser visto na Figura 2.1, no diagrama de casos de uso do sistema. Neste capítulo uma versão inicial do sistema é apresentada (Figura 2.2), este por sua vez será o ponto de partida da modelagem do sistema durante um processo de desenvolvimento no Capítulo 5.2.

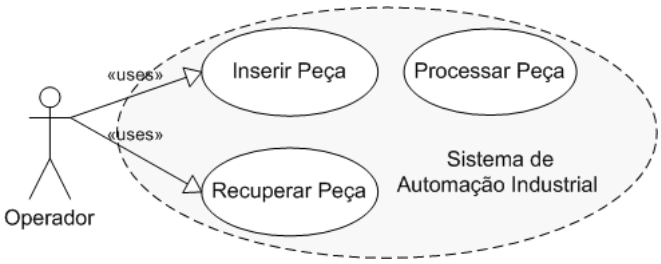


Figura 2.1. Caso de Uso do Sistema de Automação Industrial

Na Figura 2.2 é apresentado (a esquerda) o diagrama de classes do sistema, neste diagrama encontramos a declaração das cápsulas Storage e ProdSys. A cápsula Storage é um elemento de fronteira utilizado para armazenar peças industriais (representadas pela classe Pieces). A cápsula ProdSys tem como responsabilidade processar peças industriais. Além disto ProdSys representa todo o sistema, e, conseqüentemente, possui uma associação com as outras cápsulas do modelo. Na realidade as cápsulas Storage e ProdSys não foram identificadas ao acaso, elas representam elementos de análise extraídos a partir dos casos de uso do sistema. Graficamente, a declaração destas cápsulas é representada por uma caixa com um estereótipo Capsula e um símbolo em cima à esquerda. Eles são formados por três compartimentos, o primeiro indica a declaração de atributos, o segundo o de métodos e o terceiro o de portas, respectivamente de cima para baixo. Por exemplo, a cápsula ProdSys possui um atributo p para representar a peça que esta processando no momento, um método process para processar peças, e duas portas pi e po para interagir com as outras cápsulas do sistema. Note que neste diagrama não é indicado qual instâncias de cápsulas interagem entre si, isto é descrito no diagrama de estrutura.

Estas cápsulas também tem relacionamentos com protocolos, que são utilizados para dirigir a comunicação entre eles. Graficamente a declaração destes protocolos é representada por uma caixa com um estereótipo Protocol e um símbolo em cima à esquerda. Eles possuem dois compartimentos, o primeiro para descrever os sinais de entrada e o segundo para os sinais de saída do protocolos, respectivamente de cima para baixo. O protocolo STI governa inserção de peças em uma cápsula (representado únicamente pelo sinal input), enquanto STO a recuperação de peças (o sinal req é utilizado para requerer peças, enquanto o sinal output para enviar tais peças). Quando sinais são utilizados para

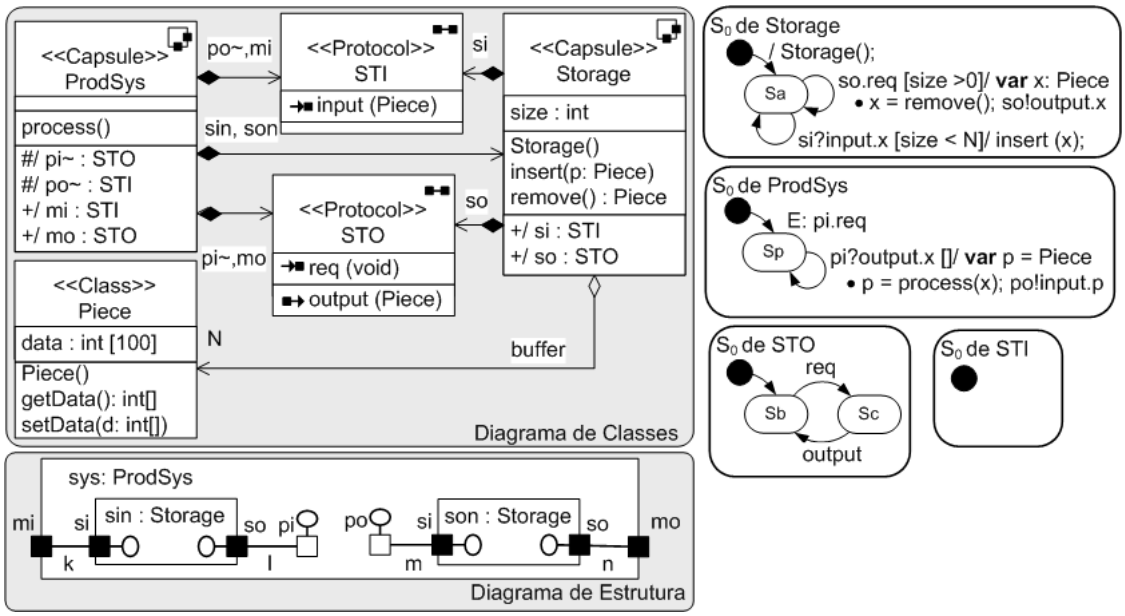


Figura 2.2. Modelo do Sistema de Automação Industrial

enviar mensagens de dados eles possuem uma classe associada ao tipo do dado, quando não o tipo associado é void.

Por sua própria natureza, cápsulas provém um alto nível de *information hiding*. Como seu mecanismo de comunicação é realizado através da passagem de mensagens por suas portas, todos os atributos e métodos das cápsulas são visíveis somente dentro do escopo da cápsulas; somente portas são visíveis externamente para serem conectadas à portas de outras cápsulas. Este desacoplamento faz com que cápsulas sejam altamente reutilizáveis, em adição cápsulas podem ser definidas hierarquicamente a partir da composição de outras cápsulas dentro de sua estrutura interna (representadas no diagrama de estrutura); cada uma delas com uma máquina de estados associada e um possível estrutura hierárquica compostas de outras cápsulas. Chamaremos aqui, as instâncias de cápsulas dentro da estrutura outra como sub-cápsulas ou cápsulas componentes.

Assumimos que a comunicação entre duas cápsulas conectadas é síncrona (como o modelo para UML-RT descrito em [8]), indicando que a cápsula de destino de um sinal sempre estará pronto para recebe-lo. Condições de mapeamento de um modelo síncrono para um assíncrono em UML-RT podem ser vistas em [8].

Um diagrama de estrutura estende o diagrama colaboração para indicar a interação entre cápsulas. Mostrando não somente dependências entre objetos, como em um diagrama de colaboração, mas indicando a configuração da arquitetura através da composição hierárquica de cápsulas, das conexões entre os pontos de interação de cada instância de cápsula da estrutura, dos protocolos associados a cada um destas conexões. Definindo assim a composição estrutural do modelo. A decomposição estrutural de ProdSys é mostrado na figura 2.2 (acima à direita). Ele é composto das instâncias sin, pro e son, respectivamente das cápsulas Storage, Processo e Storage. A cápsula sin é responsável por armazenar peças não processadas, son por armazenar peças processadas, e sys por

recuperar as peças oriundas de sin, processa-las e depositá-las em son. Sendo estas as representações gráficas para cápsulas em um diagrama de estrutura. Pequenos quadrados nas instâncias representam suas portas, onde os brancos indicam portas conjugadas (as direções dos sinais são invertidos em relação a definição do protocolo). Portas normais devem ser conectadas à portas conjugadas para que o sinais de saída de uma seja visto como o sinal de entrada da outra. As portas mi e mo são portas *relay* e servem unicamente para interligar as portas si e so ao mundo externo; por esta razão elas não precisam ser conjugadas. Enquanto as restantes são portas finais e são utilizadas para interligar os diagramas de estados das cápsulas ao ambiente externo. Para facilitar a visualização da arquitetura, a estrutura de todas as cápsulas são visíveis, porém normalmente não se é possível visualizar a partir do diagrama de estrutura de ProdSys o conteúdo e o tipo das portas contidos no diagrama de estrutura de sub-cápsulas.

O comportamento das cápsulas e do protocolos do modelo é descrito em termos de um diagrama de estados em UML-RT, que especializa diagramas de estado de UML [70] para adequá-los aos conceitos de ROOM Charts [86]. Estritamente, por serem usados para descrever objetos ativos, estes diagramas não possuem estados finais, tampouco suas transições são disparadas por eventos implícitos; todos os eventos devem ser associados a sinais externos. Assim como em UML, um diagrama de estados é composto por transições e estados; em geral, uma transição possui a formato `p.e[g]/a`, onde `e` é um sinal de entrada (ou um conjunto de sinais de entrada), `p` indica a porta de origem pelo qual o sinal foi recebido, `g` é uma guarda e `a` é uma ação. O recebimento de um sinais de entrada em um estado ativo e a avaliação da guarda correspondente como verdadeira disparam uma transição. Como resultado, a ação correspondente é executada e um novo estado é ativado.

Em UML, estados podem ser classificados como iniciais, escolha, compostos ou simples. Um estado inicial é um estado transiente que indica o ponto de início de uma máquina de estados. Um estado composto agrupo outros estados, enquanto um estado simples não possui nenhum outro dentro dele. Um estado de escolha corresponde aos estados que envolvem uma decisão de qual caminho (estado) deve ser seguido (ativado) em função de uma guarda associada ao estado; existem apenas duas transições de saída do estado: uma é disparada quando a guarda é verdadeira, e outra quando a guarda é falsa. Estados compostos são divididos em dois tipos: *Or-States* e *And-States*. *Or-State* definem uma composição sequêncial de estados, onde somente um deles é ativado por vez, enquanto *And-States* contém regiões (separados por uma linha pontilhada) que executam seqüencias de estados em paralelo, e permitem que cada região possua um estado ativo por vez. *And-States* são considerados como decisões de projetos; de fato eles podem ser representados pela composição de duas cápsulas que possui cada uma das regiões como máquina de estado. (ver Capítulo 4).

Para cada máquina de estados em UML-RT, assumimos que existe uma estado composto que possui todos os outros; ele é chamado de estado topo (ou S_0 e seu estado inicial é implicitamente disparado quando a instância da cápsula (ou porta) é criada. Em cada estado composto transições são associados diretamente a um ponto de junção (exibido graficamente por um circulo preto) na borda do estado. Se não é uma transição do ponto de junção com destino em um sub-estado, o estado inicial será ativado; caso tenha destino

a um sub-estado, este será ativado. Transições de saída de um estado composto podem emanar de um sub-estado ou diretamente da borda do estado. O último caso interrompe o estado e seus sub-estados em qualquer situação; ações de entrada e saída (ações executadas antes do estado ser tornar ativo e inativo, respectivamente) são sempre executadas normalmente.

A Figura 2.2 (abaixo) apresenta o diagramas de estados das cápsulas Storage, ProdSys e dos protocolos STO e STI , eles são denominados graficamente como os estados S_0 dos respectivas cápsulas. Nos dois primeiros casos o comportamento das cápsulas são descritas por um estado composto do tipo *Or-State*, que executa seqüencialmente seus estados. Nenhum comportamento em ProdSys esta associado à sua máquina de estados, podendo ser explicado unicamente pela composição das sub-cápsulas que compõe. Na cápsula Storage, por exemplo, após a sua criação, o estado inicial se torna ativo e ação contendo o método Storage() é executada, isto é equivalente à execução de um construtor da cápsula. Após isto o estado Sa se torna ativo e espera pelos sinais req e input, quando sinais chegarem e suas respectivas guardam forem avaliadas como verdadeiras as ações associadas a estes estados são executadas, retornando assim ao estado Sa; uma descrição mais detalhada sobre o comportamento da máquina de estado de Storage será descrita no Capítulo 3.

Nós assumimos que eventos, guardas e ações são expressas utilizando a notação de *OhCircus*. Por exemplo, no diagrama de estados de ProdSys, a ação da transição inicial é representada pelo envio de um sinal req através da porta pi (pi.req), e o evento que dispara a única transição de saída de Sp é expresso por pi?output.x, indicando que o recebimento do sinal output através da porta pi, cujo valor é atribuído a uma variável local x. A ação desta transição executa um método process() que modifica o valor de uma variável p e após isto envia o evento po!input.p, que comunica o valor p pelo sinal input através da porta po. Note o envio de sinais sempre está contido nas ações do diagrama de estado, e que os prefixos ? e ! sempre estão relacionados a eventos que comunicam dados de entrada e saída, respectivamente.

2.4 UML 2.0

A versão 2.0 de UML introduz novos conceitos à versão 1.5 e altera outros já existentes[1]. Nesta seção, falamos dos conceitos da versão 2.0 referenciando minimamente o metamodelo de UML (o núcleo da definição semântica da linguagem). Apesar disto, para apresentar os relacionamentos entre esses conceitos, precisamos descrever certos aspectos do metamodelo, como em particular, o conceito de classificadores, um tipo que pode ter instâncias; por exemplo, classes são classificadores cujas as instâncias são objetos. Este conceito é importante porque dois dos elementos de UML que vamos discutir, classes e componentes, são classificadores. Assim, quando descrevemos que um conceitos pode ser aplicado a classificadores, estamos, conseqüentemente, descrevendo como eles podem ser aplicados à classes e componentes.

Esta versão combina várias características encontradas em alguns ADLs para espe-

[01] Até a data de entrega desta dissertação a verão 2.0 de UML ainda não havia sido publicada como uma versão oficial da OMG.

cificação de componente à UML [88], como por exemplo decomposição hierárquica de componentes. Todas estas características são importantes para formar um novo conceito de componentes, tratados agora como componentes de software ao invés de meros fragmentos físicos do software para propósitos de implantação. Isto indica que componentes são pedaços modularizados, reutilizáveis, e implantáveis do software disponíveis durante o desenvolvimento, e durante a execução do sistema.

Cada classificador (classes e componentes) podem ser compostos por outros elementos contidos dentro da estrutura interna do classificador. Esta estrutura interna descreve como os elementos podem ser compostos por outros elementos, chamados de partes ou elementos conectáveis. Estes não são nunca classificadores, porém instâncias ou conjunto de instâncias de classificadores. O relacionamento entre estes elementos e os de sua estrutura interna é a de agregação, no sentido que os as partes não podem existir sem os elementos que as contém.

Adicionalmente esta estrutura indica como o componente interage com o ambiente através de suas interfaces, ou portas. Interfaces e portas são utilizadas para desacoplar as instâncias do ambiente; onde, sob o ponto de vista a instância é vista como uma caixa preta com vários pontos de interação (interfaces ou portas). Interface se diferenciam de portas, por oferecerem seus serviços como métodos. Enquanto portas oferecem seus serviços através de conjunto de sinais.

Máquinas de estados foram subdivididas em máquinas de estados comportamentais e de protocolo. A primeira é utilizada para o mesmo propósito que antes, descrever o comportamento interno de uma classificadores, enquanto a segunda é usada para definir o comportamento abstrato associado a interfaces e portas.

2.5 CONCLUSÕES

Apesar de UML-RT não ter sido criado por organizações de padronização (como a OMG), sendo mantida pelas empresas que suportam suas ferramentas, vários conceitos desta linguagem estão diretamente presentes em outras linguagens para a descrição de arquiteturas (ADLS) ou de componentes (UML 2.0). Unido-os em uma única linguagem, que tem como vantagens, uma separação clara entre elementos ativos e passivos, os quais possuem uma semântica diferente para a passagem de mensagens, decomposição hierárquica de seus componentes, e a definição explícita de inter-conexões e interfaces de seus componentes.

Além disto, possui um pequeno conjunto de diagrams, capazes de descrever os aspectos estruturais e dinâmicos do modelo. Desta forma, UML-RT é uma linguagem enxuta é descritiva para a modelagem de componentes ativos. Apesar disto, possui como desvantagem, não poder descrever explicitamente conexões com diferentes comportamentos; isto pode ser representado implicitamente através de componentes que funcionam como adaptadores desta conexão. Apesar disto, esta linguagem não ser, neste aspectos, menos representativa que outras linguagens para a descrição de arquiteturas, e todas as contribuições que possam ser incorporadas ao desenvolvimento em UML-RT podem ser, conseqüentemente, futuramente aplicadas a UML 2.0, linguagem ainda não oficial sob fase de elaboração pela OMG, .

CAPÍTULO 3

FORMALIZAÇÃO DE UML-RT

Neste capítulo, mostraremos a formalização de UML-RT através de seu mapeamento sintático na linguagem de especificação *OhCircus* [16]. Com esta formalização damos significado aos elementos de UML-RT, em termos da semântica de um subconjunto (*Circus*) de *OhCircus*.

A motivação para uso desta linguagem formal é que ela oferece um cálculo de refinamentos que auxilia na prova de transformações de modelos em UML-RT usando o mapeamento em *OhCircus*. Além disto, objetos ativos e suas conexões podem ser facilmente representados através de processos e canais de comunicação de *OhCircus*.

Apresentamos, na Seção 3.1, uma breve introdução à linguagem *OhCircus*, incluindo noções de sua sintaxe, semântica e cálculo de refinamentos. Como mostramos na Seção 3.1.2, a semântica e o cálculo de refinamentos é restrito ao subconjunto *Circus*, mas isso é suficiente para lidarmos com as construções que UML-RT adiciona a UML. Um mapeamento de UML em *OhCircus* é proposto em [10]. Na Seção 3.2, apresentamos o mapeamento dos principais elementos de UML-RT em *OhCircus*.

3.1 *OHCIRCUS*

OhCircus é uma linguagem orientada a objetos projetada para lidar com a especificação e o refinamento de sistemas concorrentes. *OhCircus* combina a álgebra de processos CSP [79] com a linguagem baseada em modelos Z [95] com o intuito de capturar aspectos relacionados ao estado e à comunicação do sistema em uma mesma especificação, como em [94].

Para expressar a comunicação de sistemas concorrentes, *OhCircus* inclui a noção de processo, cujo estado é definido por um esquema em Z e o comportamento por uma ação na notação de CSP. A interação entre processos é realizada somente através de canais, utilizados para comunicar valores ou somente para a sincronização.

Além de sua utilização para especificação de sistemas concorrentes e reativos, *OhCircus* foi projetada para suportar uma teoria de refinamentos. O objetivo desta teoria é dar uma base sólida ao desenvolvimento de sistemas concorrentes e distribuídos, utilizando um cálculo de refinamentos semelhante ao encontrado em [67].

3.1.1 Sintaxe

Da mesma maneira que especificações em Z, programas em *OhCircus* são formados por uma seqüência de parágrafos:

$$\mathsf{Program} \quad ::= \quad \mathsf{OhCircusParagraph}^*$$

Onde, $\mathsf{OhCircusParagraph}^*$ denota um lista com zero ou mais elementos da categoria

sintática OhCircusParagraph. Cada um destes parágrafos pode ser um parágrafo de Z (denotados aqui pela categoria sintática Paragraph, conforme definido em [95]), ou a definição de canais, grupo de canais, processos ou classes (denotados, respectivamente, pelas categorias sintáticas ChannelDefinition, ChanSetDefinition OhProcessDefinition, ClassDefinition).

$$
\begin{array}{lcl}
\mathsf{OhCircusParagraph} & ::= & \mathsf{Paragraph} \\
 & | & \mathsf{ChannelDefinition} \mid \mathsf{ChanSetDefinition} \\
 & | & \mathsf{OhProcessDefinition} \mid \mathsf{ClassDefinition}
\end{array}
$$

Os principais construtores de *OhCircus* são ilustrados através de uma parte da especificação do sistema de automação industrial (previamente introduzido na Seção 2.3); a gramática completa de *OhCircus* pode ser encontrada no Apêndice A. Na Figura 3.1, apresentamos a especificação da cápsula Storage, utilizada para armazenar objetos da classe Piece, através de um processo $Chart_{\mathsf{Storage}}$ em *OhCircus*. Utilizaremos o mesmo fonte da Figura 2.2 para identificadores (por exemplo, nomes de atributos, classes e cápsulas) encontrados diretamente no modelo em UML-RT.

O tamanho máximo de armazenamento é representado através de uma constante positiva N, declarada no primeiro parágrafo da Figura 3.1.

$$
\begin{array}{l}
\mid N : \mathbb{N} \\
T_{\mathsf{STI}} ::= \mathsf{input} \ll \mathsf{Piece} \gg \\
\textbf{channel}\ \mathsf{si} : T_{\mathsf{STI}} \\
T_{\mathsf{STO}} ::= \mathsf{req} \mid \mathsf{output} \ll \mathsf{Piece} \gg \\
\textbf{channel}\ \mathsf{so} : T_{\mathsf{STO}} \\
\textbf{process}\ Chart_{\mathsf{Storage}} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ StorageState \mathrel{\widehat{=}} [\mathsf{buff} : \mathrm{seq}\,\mathsf{Piece};\ \mathsf{size} : 0..N \mid \mathsf{size} = \#\mathsf{buff} \leq N] \\
\quad StorageInit \mathrel{\widehat{=}} [StorageState' \mid \mathsf{buff}' = \langle\rangle \wedge \mathsf{size}' = 0] \\
\quad \mathsf{insert} \mathrel{\widehat{=}} [\Delta StorageState;\ x? : \mathsf{Piece} \mid \mathsf{size} < N \wedge \\
\qquad \mathsf{buff}' = \mathsf{buff} \frown \langle x? \rangle \wedge \mathsf{size}' = \mathsf{size} + 1] \\
\quad \mathsf{remove} \mathrel{\widehat{=}} [\Delta StorageState;\ x! : \mathsf{Piece} \mid \mathsf{size} > 0 \wedge x! = head\,\mathsf{buff} \wedge \\
\qquad \mathsf{buff}' = tail\,\mathsf{buff} \wedge \mathsf{size}' = \mathsf{size} - 1] \\
\quad \mathsf{Sa} \mathrel{\widehat{=}} (\mathsf{size} < N\ \&\ \mathsf{si?input.x} \rightarrow \mathsf{insert(x)};\ \mathsf{Sa}) \\
\qquad \Box\ (\mathsf{size} > 0\ \&\ \mathsf{so.req} \rightarrow (\textbf{var}\,\mathsf{x} : \mathsf{Piece} \bullet \mathsf{x} = \mathsf{remove}();\ \mathsf{so!output.x});\ \mathsf{Sa}) \\
\bullet\ StorageInit;\ \mathsf{Sa} \\
\textbf{end}
\end{array}
$$

Figura 3.1. Especificação da cápsula Storage

Todos os canais utilizados por um processo devem ser declarados. As categorias sintáticas Expression e Schema−Exp são utilizadas para designar expressões em Z e expressões de esquema definidas em [95]. A categoria sintática N é utilizada para identificadores válidos em Z.

$$
\begin{array}{lcl}
\mathsf{ChannelDefinition} & ::= & \textbf{channel}\ \mathsf{CDeclaration} \\
\mathsf{CDeclaration} & ::= & \mathsf{SimpleCDeclaration} \mid \mathsf{SimpleCDeclaration};\ \mathsf{CDeclaration} \\
\mathsf{SimpleCDeclaration} & ::= & \mathsf{N}^{+} \mid \mathsf{N}^{+} : \mathsf{Expression} \mid \mathsf{Schema-Exp}
\end{array}
$$

A declaração de uma canal é composta por um nome e o valor comunicado pelo canal. Caso o canal seja utilizado apenas para a sincronização de processos, sem comunicação de valor, a declaração conterá apenas o nome do canal, sem um tipo associado.

Uma mesma declaração pode introduzir mais de um canal, desde que tenham o mesmo tipo. Neste caso, ao invés de um único nome, temos uma lista de nomes de canais separados por vírgula.

Em nosso exemplo, o processo armazena e fornece objetos através dos canais si e so, respectivamente. Os tipos T_{STI} e T_{STO} indicam que valores podem ser comunicados por estes canais; neste exemplo, esses tipos são declarados como tipos enumerados. Em T_{STO}, req é um construtor (de tipo) utilizado no exemplo para representar a requisição de objetos do processo, e output um construtor utilizado para representar o fornecimento dos objetos requisitados. E input, em T_{STI}, é utilizado para representar o armazenamento de objetos do tipo *Piece*.

A declaração de um processo é composta por seu nome e sua especificação. Apesar de *OhCircus* possibilitar a herança de processos, (utilizando-se a palavra-chave **extends**), esta característica não é abordada neste trabalho.

$$
\mathsf{OhProcessDefinition} \quad ::= \quad \textbf{process}\ \mathsf{N} \mathrel{\widehat{=}} [\textbf{extends}\ \mathsf{N}]\ \mathsf{Process}
$$

Um processo pode ser explicitamente definido, ou pode ser definido em termos de outros processos. Uma definição explícita de um processo é delimitada pelas palavras-chaves **begin** e **end**; ela é formada por uma seqüência de parágrafos do processo e uma ação principal (no final da definição, após o símbolo $\bullet$), que define o comportamento do processo. Um destes parágrafos é utilizado para descrever o estado do processo através da notação Z, após a palavra-chave **state**.

$$
\begin{array}{lcl}
\mathsf{Process} & ::= & \textbf{begin}\ \mathsf{PParagraph}^{*} \\
 & & \quad [\textbf{state}\ \mathsf{Schema-Exp}] \\
 & & \quad \mathsf{PParagraph}^{*} \\
 & & \quad [\bullet\ \mathsf{Action}] \\
 & & \textbf{end} \\
 & \mid & \mathsf{CompProcess}
\end{array}
$$

Em nosso exemplo, o processo $Chart_{\mathsf{Storage}}$ encapsula dois componentes de estado no esquema *StorageState*: uma lista de objetos e o tamanho desta lista.

Na definição explícita de um processo, além da definição de seu estado e ação principal, o seu corpo é formado por parágrafos em Z e ações. Estes parágrafos são utilizados para estruturar a especificação e são referenciados pela ação principal.

$$
\mathsf{PParagraph} \quad ::= \quad \mathsf{Paragraph} \mid \mathsf{N} \mathrel{\widehat{=}} \mathsf{ParAction} \mid \textbf{nameset}\ \mathsf{N} == \mathsf{NSExp}
$$

Uma ação pode ser um esquema, um comando guardado, uma invocação para uma ação previamente definida, ou uma combinação de outras ações utilizando operadores de

CSP.

$$\mathsf{Action} \quad ::= \quad \mathsf{Schema-Exp} \mid \mathsf{CSPActionExp} \mid \mathsf{Command} \mid \mathsf{N}$$

O corpo de parágrafos do processo $Chart_{\mathsf{Storage}}$ é formado por três esquemas em Z que modificam o estado: *StorageInit*, insert e remove. O esquema *StorageInit* especifica a operação de inicialização do processo, definindo a sua lista de objetos como vazia. A operação insert adiciona um objeto à lista e atualiza o atributo size com seu novo tamanho, enquanto a operação remove modifica o estado, removendo o primeiro elemento da lista e decrementando o valor de size (ver Figura 3.1).

Assim como em Z, interrogações (?), exclamações (!) e apostrofos ($'$) são utilizados para decorar variáveis de entrada, de saída ou com estado final (valor após a operação), respectivamente.

Três ações primitivas de CSP estão disponíveis em *OhCircus*: *Skip*, *Stop* e *Chaos*. A ação *Skip* não comunica qualquer valor, tampouco alterar o estado: ela termina imediatamente. A ação *Stop* representa *deadlock*, e a ação *Chaos* representa divergência. A única garantia em ambos os casos é que o invariante do estado é mantido.

$$\begin{array}{lcl}
\mathsf{CSPActionExp} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \\
 & \mid & \mathsf{Communication} \rightarrow \mathsf{Action} \mid \mathsf{Predicate} \ \& \ \mathsf{Action} \\
 & \mid & \mathsf{Action}; \ \mathsf{Action} \mid \mathsf{Action} \ \Box \ \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \ |[\ \mathsf{NSExp} \mid \mathsf{CSExpression} \mid \mathsf{NSExp}\ ]| \ \mathsf{Action} \\
 & \mid & \mathsf{Action} \ \|[\mathsf{NSExp} \mid \mathsf{NSExp}]\| \ \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CSExpression} \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \mid \ldots \\
\mathsf{Communication} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\mathsf{N} \mid ?\mathsf{N} : \mathsf{Predicate} \mid !\mathsf{Expression} \mid .\mathsf{Expression}
\end{array}$$

Comunicações (representadas pela categoria sintática Communication) em *OhCircus* obedecem ao mesmo padrão de CSP, porém guardas podem ser associadas a elas; a categoria sintática Predicate, utilizada para predicados em Z, é definida em [95]. Por exemplo, dado um predicado p em Z, se a condição p for verdade, a ação $p \ \& \ (c?x \rightarrow A)$ recebe um valor através do canal c e o atribui à variável x no mesmo escopo, executando, após isto, a ação A. No entanto, se a condição p for falsa, esta ação é bloqueada (refutada). Isto possibilita que condições, como a guarda p, possam ser associadas a qualquer ação.

Todas as variáveis livres precisam estar no mesmo escopo que uma ação. Todos os componentes do estado estão em um escopo global para qualquer ação do processo. Valores de entrada de canais introduzem novas variáveis no escopo, e atribuem este valor a elas; estas variáveis não podem receber novas atribuições. Os operadores de composição de CSP para ações de seqüência (;), escolha interna ($\sqcap$) e externa ($\Box$), paralelismo ($|[\mathit{NSExp} \mid \mathit{CSExpression} \mid \mathit{NSExp}]|$), *interleaving* ($\|[\mathit{NSExp} \mid \mathit{NSExp}]\|$), e *hiding* ($\setminus$) podem ser utilizados para compor ações. Operadores de CSP para a definição de comunicação e de recursividade (μ) também são permitidos no nível de ações. Outros operadores de composição de CSP podem ser encontrados na gramática de *OhCircus* (ver Apêndice A).

Em nosso exemplo, a ação principal inicializa o processo $Chart_{\mathsf{Storage}}$ e oferece continuamente a escolha para o ambiente dos eventos si?input.x e so.req. Esta escolha pelo

ambiente é definida através do operador de escolha externa ($\Box$). Tal escolha é continuamente oferecida através do uso do operador de recursividade (μ).

$$\begin{aligned}
&\mathsf{Sa} \mathrel{\widehat{=}} (\mathsf{size} < N \ \&\ \mathsf{si?input.x} \rightarrow \mathsf{insert(x)};\ \mathsf{Sa}) \\
&\qquad \Box\ (\mathsf{size} > 0 \ \&\ \mathsf{so.req} \rightarrow (\mathbf{var}\, \mathsf{x} : \mathsf{Piece} \bullet \mathsf{x} = \mathsf{remove}();\ \mathsf{so!output.x});\ \mathsf{Sa}) \\
&\bullet\ \mathit{StorageInit};\ \mathsf{Sa}
\end{aligned}$$

O processo somente aceita input se existir espaço para armazenar o novo objeto; o sinal é guardado por $\mathsf{size} < N$. Caso a guarda seja satisfeita, o objeto é adicionado à lista através da operação insert, que também incrementa a variável size. A ação $\mathsf{si?input.x} \rightarrow \mathsf{insert(x)}$ é escrita no estilo de CSP. Nesta ação, a variável x é declarada dinamicamente com o tipo dado pelo canal si e com o valor correspondente ao comunicado através do mesmo. O evento so.req somente é habilitado caso o processo contenha algum objeto em sua lista; caso esta condição seja verdadeira, o retorno de uma ação remove() (o objeto inicial da lista buff) é atribuído a uma variável x, seguido do envio do valor de x através do evento so!output.x.

No nível de ações, os operadores de paralelismo e *interleaving* são levemente diferentes daqueles de CSP. Com a intenção de evitar conflitos no acesso às variáveis em escopo, estes operadores de ações precisam declarar dois conjunto que particionam todas as variáveis em escopo: componentes de estado, variáveis locais e de entrada. O operador de paralelismo de *OhCircus* segue uma abordagem alfabetizada, adotada em [79]. Quando processos são postos em paralelo, o conjunto de eventos em que eles sincronizam necessitam ser explicitamente especificado; eventos que não são listados ocorrem independentemente. Por exemplo, no paralelismo $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, as ações A_1 e A_2 sincronizam nos canais do conjunto cs. A ação A_1 pode modificar somente os valores das variáveis em ns_1; similarmente, a ação A_2 pode modificar valores das variáveis em ns_2. As ações compostas pelo operador de *interleaving* se comportam de maneira similar ao operador de paralelismo em relação às suas partições.

Após definidos explicitamente, os processos podem ser combinados utilizando-se operadores de CSP para formar novos processos.

$$\begin{aligned}
\mathsf{CompProcess} \ ::= \ & \mathsf{N} \mid \mathsf{Proc};\ \mathsf{Process} \mid \mathsf{Process} \setminus \mathsf{CSExpression} \\
\mid \ & \mathsf{Process} \Box \mathsf{Process} \mid \mathsf{Process} \sqcap \mathsf{Process} \mid \\
\mid \ & \mathsf{Process} \mid\mid\mid \mathsf{Process} \mid \mathsf{Process} \llbracket \mathsf{CSExpression} \rrbracket \mathsf{Process} \mid ...
\end{aligned}$$

Dois Processos P_1 e P_2 podem ser combinados, por exemplo, em seqüência ($P_1;\ P_2$), onde o processo P_2 executa após o término com sucesso da execução de P_1, ou em paralelo $P_1 \llbracket cs \rrbracket P_1$, sincronizando sobre eventos de cs; seguindo uma abordagem alfabetizada como a adotada em [79].

No nosso exemplo do sistema de automação (Seção 2.3), o processo que representa a cápsula MAIN é definido pela composição paralela de todos os processos do sistema (cápsulas), sincronizando em seus canais de comunicação, como apresentado na Seção 3.2. Por exemplo, de maneira simplificada, os processos relativos às instâncias das cápsulas Storage e Processo são compostos utilizando o operador de paralelismo $\mathsf{Storage[so := c2]} \llbracket \{ \mathsf{c2} \} \rrbracket \mathsf{Processo[pi := c2]}$. Neste caso, para que os dois processos sincronizem em um canal de comunicação comum c2, os canais so e pi devem ser renomeados para c2.

A sintaxe para a definição de classe é expressa por seu nome, sua super classe através da cláusulas **extends**, e seus atributos e métodos delimitados pelas cláusula **begin** e **end**. Por não abordarmos herança neste trabalho, toda classe deverá omitir a cláusula **extends**. Assim como em processos, a definição de atributos e métodos de uma classe consiste em uma seqüência de parágrafos, onde o parágrafo com a cláusula **state** identifica o esquema de estado da classe com seus componentes de estado (atributos). O esquema de estado de uma classe é definido em Z assim como os estados de um processo, porém inclui qualificadores (**public**, **protected**, **private** e **logical**) na declaração de seus componentes. Caso um atributo não seja qualificado, este é assumido como privado.

$$
\begin{array}{lcl}
\mathsf{ClassDefinition} & ::= & \textbf{class}\ \mathsf{N} \mathrel{\widehat{=}} [\textbf{extends}\ \mathsf{N}]\textbf{begin}\ \mathsf{CParagraph}^* \\
 & & \quad [\textbf{state}\ \mathsf{StateSchema}]\ \mathsf{CParagraph}^* \\
 & & \quad [\textbf{initial}\ \mathsf{Schema-Exp}]\ \mathsf{CParagraph}^* \\
 & & \textbf{end} \\
\mathsf{CParagraph} & ::= & \mathsf{Paragraph} \mid \mathsf{Qualifier}\ \mathsf{N} \mathrel{\widehat{=}} \mathsf{ParametrisedCommand} \\
\mathsf{Qualifier} & ::= & \textbf{public} \mid \textbf{protected} \mid \textbf{private} \mid \textbf{logical}
\end{array}
$$

A cláusula **initial** introduz o construtor da classe. Este não é introduzido para ser um método, apenas para definir a criação da classe, não devendo ter em sua especificação variáveis de saída. Métodos (categoria sintática CParagraph) são declarados como parágrafos em Z, diferenciando-se porém pela possibilidade de utilizarem qualificadores de acesso, indicando em que escopo podem ser chamados.

Na Figura 3.2, a especificação de Piece é descrita através de um parágrafo com o estado da classe e três parágrafos com operações. O esquema de estado é composto de um único componente data, representado por uma seqüência de naturais de tamanho 100. O parágrafo *PieceInit* representa o construtor da classe Piece, inicializando todos os valores de data com 0. Os métodos setData e getData são utilizados somente para alterar e recuperar o valor do componente data.

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Piece} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ \mathit{PieceState} \mathrel{\widehat{=}} [\mathsf{data} : \mathrm{seq}\,\mathbb{N} \mid \#\mathsf{data} = 100] \\
\quad \textbf{initial}\ \mathit{PieceInit} \mathrel{\widehat{=}} [\mathit{PieceState}' \mid \mathrm{ran}\,\mathsf{data}' = \{0\}] \\
\quad \textbf{public}\ \mathsf{setData} \mathrel{\widehat{=}} [\Delta \mathit{StorageState};\ \mathsf{d?} : \mathrm{seq}\,\mathbb{N} \mid \mathsf{data}' = \mathsf{d?}] \\
\quad \textbf{public}\ \mathsf{getData} \mathrel{\widehat{=}} [\Xi \mathit{StorageState};\ r! : \mathrm{seq}\,\mathbb{N} \bullet r! = \mathsf{data}] \\
\textbf{end}
\end{array}
$$

Figura 3.2. Especificação da classe Piece

3.1.2 Semântica

Apesar de utilizamos *OhCircus* para o mapeamento de um subconjunto compreensível de UML-RT (Seção 3.2), classes do sistema são semanticamente vistas, aqui, apenas como registros não compartilhados de dados, representados em Z. Não sendo necessário assim

abordar todas as características de orientação a objetos de UML. A semântica de classes para UML em Z (e portanto para UML-RT) é considerada em diversos trabalhos [30, 12, 69]. Como este trabalho foca apenas na definição semântica dos elementos que UML-RT adiciona à UML, a utilização de um subconjunto de *OhCircus* (*Circus*) que não utiliza orientação a objetos é satisfatória para a finalidade deste trabalho..

O modelo semântico de *Circus* é baseado na Teoria Unificada da Programação (UTP) [47]. A UTP é um framework no qual a teoria de relações são utilizadas como uma base unificada para a ciência da programação, através de seus diferentes paradigmas: procedural e declarativo, seqüencial e paralelo, acoplado e distribuído, e *hardware* e *software*.

Programas, projetos, e especificações são interpretados como relações entre uma observação inicial e subseqüente, que pode ser uma observação intermediária ou final, do comportamento da execução de um programa. Leis do cálculo relacional podem ser utilizadas como ferramental de prova.

Desta maneira, idéias, como composição seqüencial, condicional, não-determinismo, e paralelismo são compartilhadas por diferentes teorias. Em cada teoria, variáveis de observação são identificadas para descrever seus aspectos mais importantes, e sub-teorias são criadas a partir de restrições do relacionamento entre estas variáveis de observação. Um exemplo é a própria linguagem *Circus*, que estende a teoria de CSP para processos reativos. Em *Circus* a definição de um processo, por exemplo, possui ambientes para armazenar informações de canais e variáveis em escopo, componentes de estado, parágrafos e processos que compõe.

Neste trabalho, não entramos no detalhe da semântica de processos ou canais em *Circus*; a semântica completa de *Circus* pode ser encontrada em [100]. Entretanto, mostramos como os operadores de composição de processos, podem ser representados através de processos simples, cuja semântica é apresentada em [100].

3.1.2.1 Expressões de Processos Expressões de processos utilizam operadores de CSP para combinar processos existentes. A semântica de uma expressão $R \,\mathsf{op}\, Q$, onde op é uma operação binária qualquer em CSP, exceto paralelismo e *interleaving*, é definida da seguinte forma.

$$
\begin{array}{l}
R \,\mathsf{op}\, Q = \mathbf{begin} \\
\qquad\qquad \mathbf{state}\ State \mathrel{\widehat{=}} R.st \wedge Q.st \\
\qquad\qquad R.pps \uparrow Q.st \\
\qquad\qquad Q.pps \uparrow R.st \\
\qquad\qquad \bullet\ R.act \,\mathsf{op}\, Q.act \\
\qquad\quad \mathbf{end}
\end{array}
$$

O estado dos processos R e Q são denotados por $R.st$ e $Q.st$, respectivamente. De uma maneira similar, a notação $R.pps$ e $Q.pps$ são utilizadas para se referir aos parágrafos na definição dos processos R e Q. Suas ações principais são denotadas com $R.act$ e $Q.act$. As operações do processo composto são formadas pelos esquemas de $R.pps \uparrow Q.st$ e $Q.pps \uparrow Q.st$. O termo $R.pps \uparrow Q.st$ indica que as operações em $R.pps$ não podem alterar os componentes do estado em $Q.st$; para garantir isto, é adicionado à operação a expressão ΞQ. Similarmente, $Q.pps$ não altera os componentes em $R.st$.

Como discutido na Section 3.1.1, os operadores de paralelismo e *interleaving* para ações são levemente diferentes daqueles utilizados para processos; no nível de ações, partições com todas as variáveis no escopo precisam ser declaradas. Por esta razão, um novo operador $R.pps \uparrow_{PAR} Q.st$ é utilizado na definição de paralelismo e *interleaving*. Este operador libera as operações em $R.pps$ para alterar todas as variáveis de ambiente, utilizando ΔQ ao invés da expressão ΞQ. Apesar disto, esta permissão é ignorada, já que os componentes de estado de $Q.st$ estão declarados na mesma partição utilizada pelas ações de $Q.act$; de forma similar, $R.st$ pertencem à partição utilizada pelas ações de $R.act$. Assim a semântica do operador de paralelismo é definida como segue.

$$
\begin{array}{l}
R \llbracket cs \rrbracket Q = \textbf{begin} \\
\qquad\qquad \textbf{state}\ State \mathrel{\widehat{=}} R.st \wedge Q.st \\
\qquad\qquad R.pps \uparrow_{PAR} Q.st \\
\qquad\qquad Q.pps \uparrow_{PAR} R.st \\
\qquad\qquad \bullet\ R.act \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Q.act \\
\qquad\quad \textbf{end}
\end{array}
$$

Na definição do operador de paralelismo, $R.st$ está contido em ns_1, e $Q.st$ em ns_2. O operador de *interleaving* é definido de forma similar.

A semântica do operador de *hiding* é mais simples: os parágrafos do processo são incluídos sem alterações, somente a ação principal é modificada para esconder os canais usados como argumento do *hiding*.

$$
\begin{array}{l}
R \setminus cs = \textbf{begin} \\
\qquad\qquad \textbf{state}\ R.st \\
\qquad\qquad R.pps \\
\qquad\qquad \bullet\ R.act \setminus cs \\
\qquad\quad \textbf{end}
\end{array}
$$

A semântica de outros operadores pode ser encontrada em [99]

3.1.3 Noções de Refinamento e Equivalência

Uma das noções centrais na UTP é a de refinamento, que é definida em termos de implicação: uma implementação P satisfaz uma especificação S se, e somente se, $[P \Rightarrow S]$, onde os colchetes denotam o quantificador universal sobre o alfabeto utilizado por P e S, como em [21]; o alfabeto deve ser o mesmo para a implementação e para especificação. A noção de equivalência, é igual à demonstração do refinamento em ambas as direções; na UTP, $[P = S]$ se e somente se $[P \Rightarrow S]$ e $[S \Rightarrow P]$.

Em *Circus* (e, portanto, em *OhCircus*), a noção básica de refinamento para ações e dada pela Definição 3.1 [15].

Definição 3.1 (Refinamento de Ações) *Para ações A_1 e A_2 no mesmo espaço de estados, $A_1 \sqsubseteq_{\mathcal{A}} A_2$ se e somente se $[A_1 \Rightarrow A_2]$.* □

Processos, que possuem seus estados privados, possuem uma definição levemente diferente. A ação principal de um processo define seu comportamento. Por esta razão,

refinamento de processos é definido em termos de refinamento de ações de blocos locais. A seguir, $P_1.st$ e $P_1.act$ denotam o estado local e a ação principal do processo P_1; similarmente para o processo P_2.

Definição 3.2 (Refinamento de Processos) $P_1 \sqsubseteq_{\mathcal{P}} P_2$ *se e somente se* $(\exists\, P_1.st;\ P_1.st' \bullet P_1.act) \sqsubseteq_{\mathcal{A}} (\exists\, P_2.st;\ P_2.st' \bullet P_2.act)$ □

As ações $P_1.act$ e $P_2.act$ podem agir em diferentes espaços de estado. As ações destes processos são comparadas ignorando-se seus componentes de estado; o espaço de estado contém somente as variáveis observadas na UTP.

Como dito anteriormente, o estado de um processo é privado, permitindo que seus componentes sejam modificados durante um refinamento, através de refinamento de dados [68]. Em [17], a técnica de simulação aplicada em Z foi adotada para lidar com processos e ações. Uma simulação é uma relação entre os estados de dois processos que satisfazem um conjunto de propriedades.

Definição 3.3 (*Forwards Simulation*) *Uma* forwards simulation *entre as ações* A_1 *e* A_2 *de processos* P_1 *e* P_2*, com um estado local* L*, é uma relação* R *entre* $P_1.st$*,* $P_2.st$*, e* L*, satisfazendo.*

- *(Aplicabilidade)* $\forall\, P_2.st;\ L \bullet (\exists\, P_1.st \bullet R)$
- *(Corretude)* $\forall\, P_1.st;\ P_2.st;\ P_2.st';\ L \bullet R \wedge A_2 \Rightarrow (\exists\, P_1.st';\ L' \bullet R' \wedge A_1)$ □

Neste caso, nós escrevemos $A_1 \preceq_{P_1,P_2,R,L} A_2$. Uma *forwards simulation* entre P_1 e P_2 é uma *forwards simulation* entre suas ações principais.

Como destacado em [17], diferentemente da definição usual de *forwards simulation*, na Definição 3.3 não existe um requisito de aplicabilidade em relação às pré-condições. Isto se deve ao fato de que ações são totais. Se uma ação é executada fora de sua pré-condição, ela diverge; porém, seu comportamento não é arbitrário, já que o invariante de estado é implicitamente mantido, e novas sincronizações arbitrárias podem ser observadas, mas observações passadas não são afetadas. Além disto, nenhuma condição especifica é imposta em sua inicialização: qualquer inicialização de estado precisa ser explicitamente incluída na ação principal.

Um teorema em [17] indica que se existe uma relação de *forwards simulation* existe também uma relação de refinamento. Uma estratégia de refinamento para *Circus* foi definida em [17]. Esta estratégia, pode envolver várias iterações com três passos: refinamento de dados, refinamento de ações, e refinamento de processos. Os dois primeiros passos são utilizados para reorganizar a estrutura de um processo, alterando-se seus componentes de estados e suas ações. O terceiro passo é utilizado para refinar (tipicamente, decompor) processos, preservando o comportamento.

3.1.4 Leis para refinamento de Processos

Leis para o refinamento de processos lidam simultaneamente com estado e comportamento. Nesta seção, apresentaremos duas leis para o refinamento de processos; mais leis podem ser encontradas em [17].

A primeira lei indica que podemos introduzir novos processos em uma especificação, assumindo que eles não sejam utilizados. Como na introdução de novas variáveis em uma linguagem imperativa, o fato de o processo não ser utilizado é suficiente para garantir que sua introdução não afeta os demais processos.

Lei 3.1 *Introdução de declaração de processo*

$$cp = pd\ cp$$

desde que: *o processo declarado em pd não seja referenciado nem declarado na seqüência de parágrafos do programa em* Circus *cp.* □

Apesar desta lei expressar uma propriedade simples, sua importância se torna evidente em sua composição com outras leis.

A lei a seguir é aplicada para particionar um processo cujos componentes de estado podem ser divididos em dois conjuntos, de uma maneira tal que cada um deles possui seu próprio conjunto de parágrafos, que não interfere no outro. A forma geral é apresentada pelo P abaixo.

$$\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat{=}} Q.st \wedge R.st \\
\quad Q.pps \uparrow R.st \\
\quad R.pps \uparrow Q.st \\
\quad \bullet\ F(Q.act, R.act) \\
\textbf{end}
\end{array}$$

Como na definição da expressão de processos (Seção 3.1.2.1), o estado do processo P é definido pela conjunção de outros dois esquemas $Q.st$ e $R.st$. Além disto, os parágrafos de P são também particionados de maneira que os parágrafos em $Q.pps$ não alteram os componentes de $R.st$; de uma maneira similar, os parágrafos em $R.pps$ não alteram os componentes em $Q.st$. Quando dois conjuntos de parágrafos satisfazem estas condições, dizemos que eles são disjuntos com respeito a $R.st$ e $Q.st$. Finalmente, a ação principal de P é definida como uma composição F das ações $Q.act$ e $R.act$, onde F é um contexto (função sobre processos) envolvendo operadores de CSP que podem ser utilizados na composição de processos.

A Lei 3.2, apresentada abaixo, transforma um processo particionado P (como definido acima) em três processos, onde os dois primeiros incluem as partições dos estados e dos parágrafos de P, e o terceiro processo é definido em termos dos dois primeiros, utilizando a mesma função F que define a ação principal de P.

Lei 3.2 (Decomposição de processo) *Seja qd e rd declarações dos processos Q e R, determinados por $Q.st$, $Q.pps$, e $Q.act$, e $R.st$, $R.pps$, e $R.act$, respectivamente, e pd a declaração do processo P. Então:*

$$pd = (qd\ rd\ \textbf{process}\ P \mathrel{\widehat{=}} F(Q, R))$$

desde que: *$Q.pps$ e $R.pps$ sejam disjuntos com respeito a $R.st$ e $Q.st$, respectivamente.* □

3.2 MAPEAMENTO

Nesta seção, atribuímos significado semântico aos elementos de UML-RT através de seu mapeamento sintático para *OhCircus*. Esta tradução provê um mapeamento tanto da visão estrutural quanto comportamental dos elementos de UML-RT para *OhCircus*.

Em nossa estratégia de mapeamento, o destino da tradução é uma especificação em *Circus* que possui o significado do modelo original. Classificadores de UML-RT com um comportamento associado (cápsulas e protocolos) são mapeados em processos, e portas em canais.

Como mencionado anteriormente, o mapeamento de classes já foi considerado por diversos autores [12, 29, 10], e está fora do escopo deste trabalho. Assumimos que classes em UML podem ser mapeadas diretamente em classes de *OhCircus*, como exposto em [10]. Além disto, assumimos um modelo em UML-RT mais concreto, onde associações são diretamente interpretadas como atributos de classe, e invariantes devem ser locais ao escopo da classe. Apesar destas considerações serem decididas somente durante o projeto do modelo, a antecipação dessas decisões são comumente realizadas na concretização do modelo e são justificadas através de refinamentos de modelo UML, como apresentado em [10].

Aqui somente associações unidirecionais são utilizadas, indicando um atributo da classe, e não relacionamentos entre todas as instâncias da classe. Assim, restrições sobre estas associações são confinadas dentro dos invariantes de classe (ou cápsula). Além disto, invariantes podem incluir somente referências para variáveis no escopo da classe (ou cápsula), descrita por atributos da classe (ou cápsula) ou por atributos de classes acessados via navegação.

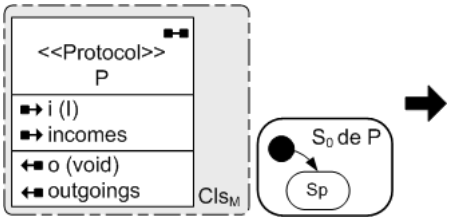
Consideramos tanto uma visão *intensional* quanto *extensional* do modelo. A visão intencional é mapeada diretamente nos elementos sintáticos de UML-RT. A visão extensional é implicitamente definida pelo diagrama de estrutura de suas cápsulas; eles contém o conjunto de todas as instâncias de cápsulas. Para lidar com a sua estrutura hierárquica em *Circus*, assumimos que todas as instâncias de cápsulas, portas e conectores têm um único nome no modelo.

Iniciamos o mapeamento a partir dos protocolos do modelo, e , em seguida, a cápsula que representa todo o sistema; incrementalmente, cada componente de cápsula é mapeado para *Circus*, seguindo uma apresentação *top-down*. No exemplo de automação industrial (Figura 2.2), a cápsula mais externa é ProdSys.

Quando mapeamos elementos declarados em uma lista (como os sinais de um protocolo, e atributos e métodos de uma cápsula), por convenção destacamos um de seus elementos, apresentamos seu mapeamento, e invocamos uma meta-função ($\mathcal{TL}()$) para traduzir os outros elementos da lista. Por simplicidade, assumimos que $\mathcal{TL}()$ é sobrecarregada (*overloaded*), para cada tipo de lista (atributos, métodos, sinais, portas, instâncias de cápsula). Na prática, obviamente, cada lista de elementos pode ser vazia, porém evitamos este caso trivial, assumindo que ela possui pelo menos um elemento.

3.2.1 Mapeamento estrutural

Uma declaração de um protocolo em UML-RT encapsula definições de elementos de comunicação (sinais) e um comportamento de controle (diagrama de estados). Em *OhCircus*, isto gera dois elementos: um processo sem estado que captura o comportamento do protocolo e um canal para representar os elementos de comunicação. Em relação aos sinais, um possível mapeamento poderia ser feito introduzindo-se um canal para cada um dos sinais do protocolo. Ao invés disto, nós utilizamos um único canal para comunicar todos os sinais de um protocolo. Este canal comunica valores de um tipo enumerado, onde cada um de seus construtores representa um sinal. Utilizar um único canal facilita o mapeamento de cápsulas apresentado a seguir. Por exemplo, quando nos referimos aos sinais envolvidos em uma sincronização ou mudança de nome (*renaming*), é mais conveniente utilizar um único canal.



$$T_{\mathsf{P}} ::= \mathsf{i} \ll \mathcal{T}(\mathsf{I}) \gg | \ \mathsf{o} \mid \mathcal{TL}(\mathsf{incomes}) \mid \mathcal{TL}(\mathsf{outgoings})$$
$$\textbf{channel } chan_{\mathsf{P}} : T_{\mathsf{P}}$$
$$\textbf{process } \mathsf{P} \mathrel{\widehat{=}} \textbf{begin} \bullet \mathcal{H}(\mathsf{S_P}) \textbf{ end}$$

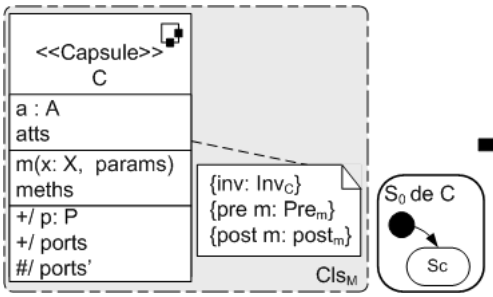
O protocolo P é mapeado no processo de mesmo nome e no canal $chan_{\mathsf{P}}$. Em nomes com índice subscrito, como $chan_{\mathsf{P}}$ acima, nós assumimos que P é um nome que deverá ser substituído pelo nome do protocolo sendo mapeado. Desta forma, o mapeamento do protocolo STO no estudo de caso (Figura 2.2) gera um canal correspondente $chan_{\mathsf{STO}}$. O canal $chan_{\mathsf{P}}$ comunica valores de tipo enumerado T_{P}; cada valor representa um sinal. Sinais sem parâmetros, como o sinal de saída o acima, são traduzidos em constantes; sinais parametrizados são mapeados em construtores de tipos de dados (como i). O tipo do parâmetro é traduzido em seu tipo correspondente em *OhCircus* pela função $\mathcal{T}()$. Os sinais restantes (incomes e outgoings) são mapeados pela função $\mathcal{TL}()$, como previamente explicado.

O comportamento do protocolo P é representado por $\mathcal{H}(\mathsf{S_P})$, onde $\mathsf{S_P}$ representa o estado que encapsula todos os outros estados do diagrama de estados de P. Note que, apesar do diagrama de estados de um protocolo poder ser sempre representado por um estado $\mathsf{S_0}$, preferimos destacar todo o comportamento de P em um subestado $\mathsf{S_P}$, onde o comportamento de $\mathcal{H}(\mathsf{S_0})$ é equivalente ao de $\mathcal{H}(\mathsf{S_P})$. A função $\mathcal{H}()$, que traduz um diagrama de estados em ações de *OhCircus*, é definida na Seção 3.2.2.

Em nosso exemplo da Figura 2.2, o mapeamento do protocolo STO possui o formato a seguir. O comportamento do protocolo é definido pelo processo homônimo P e seus elementos de comunicação (sinais) são definidos pelo canal $chan_{\mathsf{STO}}$ que comunica valores do tipo enumerado T_{STO}, explicado anteriormente na Seção 3.1.1. A ação principal de STO é expressa pela aplicação da função $\mathcal{H}$ ao seu estado mais externo S0.

$$T_{\mathsf{STO}} ::= \mathsf{req} \mid \mathsf{output} \ll \mathcal{T}(\mathsf{Piece}) \gg$$
$$\textbf{channel } chan_{\mathsf{STO}} : T_{\mathsf{STO}}$$
$$\textbf{process } \mathsf{STO} \mathrel{\widehat{=}} \textbf{begin} \bullet \mathcal{H}(\mathsf{S_0}) \textbf{ end}$$

Cápsulas são definidas também como processos, com métodos definidos em esquemas de operações e atributos mapeados em esquemas de estado em Z (o estado do processo). Cada porta gera um canal com o mesmo tipo de dados do canal correspondente ao seu protocolo, e tem seu comportamento descrito pelo processo obtido do mapeamento de seu protocolo sincronizado com o obtido do diagrama de estados da cápsula. Observe que, em UML-RT, o tipo de uma porta é o próprio protocolo. Em *OhCircus*, o tipo de um canal originado por uma porta é um tipo enumerado que representa seus sinais (como explicado no mapeamento de protocolos).



$$
\begin{array}{l}
\textbf{channel}\ \mathsf{p} : T_{\mathsf{P}};\ \mathcal{TL}(\mathsf{ports});\ \mathcal{TL}(\mathsf{ports}') \\
\textbf{process}\ \mathit{Chart}_{\mathsf{C}} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ \mathsf{C}_{state} \mathrel{\widehat{=}} [\mathsf{a} : \mathcal{T}(\mathsf{A});\ \mathcal{TL}(\mathsf{atts}) \mid \mathsf{Inv}_{\mathsf{C}}] \\
\quad m \mathrel{\widehat{=}} [\Delta \mathsf{C}_{state};\ \mathsf{x} : \mathcal{T}(\mathsf{X}); \\
\qquad\qquad\qquad \mathcal{TL}(\mathsf{params}) \mid \mathsf{Pre}_{\mathsf{m}} \wedge \mathsf{Post}_{\mathsf{m}}] \\
\quad \mathcal{TL}(\mathsf{meths}) \\
\quad \bullet\ \mathcal{H}(\mathsf{S}_{\mathsf{c}}) \\
\textbf{end}
\end{array}
$$

No mapeamento acima, o processo $\mathit{Chart}_{\mathsf{C}}$ lida com as visões representadas pelos diagramas de classe e estado, encapsulando todas as ações que manipulam os atributos privados da cápsula C. Na cápsula C acima, os compartimentos correspondem à listas de atributos, métodos e portas. Então, a, m e p são aqueles que destacamos. O atributo a é mapeado para um atributo do estado de $\mathit{Chart}_{\mathsf{C}}$ com seu tipo correspondente em *OhCircus*, dado por $\mathcal{T}(\mathsf{A})$; os outros atributos atts são mapeados pela função $\mathcal{TL}()$, como explicado previamente. O invariante $\mathsf{Inv}_{\mathsf{C}}$ se origina da nota em UML-RT no contexto à esquerda, e assume-se que este já esteja descrito em *OhCircus*. O método m() é mapeado para uma operação que pode modificar qualquer componente do estado, e cujos parâmetros são mapeados em atributos do esquema. Como o invariante $\mathsf{Inv}_{\mathsf{C}}$, assume-se que a pré- e a pós-condição $\mathsf{Pre}_{\mathsf{m}}$ e $\mathsf{Post}_{\mathsf{m}}$ sejam escritas em *OhCircus*(ver Seção 2.3). Similarmente a função $\mathcal{TL}()$ mapeia os outros métodos meths. A porta p é mapeada para um canal com o mesmo tipo T_{P} do canal $\mathit{Chan}_{\mathsf{P}}$ utilizado pelo protocolo P. A ação principal de $\mathit{Chart}_{\mathsf{C}}$ é expressa por $\mathcal{H}(\mathsf{S}_{\mathsf{C}})$, que representa o mapeamento dos diagramas de estado da cápsula C, definido na Seção 3.2.2

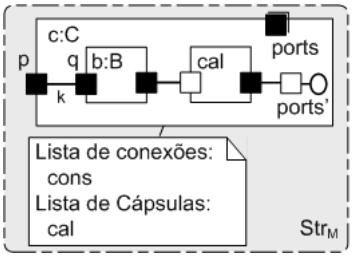
Em nosso exemplo da Figura 2.2, o mapeamento da cápsula SysProd é representado pelo processo $\mathit{Chart}_{\mathsf{SysProd}}$. Definido por um estado com um único componente p, uma operação process que realiza um processamento qualquer em p e uma ação principal, que

será exibida na Seção 3.2.2.

$$
\begin{array}{l}
\textbf{channel}\ \mathsf{pi}, \mathsf{mo} : T_{\mathsf{STO}} \\
\textbf{channel}\ \mathsf{po}, \mathsf{mi} : T_{\mathsf{STI}} \\
\textbf{process}\ \mathit{Chart}_{\mathsf{SysProd}} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ \mathsf{SysProd}_{state} \mathrel{\widehat{=}} [\mathsf{p} : \mathsf{Piece}] \\
\quad \mathit{process} \mathrel{\widehat{=}} [\Delta \mathsf{SysProd}_{state} \mid \forall\, i \in \operatorname{ran} \mathsf{p}'.\mathsf{data} \mid i \neq 0 \bullet \mathsf{p.data}(i) = \mathit{fat}(i)] \\
\quad \ldots \\
\quad \bullet\ \mathcal{H}(\mathsf{S}_0) \\
\textbf{end}
\end{array}
$$

De fato, a declaração dos atributos e métodos de SysProd são bastantes simples, a complexidade maior desta cápsula se encontra em seu diagrama de estrutura. Um mapeamento mais elaborado da declaração dos métodos e atributos da cápsula Storage podem ser encontrados na Seção 3.1.1.

Para a formalização de cápsulas, precisamos considerar também diagramas de estrutura. O mapeamento que lida com a visão correspondente a este diagrama, representa o comportamento observado por C considerando as restrições impostas pelo comportamento de suas portas aos seus respectivos canais de comunicação. É necessário considerar também, em paralelo, o comportamento de todas as cápsulas conectadas a sua estrutura.



$$
\begin{array}{l}
\textbf{channel}\ \mathsf{k} : T_{\mathsf{P}};\ \mathcal{TL}(\mathsf{cons}) \\
\textbf{chanset}\ \mathit{cc} \mathrel{\widehat{=}} \{\!|\ k\ |\!\} \cup \{\!|\ \mathsf{cons}\ |\!\} \\
\textbf{chanset}\ \mathit{hchan} \mathrel{\widehat{=}} \mathit{cc} \cup \{\!|\ \mathsf{ports}'\ |\!\} \cup \{\!|\ \mathsf{ports}_{\mathsf{B}}\ |\!\} \cup \{\!|\ \mathsf{ports}_{\mathsf{Cap}}\ |\!\} \\
\textbf{process}\ \mathit{Struct}_{\mathsf{C}} \mathrel{\widehat{=}} ((\mathit{Chart}_{\mathsf{C}}\ \ [\![\ \mathsf{cc}\ ]\!]\ (\mathsf{P}[\mathit{chan}_{\mathsf{P}} := \mathsf{k}]\ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad (\mathcal{TL}(\mathit{ports})\ [\![\ \mathsf{cc}\ ]\!]\ \mathcal{TL}(\mathsf{ports}')))) \ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad ((\mathit{Struct}_{\mathsf{Cin}}[\mathsf{q}, \mathsf{ports}_{\mathsf{Cin}} := \mathsf{k}, \mathsf{cons}])\ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad \mathcal{TL}(\mathit{cal})))[\mathsf{k}, \mathsf{cons} := \mathsf{p}, \mathsf{ports}] \setminus \mathit{hchan}
\end{array}
$$

No processo ($\mathit{Struct}_{\mathsf{C}}$), o comportamento de $\mathit{Chart}_{\mathsf{C}}$ é sincronizado com o comportamento de todas as portas da cápsula C; destacamos aqui a porta p, cujo comportamento é representado pelo processo P. O canal $\mathit{chan}_{\mathsf{P}}$ utilizado por P necessita ser renomeado com o nome do conector (k) que liga a porta p a outras portas no sistema. As outras portas são similarmente mapeadas através da função $\mathcal{TL}()$. Nós destacamos o componente de cápsula b (do tipo B) da lista dos outros componentes Cal, que tem o nome de todas as suas portas públicas (q e $\mathsf{ports}_{\mathsf{B}}$) alterados para o das conexões associadas a eles (k e cons); $\mathsf{ports}_{\mathsf{B}}$ representa a lista de portas públicas de B. Após paralelizar todos os componentes no diagrama de estrutura C, uma função injetiva é necessária para alterar o nome de conexões às portas públicas associadas a elas ($[\mathsf{m}, \mathsf{cons} := \mathsf{p}, \mathsf{ports}]$). Além disto é necessário esconder os canais associados a conexões, portas protegidas de C, e portas públicas de B e de Cal, visíveis somente no diagrama de estrutura de C.

Em nosso exemplo da Figura 2.2, o mapeamento do diagrama de estrutura da cápsula SysProd é representado por um processo $\mathit{Struct}_{\mathsf{SysProd}}$.

$$
\begin{array}{l}
\textbf{channel}\ \mathsf{k}, \mathsf{m} : T_{\mathsf{STI}} \\
\textbf{channel}\ \mathsf{l}, \mathsf{n} : T_{\mathsf{STO}} \\
\textbf{chanset}\ cc \mathrel{\widehat{=}} \{\!|\ \mathsf{k}, \mathsf{m}, \mathsf{l}, \mathsf{n}\ |\!\} \\
\textbf{chanset}\ hchan \mathrel{\widehat{=}} cc \cup \{\!|\ \mathsf{pi}, \mathsf{po}\ |\!\} \\
\textbf{process}\ Struct_{\mathsf{SysProd}} \mathrel{\widehat{=}} (Chart_{\mathsf{SysProd}}[\mathsf{pi}, \mathsf{po} := \mathsf{l}, \mathsf{m}]\ \ [\![\ \mathsf{cc}\ ]\!]\ \mathsf{STI}[chan_{\mathsf{STI}} := \mathsf{k}]\ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad\qquad\qquad \mathsf{STI}[chan_{\mathsf{STI}} := \mathsf{n}]\ [\![\ \mathsf{cc}\ ]\!]\ \mathsf{STO}[chan_{\mathsf{STO}} := \mathsf{l}]\ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad\qquad\qquad \mathsf{STO}[chan_{\mathsf{STO}} := \mathsf{m}]\ [\![\ \mathsf{cc}\ ]\!]\ \mathsf{Struct}_{\mathsf{Storage}}[\mathsf{si}, \mathsf{so} := \mathsf{k}, \mathsf{l}]\ [\![\ \mathsf{cc}]\!] \\
\qquad\qquad\qquad\qquad Struct_{\mathsf{Storage}}[\mathsf{si}, \mathsf{so} := \mathsf{m}, \mathsf{n}])[\mathsf{k}, \mathsf{n} := \mathsf{mi}, \mathsf{mo}] \setminus hchan
\end{array}
$$

O processo $Struct_{\mathsf{SysProd}}$ é composto pelo paralelismo de $Chart_{\mathsf{SysProd}}$, dos processos que representam suas portas e daqueles que representam suas subcápsulas, sincronizados nos canais associados às conexões internas de SysProd $(\mathsf{k}, \mathsf{m}, \mathsf{l}, \mathsf{n})$. No processo $Chart_{\mathsf{SysProd}}$, as portas pi e po são renomeadas nas conexões l e m, respectivamente. As portas pi, po, mi e mo de SysProd são representadas pelos processos $\mathsf{STI}[chan_{\mathsf{STI}} := \mathsf{k}]$, $\mathsf{STI}[chan_{\mathsf{STI}} := \mathsf{n}]$, $\mathsf{STO}[chan_{\mathsf{STO}} := \mathsf{l}]$ e $\mathsf{STO}[chan_{\mathsf{STO}} := \mathsf{m}]$, respectivamente; cada um deles formados pelo respectivo processo do protocolo relativo à porta e renomeados pelas conexões associadas a ela. As subcápsulas sin e son são representadas pelos processos $Struct_{\mathsf{Storage}}[\mathsf{si}, \mathsf{so} := \mathsf{k}, \mathsf{l}]$ e $Struct_{\mathsf{Storage}}[\mathsf{si}, \mathsf{so} := \mathsf{m}, \mathsf{n}]$ respectivamente; estes são formados pelo processo associado a cápsula Storage (da qual são instâncias), renomeada pelos respectivos canais de suas conexões no diagrama SysProd.

Devido a representarmos conexões entre portas através da sincronizações entre processos, necessitamos assumir que portas devem estar conectadas entre si no menor nível de estrutura que as contenha. Na realidade, esta suposição é uma boa prática de projeto pois evita indireções e reduz a complexidade do modelo. Na Figura 3.3, as instâncias de cápsulas b e c devem ser conectadas no menor nível de estrutura em $\mathsf{Str}_{\mathsf{M}}$ que os contêm, neste caso o diagrama de estrutura de A.

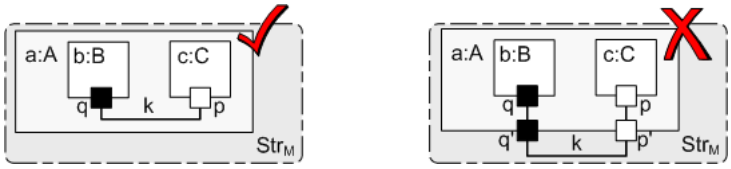
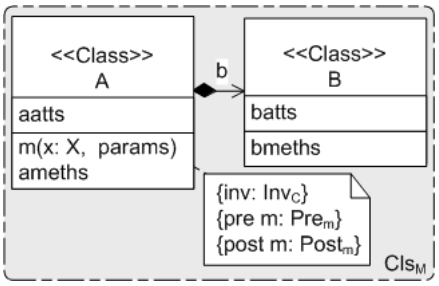


Figura 3.3. Exemplo de uma prática correta (a esquerda) e uma errada (a direita) da conexão de subcápsulas

Na Figura 3.3, o mapeamento das subcápsulas B e C do contexto à esquerda teria o formato $(Struct_{\mathsf{B}}[\mathsf{q} := \mathsf{k}])\ [\![\ \mathsf{k}\ ]\!]\ (\mathsf{Struct}_{\mathsf{C}}[\mathsf{p} := \mathsf{k}])$, enquanto o contexto do lado direito teria uma formato semelhante ao $(Struct_{\mathsf{B}}[\mathsf{q} := \mathsf{q}']\ [\![\ k\ ]\!]\ Struct_{\mathsf{C}}[\mathsf{p} := \mathsf{p}'])[\mathsf{q}, \mathsf{q}' := \mathsf{k}, \mathsf{k}]$. Como a função $[\mathsf{q}, \mathsf{q}' := \mathsf{k}, \mathsf{k}]$ não é injetiva (as duas portas são renomeadas em uma mesma porta), ela não distribui pelo operador de paralelismo (Ver Lei B.1, no Apêndice B) e difere o comportamento do mapeamento dos dois contextos (apesar de em UML-RT eles possuírem um mesmo comportamento).

Diferentemente de cápsulas e protocolos, classes são mapeadas em classes em *OhCircus*. Estas classes tem o mapeamentos de seus elementos semelhante ao mapeamentos de atributos e métodos de uma cápsula, definidos como parágrafos em Z.



$$
\begin{array}{l}
\textbf{class } \mathsf{A} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state } \mathsf{A}_{state} \mathrel{\widehat{=}} [\mathsf{b} : \mathcal{T}(\mathsf{B});\ \mathcal{TL}(\mathsf{aatts}) \mid \mathsf{Inv_A}] \\
\quad\ m \mathrel{\widehat{=}} [\Delta \mathsf{A}_{state};\ \mathsf{x} : \mathcal{T}(\mathsf{X}); \\
\qquad\qquad\qquad \mathcal{TL}(\mathsf{params}) \mid \mathsf{Pre_m} \wedge \mathsf{Post_m}] \\
\quad \mathcal{TL}(\mathsf{ameths}) \\
\textbf{end}
\end{array}
$$

Como assumimos que classes não possui um comportamento explícito associado, necessitamos apenas mapear a visão de uma classe em diagrama de classes em *OhCircus*. Na classe A acima, os compartimentos correspondem à listas de atributos e métodos. Então, b, m são aqueles que destacamos. O atributo b é mapeado para um atributo do estado de A com seu tipo correspondente em *OhCircus*, dado pela classe B; os outros atributos aatts são mapeados pela função $\mathcal{TL}()$, como explicado previamente. O invariante $\mathsf{Inv_A}$ se origina da nota em UML-RT no contexto à esquerda, e assume-se que este já esteja descrito em *OhCircus*. O método m() é mapeado para uma operação que pode modificar qualquer componente do estado, e cujos parâmetros são mapeados em atributos do esquema. Como o invariante Y, assume-se que a pré- e a pós-condição $\mathsf{Pre_m}$ e $\mathsf{Post_m}$ sejam escritas em *OhCircus*. Similarmente a função $\mathcal{TL}()$ mapeia os outros métodos ameths.

3.2.2 Mapeamento comportamental

Nosso mapeamento dos diagramas de estado de cápsulas e protocolos em *OhCircus* é baseado no trabalho reportado em [69], que apresenta a formalização de diagramas de estado em CSP. No entanto, é necessário estendê-lo para considerar ações de *OhCircus*, lidar com paralelismo (*And-States*) e estados compostos com múltiplos pontos iniciais.

Seja M uma máquina de estados e S_M o conjunto de estados de M. O conjunto de eventos de M é denotado por E_M, e suas ações e guardas booleanas por A_M e G_M, respectivamente. Além disto, SI_M denotará o conjunto de estados inicias, SCh_M o conjunto de estados de escolha, SS_M o conjunto de estados simples e SCo_M o conjunto de estados compostos de S_M.

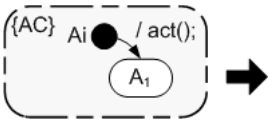
Como um diagrama de estados pode ser identificado como um estado composto que contém todos os outros, seu mapeamento de estados pode ser visto como o de um único estado. Assim, podemos indicar este mapeamento por uma função $\mathcal{H}$ que leva um estado a sua representação em *OhCircus*.

$$\mathcal{H} : S_M \rightarrow CSPAction$$

Nós assumimos que uma ação em A_M é expressa por uma chamada de métodos, e, portanto, não precisa ser traduzida. Como outros predicados, guardas em A_M são escritas utilizando a sintaxe de *OhCircus*. Como portas possuem um canal associado em *OhCircus* com o seu nome, e sinais são expressos pelos valores do tipo destes canais, um sinal e de uma porta p pode ser diretamente escrito pelo casamento de padrão de eventos em *OhCircus* identificado, correspondente ao sinal e.

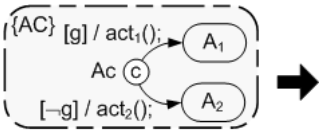
Cada padrão aplicado à função de mapeamento de estados $\mathcal{H}(\cdot)$ gera um equação distinta. No lado esquerdo de cada equação, ilustramos o padrão de aplicação como um *template* para um tipo de estado específico.

Nossa primeira tradução corresponde ao estado inicial. Este possui somente uma transição de saída, e nenhuma ação de entrada ou saída. Daí, seja Ai um estado inicial, com $\mathsf{Ai} \in SI_M$, $\mathsf{act}()$ a ação de sua transição de saída, e $\mathsf{A_1}$ o destino desta transição, então:



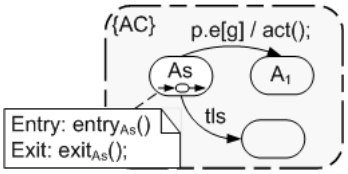
$$\mathcal{H}(\mathsf{Ai}) = \mathsf{act}();\ \mathcal{H}(\mathsf{A_1})$$

Para um estado de escolha, a tradução é como segue. Seja Ac um estado de escolha, com $\mathsf{Ac} \in SCh_M$, que tem somente duas transições e uma guarda g (estas transições não possuem evento e são disparadas de acordo com a avaliação de g), então:



$$\mathcal{H}(\mathsf{Ac}) = (\mathsf{g}\ \&\ \mathsf{act_1}();\ \mathcal{H}(\mathsf{A_1})) \sqcap (\neg\ \mathsf{g}\ \&\ \mathsf{act_2}();\ \mathcal{H}(\mathsf{A_2}))$$

Em relação a um estado simples, sua tradução corresponde à ação de processo abaixo. Seja As um estado simples, com $\mathsf{As} \in SS_M$, AC o estado composto que o contém, com $\mathsf{AC} \in SCO_M$, tls sua lista de transições de saída e $\mathsf{p.e[g]/act}()$ uma das transição destacada desta lista, então:

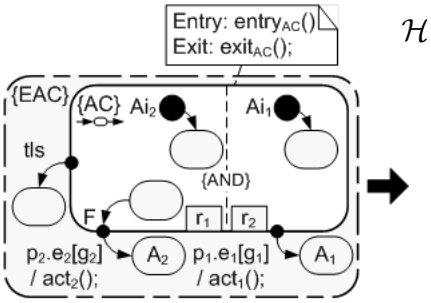


$$\begin{aligned}\mathcal{H}(\mathsf{As}) = \mathsf{entry_{As}}();\ &((\mathsf{g}\ \&\ \mathsf{p.e} \rightarrow \mathsf{exit_{As}}();\ \mathsf{act}();\ \mathcal{H}(\mathsf{A_1}))\\ &\Box\ ((\mathit{notevent}_{\mathsf{AC}} \rightarrow \mathsf{exit_{As}}();\ \mathit{exitevent}_{\mathsf{AC}} \rightarrow \mathit{SKIP})\\ &\Box\ \mathcal{TL}(\mathsf{tls})))\end{aligned}$$

Quando o estado As é ativado, ele executa a ação de entrada $\mathsf{entry_{As}}()$ e espera por um novo sinal oriundo do ambiente (a escolha externa captura esta decisão). Destacamos aqui a transição disparada pelo sinal e da porta p, dentre a lista de transições tls. O evento $\mathsf{p.e}$ é executado somente quando sua guarda g for satisfeita. Caso isto ocorra, a ação de saída $\mathsf{exit_{As}}()$ e a ação $\mathsf{act}()$ associada à transição disparada, são executadas, e o estado $\mathsf{A_1}$ torna-se ativo. Além de esperar por sinais que disparem suas transições de saída, um estado simples pode ficar inativo pelo disparo de uma transição de saída (de grupo) do estado composto que o contém; neste caso, um evento de notificação $\mathit{notevent}_{\mathsf{AC}}$, onde AC representa o estado composto, é disparado para garantir que a ação de saída de As seja executada antes da ação de saída de AC. Após a execução da ação de saída de As, um evento $\mathit{exitevent}_{\mathsf{AC}}$ é enviado, permitindo que a ação de saída do estado AC seja executada.

Para mapear estados compostos, é necessário apenas a apresentação da tradução de um estado composto do tipo *And-State*. Como suas regiões concorrentes são dadas pela composição seqüencial de estados (tal qual um *Or-State*), a tradução de um estado composto do tipo *Or-State* pode ser visto por um *And-State* com uma única região. Seja

AC e EAC estados compostos pertencentes à SCO_M, onde EAC é o estado composta que contém AC e AC um *And-State* com as seguintes regiões r_1 e r_2, Ai_1 e $\mathsf{Ai}_2 \in SI_M$ os estados iniciais das regiões de AC, A_1 e $\mathsf{A}_2 \in S_M$ estados contido em EAC, e $[R]$ uma função de renomeação que substitui um canal associado a uma porta à um canal associado a conexão que esta porta está conectada, então:



$$\begin{aligned}
\mathcal{H}(\mathsf{AC}) = {} & \mathsf{entry}_{\mathsf{AC}}();\ (\mathcal{H}(\mathsf{Ai}_1)[R] \llbracket \mathsf{C}_{\mathsf{state1}} \mid cc \mid \mathsf{C}_{\mathsf{state2}} \rrbracket \\
& \mathcal{H}(\mathsf{Ai}_2)[R] \llbracket \mathsf{C}_{\mathsf{state1}} \cup \mathsf{C}_{\mathsf{state2}} \mid cc \mid \mathsf{C}_{\mathsf{state3}} \rrbracket \\
& \Box\ (\mathsf{g}_1\ \&\ \mathsf{p}_1.\mathsf{e}_1 \rightarrow notevent_{\mathsf{AC}} \rightarrow exitevent_{\mathsf{AC}} \rightarrow \mathsf{exit}_{\mathsf{AC}}(); \\
& \quad \mathsf{act}_1();\ \mathcal{H}(\mathsf{A}_1)) \\
& \Box\ (\mathcal{TL}(\mathsf{tls})) \\
& \Box\ (\mathsf{g}_2\ \&\ \mathsf{p}_2.\mathsf{e}_2 \rightarrow exitevent_{\mathsf{AC}} \rightarrow \mathsf{exit}_{\mathsf{AC}}();\ \mathsf{act}_2();\ \mathcal{H}(\mathsf{A}_2)) \\
& \Box\ (notevent_{\mathsf{EAC}} \rightarrow notevent_{\mathsf{AC}} \rightarrow exitevent_{\mathsf{AC}} \rightarrow \mathsf{exit}_{\mathsf{AC}}(); \\
& \quad exitevent_{\mathsf{EAC}} \rightarrow SKIP))
\end{aligned}$$

Quando o *And-State* AC é ativado, ele executa a ação de entrada $\mathsf{entry}_{\mathsf{AC}}()$, e depois o estado inicial de cada uma de suas regiões. Para que as regiões r_1 e r_2 sincronizem adequadamente suas ações, é necessário renomear os canais associados às portas conectadas em $\mathsf{Str}_{\mathsf{M}}$ pelos canais em cc correspondentes a suas conexões, através da função R. Assim sinais de saída de uma região podem ser sincronizados à sinais de saída de outra. Assumimos que as ações dos estados r_1 e r_2 podem modificar somente valores de variáveis em $\mathsf{C}_{\mathsf{state1}}$ e $\mathsf{C}_{\mathsf{state2}}$, respectivamente, enquanto a ação de saída e as associadas às transições de saída de $\mathsf{state2}$ modificam somente valores de variáveis em $\mathsf{state3}$. O estado da cápsula contém a união de $\mathsf{C}_{\mathsf{state1}}$, $\mathsf{C}_{\mathsf{state2}}$ e $\mathsf{C}_{\mathsf{state3}}$. E a qualquer momento, o estado AC pode receber um evento externo e interromper os subestados de $\mathsf{r1}_1$ e r_2, deixando-os inativos.

Um estado composto pode ficar inativo de três formas: através de uma transição de saída com origem no próprio (transições de grupo), com origem em um de seus subestados, ou com origem em um de seus superestados. Nos dois primeiros casos, o comportamento associado à transição é descrito em AC, enquanto no último este comportamento é descrito em EAC. E em todos eles, eventos de notificação são criados para garantir que a ordem de execução das ações de saída dos estados seja realizada do estado mais interno ao mais externo, que contém os demais. A lista de transições localizadas no dois primeiros casos é representada aqui por tls, onde foram destacadas aquelas disparadas pelos sinais $\mathsf{p}_1.\mathsf{e}_1$ e $\mathsf{p}_1.\mathsf{e}_1$, respectivamente, para o primeiro e o segundo caso.

Quando o estado AC recebe o evento $\mathsf{p}_1.\mathsf{e}_1$ e avalia a guarda g como verdadeira, um evento $notevent_{\mathsf{AC}}$ é enviado para solicitar que todos os subestado ativos executem suas ações de saída. Quando uma notificação $exitevent_{\mathsf{AC}}$ chega, a ação de saída de AC é finalmente executada, seguida pela ação act_1 da transição. Após este conjunto de ações, o estado A_1 torna-se então ativo. Transições que emanam de um subestado, através de um ponto de junção de AC se comportam de forma similar. Este tipo de junção é mapeado como um subestado transiente final de AC, e, portanto, não possui qualquer ação associada. Assim, AC não necessita esperar pela execução de qualquer ação de saída, e não precisa lidar com o evento $notevent_{\mathsf{AC}}$. No entanto o evento $exitevent_{\mathsf{AC}}$ é necessário

para sincronizar com este pseudo-estado de saída de AC. Para um ponto de junção F, sua tradução corresponde a $\mathcal{H}(F) = exitevent_{\mathsf{AC}} \rightarrow SKIP$. As outras transições de saída de AC em tls podem ser similarmente traduzidas pela função $\mathcal{TL}(\mathsf{tls})$, que mapeia a lista tls de transições da mesma similar à transições disparadas pelos eventos $\mathsf{p}_1.\mathsf{e}_1$ e $\mathsf{p}_1.\mathsf{e}_1$, que destacamos aqui.

Assim como estados simples, estados compostos podem estar inclusos dentro de outros estados compostos; no contexto anterior, AC é um subestado de EAC. Nesta situação, AC pode a qualquer momento receber uma notificação $notevent_{\mathsf{EAC}}$ indicando a ocorrência de uma transição de saída em EAC. Quando este evento é recebido, AC precisa enviar o evento $notevent_{\mathsf{AC}}$ para seus subestados ativos a fim de que eles executem suas ações de saída antes de sua própria ação ($\mathsf{exit}_{\mathsf{AC}}$). Quando AC receber $exitevent_{\mathsf{AC}}$, a ação $\mathsf{exit}_{\mathsf{AC}}$ é executada, e então $exitevent_{\mathsf{EAC}}$ é enviado para que EAC execute sua própria ação de saída.

Em nosso exemplo da Figura 2.2, o mapeamento do comportamento do protocolo STO será definido por um processo homônimo, cuja ação principal é dada pelo mapeamento do estado S_0, que contém todos os outros.

$$
\begin{array}{l}
\textbf{process}\ \mathsf{STO} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \mathsf{Sb} \mathrel{\widehat{=}} chan_{\mathsf{STO}}.\mathsf{req} \rightarrow \mathsf{Sc} \\
\quad \mathsf{Sc} \mathrel{\widehat{=}} chan_{\mathsf{STO}}.\mathsf{output} \rightarrow \mathsf{Sa} \\
\quad \bullet\ \mathsf{Sb} \\
\textbf{end}
\end{array}
$$

Na máquina de estados de um protocolo, não encontramos ações ou guardas, sendo representado apenas pela ordem dos eventos no canal associados a ele, que em STO é representado por $chan_{\mathsf{STO}}$. Como podemos notar, à exceção do estado inicial, cada um dos estados de STO é representado por uma parágrafo no corpo do processo (Sb e Sc). A ação principal de STO ($\bullet$ Sb) é igual ao comportamento correspondente ao mapeamento do estado inicial STO, cuja transição tem como destino o estado Sb.

Diferentemente de protocolos, as máquinas de estados de cápsulas possuem ações e guardas, como é visto no mapeamento associado à ação principal de $Chart_{\mathsf{SysProd}}$ (abaixo) e $Chart_{\mathsf{Storage}}$ (ver Seção 3.1.1) .

$$
\begin{array}{l}
\ldots \\
\mathsf{Sp} \mathrel{\widehat{=}} \mathsf{pi?output.x} \rightarrow \mathsf{p} = \mathsf{x};\ \mathsf{process}();\ \mathsf{po!input.x};\ \mathsf{Sp} \\
\bullet\ \mathsf{pi.req} \rightarrow \mathsf{Sp}
\end{array}
$$

Na ação principal de $Chart_{\mathsf{SysProd}}$, pi.req correspondente à ação da transição inicial de SysProd, que possui como destino o estado Sp. Este estado é representado por um parágrafo de mesmo nome e oferece um único evento pi?output.x, o qual dispara a única transição de Sp em SysProd. A ação executada após este evento é a mesma descrita na Figura 2.2, que utiliza uma notação semelhante para a chamada de métodos e o envio de sinais.

Como desejamos neste trabalho um mapeamento puramente sintático, não lidamos com história em estados. Já que para isto deveríamos atribuir variáveis a todos estados

compostos do modelo, criando uma representação semântica de cada um deles. Esta variável seria utilizada para guardar a última configuração de subestados ativos antes que uma transição de saída fosse disparada. Assim, quando um estado composto se tornasse ativo novamente, essa configuração seria restaurada.

3.3 CONCLUSÕES

Através do mapeamento de UML-RT para *OhCircus*, possibilitamos a atribuição de uma semântica a UML-RT através de sua integração com uma notação formal. Como conseqüência, podemos provar propriedades de um modelo em UML-RT através de demonstrações de sua especificação em *OhCircus*, e, portanto, a preservação de comportamento de um modelo durante a aplicação de leis transformação em UML-RT.

CAPÍTULO 4

LEIS DE TRANSFORMAÇÃO PARA UML-RT

Neste capítulo, focamos em leis de transformação que expressam pequenas reestruturações de um modelo em UML-RT, sem que haja alteração do comportamento do sistema modelado. Particularmente, concentramo-nos nos elementos adicionais que UML-RT adiciona a UML (cápsulas, protocolos e portas) e em seus relacionamentos, complementando assim outros trabalhos que lidam somente com elementos de UML (como classes e interfaces) [41, 30, 38]. Observe que como cápsulas não compartilham classes, e que como a comunicação entre cápsulas e classes é através da chamada de métodos, transformações num modelo de classes, sem mudar suas interfaces com as cápsulas a que dependem, são livremente permitidas.

Focamos, principalmente, em leis que envolvem as visões relacionadas à declaração de elementos do modelo e à configuração de sua arquitetura, representados pelos diagramas de classe e estrutura. Além disto, algumas leis relacionadas à declaração do comportamento de elementos (leis para diagramas de estado) são exibidas, porém em uma quantidade necessária para descrevermos a abrangência de nossas leis quanto as duas visões citadas; esta abrangência será discutida no Capítulo 5.

Algumas transformações básicas (criação, remoção ou alteração de elementos) são realizadas, usualmente, em elementos isolados do modelo. Outras transformações são utilizadas para substituir elementos. Assim, noções de equivalência, extraídas da semântica de *OhCircus*, permitem que as leis lidem simultaneamente com os aspectos estáticos e dinâmicos do modelo, preservando o comportamento observado, através de mudanças em seus diagramas de classe, estado e estrutura.

A razão para lidarmos com diagramas de estrutura é porque eles representam interações entre cápsulas que não são expressas por outros diagramas; por isto, eles devem ser considerados para que o comportamento global do sistema gerado seja preservado.

Inicialmente, apresentamos algumas leis básicas (Seção 4.1) que são usadas em seguida para justificar leis mais elaboradas (Seção 4.2). A partir da composição destas leis básicas, é possível criar transformações utilizadas usualmente durante a análise e projeto do sistema, como *refactorings*. Apesar de nosso propósito maior ser a sistematização do desenvolvimento em UML-RT, utilizando transformações de modelo, a partir do mapeamento de modelos em UML-RT para *OhCircus* (Capítulo 3) obtemos o benefício de utilizar uma semântica formal, bem como leis de refinamento de *OhCircus*, para provar a corretude de algumas das leis de transformação apresentadas neste capítulo (Seção 4.3).

4.1 LEIS BÁSICAS

Cada lei é definida por uma equação, onde os termos descrevem o contexto dos elementos do modelo antes e após a aplicação da lei. A aplicação de uma lei pode ser feita

em qualquer um dos sentidos (partindo-se do termo da esquerda ou da direita), porém obedecendo-se as condições impostas para aplicação da lei no sentido desejado.

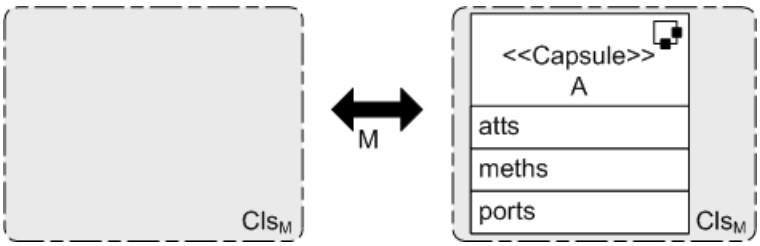
A relação entre os dois termos é representada na lei por uma relação de equivalência de modelos (uma seta bidirecional) com um subscrito M, que representa o contexto do restante do modelo. A noção de equivalência utilizada é baseada na semântica de *OhCircus*, que é definida na *Unifying Theories od Programming* [47]. Na realidade, apenas um subconjunto de *OhCircus* (*Circus*) tem uma semântica completamente definida. Apesar das leis propostas não modificarem M, algumas das leis impõem condições sobre algumas visões deste modelo, que por esta razão são explicitamente apresentadas durante a lei: $\mathsf{Cls_M}$ representa as declarações de cápsulas, protocolos e classes em M, composto por seu conjunto de diagramas de classe; $\mathsf{Str_M}$ simboliza a estrutura (ou a configuração da arquitetura) de M, expresso pelos diagramas de estrutura de suas cápsulas. Um terceira visão do modelo, $\mathsf{Sta_M}$, representa a declaração do comportamento, isolado, de todas as cápsulas e protocolos do sistema, expresso por suas respectivas máquinas de estado. Apesar de, dinamicamente, estas máquinas de estados se comunicarem, de acordo com as conexões expressas em $\mathsf{Str_M}$, elas são independentes entre si. Por esta razão, usualmente, o diagrama de estados de cada um destes elementos será explicitamente mostrado nos termos da leis, ao invés de recorrermos a $\mathsf{Sta_M}$; para um elemento A, indicaremos seu diagrama de estados como $\mathsf{S_0}$ de A.

Diagramas e anotações descrevendo propriedades (invariantes, pré- e pós-condições) do modelo são apresentadas nas leis quando necessário. Em alguns casos, existe o interesse que somente uma parte (auto-contida) destes diagramas seja mostrada; neste caso, uma linha pontilhada na borda dos diagrama é utilizada. Adicionalmente, relacionamentos entre os elementos apresentados no diagrama podem existir, mesmo que não explicitados.

Leis relacionadas à declaração de elementos, usualmente, são responsáveis por criar novos elementos em um modelo UML-RT. Em geral, elas possuem como condição para a introdução de um elemento (aplicação da esquerda para a direita da lei) que este não possua o mesmo nome de outro elemento já utilizado no modelo, e para a remoção de um elemento (aplicação da direita para esquerda) que o elemento não seja utilizado pelo restante do modelo.

A primeira lei estabelece quando é possível introduzir uma nova cápsula ao modelo. Observe que apesar do contexto no lado esquerdo da lei estar vazio, o M subscrito fixa o contexto para aplicação da lei.

Lei 4.1 *Declarar Cápsula*



Condições:

($\rightarrow$) $\mathsf{Cls_M}$ *não possui a declaração de nenhum elemento, no mesmo pacote, chamado* A.

(←) *Nenhuma cápsula em* M *tem uma relação com a cápsula* A *em qualquer diagrama.*

Usamos o símbolo (→) antes de uma condição para indicar que ela é requerida somente para aplicações da lei da esquerda para a direita. Similarmente, utilizamos (←) para indicar que a condição é necessária somente para aplicações da lei da direita para a esquerda, e usamos (↔) para indica que a condição é necessária em ambas as direções.

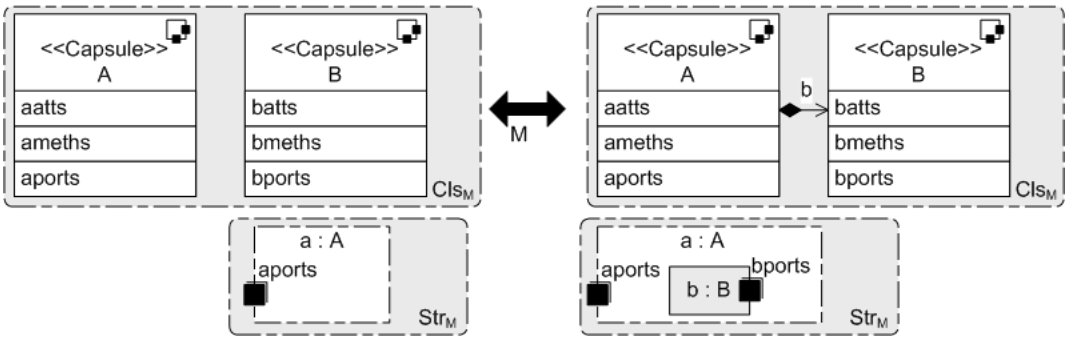
A condição para remover a cápsula A (aplicação da lei da direita para esquerda) é que nenhuma outra estende, tem uma associação em $\mathsf{Cls_M}$ ou é conectada com ela em $\mathsf{Str_M}$. Em relação à aplicação da lei da esquerda para a direita, já que em UML (e, por sua vez, em UML-RT) não é permitido que dois elementos possuam o mesmo nome em um mesmo pacote (contexto local), existe uma condição expressando que a declaração de nova cápsula em $\mathsf{Str_M}$ deve possuir sempre um novo nome no modelo. Assumimos que atts, meths e ports representam, respectivamente, os conjuntos de atributos, métodos e portas da cápsula.

Nas duas direções de aplicação da lei, a cápsula A não está associada a nenhuma outra no modelo, portanto, a apresentação do diagrama de estrutura e o de estados é irrelevante. Na realidade, estes diagramas podem possuir qualquer conteúdo, que obedeça às condições da lei.

Apesar de sua simplicidade, esta lei é bastante importante, já que uma cápsula só pode fazer parte da arquitetura do sistema modelado, caso ela tenha sido declarada previamente. Semelhante à Lei 4.1, as leis C.1 e C.2 apresentadas no Apêndice C capturam a introdução de protocolos e classes em $\mathsf{Cls_M}$.

A próxima lei estabelece quando podemos adicionar ou remover uma associação entre cápsulas. Assumimos que, quando uma cápsula A é criada, é criado também seu diagrama de estrutura; este diagrama contém as instâncias de todas as cápsulas com as quais A possui uma associação.

Lei 4.2 *Introduzir Associação Cápsula-Cápsula*



Condições:

(→) *Não existe outra instância de cápsula chamada* b *no diagrama de estrutura de* A.

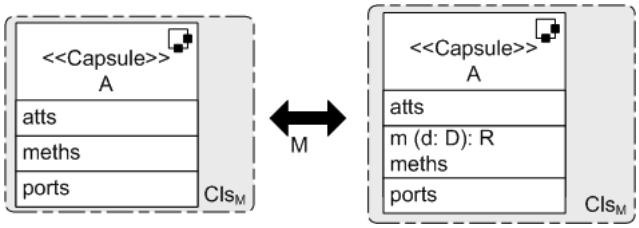
(←) *A instância* b *da cápsula* B *não está conectada a nenhuma outra na estrutura de* A, *inclusive à instância* a *que a contém.*

Na Lei 4.2, a aplicação da esquerda para a direita gera como resultado uma associação entre as cápsulas A e B. Como conseqüência, uma nova instância de B é criada no diagrama de estrutura de A, em qualquer contexto de $\mathsf{Str_M}$ onde uma instância a de A apareça. Nesta lei, o conjunto de portas aports e bports são representados por portas múltiplas em $\mathsf{Str_M}$; na realidade, estes conjuntos podem possuir uma ou mais portas distintas com diferentes cardinalidades e tipos. Pode ainda haver conexões envolvendo as portas de A e B, não explicitado na lei.

Relativo à semântica de um cápsula isolada em $\mathsf{Str_M}$, esta é vista como processo em *OhCircus* que não comunica com nenhum outro na especificação e que possivelmente estará sempre bloqueada. Como o comportamento de cápsulas é descrito por máquinas de estado que não possuem um estado final, um sistema em UML-RT é representado em *OhCircus* por um processo que nunca termina, e que, por esta razão, não terá seu comportamento alterado pela inserção de uma nova cápsula isolada do restante do modelo.

A próxima lei estabelece quando é permitido adicionar ou remover métodos em uma cápsula.

Lei 4.3 *Introduzir Método*



Condições:

(→) *Não existe um método chamado* m *em* A.

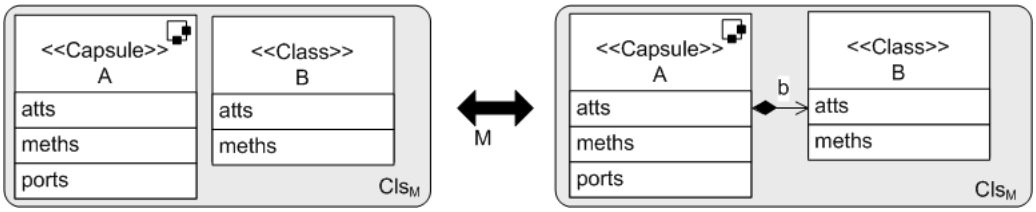
(←) *O método* m *não é utilizado por outro método em* meths, *nem tampouco no diagrama de estados de* A, *ou em um predicado de* A.

A Lei 4.3 é similar ao refatoramento para adicionar métodos em classe [33], possuindo restrições quanto ao nome do método e quanto ao seu uso. Da direita para a esquerda, o método m não pode ser utilizado por outro método, ação do diagrama de estados ou predicado (invariante, pré- ou pós-condição) de A. Além disto, o padrão da lei possui como condições implícitas que os tipos de dados D e R já existam em M. Note que eliminar um método de uma cápsula é bem mais simples do que de uma classe, pois métodos em uma cápsula são inerentemente privados.

A seguir, é apresentado um caso específico de uma lei que introduz um atributo em uma cápsula, como conseqüência da criação de uma associação da cápsula com uma classe.

Como outras leis que introduzem novos elementos, a Lei 4.4 possui em suas condições de aplicação restrições quanto ao nome e ao uso do elemento introduzido (neste caso a associação b). Assumimos que a introdução de uma associação b entre uma cápsula A e uma classe B introduz implicitamente um atributo b em A, por esta razão a existência de uma condição que indica que não pode haver outro atributo chamados b em A.

Lei 4.4 *Introduzir Associação Cápsula-Classe*



Condições:

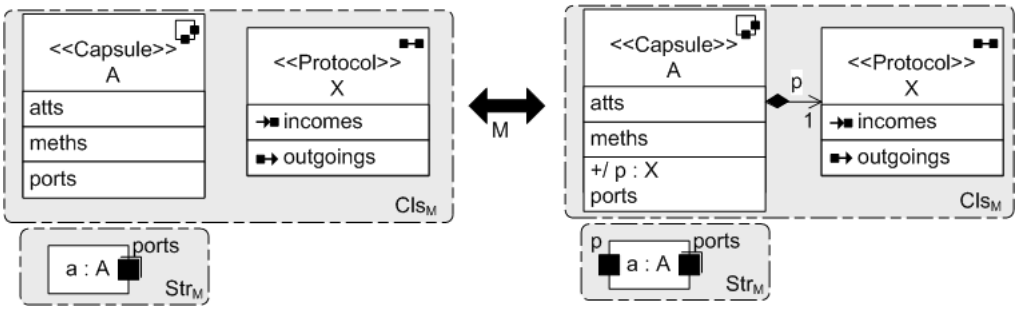
(→) *Não há um atributo chamado* b *em* A.

(←) *O atributo* b *não é utilizado por nenhum método, diagrama de estados ou predicado de* A.

Apesar da Lei 4.4 ser específica para classes, uma lei análoga para a introdução de atributos (possivelmente de tipos primitivos) pode ser facilmente deduzida a partir dela.

A próxima lei estabelece quando podemos adicionar ou remover uma associação entre um protocolo e uma cápsula. Como conseqüência da introdução desta associação, é criada uma instância deste protocolo (porta) na cápsula.

Lei 4.5 *Introduzir Associação Cápsula-Protocolo*



Condições:

(→) *Não existe outra porta chamada* p *na cápsula* A.

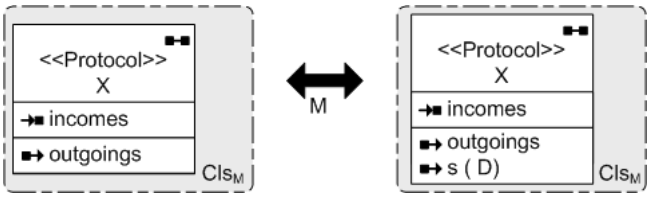
(←) *A porta* p *não é utilizada no diagrama de estados de* A*; não existe conexão ligada a* p *em* $\mathsf{Str_M}$.

Em uma visão geral, a Lei 4.5 é similar ao *refactoring* "Adicionar Atributo" [33]; portas, apesar de sua declaração em um compartimento diferente, podem ser vistas como atributos de uma cápsula. Por esta razão, a lei possui como condição que não pode haver duas ou mais portas com um mesmo nome em uma cápsula. Para se remover uma porta é necessário que esta não seja usada no diagrama de estados (assumimos que ações para o envio de sinais não podem estar localizados em métodos da cápsula); nem tampouco a porta pode ser usada na conexão com outras cápsulas em $\mathsf{Str_M}$. Note que portas são criadas como conseqüência da associação entre cápsulas e protocolos, sendo visualmente vista tanto no diagrama de classes quanto no diagrama de estrutura; por simplicidade, o conjunto de portas ports é mostrado no diagrama de estrutura como uma porta múltipla.

Devido a condição da lei de que a porta p nunca é utilizada no diagrama de estados de A, então a apresentação deste diagrama é desnecessária.

A próxima lei estabelece quando é permitido adicionar ou remover sinais de saída em um protocolo.

Lei 4.6 *Introduzir Sinal de Saída*



Condições:

(→) *Não existe sinal chamado* s *no protocolo* X.

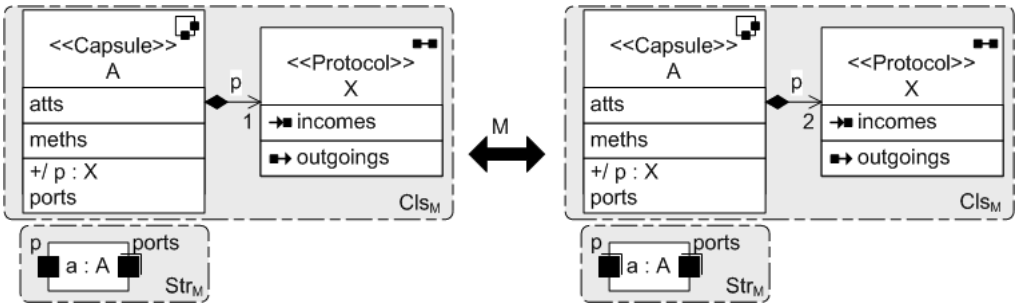
(←) *Nenhum diagrama de estados em* $\mathsf{Sta_M}$ *usa o sinal* s.

Semelhante à Lei 4.5, a Lei 4.6 possui uma restrição quanto ao uso do nome do novo elemento (sinal) em seu contexto local (protocolo) e quanto ao uso deste elemento nos diagramas de estado do modelo. A primeira condição indica que não existe previamente nenhum sinal s de entrada ou saída em X no modelo do lado esquerdo. A segunda condição indica que o sinal s não é utilizado nos diagramas de estado do protocolo X ou de nenhuma cápsula com uma porta de tipo X. Note que apesar de s possuir um parâmetro do tipo de dado D, este pode ser inclusive void. Em geral, há uma condição implícita de que D deve ser um tipo válido em $\mathsf{Cls_M}$.

Apesar do sinal s estar localizado no mesmo compartimento de sinais de saída de X, uma lei análoga para um sinal de entrada possui as mesmas condições que a Lei 4.6. A única diferença entre estas leis é o compartimento de localização do sinal, o qual indica o seu sentido. A lei para introduzir um sinal de entrada está no Apêndice C (na Lei C.4).

A próxima lei captura o incremento da multiplicidade da associação entre um protocolo e uma cápsula e, conseqüentemente, o respectivo incremento da multiplicidade da porta que representa esta relação.

Lei 4.7 *Incrementar Multiplicidade da Associação Cápsula-Protocolo*



Condições:

(←) *O número de conexões em* $\mathsf{Str_M}$ *envolvendo a porta* p *é inferior à sua multiplicidade.*

A condição relativa à quantidade de conexões envolvendo uma porta é uma restrição da própria linguagem UML-RT. A Lei 4.7, apesar de sua simplicidade, é muito importante para sincronização de cápsulas em diagramas de estrutura (Lei 4.8).

A próxima lei estabelece quando é permitido se conectar duas cápsulas. Apesar de suas similaridades com outras leis, esta lei afeta (exclusivamente) o diagrama de estrutura do modelo.

Lei 4.8 *Introduzir Conexão*

Condições:

(↔) *A seqüência de sinais aceitos por* b *e* c *é inalterada;* r *e* s *devem ser instâncias de mesmo um protocolo.*

(→) *A quantidade de conexões ligadas à* r *e* s *é inferior a multiplicidade destas porta.*

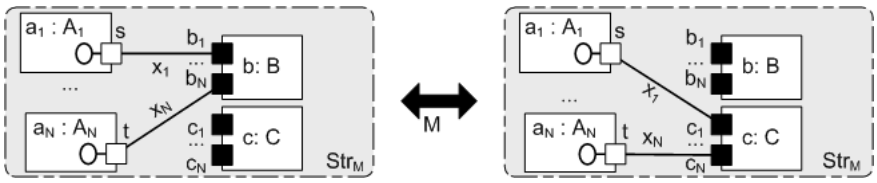
A Lei 4.8 indica que duas cápsulas somente podem ser conectadas se o seu comportamento for inalterado: a seqüência de sinais aceitos por ambas as cápsulas (sinais que disparam suas transições), através das portas r e s, não deve ser mudada. A linha pontilhada em $\mathsf{Str_M}$ indica que nada é dito quanto às conexões das portas r e s com o restante do modelo, ambas podendo, portanto, possuir conexões com outras cápsulas em $\mathsf{Str_M}$. Note a condição quanto à seqüencia de sinais através de b e c é uma condição dinâmica, e que não pode ser verificada sintaticamente. Para a instância de cápsula a, temos que verifica se todos os sinais provenientes da porta s de b, que sejam aceitos por a, sejam sincronizados com outra cápsula c' no modelo, de forma que o paralelismo de c' com b sincronizada em r é equivalente a c'; o mesmo acontece para instância de cápsula b e os sinais provenientes de r.

Em um caso mais simples desta lei, não existem outras portas conectadas a r ou s. Neste caso, para que o comportamento de A ou B não seja alterado, a máquina de estados destas cápsulas não devem possuir transições disparadas pelos novos sinais provenientes desta conexão. Uma condição menos restritiva, que ainda conserva as condições de aplicação da Lei 4.8, é que pelo menos uma das portas r ou s não é utilizada em sua cápsula; isto é, não é utilizada pelo diagrama de estado de sua respectiva cápsula.

No caso mais complexo, estas portas estão conectadas a porta de outras cápsulas do modelo; o que acarretaria um condição implícita, no lado esquerdo, de que a multiplicidade desta portas comportassem a nova conexão c. Neste caso, para que a seqüência de sinais recebidos fosse inalterada, todos novos sinais que trafeguem por c devem esta sincronizados a outros sinais enviados ou recebidos nessas outras cápsulas do modelo. Desta maneiras, após a aplicação da esquerda para a direita, as cápsulas continuam a aceitar e enviar os mesmos sinais que outrora, incluindo o valor de seus dados, sem que isto gere uma mudança na comunicação ou *deadlock*.

A próxima lei indica quando uma cápsula pode substituir outra na arquitetura do sistema, afetando, exclusivamente, o diagrama de estrutura do modelo.

Lei 4.9 *Substituição de cápsula*



Condições:

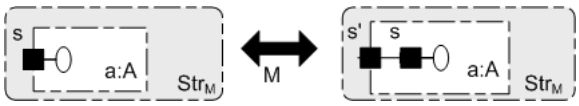
(↔) B *e* C *possuem as assinaturas dos conjuntos de portas* $\mathsf{b}_1, .., \mathsf{b}_N$ *e* $\mathsf{c}_1, .., \mathsf{c}_N$ *e o seus comportamentos, quando visto em relação este conjunto de portas, equivalentes.*

A Lei 4.9 possui como condição de aplicação que as cápsulas B e C possuam compatibilidade de assinatura e comportamento. A compatibilidade de assinatura indica que existe uma relação de bijeção entre o conjunto de portas de cada uma das cápsulas, isto é, cada porta de B possui uma porta correspondente, distinta e do mesmo protocolo em C, e vice e versa. A compatibilidade de comportamento indica que instâncias de B e C possuem o mesmo comportamento observado e que a substituição de uma pelo outra no sistema não poderá ser notada. A verificação desta compatibilidade é claramente uma condição dinâmica, mas pode ser simplificada, verificando-se uma região da máquina de estados de C que seja equivalente a uma região da máquina de estado de B. A equivalência das máquinas de estados é comprovada pela conservação da seqüência dos sinais aceitos e enviados por cada um das portas de uma cápsula e sua correspondente na outra cápsula; um trabalho que lida com a equivalência de máquinas de estado pode ser encontrado em [34].

Apesar desta lei ser bastante semelhante à Lei 4.8, ela não pode ser justificada a partir de diversas aplicações da Lei 4.8 sem que não introduzamos *deadlock* ao modelo. A Lei 4.9 só poderia ser derivada a partir da Lei 4.8, caso cada um das portas da cápsula que será substituída estivesse associada a uma região da máquina de estados da cápsula que não utilizasse as outras portas da cápsula. Um exemplo desta configuração pode ser encontro na Lei 4.20.

Na lei a seguir, é possível verificar sob que condições uma *relay port* pode ser criado. Este tipo de porta serve apenas como uma ponte para conexões de fora da estrutura de uma cápsula com portas protegidas dentro desta estrutura.

Lei 4.10 *Criar Relay Port*



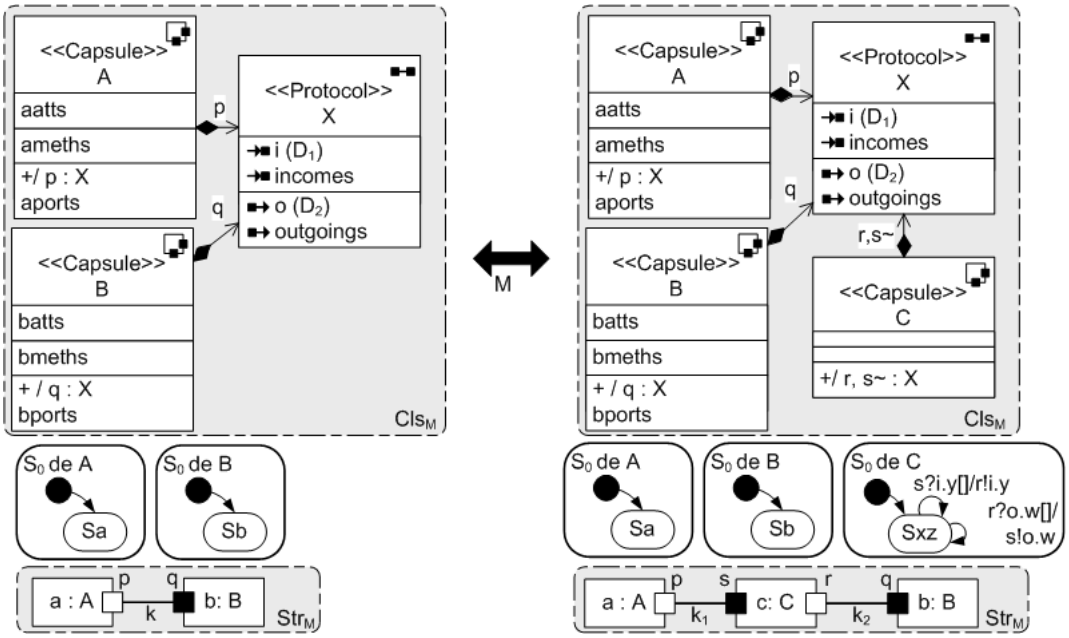
Condições:

(↔) *O protocolo associado à porta* p *possui uma máquina de estados determinística.*

Para que a inserção da nova porta na comunicação de a não altere o comportamento do sistema, devemos garantir que o diagrama de estados do protocolo associado à porta inserida é determinística.

A próxima lei expressa como uma instância de uma cápsula pode ser inserida para intermediar a comunicação de outras duas cápsulas.

Lei 4.11 *Inserir Cápsula para Intermediar Comunicação*



Condições:

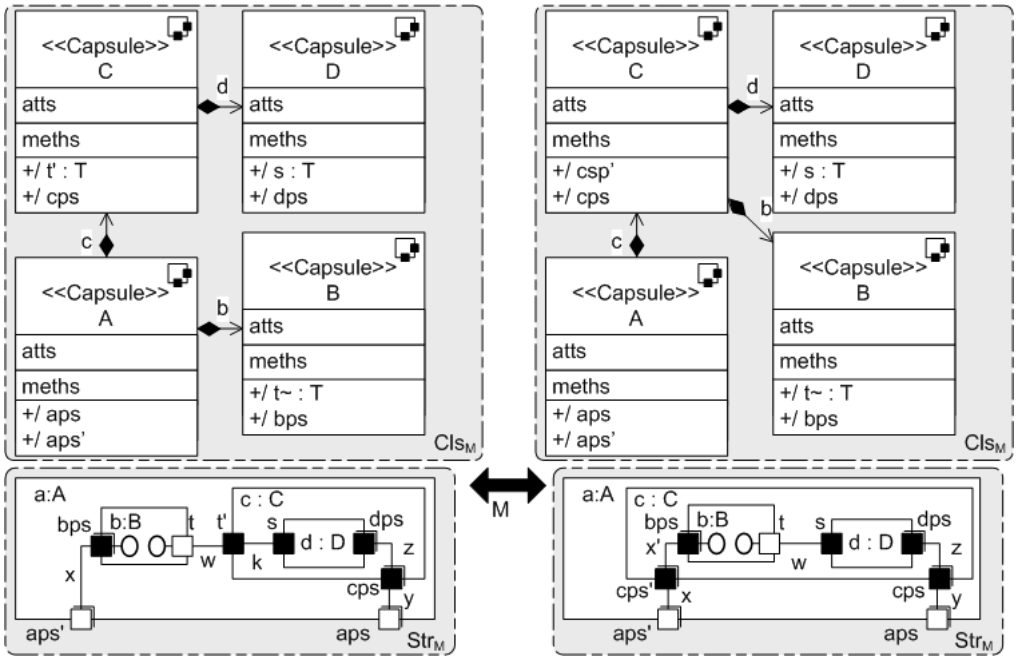
(↔) *A seqüência de sinais transmitidos pela conexão* k *é a mesma que a transmitida pelas conexões* k_1 *e* k_1.

No lado direito da Lei 4.11, a cápsula C aceita e envia os mesmo sinais que poderiam ser transmitidos através de k. Esta é uma condição dinâmica, e indica que a máquina de estado de XC não pode ser mais restritiva que a máquina de estados de X. Em sua forma mais geral, todos os sinais de entrada do protocolo X são aceitos por r e reenviados a partir de s; similarmente, os sinais de saída de X são aceitos por s e reenviados a partir de r. Destacamos aqui, somente os sinais i e o de X, enquanto outras transições internas de Sxz são responsáveis pelos demais sinas do protocolo para que a condição seja satisfeita.

A condição de aplicação desta é uma condição dinâmica, e não pode ser verificada sintaticamente. Para demonstrarmos a condição precisamos provar que todo sinal recebido por r deve ser enviado através de s, e vice-versa. Apesar de leis responsáveis por alterar somente o diagrama de estados do modelo não serem o foco deste trabalho, apresentaremos a seguir algumas leis relacionadas a esta visão. Algumas outras transformações em digramas de estados podem ser encontradas em [34, 96]

A próxima lei expressa como a associação entre duas cápsulas pode ser movida para outra cápsula.

Lei 4.12 *Encapsular Cápsula*



Condições:

(→) *Nenhum nome de porta em* cps′ *coincide com um nome em* csp*; em todo contexto em* $\mathsf{Str_M}$ *onde existir uma instância de* C*, a porta* t′ *estará conectada à porta* t *de uma instância de* B*.*

(←) *Não existe uma porta chamada* t′ *em* C*.*

(↔) *O protocolo* T *e todos protocolos associados às portas em* bps *e* aps′ *possuem uma máquina de estados determinística.*

No lado direito da Lei 4.12, como conseqüência da mudança da associação b de A para C, a instância b é transferida para o diagrama de estrutura de C. Assim, a instância b passa a se comunicar diretamente com as outras instâncias na estrutura de c e necessita de um conjunto de portas cps′ em c para ter acesso ao mundo externo; todas as conexões que existiam em bps passam a se conectar à cps′. A porta t′ deixa de ter utilidade no modelo.

A aplicação da direita para a esquerda desta lei mostra como a hierarquia de um diagrama de estrutura pode ser dissolvida passo-a-passo, através da mudança das associações entre cápsulas. Note que apesar da hierarquia de cápsulas indicar uma ordem na criação destas cápsulas (cápsulas mais externas seriam criadas antes das mais internas), não há garantia quanto à ordem de execução de suas ações iniciais. Desta forma, uma estrutura plana de cápsulas, sem prioridades quanto à criação, teria o mesmo comportamento que uma estrutura hierárquica. De fato, esta hierarquia é usada somente para agrupar conceitos ou computabilidade em processos de execução.

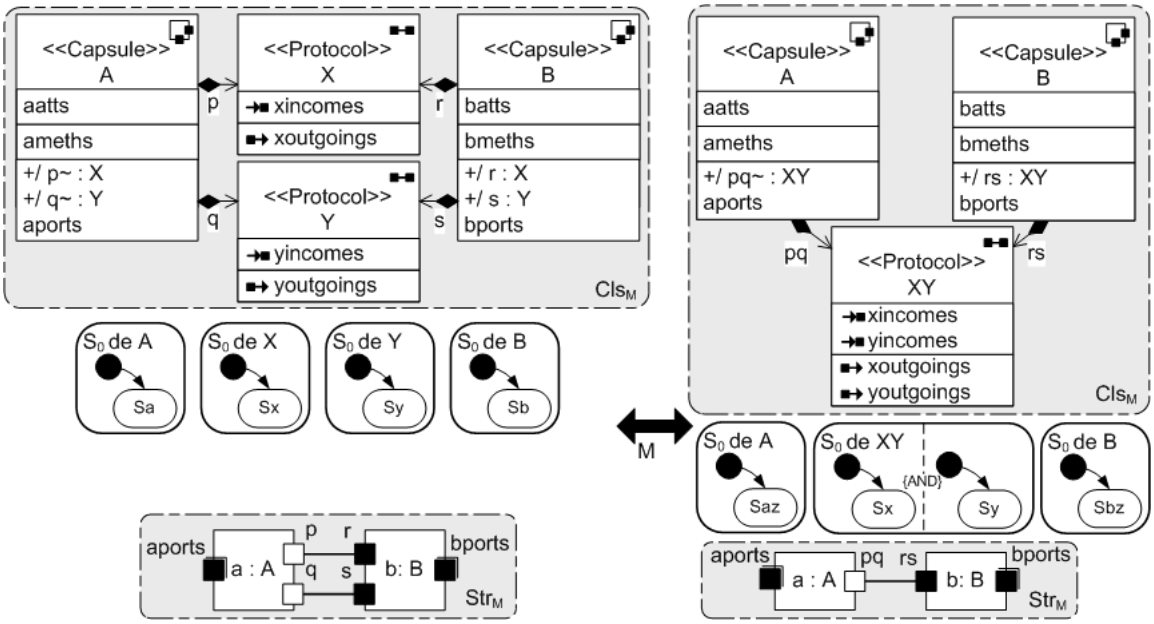
Devido a lei incorporar novas portas no modelo, ela possui uma condição de aplicação de que o comportamento destas portas e, conseqüentemente, seus protocolos, são determinísticos. Desta maneira a conexão de uma nova porta em $\mathsf{Str_M}$ não ira alterar

o comportamento do modelo. Apesar desta ser uma condição dinâmica, na prática, é facilmente verificada por que, normalmente, protocolos não têm máquinas de estados associadas.

Semelhante à Lei 4.12, as leis C.5 e C.6 apresentadas no Apêndice C capturam dois outros possíveis padrões para a mudança de uma associação entre cápsulas.

A lei a seguir indica que um conjunto de portas utilizadas na comunicação entre duas cápsulas podem ser combinadas em apenas duas portas, cujos tipos compreendem os sinais e o comportamento do conjunto original.

Lei 4.13 *Compor Protocolos*



Condições:

(→) X *e* Y *não possuem sinais com nomes em comum.*

(←) Sx *utiliza somente os sinais em* xincomes *e* xoutgoings*;* Sy *utiliza somente os sinais em* yincomes *e* youtgoings.

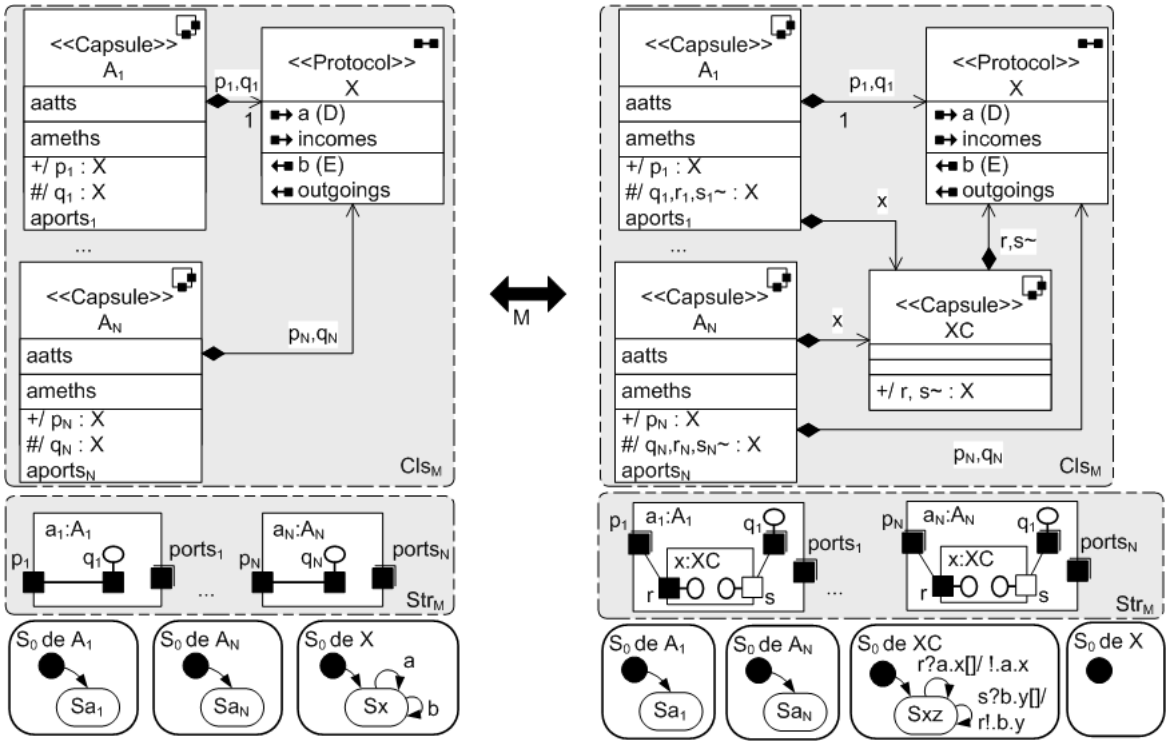
(↔) Sa *e* Sb *são isomórficos a* Saz *e* Sbz*, respectivamente, exceto que todas as ocorrências de* p *e* q *em* Sa *são trocadas por* pq *em* Saz*; similarmente,* r *e* s *são trocadas por* rs *em* Sbz.

No lado esquerdo Lei 4.13, utilizações redundantes dos protocolos X e Y são simplificadas para utilizarem um único protocolo XY, como conseqüência a quantidade de associações e portas no modelos são reduzidas. As máquinas de estados das cápsulas que utilizam tais protocolos continuam a descrever o mesmo comportamento, porém utilizando novas portas, associadas à protocolo XY. Em sentido contrário, a lei pode ser utilizada para aumentar a reusabilidade do protocolo XY, dividindo-o em dois novos protocolos X e Y. Note que a visão Sta_M é representada nesta lei através dos diagramas de estado de das cápsulas e protocolos envolvidas na lei. Esta Lei possui como condição de aplicação que todos estados dos diagramas de estado de A, C, X, e Y estejam encapsulados

dentro de um estado composto que representa toda a máquina de estados; Sa, Sc, Sx, e Sy, respectivamente. Este formato pode ser alcançado através da aplicação da Lei 4.15.

A próxima lei estabelece sob que condições podemos mover o comportamento de um protocolo para as cápsulas que possuem uma associação com este protocolo.

Lei 4.14 *Mover Comportamento Protocolo-Cápsulas*



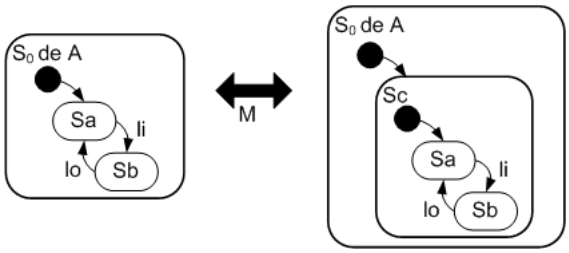
Condições:

(↔) *As máquinas de estados em* Sx *e* Sxz *são isomórficas, exceto que toda transição* a[]\ *disparadas por um sinal de entrada na maquina de estados de* X *é representada por uma transição* r?a.x[]\s!a.x*; similarmente toda transição* b[]\ *disparadas por um sinal de saída* X *é representada por uma transição* s?b.y[]\r!b.y.

No lado esquerdo da Lei 4.14, toda comunicação com a máquina de estados de A_1 pela porta q_1 é filtrada pela máquina de estados de X, que governa o fluxo de comunicação por esta porta; o mesmo acontece nas outras cápsulas do sistema que utilizam portas do tipo X. No lado direito da Lei 4.14, toda comunicação com a máquina de estados de A_1 por uma porta q_1 é sincronizada com as portas r e s, restringindo seu comportamento da mesma maneira que X o fazia no lado esquerdo da lei; similarmente, o mesmo acontece para outras cápsulas A_N que utilizem X. Através desta lei, é possível retirar todas as regras do modelo situadas nos protocolos e movê-las paras as máquinas de estado das cápsulas.

A próxima Lei mostra que um conjunto de estados sempre pode ser visto como um único estado que, por sua vez, engloba todos os outros.

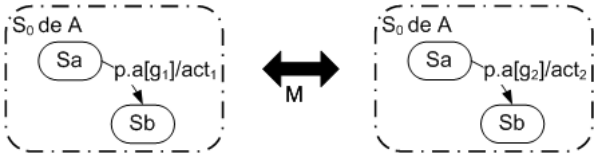
Lei 4.15 *Encapsular estados*



No lado direito da Lei 4.15 toda a submáquina de estados de Sa, do lado esquerdo, esta contida em Sc; Sa, do lado esquerdo, é isomórfica a Sc. Esta lei server apenas para reestruturar o diagrama de estados de A, sem alterar seu comportamento. Assumimos que li e lo representam respectivamente as transições de saída e entrada de Sa, e que tanto Sa quanto Sb podem ser compostos de outros estados.

A próxima lei mostra como a substituição de ações em um diagrama de estados de uma cápsula pode não alterar o comportamento observado desta cápsula. Apesar desta lei ser relativa apenas a ações de transições, podemos assumir que ações de saída ou entrada de estados podem sempre ser reescritas como ações em transições de saída ou entrada destes estados.

Lei 4.16 *Reescrever Ação de Transição*



Condições:

$(\leftrightarrow)$ $\mathsf{act}_1 \equiv \mathsf{act}_2$; $\mathsf{g}_1 \equiv \mathsf{g}_2$

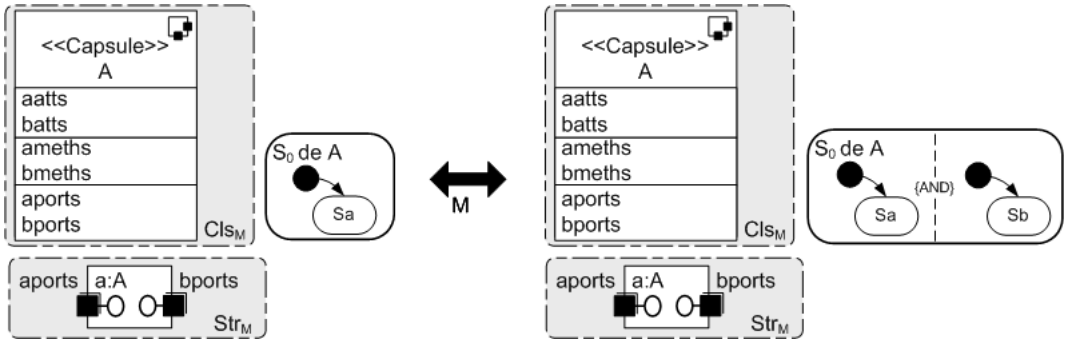
Na Lei 4.16, apenas uma parte dos diagramas de estado de A é apresentado, expressado pela linha pontilhada no diagrama. A equivalência entre ações (act1 $\equiv$ act2) indica que atributos ou parâmetros utilizados por outras ações no diagrama de estados não terão seus valores alterados após a reescrita das ações, nem tampouco o comportamento observado da cápsula será alterado. Como, no contexto deste trabalho, ações são escritas na notação de *Circus*, a equivalência de ações pode ser determinada utilizando-se a noção de leis reportada em [15]. Esta condição é, claramente, semântica. Já que diagramas de estados de protocolos não possuem ações, está implícito nesta lei que o diagrama apresentado é de uma cápsula A de M.

Devido a esta lei não interferir em nenhuma declaração de Cls_M, nem tampouco alterar suas conexões em Str_M, a apresentação dos diagramas de classe e estrutura é desnecessária.

A próxima lei estabelece quando é possível criar uma nova região um AND-State de uma máquina de estados. Uma condição para a aplicação da lei é que a cápsula seja formada por partições. Uma partição de uma cápsula é composto por uma tupla com fragmentos de cada um dos conjuntos de elementos declarados em uma cápsula (atributos, métodos, predicados, portas e máquina de estado), estes fragmentos podem referenciar

somente outros elementos da tupla, e não podem ser referenciados por nenhum outro elemento declarado na cápsula. Assim, uma cápsula particionada é fracamente coesa, cujas partições não compartilham elementos entre si.

Lei 4.17 *Criar Região*



Condições:

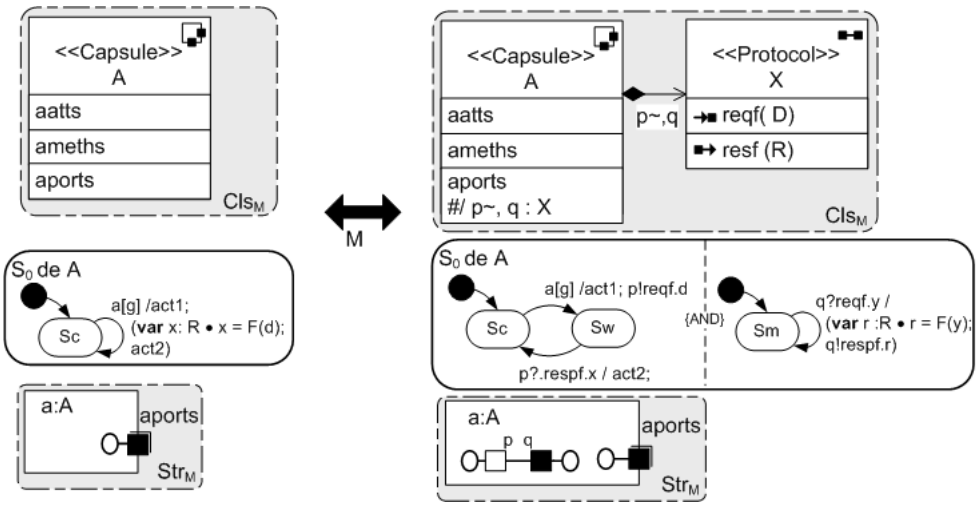
(↔) *A cápsula* A *é formada por duas partições:* ⟨aatts, ainv, ameths, Sa, aports⟩ *e* ⟨batts, binv, bmeths, Sb, bports⟩*; as portas em* bports *não devem estar conectadas a nenhuma outra cápsula em* $\mathsf{Str}_{\mathsf{M}}$.

(←) *As portas em* bports *não estão conectadas a outras portas em* Str_{sfM}.

A Lei 4.17 indica que uma nova região pode ser adicionada a uma cápsula caso ela não interfira no comportamento das regiões restantes da cápsula. No lado direito da Lei 4.17 a máquina de estados de A é formado por um AND-State composto por duas regiões (que contém os estados Sa e Sb), que interagem entre si ou com o ambiente. Além disto, nas ações em Sa, somente atributos aatts e os métodos ameths (que podem referenciar somente atributos em aatts) são usados; analogamente, ações de Sb utilizam somente atributos batts e métodos bmeths (que referenciam somente atributos de batts). O invariante de A é a conjunção ainv ∧ binv, onde ainv inclui somente variáveis livres en aatts, e binv somente em batts. Finalmente, as portas em aports só podem ser referenciadas em Sa e bports em Sb. Quando uma cápsula obedece estas condições, nós dizemos que ela está semi particionada. Neste caso, existem duas semi partições: ⟨aatts, ainv, ameths, Sa, aports⟩ e ⟨batts, binv, bmeths, Sb, bports⟩.

A próxima lei estabelece como uma ação do diagrama de estados de uma cápsula pode ser isolado em uma nova partição da cápsula.

Lei 4.18 *Isolar Ação*



Condições:

(→) *A cápsula* A *não possui a declaração de nenhuma porta, chamada* p *ou* q*; a única variável local utilizada pela ação* F *é a variável* d *do tipo* D.

(←) *O estado* Sw *não possui subestados;*

(↔) *A ação* F *não utiliza atributos em* aatts *que sejam utilizados por outras ações na máquina de estados de* A

Na lado direito da Lei 4.18, o acesso à ação F passa a ser realizado através de um sinal reqf de requisição à porta p, ao invés de executa-lo diretamente. Esta requisição é capturada por uma outra partição na cápsula, que processa a ação F e envia o retorno deste processamento através do sinal resf. Um caso mais simples desta lei é quando F é igual a um método m em ameths, onde m deve possuir um parâmetro de tipo D e retorno de tipo R

4.2 LEIS DERIVADAS E REFATORAMENTOS

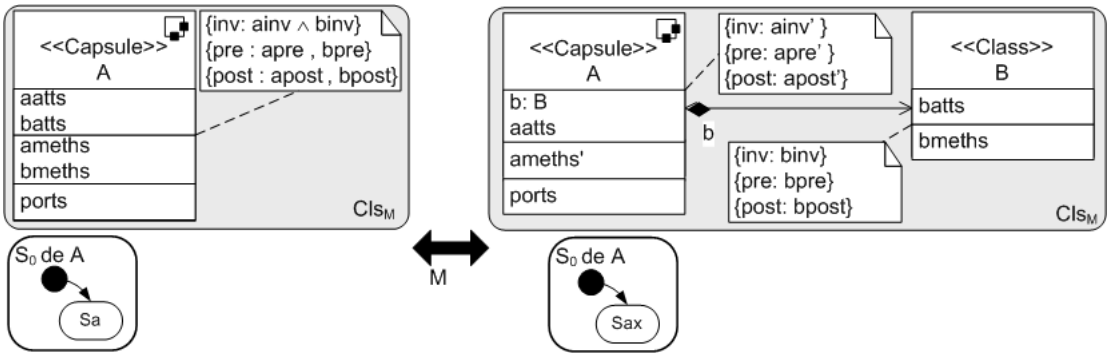
Diferentemente das leis apresentadas na seção anterior, nesta seção consideramos leis de maior granularidade, usualmente aplicadas durante a análise e projeto de *sistemas de software*. Transformações de modelo como estas podem ser justificadas a partir da composição seqüencial de leis básicas, aplicadas passo-a-passo. As condições de aplicação destas leis são criadas a partir da composição das condições de aplicação das leis básicas que as formam.

As transformações apresentadas aqui podem ser consideradas refatoramentos de modelos (*refactorigs*) [96], por serem usadas para reestruturar o modelo e por manterem o comportamento do sistema modelado inalterado após sua aplicação. Mostramos, nesta seção, somente um pequeno conjunto destas leis, focando naquelas que são mais importantes para a descrição dos principais passos da estratégia de normalização de um modelo em UML-RT (Seção 4.2). Após a apresentação de cada lei, serão realizadas justificativas informais sobre a derivação destas transformações a partir das mais básicas; algumas provas formais serão apresentadas na Seção 4.3.

Em um processo de desenvolvimento, é comum identificar uma classe (ou cápsula) durante a análise que depois, durante o projeto, irá ser representada por mais do que uma única abstração. Então, é importante uma transformação para promover esta abstração escondida em uma classe ou cápsula do modelo.

Leis para extração de dados de uma classe ou de uma cápsula são bem semelhantes. Mostramos apenas a lei para extração de dados a partir de uma cápsula. Este tipo de lei, por exemplo, facilita que sistemas sejam modelados como componentes e, após um certo estágio do desenvolvimento, classes sejam extraídas destes componentes (ou cápsulas).

Lei 4.19 *Extrair Classe*



Condições:

(→) bmeths, binv, bpre *e* bpost *acessam somente métodos em* bmeths *ou atributos em* batts.

(←) *Nenhuma cápsula, exceto* A, *utiliza* B.

(↔) *A estrutura interna de* Sa *é isomórfica a* Sax *e os métodos em* amths′ *possuem a mesma assinatura e o corpo dos métodos em* ameths, *exceto que métodos em* bmeths *e atributos em* batts *são acessados a partir de* b.

No lado esquerdo da Lei 4.19, a cápsula A é representada pelo conjuntos de atributos aatts e batts, de métodos ameths e bmeths e de portas ports. A máquina de estados de A é representada por um estado Sa que engloba todos os outros estados da máquina de estado, e tem acesso a qualquer elemento descrito anteriormente. O invariante de A é a conjunção ainv ∧ binv, e a pré- e pós-condições dos métodos de A são expressos por apre, bpre, apost e bpost. Os elementos batts, bmeths, bpre e bpost são fortemente coesos, e não fazem referência a nenhum outro elemento de A; estes são os elementos que podem ser extraídos para uma nova classe. Na lado direito da Lei 4.19, toda ação em Sax, método de ameths′ ou predicado (ainv′, apre′ ou apost′) que anteriormente acessava um atributo i de batts ou método m de bmeths deverá ser reescrita para acessá-los como b.i ou b.j.

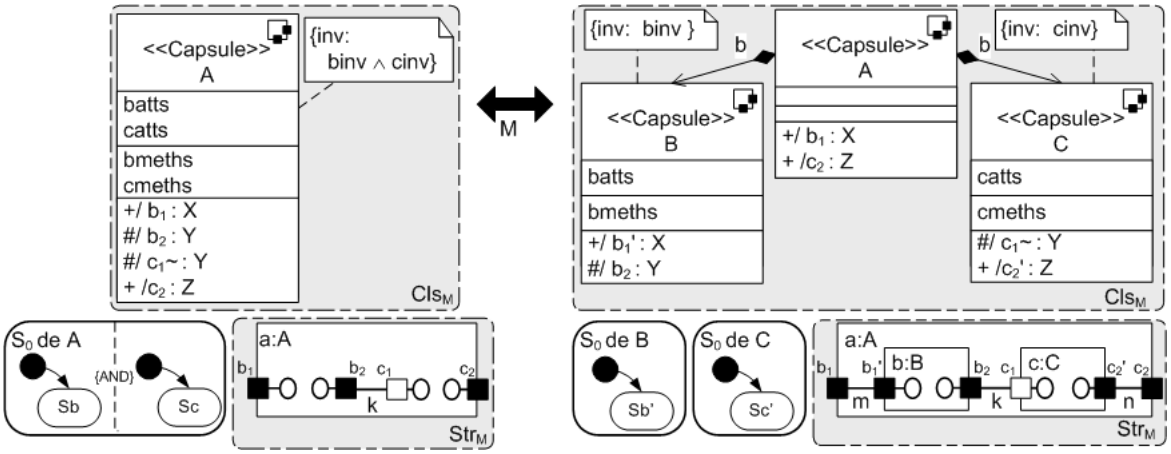
Note que como B é uma nova abstração no modelo, pode-se desejar que esta esteja livre de certas regras impostas ao contrato de A. Desta forma, pode-se atribuir qualquer invariante binv a B, tal que ainv ∧ binv seja aceitável pelo contrato de A. Da mesma forma pré- e pós-condições indesejáveis em B devem ser colocadas em métodos em ameths que deleguem todas as suas ações a métodos em bmeths.

A justificativa da Lei 4.19 a partir de leis mais básicas é feita aplicando-se a Lei 4.4 para se introduzir o atributo b em A, do tipo de uma classe B (criada a partir da Lei C.2)

como descrita na Lei 4.19, e aplicando-se a Lei 4.16 para se reescrever as ações da máquina de estados de A em ações equivalentes que utilizam b. Por fim, atributos e métodos não utilizados mais em A podem ser removidos aplicando-se as Leis 4.3 e 4.4. Um exemplo de uma lei de transformação para a extração de classes a partir de uma cápsula pode ser visto em [83]; esta lei mostra a extração de um atributo, representando um conjunto de classes, para uma classe que mediará o acesso a este conjunto.

A lei a seguir decompõe uma cápsula A no paralelismo de instâncias de cápsulas (B e C) com o propósito de diminuir sua complexidade e de, potencialmente, aumentar seu reuso.

Lei 4.20 *Decomposição Paralela de uma Cápsula*



Condições:

(→) $\langle \mathsf{batts}, \mathsf{binv}, \mathsf{bmeths}, (\mathsf{b}_1, \mathsf{b}_2), \mathsf{Sb} \rangle$ *e* $\langle \mathsf{catts}, \mathsf{cinv}, \mathsf{cmeths}, (\mathsf{c}_1, \mathsf{c}_2, \mathsf{Sc} \rangle$ *particionam* A*;*

(↔) *As máquinas de estado* Sb *e* Sc *são isomórficas à* Sb′ *e* Sc′*, respectivamente, exceto que todas as ocorrências de* b_1 *e* c_2 *em são substituídas respectivamente, por* b'_1 *e* c'_2*; os protocolo* X *e* Z *possuem uma máquina de estados determinística.*

Analogamente à Lei4.17, a Lei 4.20 requer que a cápsula A seja particionada, onde cada partição deve ser auto-contida e fazer uso somente dos atributos e métodos da partição. Além disto, as únicas portas utilizadas em uma partição são aquelas que ela contém.

No lado esquerdo da Lei 4.20, a máquina de estado de A é formada por um AND-State, composto de duas regiões. Cada uma destas regiões possui um estado que engloba todos os outros (Sb e Sc), que interagem entre si (comunicação interna) através das portas conjugadas b_2 e c_1 (como capturado pelo diagrama de estrutura). As outras duas portas (b_1 e c_2) são utilizadas para comunicações externas dos estados Sb e Sc, respectivamente. As portas b_1 e b_2 são utilizadas somente pelo estado Sb, enquanto Sc faz uso somente das portas c_1 e c_2. Como dito, anteriormente, as ações de Sb utilizam somente atributos e métodos de batts e bmeths; analogamente Sc faz uso somente de catts e cmeths. Finalmente, o invariante de A é a conjunção binv ∧ cinv, onde binv inclui somente batts como variáveis livres, e cinv inclui somente catts.

O efeito da decomposição é criar duas novas cápsulas componentes, b e c, uma para cada partição, e redimensionar a cápsula original A para agir como um mediador. Em geral, o novo comportamento de A irá depender da forma particular da decomposição. A Lei 4.20 captura uma decomposição paralela. No lado direito da lei, A não possui uma máquina de estados, delegando completamente seu comportamento original para B e C através de conexões com componentes destes tipos no diagrama de estrutura.

Em relação ao diagrama de estrutura no lado direito da lei, ele mostra como A encapsula b e c. Quando a é criada, ele automaticamente cria instâncias de b e c, que executam concorrentemente. As portas públicas b_1 e c_2 são preservadas em A, e, assim, sua interface com o ambiente externo. A cápsula B tem como porta pública uma imagem de b_1, chamada $\mathsf{b}_1^{'}$. Apesar desta porta ser pública em B, ela somente é vista dentro do diagrama de estrutura de A. O papel desta porta é possibilitar que B receba sinais externos recebidos por A através de b_1, como é capturado pela conexão entre $\mathsf{b}_1^{'}$ e b_1 no diagrama de estrutura de A. Analogamente, c_2 e $\mathsf{b}_2^{'}$ têm o mesmo relacionamento, relativo às cápsulas A e C. As portas internas b_2 e c_1 foram movidas para as cápsulas B e C, respectivamente, e realizam os mesmo papéis que antes.

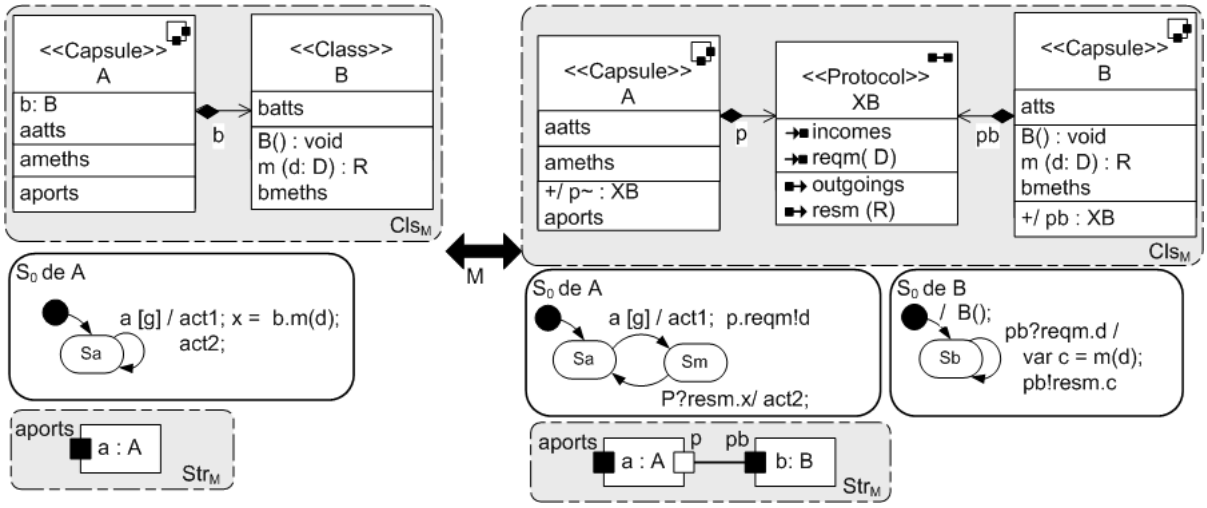
A justificativa da Lei 4.20 a partir das leis básicas pode ser feita movendo-se, no lado direito da lei, passo-a-passo o comportamento de B e C para A. Inicialmente movemos o comportamento de B para uma outra cápsula A', que futuramente irá substituir A. O primeiro passo é desencapsular o componente de cápsula b (Lei 4.12) da estrutura de A. Criar outra cápsula A' (Lei 4.1) com os atributos (batts), invariante (binv), métodos (bmeths) e portas (b_1 e b_2) de B, e então, inserir uma instância a' de A' em todo contexto de $\mathsf{Str_M}$ em que uma instância de A aparece (Lei 4.2). Em seguida, b é substituído por a' através da aplicação da Lei 4.9, já que as máquinas de estados de B e A' são isomórficas. Finalmente, b é removida de $\mathsf{Str_M}$ (Lei 4.2).

Para se mover o comportamento de C para A', os atributos catts, os métodos cmeths e as portas c_1 e c_2 são adicionados à A' (Leis 4.3 e 4.5). Então, uma região é adicionada ao digrama de estados de A' com o estado Sc (Lei 4.17). Já que Sc não compartilha nenhum elemento (atributo, método ou porta) com Sb, o comportamento de A continua inalterado. No passo seguinte, o componente de cápsula c é desencapsulado (Lei 4.12) do diagrama de estrutura de A, substituída por a' (Lei 4.9), e removida de $\mathsf{Str_M}$ (Lei 4.2), seguindo um processo semelhante ao que aplicado à instância de c.

No final, A não é mais utilizada no sistema, podendo ser retirada do modelo (Lei 4.1), e assim A' pode ter seu nome renomeado para A (Lei C.3), obtendo assim o mesmo contexto apresentado no lado esquerdo da Lei 4.20.

Motivado por práticas já existentes de desenvolvimento, precisamos de uma lei que transforma classes, possivelmente encontradas durante a análise, em cápsulas, durante o projeto.

Lei 4.21 *Transformar Classe em Cápsula*



Condições:

(→) *Todos os atributos* batts *são privados.*

(↔) *Nenhuma outra cápsula, além de* A*, possui uma relação com* B.

A Lei 4.21 transforma uma classe B (lado esquerdo), que não tem nenhum comportamento associado (diagrama de estados), em uma cápsula (lado direito), que inclui um diagrama de estados que lida com as requisições para a chamada de seus métodos.

O comportamento das chamadas de métodos em B é preservado por um diagrama de estados que simula uma comunicação sincronizadas com o protocolo cliente. Então, todos os serviços (métodos públicos) da classe são acessíveis a partir de um novo protocolo XB. Note que o construtor da classe B transforma-se na ação da transição inicial de diagrama de estados da nova cápsula B.

A justificativa desta lei em função de leis básicas pode ser feita, no lado esquerdo da Lei 4.21, aplicando-se a Lei 4.19 para incluir a classe B na cápsula A, em seguida aplica-se as leis 4.17 e 4.18 para criar uma nova partição em A com os atributos em batts, métodos bmeths, m() e B().

Após particionar A, a Lei 4.20 pode ser aplicada para decompor a cápsula A em duas cápsulas, onde uma destas é a cápsula B. Então, o componente b do tipo B será desencapsulado da estrutura de A (Lei 4.12) para que obtenhamos $\mathsf{Str_M}$ como é apresentado no lado esquerdo da Lei 4.21. Finalmente uma região vazia será criada em A(Lei 4.17), e, através da aplicação da Lei 4.20, a cápsula A poderá ter a mesma declaração que no lado esquerdo da Lei 4.21.

4.3 FORMALIZAÇÃO DAS LEIS

Baseado no mapeamento semântico apresentado na Seção 3.2, podemos transformar os modelos em UML-RT presentes nos dois lados de uma lei de transformação para a notação *OhCircus*. A partir da semântica de *Circus* e de seu cálculo de refinamentos, é possível verificar a validade das leis de transformação para UML-RT. Ao invés de recorrer

diretamente à semântica de *Circus* podemos utilizar leis de refinamento [84] para realizar sua prova; na realidade, várias das leis para UML-RT foram inspiradas por leis em *Circus*.

4.3.1 Prova da Lei 4.20

Para provar esta lei, lidamos inicialmente com as visões descritas pelos diagramas de classe e estado. O mapeamento destas visões da cápsula A, do lado esquerdo da lei, como um processo em *Circus* é obtido utilizando-se a segunda regra de mapeamento apresentada na Seção 3.2.1 e a último regra da Seção 3.2.2. Escrevemos as duas partições usando o operador $\uparrow$, como na Lei 3.2. De fato, a Lei 4.20 foi inspirada pela Lei 3.2 de *Circus*. Assim, as visões representadas pelo processo $Chart_{\mathsf{A}}$ são equivalentes à simples composição paralela de $Chart_{\mathsf{B}}$ e de $Chart_{\mathsf{C}}$. Estritamente, os atributos e métodos *batts*, *catts*, *bmeths* e *cmeths* precisam ser mapeados utilizando a função $\mathcal{TL}$; aqui nós omitiremos sua aplicação para manter a legibilidade.

$$
\begin{array}{l}
\textbf{process}\ Chart_{\mathsf{A}} \mathrel{\widehat{=}} \textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat{=}} [\mathsf{batts} \wedge \mathsf{catts} \mid \mathsf{Inv_B} \wedge \mathsf{Inv_C}] \\
\quad \mathsf{bmeths} \uparrow \mathsf{catts} \\
\quad \mathsf{cmeths} \uparrow \mathsf{batts} \\
\quad \bullet\ \mathcal{H}(\mathsf{S_B})[\mathsf{b_2} := \mathsf{k}]\ [\![\ \mathsf{batts} \mid \{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\} \mid \mathsf{catts}]\!])\mathcal{H}(\mathsf{S_C})[\mathsf{c_1} := \mathsf{k}] \\
\textbf{end}
\end{array}
$$

Aplicando as funções identidade $[\mathsf{b_1} := \mathsf{b_1'}][\mathsf{b_1'} := \mathsf{b_1}]$ e $[\mathsf{c_2} := \mathsf{c_2'}][\mathsf{c_2'} := \mathsf{c_2}]$ à ação principal de $Chart_{\mathsf{A}}$, obtemos:

$$
\bullet\ \mathcal{H}(\mathsf{S_B})[\mathsf{b_1} := \mathsf{b_1'}][\mathsf{b_1'}, \mathsf{b_2} := \mathsf{b_1}, \mathsf{k}]\ [\![\ \mathsf{batts} \mid \{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\} \mid \mathsf{catts}]\!])\mathcal{H}(\mathsf{S_C})[\mathsf{c_2} := \mathsf{c_2'}][\mathsf{c_2'}, \mathsf{c_1} := \mathsf{c_2}, \mathsf{k}]
$$

A partir da Lei 3.2, obtemos a igualdade:

$$
Chart_{\mathsf{A}} = Chart_{\mathsf{B}}[\mathsf{b_1'}, \mathsf{b_2} := \mathsf{b_1}, \mathsf{k}]\ [\![\ \{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}\ ]\!]\ Chart_{\mathsf{C}}[\mathsf{c_2'}, \mathsf{c_1} := \mathsf{c_2}, \mathsf{k}]
$$

onde $Chart_{\mathsf{B}}$ e $Chart_{\mathsf{B}}$ são declarados como:

$$
\begin{array}{l}
\textbf{process}\ Chart_{\mathsf{B}} \mathrel{\widehat{=}} \textbf{begin state}\ State \mathrel{\widehat{=}} [\mathsf{batts} \mid \mathsf{Inv_B}]\ \mathsf{bmeths} \bullet \mathcal{H}(\mathsf{S_B})[\mathsf{b_1} := \mathsf{b_1'}]\ \textbf{end} \\
\textbf{process}\ Chart_{\mathsf{C}} \mathrel{\widehat{=}} \textbf{begin state}\ State \mathrel{\widehat{=}} [\mathsf{catts} \mid \mathsf{Inv_C}]\ \mathsf{cmeths} \bullet \mathcal{H}(\mathsf{S_C})[\mathsf{c_2} := \mathsf{c_2'}]\ \textbf{end}
\end{array}
$$

Agora, considerando a parte estrutural da cápsula A, obtemos o processo $Struct_{\mathsf{A}}$ em *Circus*, como é apresentado abaixo.

$$
\begin{array}{l}
Struct_{\mathsf{A}} \mathrel{\widehat{=}} (Chart_{\mathsf{A}}[\mathsf{b_2}, \mathsf{c_1} := \mathsf{k}, \mathsf{k}]\ [\![\{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}]\!]\ \mathsf{X}[chan_{\mathsf{X}} := \mathsf{b_1}]\ [\![\{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}]\!] \\
\quad \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}]\ [\![\{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}]\!]\ \mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{k}]\ [\![\{\!|\ \mathsf{k}, \mathsf{b_1}, \mathsf{c_2}\ |\!\}]\!]\ \mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c_2}]) \setminus \{\!|\ \mathsf{k}\ |\!\}
\end{array}
$$

O diagrama de estrutura de B, C e A, no lado direito da lei, são mapeados de maneira similar. Para se evitar confusões entre as duas ocorrências de A na lei, iremos nos referir à ocorrência no lado esquerdo simplesmente como A, e à do lado direito como $\mathsf{A'}$.

$$
\begin{aligned}
&\mathit{Struct}'_{\mathsf{A}} \mathrel{\widehat{=}} ((\mathit{Struct}_{\mathsf{B}}[\mathsf{b}'_1, \mathsf{b}_2 := \mathsf{m}, k] \llbracket \{\!|\ \mathsf{k}\ |\!\} \rrbracket\ \mathit{Struct}_{\mathsf{C}}[\mathsf{c}'_2, \mathsf{c}_1 := \mathsf{n}, k]) \llbracket \{\!|\ \mathsf{k}\ |\!\} \rrbracket \\
&\quad (\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{m}] \llbracket \{\!|\ \mathsf{k}\ |\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{n}]))[\mathsf{m}, \mathsf{n} := \mathsf{b}_1, \mathsf{c}_2] \setminus \{\!|\ \mathsf{k}\ |\!\} \\
&\mathit{Structs}_{\mathsf{B}} \mathrel{\widehat{=}} \mathit{Chart}_{\mathsf{B}} \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1|\!\} \rrbracket \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}'_1] \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1|\!\} \rrbracket \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{b}_2] \\
&\mathit{Struct}_{\mathsf{C}} \mathrel{\widehat{=}} \mathit{Chart}_{\mathsf{C}} \llbracket \{\!|\mathsf{c}_1,\mathsf{c}'_2|\!\} \rrbracket \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{c}_1] \llbracket \{\!|\mathsf{c}_1,\mathsf{c}'_2|\!\} \rrbracket \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}'_2]
\end{aligned}
$$

Então, queremos provar que $\mathit{Struct}_{\mathsf{A}}$ tem o mesmo comportamento de $\mathit{Struct}_{\mathsf{A}'}$, onde o último inclui em sua estrutura $\mathit{Struct}_{\mathsf{B}}$ e $\mathit{Struct}_{\mathsf{C}}$.

$\mathit{Struct}_{\mathsf{A}}$
$= [$1. Pela definição de $\mathit{Struct}_{\mathsf{A}}]$
$(\mathit{Chart}_{\mathsf{A}}[\mathsf{b}_2, \mathsf{c}_1 := \mathsf{k}, \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}]$
$\llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2]) \setminus \{\!|\mathsf{k}|\!\}$
$= [$2. Expandindo $\mathit{Chart}_{\mathsf{A}}\ ]$
$(\mathit{Chart}_{\mathsf{B}}[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{b}_1,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathit{Chart}_{\mathsf{C}}[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{c}_2,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1]$
$\llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$3. Aplicando a Lei B.6 para $\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1]$ e $\mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2]]$
$(\mathit{Chart}_{\mathsf{B}}[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{b}_1,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket$
$\mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathit{Chart}_{\mathsf{C}}[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{c}_2,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket$
$\mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$4. Reorganizando os processos utilizando as leis B.4 e B.4$]$
$(\mathit{Chart}_{\mathsf{B}}[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{b}_1,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket$
$\mathit{Chart}_{\mathsf{C}}[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{c}_2,\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket$
$\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$5. Aplicando as funções de substituição indentidade $[\mathsf{k} := \mathsf{b}_2][\mathsf{b}_2 := \mathsf{k}]$, $[\mathsf{k} := \mathsf{c}_1][\mathsf{c}_1 := \mathsf{k}]$, e distribuindo $[\mathsf{k} := \mathsf{b}_2]$ e $[\mathsf{k} := \mathsf{c}_1]$ através da Lei B.1 $]$
$(((\mathit{Chart}_{\mathsf{B}}[\mathsf{b}'_1 := \mathsf{b}_1] \llbracket \{\!|\mathsf{b}_2,\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1] \llbracket \{\!|\mathsf{b}_2,\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{b}_2])[\mathsf{b}_2 := \mathsf{k}]$
$\llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ ((\mathit{Chart}_{\mathsf{C}}[\mathsf{c}'_2 := \mathsf{c}_2] \llbracket \{\!|\mathsf{c}_1,\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{c}_1] \llbracket \{\!|\mathsf{c}_1,\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}'_2])$
$[\mathsf{c}_1 := \mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}_1]) \llbracket \{\!|\mathsf{k},\mathsf{b}_1,\mathsf{c}_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}_2])) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$6. Aplicando a função de substituição indentidade $[\mathsf{c}_2,\mathsf{b}_1 := \mathsf{c}'_2,\mathsf{b}'_1][\mathsf{c}'_2,\mathsf{b}'_1 := \mathsf{n},\mathsf{m}]$ $[\mathsf{n},\mathsf{m} := \mathsf{c}_2,\mathsf{b}_1]$, e distribuindo $[\mathsf{c}_2,\mathsf{b}_1 := \mathsf{c}'_2,\mathsf{b}'_1][\mathsf{c}'_2,\mathsf{b}'_1 := \mathsf{n},\mathsf{m}]$ através da Lei B.1 $]$
$(((\mathit{Chart}_{\mathsf{B}} \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1,\mathsf{n}|\!\} \rrbracket \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}'_1] \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1,\mathsf{n}|\!\} \rrbracket \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{b}_2])[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{m},\mathsf{k}]$
$\llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ ((\mathit{Chart}_{\mathsf{C}} \llbracket \{\!|\mathsf{c}_1,\mathsf{m},\mathsf{c}'_2|\!\} \rrbracket\ \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{c}_1] \llbracket \{\!|\mathsf{c}_1,\mathsf{m},\mathsf{c}'_2|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}'_2])$
$[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{n},\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m},\mathsf{n} := \mathsf{b}_1,\mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$7. Utilizando a Lei B.7 para restringir as sincronizações sobre os canais n e $\mathsf{m}]$
$(((\mathit{Chart}_{\mathsf{B}} \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1|\!\} \rrbracket \mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}'_1] \llbracket \{\!|\mathsf{b}_2,\mathsf{b}'_1|\!\} \rrbracket \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{b}_2])[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{m},\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket$
$((\mathit{Chart}_{\mathsf{C}} \llbracket \{\!|\mathsf{c}_1,\mathsf{c}'_2|\!\} \rrbracket \mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{c}_1] \llbracket \{\!|\mathsf{c}_1,\mathsf{c}'_2|\!\} \rrbracket \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{c}'_2])[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{n},\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket$
$\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m},\mathsf{n} := \mathsf{b}_1,\mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= [$8. Utilizando a definição de $\mathit{Struct}_{\mathsf{B}}$ e $\mathit{Struct}_{\mathsf{C}}\ ]$
$((\mathit{Struct}_{\mathsf{B}}[\mathsf{b}'_1,\mathsf{b}_2 := \mathsf{m},\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ (\mathit{Struct}_{\mathsf{C}}[\mathsf{c}'_2,\mathsf{c}_1 := \mathsf{n},\mathsf{k}] \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket$
$\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{m}]) \llbracket \{\!|\mathsf{k},\mathsf{m},\mathsf{n}|\!\} \rrbracket\ \mathsf{Z}[\mathit{chan}_{\mathsf{Z}} := \mathsf{n}])[\mathsf{m},\mathsf{n} := \mathsf{b}_1,\mathsf{c}_2]) \setminus \{\!|\ \mathsf{k}\ |\!\}$
$= \mathit{Struct}_{\mathsf{A}'}$

Durante a prova, três leis condicionais são utilizadas. A condição da Lei B.6 (Passo 3) é claramente satisfeita porque os processos X e Z são obtidos de dos protocolos X e Z, que, na Lei 4.20, assume-se que são determinísticos. A condição da Lei B.7 (Passo 7) é satisfeita já que que os processos $\mathit{Chart}_{\mathsf{B}}$, $\mathsf{X}[\mathit{chan}_{\mathsf{X}} := \mathsf{b}'_1]$ e $\mathsf{Y}[\mathit{chan}_{\mathsf{Y}} := \mathsf{b}_2]$ não utilizam o

canal n; similarmente, os processos $Chart_{\mathsf{C}}$, $\mathsf{Y}[chan_{\mathsf{Y}} := \mathsf{c}_1]$ e $\mathsf{Z}[chan_{\mathsf{Z}} := \mathsf{c'}_2]$ não utilizam m. A condição da Lei B.1 (passos 5 e 6) é satisfeita já que as seguintes funções de renomeação utilizadas na distribuição são injetivas: $[\mathsf{k} := \mathsf{b}_2]$, $[\mathsf{k} := \mathsf{c}_1]$ e $[\mathsf{c}_2, \mathsf{b}_1 := \mathsf{n}, \mathsf{m}]$.

4.3.2 Prova da Lei 4.12

No lado esquerdo da Lei 4.12, em toda ocorrência de uma instância da cápsula C, existe sempre uma instância b da cápsula B conectada a uma de suas subcápsulas d da cápsula $\mathsf{d} : \mathsf{D}$. A cápsula A representa o contexto onde esta configuração existe. A lei indica que após sua aplicação, em seu lado direito, o comportamento de C e de qualquer contexto em que apareça (A) será inalterado, caso b passe a ser um componente do diagrama de estrutura de C.

Para provar esta lei, precisamos lidar apenas com a visão estrutural do modelos ($\mathsf{Str_M}$), já que as únicas reestruturações da lei que podem afetar o comportamento do sistema são mudanças realizadas nas conexões do diagrama de estrutura de C e A. De fato a visão intencional das instâncias de cápsulas em *Circus* desconsidera as associações no diagrama de classes do modelo, mapeando unicamente a ocorrências destas instâncias em $\mathsf{Str_M}$. Apesar da remoção da porta $\mathsf{t'}$, no lado esquerdo da lei, e da adição do conjunto de portas $\mathsf{p'}$, no lado direito da lei, estas portas são somente utilizadas na visão estrutural para interconectar portas de componentes de cápsula em diferentes estruturas, e nenhuma necessidade de prova quando à sua declaração é necessária.

Devido à condição de que a única instância de C é o componente c em A, nossa demonstração precisa mostrar apenas a igualdade do comportamento de A no lado esquerdo e direito da lei. Para se evitar confusões entre as duas ocorrências de A e C na lei, iremos nos referir às ocorrência no lado esquerdo simplesmente como A e C, e às do lado direito como $\mathsf{A'}$ $\mathsf{C'}$; note que B e D continuam inalterados em ambos os lados da lei. Assim, o comportamento da cápsula A seria dado por $Struct_{\mathsf{A}}$, através da terceira regra de mapeamento da Seção 3.2.1.

$$\begin{aligned} Struct_{\mathsf{A}} \mathrel{\widehat{=}} & (\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket Struct_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \\ & Struct_{\mathsf{C}}[\mathsf{cps,t'} := \mathsf{y,w}]) \, [\mathsf{x,y} := \mathsf{aps'}, \mathsf{aps}] \setminus \{\!| \mathsf{w} |\!\} \end{aligned}$$

Similarmente, o mapeamento de B, C, $\mathsf{C'}$ e $\mathsf{A'}$ são dados por:

$$\begin{aligned} Struct_{\mathsf{B}} \mathrel{\widehat{=}} & Chart_{\mathsf{B}} \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathcal{TL}(\mathsf{bps}) \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathsf{T}[chan_{\mathsf{T}} := t] \\ Struct_{\mathsf{C}} \mathrel{\widehat{=}} & (\mathsf{T}[chan_{\mathsf{T}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{k,z} |\!\} \rrbracket Struct_{\mathsf{D}}[\mathsf{dps}, \mathsf{s} := \mathsf{z}, \mathsf{k}]) \\ & [\mathsf{z}, \mathsf{k} := \mathsf{csp}, \mathsf{t'}] \\ Struct_{\mathsf{C'}} \mathrel{\widehat{=}} & (Struct_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x'}, \mathsf{w}] \llbracket \{\!| \mathsf{x'}, \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'}, \mathsf{w,z} |\!\} \rrbracket \\ & Struct_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) \, [\mathsf{z,x'} := \mathsf{csp,cps'}] \setminus \{\!| \mathsf{w} |\!\} \\ Struct_{\mathsf{A'}} \mathrel{\widehat{=}} & (\mathcal{TL}(\mathsf{aps}) \llbracket x, y \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket x, y \rrbracket Struct_{\mathsf{C'}}[\mathsf{cps}, \mathsf{cps'} := \mathsf{y}, \mathsf{x}]) \\ & [\mathsf{x}, \mathsf{y} := \mathsf{aps'}, \mathsf{aps}] \end{aligned}$$

Como veremos na prova abaixo, o comportamento do processo $Struct_{\mathsf{A}}$ será igual ao do $Struct_{\mathsf{A'}}$, de fato a única mudança entre será na ordem da composição paralela de $Struct_{\mathsf{B}}$ e $Struct_{\mathsf{C}}$, e na renomeção de seus canais.

$Struct_{\mathsf{A}}$

$= [1.\ \text{Expandindo } \mathit{Struct}_{\mathsf{A}}]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket$
$\mathit{Struct}_{\mathsf{C}}[\mathsf{cps,t'} := \mathsf{y,w}]) [\mathsf{x,y} := \mathsf{aps',aps}] \setminus \{\!| \mathsf{w} |\!\}$
$= [2.\ \text{Expandindo } \mathit{Struct}_{\mathsf{C}}]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket \mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket$
$((\mathsf{T}[\mathit{chan}_{\mathsf{T}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{k,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps}, \mathsf{s} := \mathsf{z}, \mathsf{k}]) [\mathsf{z}, \mathsf{k} := \mathsf{csp}, \mathsf{t'}])$
$[\mathsf{cps,t'} := \mathsf{y,w}]) [\mathsf{x,y} := \mathsf{aps',aps}] \setminus \{\!| \mathsf{w} |\!\}$
$= [3.\ \text{Restrigindo } \mathsf{w} \text{ aos processos que o usam, aplicando as leis B.7 e B.2}]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket (\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket$
$((\mathsf{T}[\mathit{chan}_{\mathsf{T}} := \mathsf{k}] \llbracket \{\!| \mathsf{k,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \mathsf{k,z} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,k}]) [\mathsf{z}, \mathsf{k} := \mathsf{csp}, \mathsf{t'}])$
$[\mathsf{cps}, \mathsf{t'} := \mathsf{y}, \mathsf{w}]) \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x}, \mathsf{y} := \mathsf{aps'}, \mathsf{aps}]$
$= [4.\ \text{Aplicando os } \mathit{renamings}\ [\mathsf{t'} := \mathsf{k}] \text{ e } [\mathsf{k} := \mathsf{w}]]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket (\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket$
$((\mathsf{T}[\mathit{chan}_{\mathsf{T}} := \mathsf{w}] \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z} := \mathsf{csp}])$
$[\mathsf{cps} := \mathsf{y}]) \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'}, \mathsf{aps}]$
$= [5.\ \text{Inserindo a função indentidade } [\mathsf{x} := \mathsf{x'}][\mathsf{x'} := \mathsf{cps'}][\mathsf{cps'} := \mathsf{x}]\ ]$
$((\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket (\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x,w}] \llbracket \{\!| \mathsf{x,w,y} |\!\} \rrbracket$
$((T[\mathit{chan}_{\mathsf{T}} := w] \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z} := \mathsf{csp}])$
$[\mathsf{cps} := \mathsf{y}]) \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x} := \mathsf{x'}][\mathsf{x'} := \mathsf{cps'}][\mathsf{cps'} := \mathsf{x}]) [\mathsf{x,y} := \mathsf{aps'}, \mathsf{aps}]$
$= [6.\ \text{Aplicando os } \mathit{renamings}\ [\mathsf{x} := \mathsf{x'}],\ [\mathsf{x'} := \mathsf{cps'}]e[\mathsf{cps'} := \mathsf{x}]]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{w,y} |\!\} \rrbracket$
$(\mathsf{T}[\mathit{chan}_{\mathsf{T}} := w] \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}])$
$[\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}]) \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}]$
$= [7.\ \text{Aplicando B.7 para espandir o canal } \mathsf{x}\ ]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{w,y} |\!\} \rrbracket$
$\mathsf{T}[\mathit{chan}_{\mathsf{T}} := w] \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}])$
$[\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}])$
$= [8.\ \text{Restringindo } \mathsf{w} \text{ através da Lei B.7}]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket (((\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{y} |\!\} \rrbracket \mathsf{T}[\mathit{chan}_{\mathsf{T}} := w])$
$\llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}])$
$[\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}]$
$= [9.\ \text{Expandindo } \mathit{Struct}_{\mathsf{B}}\ ]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((((\mathit{Chart}_{\mathsf{B}} \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathcal{TL}(\mathsf{bps}) \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket$
$\mathsf{T}[\mathit{chan}_{\mathsf{T}} := t])[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{y} |\!\} \rrbracket \mathsf{T}[\mathit{chan}_{\mathsf{T}} := w]) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket$
$\mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}]) [\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}]$
$= [10.\ \text{Usando a Lei B.1 para distribuir o } \mathit{renaming}\ [\mathsf{bps,t} := \mathsf{x,w}]]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((((\mathit{Chart}_{\mathsf{B}} \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathcal{TL}(\mathsf{bps}) \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket$
$\mathsf{T}[\mathit{chan}_{\mathsf{T}} := t] \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathsf{T}[\mathit{chan}_{\mathsf{T}} := t]))[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket$
$\mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}]) [\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}]$
$= [11.\ \text{Usando a Lei B.6 para } \mathsf{T}[\mathit{chan}_{\mathsf{T}} := t]\ ]$
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps'}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((((\mathit{Chart}_{\mathsf{B}} \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket \mathcal{TL}(\mathsf{bps}) \llbracket \{\!| \mathsf{bps,t} |\!\} \rrbracket$
$\mathsf{T}[\mathit{chan}_{\mathsf{T}} := t]))[\mathsf{bps,t} := \mathsf{x'},\mathsf{w}] \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps}) \llbracket \{\!| \mathsf{x'},\mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}])$
$[\mathsf{z},\mathsf{x'} := \mathsf{csp},\mathsf{cps'}]) [\mathsf{cps}, \mathsf{cps'} := \mathsf{y},\mathsf{x}] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{x,y} := \mathsf{aps'},\mathsf{aps}]$
$= [12.\ \text{Pela definição de } \mathit{Struct}_{\mathsf{B}}\ ]$

$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps}') \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x}', \mathsf{w}] \llbracket \{\!| \mathsf{x}', \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps})$
$\llbracket \{\!| \mathsf{x}', \mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z,x}' := \mathsf{csp,cps}']) [\mathsf{cps}, \mathsf{cps}' := \mathsf{y,x}] \setminus \{\!| \mathsf{w} |\!\})$
$[\mathsf{x,y} := \mathsf{aps}', \mathsf{aps}]$

$=$ [13. aplicando a Lei B.3 para mover $\setminus \{\!| \mathsf{w} |\!\}$]
$(\mathcal{TL}(\mathsf{aps}) \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket \mathcal{TL}(\mathsf{aps}') \llbracket \{\!| \mathsf{x,y} |\!\} \rrbracket ((\mathit{Struct}_{\mathsf{B}}[\mathsf{bps,t} := \mathsf{x}', \mathsf{w}] \llbracket \{\!| \mathsf{x}', \mathsf{w,z} |\!\} \rrbracket \mathcal{TL}(\mathsf{cps})$
$\llbracket \{\!| \mathsf{x}', \mathsf{w,z} |\!\} \rrbracket \mathit{Struct}_{\mathsf{D}}[\mathsf{dps,s} := \mathsf{z,w}]) [\mathsf{z,x}' := \mathsf{csp,cps}'] \setminus \{\!| \mathsf{w} |\!\}) [\mathsf{cps}, \mathsf{cps}' := \mathsf{y,x}])$
$[\mathsf{x,y} := \mathsf{aps}', \mathsf{aps}]$

$=$ [14. Aplicação da definição de $\mathit{Struct}_{\mathsf{C}'}$]
$(\mathcal{TL}(\mathsf{aps}) \llbracket x, y \rrbracket \mathcal{TL}(\mathsf{aps}') \llbracket x, y \rrbracket \mathit{Struct}_{\mathsf{C}'}[\mathsf{cps}, \mathsf{cps}' := \mathsf{y}, \mathsf{x}]) [\mathsf{x}, \mathsf{y} := \mathsf{aps}', \mathsf{aps}]$

$=$ [15. Segundo a definição de $\mathit{Struct}_{\mathsf{A}'}$]
$\mathit{Struct}_{\mathsf{A}'}$

Durante a prova, três leis condicionais são utilizadas. A condição da Lei B.6 (Passo 11) é claramente satisfeita porque o processo T é obtido do protocolo T, que, na Lei 4.12, assume-se que é determinístico. A condição da Lei B.7 (passo 3,7,8) é satisfeita já que que nos passos onde é aplicada os processos não utilizam os canais x, w e y. A condição da Lei B.1 (Passos 10) é satisfeita já que a funções de renomeação $[\mathsf{bps,t} := \mathsf{x,w}]$ utilizadas na distribuição é injetiva.

4.4 CONCLUSÕES

Neste capítulo, apresentamos um conjunto de leis básicas em UML-RT e como elas podem ser utilizadas para justificar outras leis de transformações mais elaboradas, usualmente utilizadas na prática.

A partir da semântica *OhCircus*, foi possível também demonstrar a corretude destas leis. Esta corretude permite a inclusão das leis em um processo de desenvolvimento rigoroso, onde o desenvolvedor pode realizar suas tarefas de modelagem com uma maior garantia sobre seus resultados. Adicionalmente, a aplicação de leis com estas características pode diminuir o esforço desprendido durante o desenvolvimento em testes de verificação do modelo.

CAPÍTULO 5

NORMALIZAÇÃO E APLICAÇÃO DAS LEIS

Com o objetivo de analisar o poder de expressão do conjunto de leis proposto, neste capítulo mostraremos a abrangência dessas leis através de sua aplicação nos principais passos de uma estratégia de normalização de um modelo UML-RT em um modelo UML estendido com um único objeto ativo, responsável por todas as interações com o ambiente e por conservar o comportamento dinâmico do sistema modelado.

Este modelo UML estendido pode ser visto como uma *forma normal*, e, portanto, nossa estratégia pode ser vista como uma contribuição para uma estratégia de completude capturada por uma redução a esta *forma normal*, semelhante à estratégia apresentada em [9] para uma linguagem de programação orientada a objetos; a completude de conjuntos de leis para uma linguagem imperativa simples e para a linguagem concorrente Occam é apresentada em [46, 80], respectivamente. Nosso foco é em um conjunto de leis para diagramas de classe e estrutura do modelos; uma estratégia completa de normalização necessita de um grande número de leis para capturar transformações nos diagramas de estados, que está fora do escopo deste trabalho.

No processo de normalização as leis são aplicadas, usualmente, em um sentido inverso ao que são aplicadas durante um processo de desenvolvimento. Durante o processo de desenvolvimento, elas são aplicadas para quebrar abstrações, promover reuso e modularização, enquanto que em uma estratégia de redução a uma forma normal, as leis são aplicadas para gerar um sistema monolítico, cujo único objetivo é atestar o poder de expressão das leis.

Nas seções a seguir mostramos a aplicação das leis em seus dois sentidos (ver Figura 5.1). Na Seção 5.1, a medida que os passos de nossa estratégia de normalização são apresentados, mostramos a aplicação de algumas leis de transformação utilizadas na redução do modelo de sistema simplificado de automação industrial, apresentado anteriormente na Seção 2.3, em uma única cápsula. Enquanto na Seção 5.2, apresentamos a aplicação das leis durante o desenvolvimento do estudo de caso, partindo-se de um modelo abstrato de análise, extraido dos casos de uso do sistema, até um modelo concreto e mais elaborado de projeto em UML-RT , próximo à fase de implementação. Como tanto a estratégia de redução quanto o desenvolvimento do estudo de caso utilizam o mesmo modelo como ponto de partida, a combinação de ambos é capaz de ilustrar o conjunto de passos necessários para desenvolver um modelo monolítico representado por única cápsula em um modelo mais concreto e modularizado de projeto.

5.1 ESTRATÉGIA DE NORMALIZAÇÃO

Através do uso conjunto das leis propostas mostramos que todo o modelo pode ser representado por uma única cápsula MAIN, cujas portas públicas são utilizadas pelo

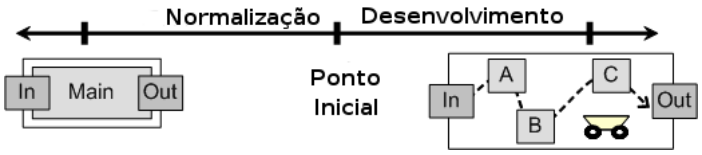


Figura 5.1. Os dois sentidos de aplicação das leis

sistema na comunicação com o ambiente externo. Inicialmente, assumimos que todas as demais cápsulas do modelo estar contidas no diagrama de estrutura de MAIN. Desta maneira, $\mathsf{Str_M}$ é expresso pelo diagrama hierárquico de estrutura de MAIN, e toda cápsula que reagir a um estimulo externo, interage direta ou indiretamente com alguma porta de MAIN. Após isto toda a hierarquia de $\mathsf{Str_M}$ é destruída e redundâncias no modelo são removidas, assegurando que todo o comportamento do modelo esta contido em suas cápsulas. Em seguida o objetivo é compor as cápsulas duas-a-duas, em uma única cápsula, até que reste uma única cápsula no modelo (MAIN).

Esta estratégia é efetuada através da seqüência de passos abaixo. As leis referenciadas em cada passo são exaustivamente aplicadas no modelo. Esta seqüencia de passos é executada ciclicamente, até que as leis não possam ser mais aplicadas.

i) Combinar portas de cápsulas. As cápsulas devem ter uma única conexão binária entre si, utilizando, cada uma delas, uma única porta (aplicação da Lei 4.13, da esquerda para a direita);

   Este passo é utilizado principalmente para adequar a comunicação entre as cápsulas ao padrão de aplicação das leis dos próximos passos. Além disto, é útil também para diminuir portas redundantes no modelo. Em nosso estudo de caso (Figura 2.2), todas as instâncias de cápsulas comunicam-se entre si através de uma única conexão binária, e por isto a aplicação desta lei é desnecessária.

ii) Eliminar hierarquia no diagrama de estrutura das cápsulas. Componentes de cápsulas (instâncias) situadas na estrutura de outra cápsula devem ser movidas para o nível de estrutura superior, que contém esta outra cápsula (aplicação das leis C.6, C.5 e 4.12, da direita para a esquerda).

   Em nosso estudo de caso, explicitamos a cápsula Main no modelo. Isto pode ser feito através da sua criação (aplicação da Lei 4.1 da esquerda para a direita) e inserção de uma instância man do tipo Main em $\mathsf{Str_M}$; a cápsula ProdSys pode ser movida para dentro do diagrama de estrutura de Main a partir da aplicação da Lei C.6 (da direita para a esquerda).

   Então, devemos eliminar as instâncias de cápsulas sin e son da hierarquia de ProdSys, resultando em arquitetura plana, sem a existência de subcápsulas, como é mostrado na Figura 5.2. Nesta figura, a Lei C.5 foi aplicada duas vezes para que a cápsula sin e son fossem retiradas da estrutura de sys.

iii) Mover comportamento para cápsulas. Todo o comportamento do modelo deve estar

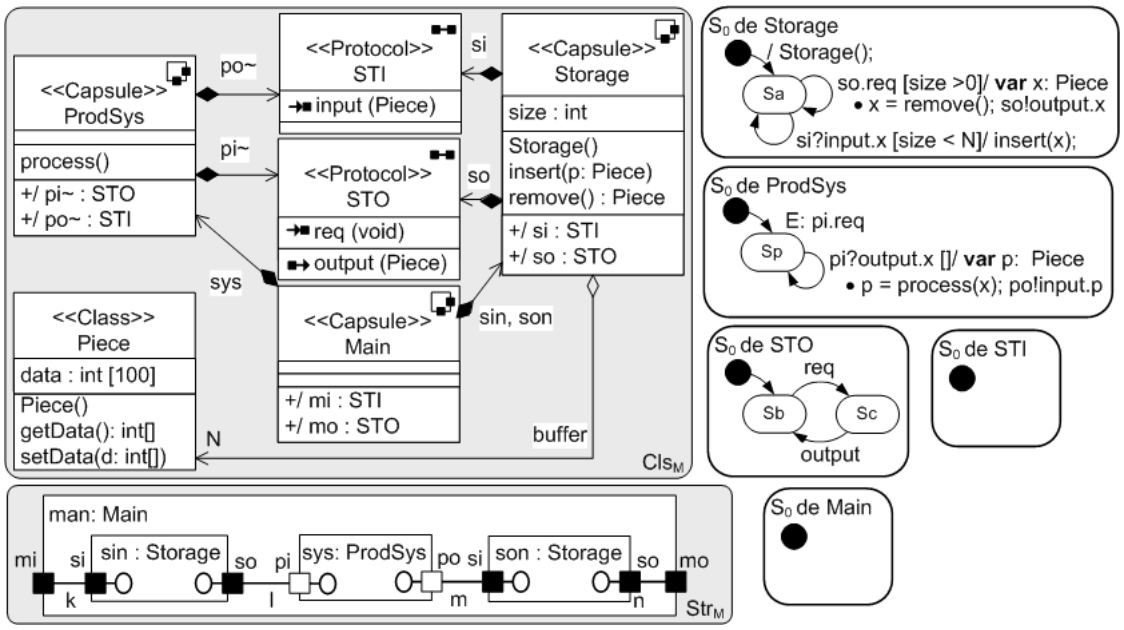


Figura 5.2. Passo ii da normalização do Estudo de Caso

situado dentro de suas cápsulas; protocolos não devem conter qualquer comportamento associado (aplicação da Lei 4.14, da esquerda para a direita);

No sistema de automação industrial, o único protocolo que possui um comportamento associado é STO. Este está associado às portas so e pi nas cápsulas Storage e ProdSys. Para que possamos aplicar a lei que move o comportamento do protocolo para uma cápsula, precisamos proteger as portas so e pi e criar *relay ports* que interliguem elas ao ambiente externo, através da Lei 4.10.

Em seguida, podemos mover o comportamento do protocolo STO para uma cápsula STOC no modelo, através da Lei 4.14. Toda as cápsulas que possuírem uma associação com STO, passaram a possuir uma nova subcápsula de tipo STOC em seu diagrama de estrutura que controla o fluxo de informações da conexão às portas do tipo STO, tal qual a porta o fazia. Observamos que o diagrama de estrutura é bastante complexo e que pode ser reduzido a uma forma mais amigável através da aplicação da Lei C.5 para desencapsular as instâncias do tipo STOC de dentro da estrutura de Storage e ProdSys, e removermos uma das instâncias redundantes que intermediam a comunicação entre sin e sys, através da aplicação da Lei 4.11, resultando no modelo da Figura 5.3.

Apesar do resultado obtido possibilitar a aplicação do próximo passo de normalização, notamos que as instâncias da cápsula STOC′ não restringem nenhuma comunicação na conexão; na realidade, as cápsulas Storage e ProdSys já obedecem a esta restrição, sempre enviando ou recebendo um sinal req antes de sincronizarem com o sinal output. Por isso, podemos remover as instâncias das subcápsula c e c′ que intermediam as conexões entre as instâncias sin, sys e son em Str_M, através da aplicação da Lei 4.11. O resultado é um modelo idêntico ao da Figura 5.2, exceto

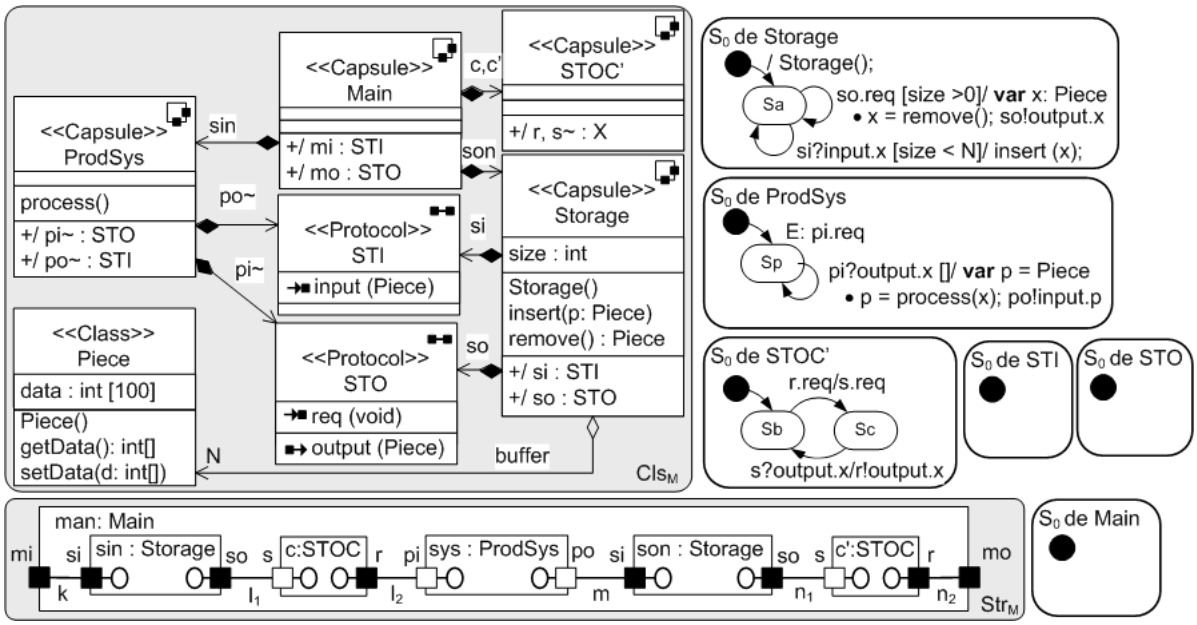


Figura 5.3. Passo iii da normalização do Estudo de Caso

que STO não possui uma máquina de estados associada.

iv) Compor cápsulas. Cada duas cápsulas que se comuniquem, devem ser encapsuladas no diagrama de estrutura de uma nova cápsula e compostas dentro deste diagrama. Para tanto, uma nova cápsula é criada a partir da aplicação da Lei 4.1, e uma instância de seu tipo é inserida no mesmo diagrama de estrutura das outras duas cápsulas que serão compostas (Lei 4.2). A duas cápsulas devem ser encapsulas dentro do diagrama de estrutura desta nova cápsula (leis C.6 e 4.12), e finalmente compostas dentro da nova cápsula (aplicação da Lei 4.20, da direita para a esquerda), reduzindo-se assim a quantidade de cápsulas do sistemas;

Em nosso estudo de caso, para que possamos compor as cápsulas sys e son devemos encapsulá-las em uma nova cápsula Controller (criada a partir da Lei 4.1), que mediará qualquer comunicação externa a sys ou a son. Para tanto, devemos aplicar a Lei C.6 para encapsular son dentro de con e depois aplicar a Lei 4.12 para encapsular sys em con, obtendo como resultado final o modelo exibido na Figura 5.4.

Após encapsularmos as duas cápsulas sys e son podemos compô-las em uma única cápsula através da aplicação da Lei 4.20. Cada região das máquinas destas cápsulas será associada a uma nova região na máquina de estados de Controller, descrevendo o comportamento ativo destas cápsulas através do paralelismo destas regiões. A Conexão entre as portas po e si continuara a ser descrita através de uma conexão interna ao diagrama de estrutura de Controller em $\mathsf{Str_M}$, como pode ser visto na Figura 5.5.

Após a primeira aplicação da Lei 4.20, notamos que o modelo (visto na Figura 5.5) se encontra em um formato que possibilita uma segunda composição de cápsulas,

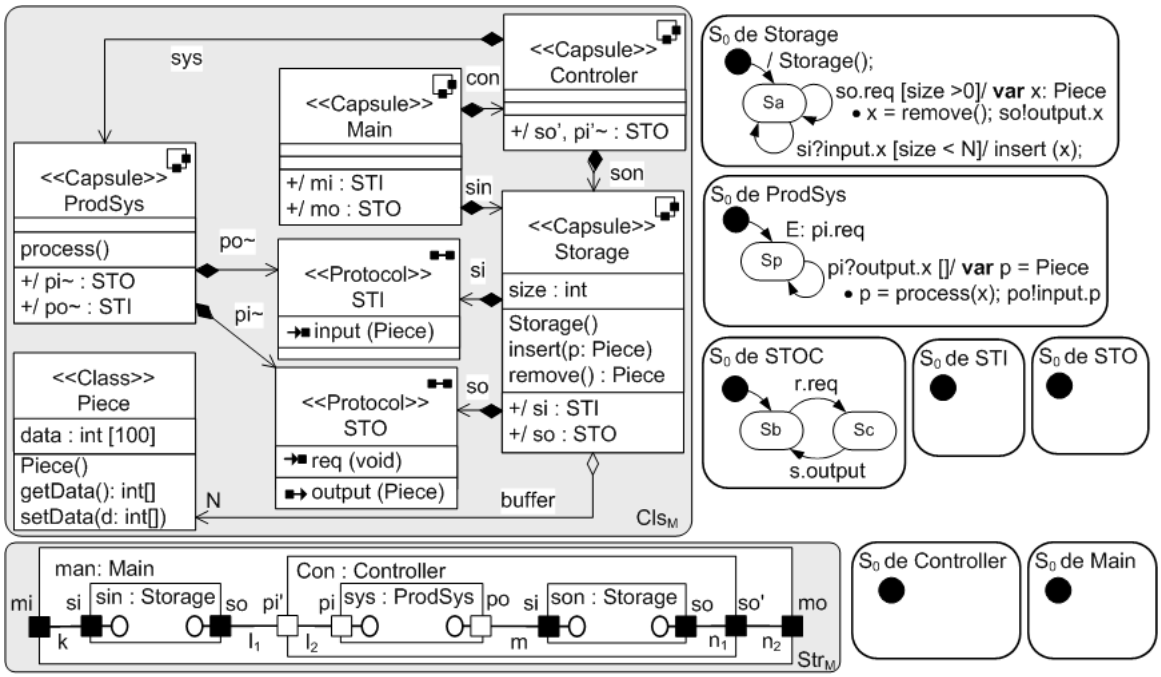


Figura 5.4. Primeiro modelo intermediário do passo iv da normalização do Estudo de Caso

desta vez sin e con. Notamos desta maneira que, nas cápsulas compostas, seus controladores são vistos como cápsulas normais do modelos, e que podem ser compostas novamente. Nesta última composição, a cápsula que contém sin e con é a propria cápsula Main que representa todo o sistema, resultando no fim de nosso processo de aplicação. O resultado desta segunda composição pode ser visto na Figura 5.6.

v) Remover declarações não utilizadas no modelo. Todas os elementos (cápsulas, protocolos, portas, métodos e atributos) não referenciados por outros elementos do modelo devem ser removidos (aplicação das Leis 4.1, C.1, 4.2, 4.5, 4.6, 4.3, C.2, 4.8 da direita para a esquerda);

   Em nosso estudo de caso, durante a aplicação das leis, diversas cápsulas deixaram de ter utilidade, como Storage, ProdSys, Controller e STOC. Apesar de não serem mostradas mais no modelo, nenhuma lei foi utilizada para remover explicitamente suas declarações. A remoção da declaração destes elementos do modelo pode ser feita através da aplicação da Lei 4.1 (da direita para a esquerda).

Esta estratégia visa unificar as cápsulas do modelo em uma única cápsula com todo o comportamento do modelo, como pode ser observado na redução de nosso estudo de caso na Figura 5.6. Desta forma o modelo resultante é formado por uma única cápsula que contém o comportamento e os dados de todo o modelo, mantendo as mesmas portas de comunicação, existentes anteriormente, com o ambiente externo. As classes do modelo não são afetadas.

A simplificação da máquina de estados da cápsula resultante, reduzindo a quantidade de regiões de S_0, não é foco deste trabalho. Isto ser feito através de leis que envolvem

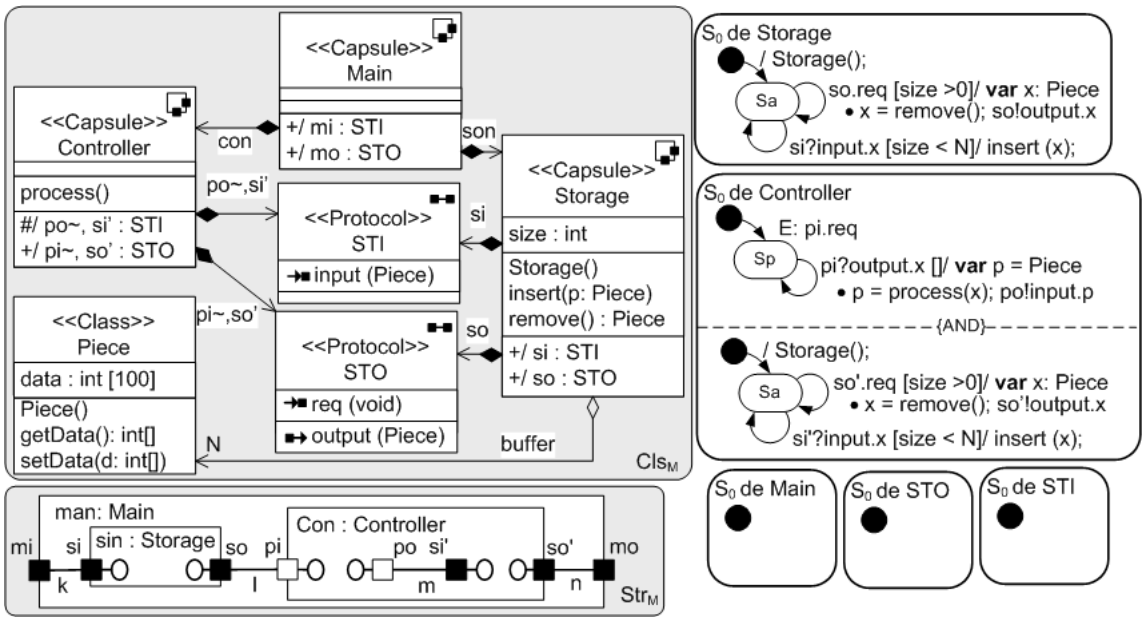


Figura 5.5. Segundo modelo intermediário do passo iv da normalização do Estudo de Caso

condições de prova na equivalência de diagrama de estados [34].

Este modelo poderia ser reduzido a uma única cápsula, sem a existência de classes no modelo. Neste caso, passos adicionais devem ser aplicados, como em [9], para reduzir todas as classes do modelo, e incluir as classes na capsula que possui uma associação para elas (aplicação da Lei 4.19, da direita para a esquerda). Note que, caso a cápsula resultante tenha um comportamento puramente passivo, ela pode ser transformada em uma classe, através da Lei 4.21. A classe final desempenharia, então, papel semelhante ao resultado da estratégia de redução em [9], que reduz um sistema orientado a objeto arbitrário na linguagem ROOL a uma única classe que representa todo o modelo.

Algumas leis não foram explicitamente referenciadas no processo de normalização. Por exemplo, as leis 4.15 e 4.17. Entretanto, estas leis são necessárias para, por exemplo, particionar uma cápsula e permitem a aplicação da Lei 4.20. Similarmente, outras leis são usadas na justificativa de leis derivadas, utilizadas no processo de normalização ou para preparar o modelo em um formato que possibilite a aplicação de outras leis, como as leis 4.10 e 4.15.

5.2 ESTUDO DE CASO

Com a finalidade de ilustrar a aplicação das leis propostas no Capítulo 4, apresentaremos, nesta seção, a modelagem sistematizada de um sistema simplificado de automação industrial (ver Seção 2.3). O sistema utilizado como estudo de caso já foi explorado em [53], onde a sua modelagem em UML-RT é apresentada, porém sem que nenhuma lei de transformação tenha sido proposta. Aqui, refinamos um modelo abstrato de análise, extraído dos casos de uso do sistema, em um modelo concreto e mais elaborado de projeto, próximo a uma implementação. Esta estratégia mostra que o conjunto de leis proposto

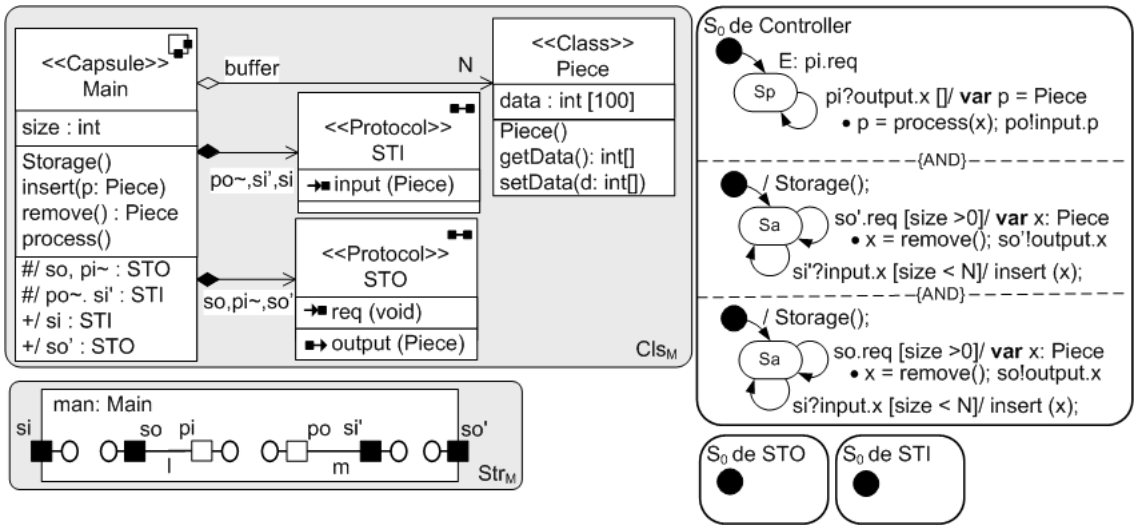


Figura 5.6. Modelo final da composição de cápsulas no passo iv da normalização do Estudo de Caso

pode justificar práticas informais de projeto. Em particular, desenvolvemos este estudo de caso seguindo algumas das práticas encontradas no Processo Unificado (RUP) [52], que são explicadas no decorrer do texto.

No sistema de automação industrial [53], toda a aplicação é responsável por processar um número de peças. O sistema consiste dos seguintes dispositivos: um repositório de entrada com peças não processadas, um repositório de saída com peças já processadas, algumas máquinas que processam as peças e alguns agentes de transporte. Cada peça não processada deve ser retirada do repositório de entrada, passar por todas as máquinas em um trajeto pré-definido e ser armazenada no repositório de saída. As máquinas e repositórios encontram-se distantes fisicamente, e necessitam requerer peças de trabalho a um elemento de transporte, chamado *holon* (um agente autônomo de transporte).

Por simplicidade, na Seção 2.3 não lidamos com todas as restrições da especificação, representando apenas o comportamento global do sistema, que servirá como ponto de partida para o desenvolvimento do estudo de caso. O objetivo, aqui, é detalhar esta modelagem, mostrando os passos necessários até que ele possua duas máquinas e um agente de transporte. Passos adicionais para se obter o projeto apresentado em [53] são discutidos no final da seção.

No RUP, modelos de análise e projeto são construídos a partir da visão de casos de uso do sistema. Para o propósito deste trabalho, as funcionalidades do sistema foram reduzidas a três casos de uso: inserir peças, recuperar peças e processar peças (veja Figura 2.1, na Seção 2.3). Inicialmente, o sistema se encontra inativo e sem nenhuma peça; à medida que o operador do sistema as insere, o sistema processa as peças e as disponibiliza para que o operador possa recuperá-las.

Em relação ao mapeamento de casos de uso no modelo de análise, baseamos nos no trabalho reportado em [101] para extrair um objeto ativo de cada caso de uso do sistema. Nesta abordagem, após a descrição dos casos de uso, o comportamento dos casos de uso

são mapeados em objetos ativos (cápsulas) que podem utilizar outros objetos passivos (classes), sintetizados a partir da descrição dos casos de uso. A especificação conjunta destas duas categorias de objetos forma o modelo de análise. O modelo que representa a visão de análise de nosso estudo de caso pode ser visto na Figura 5.2, onde a cápsula Main representa todo o sistema, a instância sin da cápsula Storage representa o caso de uso inserir peça, a instância son da cápsula Storage representa o caso de uso recuperar peça e a instância sys da cápsula ProdSys representa o caso de uso processar peça. A classe Piece é sintetizado a partir da descrição da troca de mensagens entre estas cápsulas. Na estrutura de Main, é descrita a interação entre estas cápsulas e, conseqüentemente, entre os casos de uso do sistema: uma única instância sys recupera peças de trabalho diretamente da instância sin, as processa, e as envia à cápsula son.

Diferente desta abordagem, no RUP, o modelo de análise é derivado a partir de casos de uso, ignorando o uso de objetos ativos. Cada caso de uso dá origem a uma classe de controle com todas as regras de negócio do caso de uso, classes de fronteira para cada interação do sistema com um ator (pessoa ou sistema) do mundo externo e classes de entidade com os dados utilizados pelas classes restantes associadas ao caso de uso. Através desta técnica, é possível separar dados das operações associadas às regras de negócio dos casos de uso, bem como das regras de comunicação do sistema com o ambiente externo.

Apesar das diferenças, a abordagem utilizada por nós pode alcançar, através do uso de nossas leis, resultados semelhantes à técnica utilizada no RUP. A cápsula que representa todo o caso de uso pode ser decomposta em várias outras cápsulas que representem os papeis das elementos de controle e fronteira sugeridos no RUP, enquanto entidades, por serem naturalmente passivas, continuariam a ser representados através de classes. Por fim, caso cada uma desta cápsulas (de controle e de fronteira) possuam um comportamento passivo, elas podem ser transformadas em classes, resultando em um modelo de análise semelhante ao proposto pelo RUP.

Em nosso estudo de caso, as cápsulas Storage e Prodsys possuem um comportamento ativo e representam o papel de controladores de seus respectivos casos de uso, enquanto Main serve de fronteira com o mundo externo, neste caso nenhuma regra de comunicação é atribuída a Main, que não possui por este motivo uma máquina de estados associada.

Após a criação de um modelo de análise, o próximo passo é encontrar uma *arquitetura candidata*[52]. Nesta atividade do RUP, identificamos os elementos que são abstrações chaves para interação entre os casos de uso e verificamos a possibilidade de reuso. Adotamos uma arquitetura de camadas simples onde a manipulação de dados é isolada das regras de negócio. Por esta razão inserimos uma coleção de dados para representar a coleção de Pieces em Storage. Esta coleção é extraída da cápsula Storage, através da Lei 4.19, em uma nova classe PieceCollection, como é mostrado na Figura 5.7.

A arquitetura candidata é incrementalmente aperfeiçoada através da *identificação de novos elementos de projeto*. Como desejamos que nosso modelo possua uma segunda máquina de processamento, decompomos a cápsula ProdSys em duas outras cápsulas. Para isto, inicialmente necessitamos criar uma nova partição em ProdSys como parte da responsabilidade do processamento de peças de trabalho. Dois novos métodos processA() e processB() são criados em ProdSys (Lei 4.3), de forma que execução seqüencial destes métodos seja equivalente a process(). Então, a ação p = process(x) é reescrita na

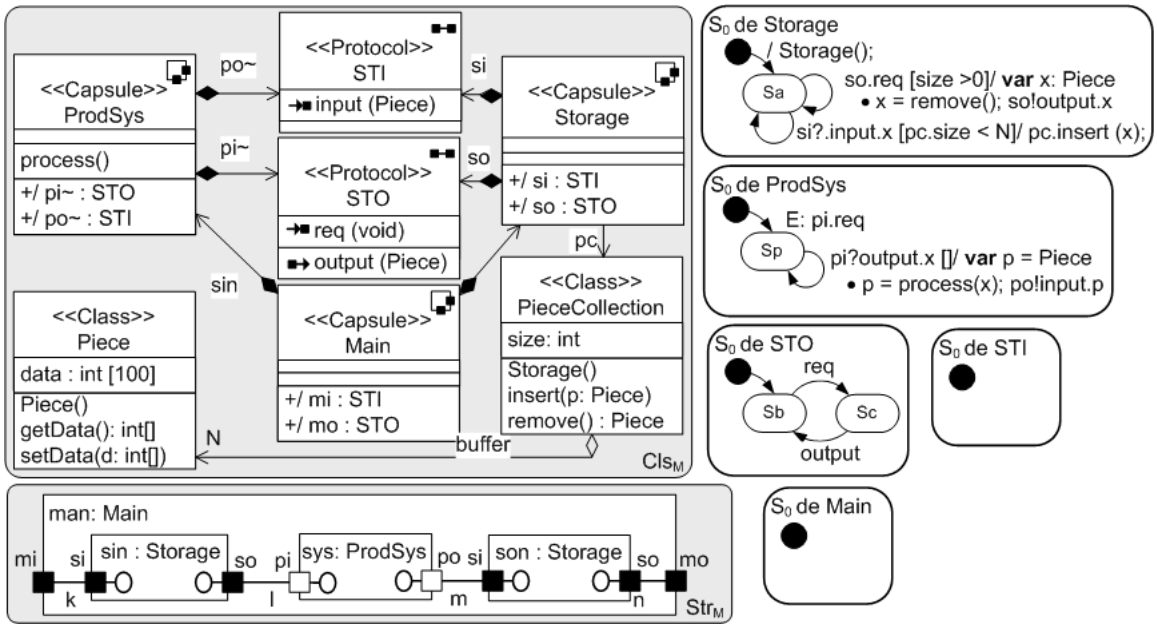


Figura 5.7. Arquitetura candidata do Estudo de Caso

ação equivalente **var** y = processA(x); p = processB(y) (Lei 4.16), e a execução da ação processB(y); po!input.p é isolada em uma nova partição (Lei 4.18). Finalmente, após possuirmos duas partições na cápsula ProdSys, esta pode ser decomposta em duas novas cápsulas (ProcessorA e ProcessorB) através da Lei 4.20, como é mostrado na Figura 5.8; por questão de clareza, algumas relações entre cápsulas e protocolos são omitidas no modelo.

Outro elemento crucial da especificação que falta ser identificado é o agente de transporte. Este elemento itermedia a comunicação entre instâncias das cápsulas Storage, ProcessorA e ProcessorB, que representam dispositivos fisicamente separados. Para dar origem a este elemento de transporte, criamos uma nova instância de cápsula em $\mathsf{Str_M}$ intermediando a comunicação entre as instâncias das cápsulas citadas, através da Lei 4.11. Em seguida, combinamos as instâncias duas a duas até formar uma especificação inicial da cápsula que representará o agente de transporte, chamada de Holon. Utilizaremos para combinar estas cápsulas a Lei C.7, uma versão simplificada da Lei 4.20 que não requer que as partições se comuniquem. O modelo resultante da aplicação destas leis pode ser visto na Figura 5.9

Outras mudanças podem ser realizadas no modelo como, por exemplo, a união das portas pi e pd, e pb e po nas cápsulas ProcessorA e ProcessorB, respectivamente, através da Lei 4.13. Não realizamos estas transformações, pois não diminuiria o potencial de reuso do sistema e não propiciaria melhoria significativa do modelo.

Outros passos de modelagem que podem ser realizado é a criação de um novo agente de transporte (instância de Holon) em sincronia com hts, através da aplicação incremental da Lei 4.8 a todas as portas de hts, e o refinamentos das máquinas de estados de ProcessorA, ProcessorB e Holon para que apenas um agente efetue o transporte de peças a cada

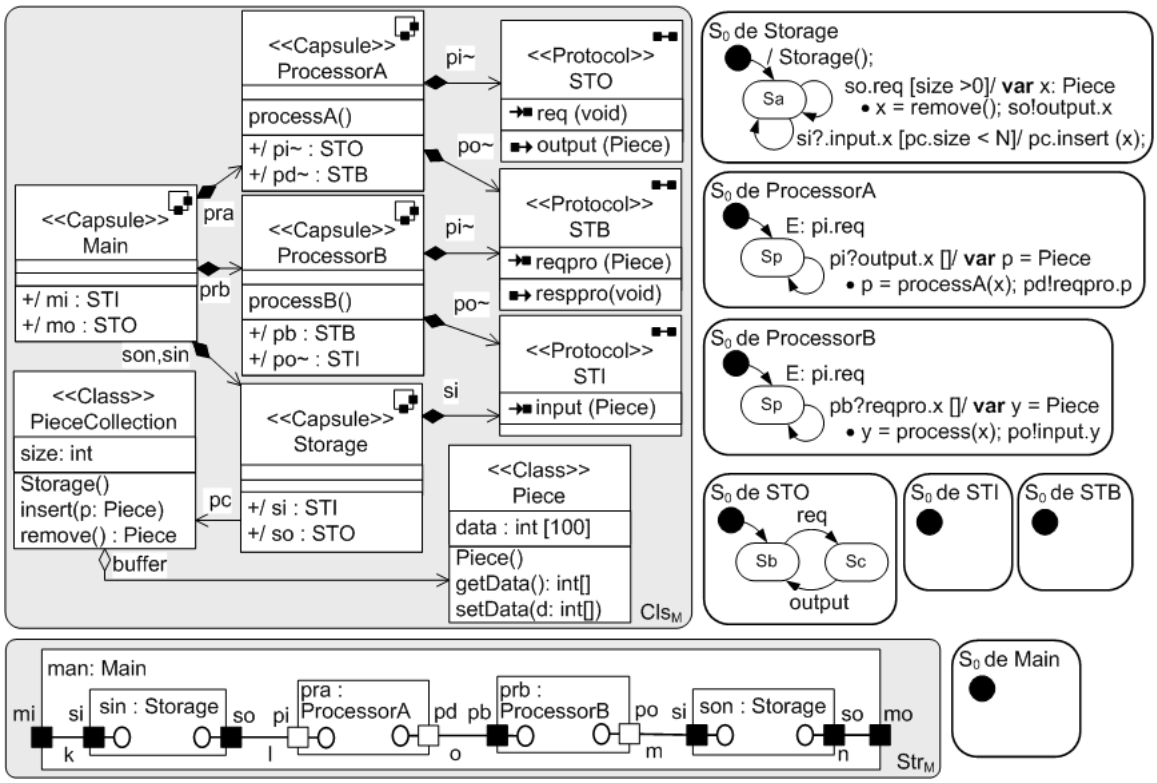


Figura 5.8. Identificação de ProcessorA e ProcessorB no Estudo de Caso

requisição de uma máquina (ProcessorA e ProcessorB), tal qual o modelo existente em [53]. Desta maneira, as máquinas de processamento negociariam o agente de transporte que possuíssem menor custo (de tempo e distância) para transporte das peças entre outras máquinas do sistema ou repositórios.

Além disto, podemos alterar a máquina de estados de Holon para que ela possua uma partição encarregada do percurso (entre as máquinas e repositórios) que as peças devem seguir no sistema e outra partição encarregada pelo transporte propriamente dito. Através da Lei 4.20, a partição com o roteiro daria origem a uma nova cápsula Driver que pertenceria à camada de negócio da arquitetura do sistema, enquanto a partição responsável pelo transporte daria origem a uma cápsula Transporter pertencente à camada de comunicação da arquitetura do sistema. Não nos atemos a estes passos da modelagem por eles estarem essencialmente relacionados a mudanças nos diagramas de estados do modelos, cujas leis não são o enfoque deste trabalho.

5.3 CONCLUSÕES

Neste capítulo, ilustramos a aplicabilidade das leis propostas em um estudo de caso real e discutimos, brevemente, como elas podem ser inseridas em um processo de desenvolvimento. Notamos que, devido a nem todos os processos de desenvolvimento incluírem a noção de objetos ativos, algumas atividades do processo devem ser adaptadas, sem que isto altere o fluxo do processo.

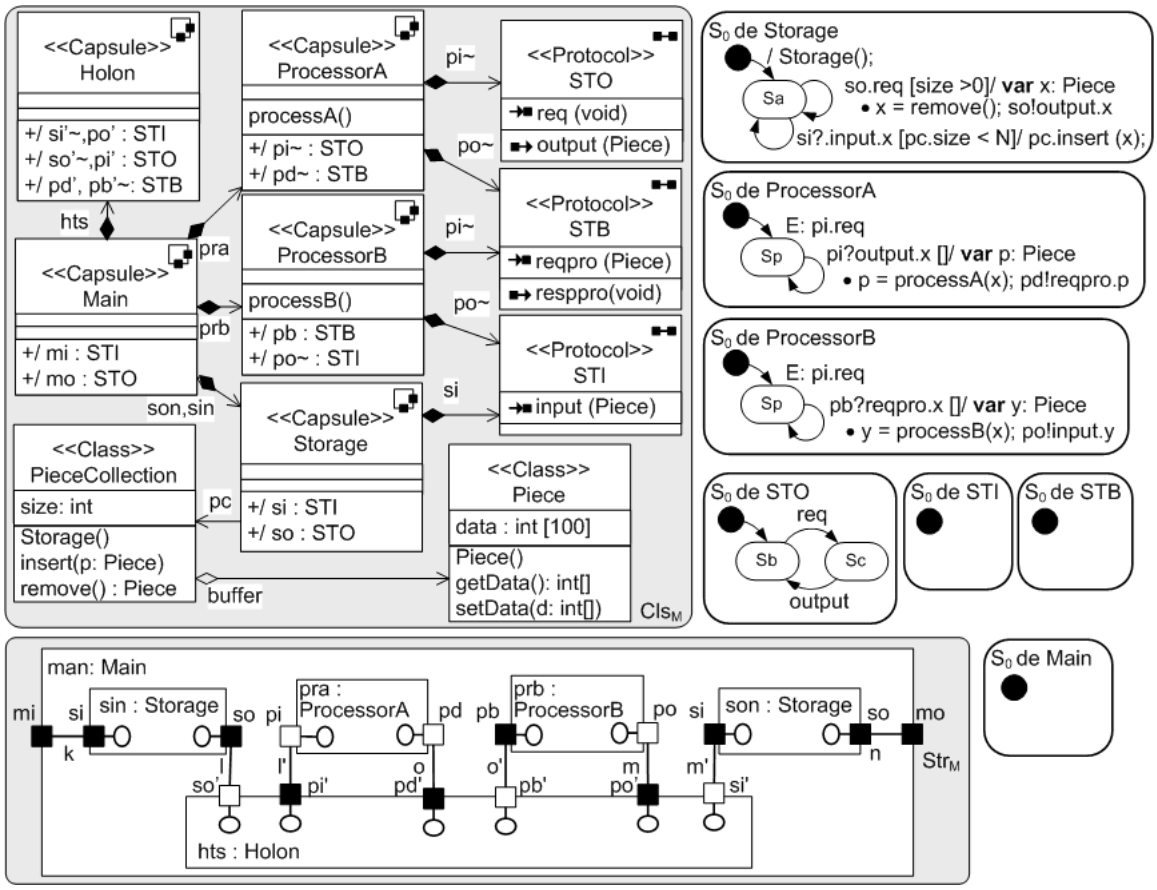


Figura 5.9. Identificação de Holon no Estudo de Caso

Outro ponto observado é que, devido ao nosso enfoque em leis estruturais do modelo, certos passos de modelagem não são completamente cobertos, principalmente quando estes requerem leis que alterem o diagrama de estados de modelo. Apesar disto, a completude de nossas leis quando à visão estrutural do modelo não é comprometida, como foi mostrado no estudo de caso.

Além disso, demonstramos a abrangência deste conjunto de leis através dos principais passos de uma estratégia de normalização. Esta estratégia foi ilustrada pela normalização do modelo de análise do sistema de automação industrial até uma *forma normal*, representada por um modelo UML estendido com uma única cápsula.

Como tanto a estratégia de redução quanto o desenvolvimento do estudo de caso utilizam o mesmo modelo como ponto de partida, a combinação de ambos é capaz de ilustrar o conjunto de passos necessários para desenvolver um modelo monolítico representado por única cápsula em um modelo mais concreto e modularizado de projeto.

CAPÍTULO 6

CONCLUSÕES

Neste trabalho, mostramos a possibilidade de um desenvolvimento rigoroso em UML-RT através do uso de leis de transformação de modelos. Estas leis são apresentadas de uma maneira incremental, onde um conjunto de leis básicas é utilizado para justificar leis mais elaboradas e usualmente utilizadas na práticas. Em contraste com a definição usual de regras de transformação, estas leis são definidas segundo uma semântica formal e garantem a preservação do comportamento do sistema modelado.

Para provarmos a corretude destas leis, necessitamos atribuir uma semântica formal a UML-RT, através de seu mapeamento para a linguagem formal *OhCircus*, enfocando os elementos que UML-RT adiciona a UML (cápsulas, protocolos, portas e conexões) e considerando uma visão integrada do modelo envolvendo digramas de classes, estado e estrutura. Baseado neste mapeamento semântico, e na semântica das leis de *OhCircus*, nós mostramos que é possível (e relativamente simples) provar tais leis de transformação de modelos, à medida que contribuímos de uma forma original à formalizarão de transformações em modelos UML-RT.

Apesar de o destino de nossa tradução ser uma linguagem de especificação (ao invés de um modelo matemático mais padronizado), a notação de *OhCircus* inclui as notações de CSP e Z, ambas formalismos bem conhecidos e maduros. Além disto, a combinação destas notações em *OhCircus* é formalmente caracterizada com base na *Unifying Theories of Programming*.

Nossas leis são apresentadas de uma forma incremental; iniciamos com leis básicas que capturam isoladamente propriedades fundamentais dos elementos de UML-RT. A partir destas, nós propomos leis mais elaboradas, que formalizam transformações utilizadas na prática. As leis lidam com diversos diagramas do modelo (classe, estado e estrutura) e sua apresentação deixa explícito os efeitos de sua aplicação sobre estes diagramas, diferentemente de outras leis que consideram estes diagramas isoladamente.

Considerando os elementos que UML-RT adiciona a UML, o conjunto de leis proposto é abrangente, e pode ser observado como parte de uma semântica axiomática destes elementos. A noção de abrangência destas leis é baseada na demonstração de que um modelo arbitrário UML-RT pode ser reduzido a um modelo UML estendido com uma única cápsula responsável por todas as interações com o ambiente. Este modelo UML estendido pode ser visto como uma *forma normal*, e, portanto, nossa estratégia pode ser vista como uma contribuição para uma estratégia de completude capturada por uma redução a esta *forma normal*, semelhante à estratégia apresentada em [9] para uma linguagem de programação orientada a objetos. Apesar de não termos apresentado uma estratégia de redução em todos os detalhes, mostramos que as leis propostas são abrangentes o suficiente para cobrir transformações que incluem cápsulas e protocolos em diagramas de classe e de estrutura.

Outro ponto importante é a relação entre classes e cápsulas, como capturado pela Lei 4.21 que transforma, sob determinadas condições, um elemento no outro. Mostramos, também, a aplicação das leis durante a execução de atividades de modelagem de análise e projeto do RUP através de um estudo de caso. Como explicado no Capítulo 4, transformações consistentes entre classes no diagrama são livremente permitidas, desde que estas não interfiram com a interface esperada por cápsulas que invoquem métodos destas classes. Isto acontece porque a comunicação entre cápsulas e classes é através da chamada de métodos e porque cápsulas não compartilham classes entre si. Assim do ponto de vista destas classes, cápsulas são simplesmente elementos externos que invocam seus serviços.

Apesar de lidarmos, neste trabalho, com um subconjunto da linguagem UML-RT e o fato desta linguagem não ter sido criada por organizações de padronização (como a OMG), sendo mantida pelas empresas que suportam suas ferramentas [19], vários conceitos desta linguagem estão diretamente presentes em outras linguagens para a descrição de arquiteturas ou de componentes, como ADLs ou UML 2.0 (versão ainda não finalizada pela OMG). Assim, todas as contribuições que possam ser incorporadas ao desenvolvimento usando UML-RT podem, em princípio, ser adaptadas para estas linguagens.

6.1 TRABALHOS RELACIONADOS

Relacionado à formalização de modelos visuais orientados a objetos, duas abordagens principais podem ser identificadas [30]: *suplementar* e *integração de métodos*. A primeira substitui linguagens naturais de documentação por expressões mais formais. A própria OCL [70] é um exemplo desta abordagem, ela é utilizada em conjunto com UML com o propósito de anotar o modelo. Na última abordagem, técnicas de modelagem orientadas a objetos são tornadas mais precisas e receptivas a análises rigorosas através de sua integração com notações de especificação formal, mapeando o modelo informal em um domínio formal e bem definido. Utilizamos a última abordagem por ela ter, em princípio, um maior apelo prático, já que o desenvolvedor pode realizar suas tarefas de modelagem na notação que possuir maior familiaridade sem explicitamente lidar com formalismos, porém baseados em uma notação única que possibilita a prova de vários aspectos do modelo. Um exemplo bem sucedido da integração de métodos formais com linguagens visuais é a linguagem Alloy [50] que descreve a declaração de classes, seus relacionamentos e propriedades em um diagrama similar ao diagrama de classes em UML. Contudo, apesar de Alloy possuir uma semântica bem definida, apenas um subconjunto das visões do modelo é expresso nesta linguagem.

Na literatura, diversos esforços têm se referido aos problemas de integração de modelos UML com linguagens formais; existem diversas abordagens de especificação [12, 28, 29, 78, 69], como a combinação de UML com Z [95] ou com CSP[79]; tipicamente, cada uma destas contribuições tende a se concentrar em uma única visão do modelo (como diagrama de estados [69] ou diagrama de classes [12, 28, 78]). Embora alguns trabalhos como, por exemplo, [58, 66] utilizem uma notação uniforme para descrever um mapeamento que considere a estrutura e o comportamento do modelo, a apresentação é informal e baseada em exemplos.

A formalização de diagramas de classes utilizando a notação Z é apresentada em [12, 28, 29]; nestas abordagens, classes, associações e o próprio modelo são mapeados em esquemas em Z. O esquema que representa o modelo possui a visão extensional de suas classes, com o conjunto de todos os objetos do modelo, e as associações do diagrama de classes. Desta maneira, quaisquer propriedades sobre as classes ou associações podem ser incorporadas como predicados deste esquema. Comparações detalhadas desta abordagem e de outras que envolvem extensões de Z com orientação a objetos (Object-Z) [93] são discutidas em [4]. Estas abordagens para Object-Z usam, em geral, conceitos semelhantes à abordagem utilizada em Z, porém através de uma representação mais sucinta do diagrama de classes, por representarem diretamente algumas das características de orientação a objetos de UML.

O mapeamento de classes assumido para *OhCircus* (uma simplificação do encontrado em [10]) é semelhante ao encontrado em [78] para Object-Z, onde classes são mapeadas diretamente em construtores da linguagens, associações são representadas pela instanciação de atributos adicionais nas classes envolvidas na relação (localizados de acordo com a navegação da associação) e todas as restrições do modelo devem ser expressos como invariantes destas classes. O trabalho reportado em [78] também mostra o poder de expressividade de Z em relação a propriedades do modelo descritas OCL, corroborando com outros trabalhos que comparam Z e OCL [49].

Com respeito à semântica formal de diagramas de estados, o mapeamento destes diagramas em processos em CSP é apresentado em [69, 75]. Em [69], máquinas de estados são mapeadas em processos CSP. Apesar de expressar a hierarquia de máquinas de estados, este trabalho não lida com paralelismo ou estados com história. Além disto, por utilizar somente CSP, esta abordagem ignora ações que envolvam operações sobre os componentes de estados ou métodos de um elemento declarado no modelo.

Em [75], máquinas de estados são mapeadas em processos CSP e esquemas em Z, utilizando a notação unificada CSP-OZ. Cada estado possui uma representação semântica através de um esquema em Z, enquanto os eventos de comunicação e a composição destes estados são realizados através de processos em CSP. Desta maneira, os principais conceitos de máquina de estados, como hierarquia, concorrência e estados com historia são mapeados em uma notação única que permite uma integração com mapeamentos de diagrama de classes. Apesar de lidar com hierarquia de estados, transições complexas (por exemplo, transições entre diferentes níveis da hierarquia que atravessam a borda de estados compostos) não são cobertos neste trabalho; as únicas transições expressas em estados compostos são transições de grupo. Apesar destes tipos de transições complexas permitirem a violação do encapsulamento de estados compostos, elas são necessárias para indicar diferentes estados de aceitação ou falha da submáquina de estados de um estado composto [92]; sendo, esta propriedade crucial em nossas leis para simplificar a representação de submáquina de estados em um único estado.

Por esta razão, optamos por estender o trabalho [69] utilizando a notação de *OhCircus*. Adaptações foram necessárias para permitir ações sobre atributos e métodos das cápsulas, e a inserção de paralelismo através de *And-States*. Neste sentido, o mapeamento de estados compostos foi completamente alterado para permitir os diversos tipos de transições sobre *And-States*, utilizando uma abordagem similar à encontrada em [75]

na utilização de eventos de comunicação para notificar a saída destes estados compostos para os seus sub-estados (ver Seção 3.2.2); estados historia não foram abordados neste trabalho por não serem necessários em nossas leis, já que não focamos em leis para diagramas de estados.

Relacionado à formalização de UML-RT, limitações similares às encontradas na integração de métodos formais e UML podem ser encontrada em trabalhos [32, 27, 24] que formalizam UML-RT utilizando CSP. Eles focam na tradução da visão estrutural de UML-RT em CSP [32], e consideram a representação comportamental das cápsulas [27, 24] somente parcialmente. Por lidarem isoladamente com as visões de um modelo UML-RT, estes trabalhos apresentam deficiências quando analisamos as diversas visões do modelo em conjunto, como discutido adiante.

Em [32] a visão estrutural (contido no diagrama de estrutura) das portas, cápsulas e conectores são representados por classes e processos em CSP-OZ, e a renomeação de canais de comunicação para o mesmo nome destas portas é utilizado para a composição de cápsulas, sendo esta a estratégia utilizada por nós na composição (conexão) de cápsula em um diagrama de estrutura. Nenhuma referência a outros elementos declarados no diagrama de estado ou classes é feita neste trabalho. Apesar de em [66] os trabalhos reportados em [32] e em [75] serem referenciados conjuntamente em uma estratégia para a tradução dos diagramas de estrutura, estados e classes de UML-RT para CSP-OZ, o mapeamento é feito de maneira informal e através de exemplos, sem nenhuma referência para: como o comportamento dos protocolos associados às portas de uma conexão é incorporado na comunicação entre as cápsulas; como regiões de *AND-State* em um diagrama de estados de uma cápsula devem ser sincronizadas entre si ou com outras máquinas de estados do modelo; ou quais condições são necessárias para a composição de uma cápsula a outra.

O trabalho reportado em [27, 24] apresenta brevemente algumas noções que podem ser utilizadas como base para o mapeamento de UML-RT em CSP, porém baseado somente nestas noções parece ser difícil o mapeamento de sistemas complexos em todos os seus aspectos. Estes trabalhos citam brevemente máquinas de estados simples de cápsulas [27] e protocolos [24] e contribuem com o mapeamento do diagrama de estruturas através do mapeamento de conectores como processos CSP [27], apresentando ainda assim os mesmos problemas atribuídos aos trabalhos [66, 32, 75], além disto, não lidam com a hierarquia de diagramas de estrutura e estado.

Relacionado a leis para UML, existem diversos trabalhos [41, 28, 29, 38, 55, 96] que consideram transformações para classes e diagramas de estados que consideramos complementares ao nosso trabalho. Diferentemente das leis apresentadas no Capítulo 4, as leis apresentadas nestes trabalhos lidam com um modelo mais abstrato, não considerando associações como atributos de classes no diagrama de classes ou ações no diagrama de estados. Além disto, as leis propostas nestes trabalhos consideram restrições globais ao modelo, como aquelas que envolvem o número de instâncias de classes em todo o sistema; em nosso trabalho, propriedades são restritas aos invariantes das classes e cápsulas.

Em [28, 29, 37] leis sobre diagramas de classe são consideradas incluindo uma semântica formal para UML, utilizando Alloy [37] e Z [28, 29], bem como a prova da validade destas leis; no trabalho reportado em [41], regras de transformação são apresentadas baseadas

na linguagem semi-formal OCL. Destacamos o trabalho reportado em [37] por apresentar um conjunto abrangente de leis básicas para Alloy, e que podem ser facilmente aplicadas a diagramas de classe. Semelhante ao nosso trabalho, em [37] leis mais elaboradas e usualmente aplicadas na prática são derivadas da composição de leis mais básicas. Como dito anteriormente, estes trabalhos interpretam o modelo em um nível mais abstrato, onde associações não são necessariamente direcionais nem tampouco geram atributos nas classes que relacionam. Por esta razão, estas leis são mais gerais que as nossas e permitem descrever uma quantidade maior de propriedades sobre as classes do modelo. Contudo, leis para classes não são o foco de nosso trabalho e somente são utilizadas para complementar as leis que propomos para os elementos que UML-RT adiciona a UML.

As transformações de modelos UML apresentadas em [55, 96] contemplam diagramas de estado, em adição à diagramas classe, porém os efeitos em cada diagrama são considerados isoladamente, e portanto qualquer interferência entre aspectos estáticos e dinâmicos são ignorados, diferentemente da nossa abordagem que considera estes efeitos simultaneamente. Em particular, [55] apresenta uma semântica formal para UML, OCL e *statecharts* em termos de *Real-time Action Logic* (RAL), e apresenta e prova algumas transformações. Em [96], uma relação entre os conceitos de *refactoring* para código e transformações de modelos são apresentadas de maneira informal, mostrando a preservação de comportamento em algumas transformações para diagramas de estado e de classe.

Relativo a leis de transformação para UML-RT, poucos trabalhos [85, 25, 63] foram encontrados; em sua maioria, apenas a utilidade destas leis são mencionadas em um processo de desenvolvimento passo-a-passo [85] ou na verificação de consistência do modelo durante sua evolução [25], utilizando UML-RT.

Em [85], é proposto um processo de desenvolvimento incorporando noções de refinamento, baseadas em princípios particulares de *refinamento comportamental de interfaces* do componentes (cápsulas) da aplicação e *incorporação de tempo*. O primeiro tipo de transformação permite que as interfaces de uma cápsula sejam alteradas levando-se em consideração sua máquina de estados, enquanto o segundo tipo sugere que sincronismo e aspectos temporais sejam ignorados nos estágios iniciais do desenvolvimento, e que o modelo, inicialmente assíncrono, seja refinando para adiciona-los durante o desenvolvimento. Não consideramos em nosso trabalho aspectos temporais, assincronismo, ou mesmo refinamentos. Apesar disto, o *refinamento comportamental de interfaces* propõe que o comportamento observado do componente não deve ser alterado, sendo semelhante a algumas de nossas leis que lidam com a visão estrutural do modelo, sem alterar seu comportamento, como leis para a conexão (Lei 4.8), substituição (Lei 4.9) e decomposição (Lei 4.20) de cápsulas (ver Capítulo 4).

Em [25], o *principio da localidade* é explorado, formalizando a evolução de modelos através de leis básicas de transformação; este trabalho também analisa o efeito destas transformações na consistência de algumas propriedades do modelo, como *deadlock* e *compatibilidade de interface*. Nenhum destes trabalhos, todavia, apresenta a aplicação das leis de uma maneira sistemática, com fizemos aqui. Apesar de algumas leis serem introduzidas (como, por exemplo, a introdução de uma cápsula, porta ou conexão), nenhum padrão para os termos à esquerda e à direita da lei, nem tampouco as condições

explícitas para suas aplicações, são apresentados. Em [63], uma única lei para a delegação de responsabilidades de uma cápsula a subcápsulas é apresentada, tal qual a Lei 4.20. A diferença existente entre as duas é que a lei proposta em [63] considera a possibilidade de uma das partições da cápsula não ser movida para uma nova subcápsula, sendo este apenas um dos passos da derivação da Lei 4.20 (ver Seção 4.2).

Sob um determinado ponto de vista, propomos leis algébricas que descrevem propriedades básicas sobre os elementos que UML-RT adiciona a UML, de forma semelhante a outras leis algébricas para alguns paradigmas da programação, como, por exemplo, o orientado a objetos [9], o imperativa [46] e o concorrente [80], ou a linguagens de modelagem como Alloy [37]. Apesar de UML-RT possuir algumas características encontradas em alguns deste paradigmas, nossa noção de equivalência não pode ser baseada diretamente em nenhum deles, e por esta razão utilizamos em nossas leis uma noção de equivalência baseada na semântica de *OhCircus*, que por sua vez é baseada na UTP. Nossa estratégia de redução a uma forma normal também é similar a alguns destes trabalhos [9].

No contexto do desenvolvimento utilizando *Model Driven Architecture* (MDA) [65], nossas leis são classificadas como leis horizontais, por não alterarem o nível de abstração do modelo quanto a sua plataforma. Além disto, os modelos utilizados por nós encontram-se em um nível intermediário de abstração da plataforma, por incorporarem uma visão próxima da implementação; nossos modelos utilizam uma linguagem para a descrição de ações no diagrama de estados e lidam com associações como atributos. Apesar disto, o modelo possui ainda assim um grande nível de abstração e pode ser mapeado para diversas plataformas específicas, que descrevam, por exemplo, a tecnologia de comunicação das cápsulas e a representação destas na linguagem destino. Além disso, nosso mapeamento para *OhCircus* pode ser visto como uma transformação para um modelo transversal com o intuito de verificar e validar propriedades do modelo. Nossas leis ainda podem ser aplicadas em um processo de desenvolvimento tradicional, como o RUP, que utilize UML-RT [5, 42, 1].

6.2 TRABALHOS FUTUROS

Apesar de termos proposto um conjunto abrangente de leis básicas, e discutido sua abrangência através de uma estratégia de redução a uma forma normal, a completude deste conjunto não foi provada formalmente. As leis cobrem os principais passos da estratégia, mas leis adicionais (principalmente referente a transformações de diagrama de estados) são necessárias.

Além disto, derivações de novas leis de maior granularidade podem ser necessárias para facilitar as tarefas realizadas pelo desenvolvedor durante a modelagem de sistemas. Tais leis devem ser encontradas a partir da análise de um conjunto amplo de estudos de caso e padrões de projeto para UML-RT. Além disto, estudos sobre a incorporação de modelos e transformações de modelos em processos de desenvolvimento [60, 51, 2] ainda se encontram em um estado bastante incipiente. Maiores estudos nesta área poderão revelar novas aplicações de transformação de modelos, e , conseqüentemente, a definição de novas leis derivadas a partir das leis básicas de transformação.

Outro trabalho futuro é expandir o escopo de nossas leis para lidar com características

encontradas comumente em linguagens orientadas a objetos, como a herança de classes e cápsulas, e introduzir semântica de referência para classes. Para isto, além de necessitarmos atribuir uma semântica para herança comportamental de cápsulas, ainda não bem definida, precisaríamos formalizar tais características de orientação a objetos na semântica para *OhCircus*, que ainda está em processo de definição. Outro campo de estudo que merece especial interesse é o desenvolvimento orientado à agentes, onde se verifica uma grande semelhança entre o conceito de cápsulas e agentes. Um exemplo de uma metodologia que utiliza UML-RT para modelar agentes é encontrado em [14].

Observamos, também, a necessidade de verificar a relação entre nossas leis de modelagem e leis de programação, especialmente em linguagens de programação que lidam com objetos ativos, como JCSP [97]. Isto possibilita o desenvolvimento rigoroso em UML-RT lidar com três níveis distintos de abstração do sistema, utilizando modelos em UML-RT, especificações em *OhCircus* e implementações em JCSP, baseado em trabalhos que mostram o mapeamento entre estas linguagens [72, 74]. Este desenvolvimento seria semelhante à abordagem em [66], que mapeia UML-RT em Java e CSP-OZ, porém introduzindo leis de transformação como parte essencial de seu processo. Adicionalmente, a mecanização destas leis de transformação facilitariam sua utilização prática e automação de tarefas de modelagem.

Por fim, acreditamos que as contribuições deste trabalho podem ser incorporadas no desenvolvimento baseado em componentes ativos em UML 2.0, que traz vários dos conceitos de UML-RT em sua definição, contribuindo, assim, para a integração de métodos formais com uma linguagem de modelagem padronizada e amplamente utilizada pela comunidade.

Desta maneira, uma provável lista de trabalhos futuros inclui:

- Prova formal de todo o conjunto de leis básicas;
- Formalização da estratégia de redução do conjunto de leis básicas;
- Emprego de novos estudos de casos, e a derivação de padrões de projeto para UML-RT;
- Estudo do relacionamento de leis de transformação de modelos em UML-RT e leis para linguagens de programação;
- Estudo da integração de leis de transformação em processos de desenvolvimento rigorosos, que incorporem linguagens formais, de modelagem e de implementação;
- Mecanização das leis de transformação de modelos;
- Incorporação das contribuições deste trabalho a com outras linguagens de modelagem, como UML 2.0 e Linguagens de Descrição de Arquitetura.

APÊNDICE A

SINTAXE COMPLETA DE OHCIRCUS

Program	::=	OhCircusParagraph*
OhCircusParagraph	::=	Paragraph
	|	ChannelDefinition | ChanSetDefinition
	|	OhProcessDefinition | ClassDefinition
ChannelDefinition	::=	$\mathbf{channel}$ CDeclaration
CDeclaration	::=	SimpleCDeclaration | SimpleCDeclaration; CDeclaration
SimpleCDeclaration	::=	N^+ | N^+ : Expression | Schema$-$Exp
ChanSetDefinition	::=	$\mathbf{chanset}$ N $==$ CSExpression
CSExpression	::=	$\{\!\|\}$ | $\{\!\|$ N^+ $\|\!\}$ | N
	|	CSExpression $\cup$ CSExpression | CSExpression $\cap$ CSExpression
	|	CSExpression $\setminus$ CSExpression
OhProcessDefinition	::=	$\mathbf{process}$ N $\widehat{=}$ [$\mathbf{extends}$ N] Process
Process	::=	$\mathbf{begin}$ PParagraph*
		[stateSchema$-$Exp]
		PParagraph*
		[$\bullet$ Action]
		$\mathbf{end}$
	|	CompProcess
CompProcess	::=	N | Proc; Process | Process $\setminus$ CSExpression
	|	Process $\Box$ Process | Process $\sqcap$ Process | Process $\|\|\|$ Process
	|	Process $\llbracket$ CSExpression $\rrbracket$ Process | Process[N^+ := N^+]
	|	Declaration $\odot$ Process | ; Declaration $\odot$ Process
	|	$\Box$ Declaration $\odot$ Process | $\sqcap$ Declaration $\odot$ Process
	|	$\|\|\|$ Declaration $\odot$ Process | Declaration $\odot$ Process$\lfloor$Exp$^+\rfloor$
	|	$\|$ Declaration $\llbracket$ CSExpression $\rrbracket$ $\odot$Process
	|	Declaration $\bullet$ Process | ; Declaration $\bullet$ Process
	|	$\Box$ Declaration $\bullet$ Process | $\sqcap$ Declaration $\bullet$ Process
	|	$\|\|\|$ Declaration $\bullet$ Process | Declaration $\bullet$ Process(Exp$^+$)
	|	$\|$ Declaration $\llbracket$ CSExpression$\rrbracket$ $\bullet$ Process
PParagraph	::=	Paragraph | N $\widehat{=}$ ParAction | $\mathbf{nameset}$ N $==$ NSExp
NSExp	::=	$\{\}$ | N^+ | N | NSExp $\cup$ NSExp | NSExp $\cap$ NSExp
	|	NSExp $\setminus$ NSExp
ParAction	::=	Declaration $\bullet$ Action | Action
Action	::=	Schema$-$Exp | CSPActionExp | Command | N

CSPActionExp	::=	*Skip* | *Stop* | *Chaos*
	|	Communication $\rightarrow$ Action | Predicate $\&$ Action
	|	Action; Action | Action $\Box$ Action | Action $\sqcap$ Action
	|	Action $\|[$ NSExp | CSExpression | NSExp $]\|$ Action
	|	Action $\|\|[$NSExp | NSExp$]\|\|$ Action | ; Declaration $\bullet$ Action |
	|	Action $\setminus$ CSExpression | μ N $\bullet$ Action | ParAction(Exp$^+$)
	|	$\Box$ Declaration $\bullet$ Action |$\sqcap$ Declaration $\bullet$ Action
	|	$\|\|$ Declaration $\|[$ NSExp | CSExpression | NSExp$]\|$ $\bullet$ Action
	|	$\|\|\|$ Declaration $\|\|[$NSExp | NSExp$]||$$\bullet$ Action
Communication	::=	NCParameter*
CParameter	::=	?N |?N : Predicate |!Expression | .Expression
ClassDefinition	::=	**class** N $\widehat{=}$ [**extends** N]**begin** CParagraph*
		[**state** StateSchema] CParagraph*
		[**initial** Schema$-$Exp] CParagraph*
		end
CParagraph	::=	Paragraph | Qualifier N $\widehat{=}$ ParametrisedCommand
Qualifier	::=	**public** | **protected** | **private** | **logical**
ParametrisedCommand	::=	Schema$-$Exp | Command
	|	ParameterDeclaration $\bullet$ Command
ParameterDeclaration	::=	ParameterQualifier Declaration
	|	ParameterQualifier Declaration; ParameterDeclaration
ParameterQualifier	::=	**val** | **res**
Command	::=	N$^+$: [Pred, Pred] | N$^+$:= Expression$^+$ | μ N $\bullet$ Command
	|	**var** Declaration $\bullet$ Command | super.N(OhExpression*)
	|	OhExpression.N(OhExpression*)
	|	Command; Command | **if** GuardedCommands **fi**
GuardedCommands	::=	Predicate $\rightarrow$ Command
	|	Predicate $\rightarrow$ Command $\Box$ GuardedCommands
OhExpression	::=	Expression
	|	**this** | **null** | **new** N[(OhExpression$^+$)]
	|	OhExpression.N | OhExpression : N(OhExpression*)
	|	super.N | super.N(OhExpression*)
	|	OhExpression **instanceof** N | (N)OhExpression

APÊNDICE B

LEIS DE CSP

Devido a *Circus*utilizar a mesma semântica de vários operadores de CSP [79], leis básicas de CSP sobre estes operadores são também leis de *Circus*.

Uma destas leis é a distribuição de *renaming* através da composição paralela (Lei B.1). Referenciada também como $f[.] - [\![X]\!] - dist$.

Lei B.1 (Distribuição paralela de *renaming*)

$$f[P \,[\![X]\!]\, Q] = (f[P] \,[\![f[X]]\!] f[Q])$$

Somente se: *f é uma função injetiva.* □

A lei a seguir mostra a distribuição de *hiding* através da composição paralela (Lei B.2). Referenciada também como $hide \,[\![X]\!] - dist$.

Lei B.2 (Distribuição paralela de *hiding*)

$$(P \,[\![X]\!]\, Q) \setminus Z = (P \setminus Z) \,[\![(X]\!]\, (Q \setminus Z)$$

Somente se: $X \cap Z = \{\}$ □

A próxima lei mostra a simetria de *renaming* sobre o operador de *hiding* (Lei B.3). Referenciada também como $f[.] - hide - sym$.

Lei B.3 (Simetria de *renaming* sobre *hiding*)

$$f[P \setminus X] = f[P] \setminus f[Z]$$

Somente se: $X \cap Z = \{\}$, *e f é uma função injetiva.* □

Outras duas leis expressão a associatividade (Lei B.4) e simetria (Lei B.5) da composição paralela. Referenciada também como $[\![X]\!] - assoc$ e $[\![X]\!] - sym$.

Lei B.4 (Associatividade do paralelismo)

$$P \,[\![X]\!]\, (Q \,[\![X]\!]\, R) = (P \,[\![X]\!]\, Q) \,[\![X]\!]\, R$$

Lei B.5 (Simetria do paralelismo)

$$P \,[\![X]\!]\, Q = Q \,[\![X]\!]\, P$$

Utilizaremos os fato que o operador de paralelismo somente é idepotente sob determinadas condições. Referenciada este fato como a Lei $[\![X]\!] - idem$.

Lei B.6 (Indepotência do operador de paralelismo)

$$P = P \llbracket \alpha P \rrbracket P$$

Somente se: *P é um processo determinístico.* □

Também utilizaremos o fato que canais não utilizados no sincronismo de um operador de paralelismo são redundantes. $\llbracket cs \rrbracket - null$.

Lei B.7 (Restrição de canais)

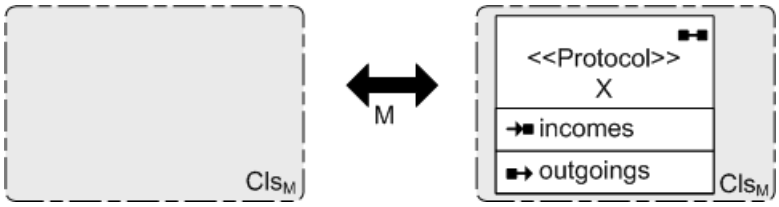
$$P \llbracket \alpha P \cup \alpha Q \cup cs \rrbracket Q = P \llbracket \alpha P \cup \alpha Q \rrbracket Q$$

APÊNDICE C

LEIS DE TRANSFORMAÇÃO PARA UML-RT ADICIONAIS

Semelhante à Lei 4.1, a próxima lei estabelece quando é possível introduzir um novo protocolo ao modelo.

Lei C.1 *Declarar Protocolo*



Condições:

(→) $\mathsf{Cls_M}$ *não possui a declaração de nenhum elemento, no mesmo pacote, chamado* X.

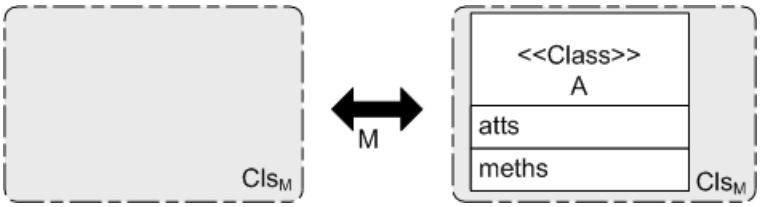
(←) *Nenhuma elemento em* $\mathsf{Cls_M}$ *tem uma relação com o protocolo* X.

Similarmente à Lei 4.1, para se remover um protocolo X é necessário que nenhum outro elemento o utilize, seja em associações, que criam como conseqüência portas de tipo X, ou em generalizações. Assim, uma porta com tipo X não pode ser usada em um diagrama de estado ou estrutura e, portanto, a apresentação destes diagramas é irrelevante (pela mesma razão que a Lei 4.1).

Assumimos que incomes e outgoings representam, respectivamente, os conjuntos de sinais de entrada e de saída do protocolo.

Semelhante a outros trabalhos que lidam com leis para classes em UML [38], A próxima lei estabelece quando é possível introduzir uma nova classe ao modelo.

Lei C.2 *Declarar Classe*



Condições:

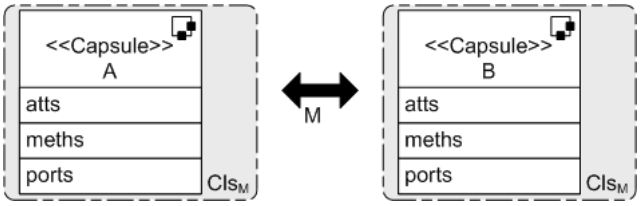
(→) $\mathsf{Cls_M}$ *não possui a declaração de nenhum elemento, no mesmo pacote, chamado* A.

(←) *Nenhuma elemento em* $\mathsf{Cls_M}$ *tem uma relação com a classe* A.

Similarmente à Lei 4.1, para se remover uma classe A é necessário que nenhum outro elemento a utilize, através de associações ou em generalizações em $\mathsf{Cls_M}$. A adição de uma classe implica $\mathsf{Cls_M}$ que esta poderá ser utilizada por outras classes ou cápsulas do modelo, que devem estar obrigatoriamente declaradas em $\mathsf{Cls_M}$, como a lei impõe condições que a classe A não é utilizada por outros elementos do modelo, a apresentação de outras visões M são irrelevante.

A próxima lei estabelece quando é permitido renomear uma cápsula.

Lei C.3 *Renomear Cápsula*



Condições:

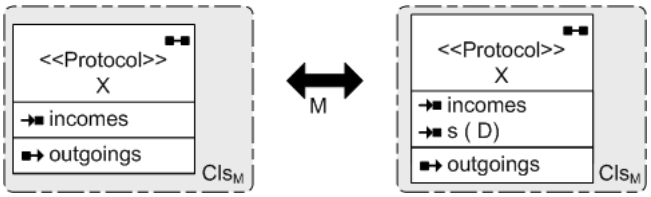
(→) $\mathsf{Cls_M}$ *não possui a declaração de nenhum elemento, no mesmo pacote, chamado* B.

(←) $\mathsf{Cls_M}$ *não possui a declaração de nenhum elemento, no mesmo pacote, chamado* A.

Como conseqüência da renomeação de uma cápsula A para B, na Lei C.3, todas as instâncias de A em $\mathsf{Str_M}$ passariam a ter o tipo B; analogamente, na aplicação da direita para a esquerda, todas as instâncias de B passariam a ter o tipo A. Apesar de sua simplicidade, esta lei é importante quando desejamos substituir uma cápsula no sistema por outra de mesmo nome; até que a cápsula original seja removida do modelo, as duas cápsulas são declaradas no modelo com nomes distintos. Após a substituição e remoção da cápsula original, a nova cápsula é renomeada.

A próxima lei estabelece quando é permitido adicionar ou remover sinais de entrada em um protocolo.

Lei C.4 *Introduzir Sinal de Entrada*



Condições:

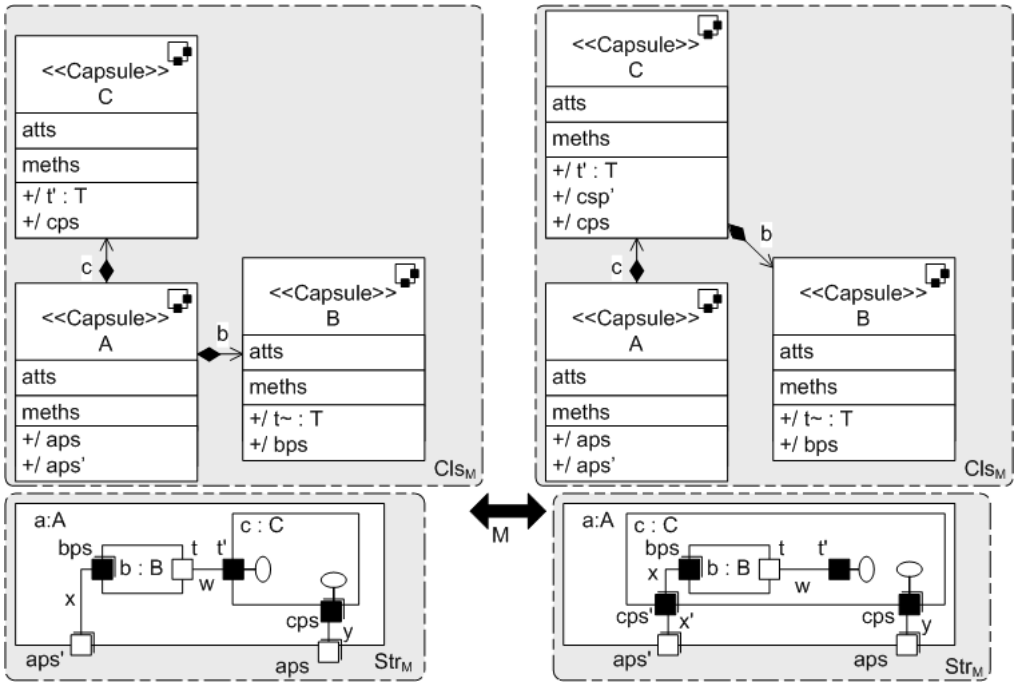
(→) *Não existe nenhum sinal chamado* s *no protocolo* X.

(←) *Nenhum diagrama de estados em* $\mathsf{Sta_M}$ *usa o sinal* s.

A Lei C.4 é análoga à Lei 4.6, diferenciando-se apenas em relação à direção do sinal inserido. Semelhante à Lei 4.6, a Lei C.4 possui uma restrição quanto ao uso do nome do novo elemento (sinal) em seu contexto local (protocolo) e quanto ao uso deste elemento nos diagramas de estado do modelo.

A próxima lei expressa como a associação entre duas cápsulas pode ser movida para outra cápsula que não possui subcápsulas.

Lei C.5 *Encapsular Cápsula 2*



Condições:

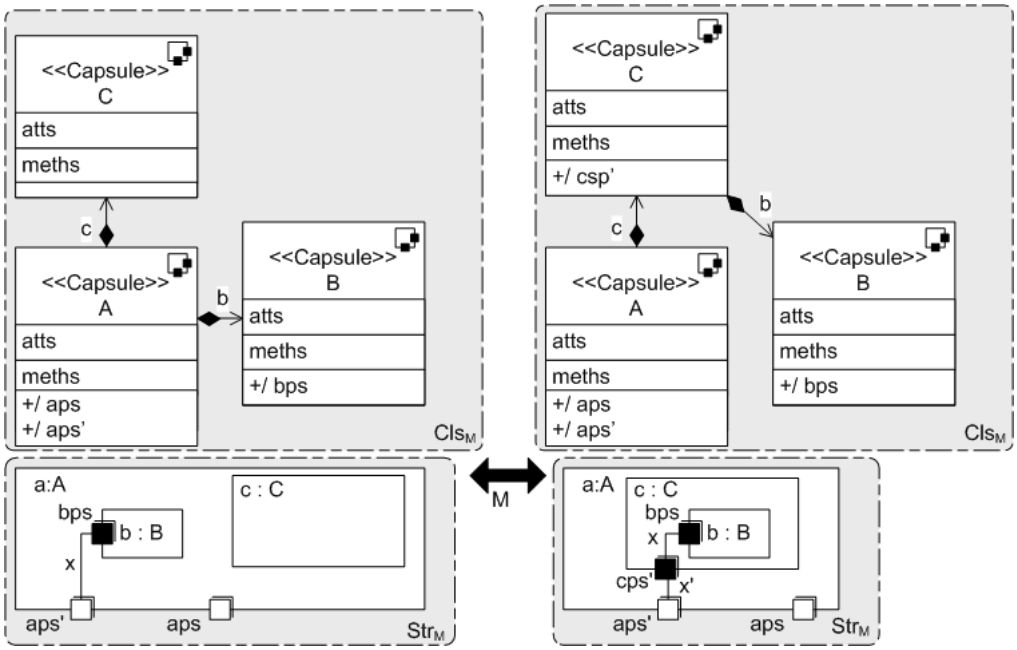
(→) *Nenhum nome de porta em* cps′ *coincide com um nome em* csp*; Em todo contexto em* $\mathsf{Str_M}$ *onde existir uma instância de* C*, a porta* t′ *estará conectada à porta* t *de uma instância de* B*.*

(↔) *O protocolo* T *e todos protocolos associados às portas em* bps *e* aps′ *possuem uma máquina de estados determinística.*

Esta lei é análoga à Lei 4.12, diferenciando-se apenas à estrutura encontrada na cápsula C. No lado direito da Lei 4.12, como conseqüência da mudança da associação b de A para C, a instância b é transferida para o diagrama de estrutura de C. Assim, a instância b passa a fazer parte da estrutura interna de C e a necessitar de um conjunto de portas cps′ em c para ter acesso ao mundo externo; todas as conexões que existiam em bps passam a se conectar à cps′.

A próxima lei expressa como a associação entre duas cápsulas pode ser movida para outra cápsula que não possui portas.

Lei C.6 *Encapsular Cápsula 3*



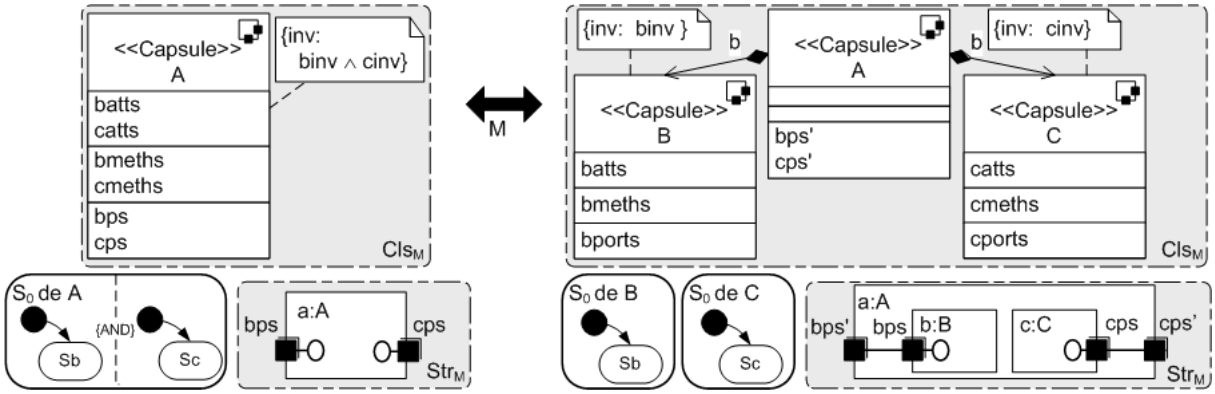
Condições:

(↔) *Todos protocolos associados as portas em* bps *e* aps′ *possuem uma máquina de estados determinística.*

Esta lei é análoga à Lei C.5, diferenciando-se apenas por C não possuir nenhuma porta associada. No lado direito da Lei 4.12, como conseqüência da mudança da associação b de A para C, a instância b é transferida para o diagrama de estrutura de C. Assim, a instância b passa a fazer parte da estrutura interna de C e a necessitar de um conjunto de portas cps′ em c para ter acesso ao mundo externo; todas as conexões que existiam em bps passam a se conectar à cps′.

A lei a seguir decompõe uma cápsula A no paralelismo de instâncias de cápsulas (B e C) que não se comunicam entre si.

Lei C.7 *Decomposição Paralela Simples de uma Cápsula*



Condições:

(↔) ⟨batts, binv, bmeths, bps, Sb⟩ *e* ⟨catts, cinv, cmeths, cps, Sc⟩ *particionam* A*; os protocolo associados às portas em* bps *e* cps *possuem uma máquina de estados determinística.*

Analogamente à 4.20, a Lei C.7 requer que a cápsula A seja particionada, onde cada partição deve ser auto-contida e fazer uso somente acesso aos atributos e aos métodos da partição. Além disto, as únicas portas utilizadas em uma partição são aquelas que ela contém.

Semelhante à Lei 4.20, esta lei possui o propósito de diminuir sua complexidade da cápsula A (lado esquerdo da lei), porém não possui como condição que as partes decompostas (que dão origem às cápsulas B e C, no lado direito da lei) comuniquem-se entre si.

O efeito da decomposição é criar duas novas cápsulas componentes, b e c, uma para cada partição, e redimensionar a cápsula original A para agir como um mediador. Em geral, o novo comportamento de A irá depender da forma particular da decomposição. A Lei C.7 captura uma decomposição paralela. No lado direito da lei, A não tem nenhuma máquina de estados. Ele delega completamente seu comportamento original para B e C através de conexões com componentes destes tipos no diagrama de estrutura.

A justificativa da Lei 4.20 a partir das leis básicas pode ser feita a partir de uma simplificação da demonstração da Lei 4.20, mudando-se apenas a aplicação das leis necessárias para desencapsular as instâncias de cápsulas b e c do diagrama de estrutura de A (no lado esquerdo da lei, utilizando-se para este fato apenas a Lei C.6 que indica que as instâncias desencapsuladas não se comunicam com nenhuma outra subcápsula de A.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Designing for Concurrency. Technical Report 084-0698, ObjecTime Limited. ObjecTime, 1998.

[2] M. Alanen, J. Lilius, I. Porres, and D. Truscan. Realizing a model driven engineering process. Technical Report 565, TUCS, Nov 2003.

[3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, May 1997. Issued as CMU Technical Report CMU-CS-97-144.

[4] N. Amalio and F. Polack. Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In *Third International Conference of B and Z*, volume 2651 of *LNCS*, pages 339–358, June 2003.

[5] 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, Chalmers University of Technology, Gothenburg, Sweden, April 2001. Honorary mention in the 2001 SNART Best Master's Thesis Award.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Reading, MA, 1998.

[7] K. Berkenkötter. Using UML 2.0 in Real-Time Development. A Critical Review. In *International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS)*, 2003. Workshop hold in conjunction with UML 2003.

[8] V. D. Bianco, L. Lavazza, M. Mauri, and G. Occorso. Towards uml-based formal specifications of component-based real-time software. In M. Pezzè, editor, *6th International Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 118–134, Warsaw, Poland, April 2003. Springer.

[9] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, 2004.

[10] R. Borges, A. Mota, and A. Sampaio. Integrando uml e métodos formais. Technical report, Universidade Federal de Pernambuco, Recife, Brasil, 2004.

[11] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison Wesley, anniversary edition, 1995.

[12] J.-M. Bruel, R. France, and M. Larrondo-Petrie. An integrated object-oriented and formal modeling environment. *Journal of Object-Oriented Programming*, 10(7):25–34, 1997.

[13] J. Buxton and B. Randell, editors. *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee*, Rome, Italy, October 1969. Brussels: NATO Scientific Affairs Division, April 1970.

[14] J. M. C. Silva, J. Castro. Detailing architectural design in the tropos methodology. In *Second International SofTware Requirements to Architectures Workshop*, pages 85–93, 2003.

[15] A. Cavalcanti, A. Sampaio, and J. Woodcock. Refinement of Actions in Circus. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002. Invited Paper.

[16] A. Cavalcanti, A. Sampaio, and J. Woodcock. A unified language of classes and processes. In *St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering*, Satellite Workshop at FM'03, 2003.

[17] A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for circus. *Formal Asp. Comput.*, 15(2-3):146–181, 2003.

[18] S. Cheng and D. Garlan. Mapping Architectural Concepts to UML-RT. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Application (PDPTA'2001)*, Las Vegas, Nevada, USA, 2001.

[19] I. Corporation. Rational rose technical developer, 2004. Available at: http://www-306.ibm.com/software/awdtools/developer/technical. Includes IBM Rational Rose RealTime.

[20] I. Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[21] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics.* Springer-Verlag New York, Inc., 1990.

[22] D. Dori. Why significant uml change is unlikely. *Communications of the ACM*, 45(11):82–85, 2002.

[23] D. D'Souza and A. Wills. *Objects, components, and frameworks with UML: the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] G. Engels, R. Heckel, and J. Küster. Rule-based specification of behavioral consistency based on the uml meta-model. In *4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 272–286, London, UK, 2001. Springer-Verlag.

[25] G. Engels, R. Heckel, J. Küster, and L. Groenewegen. Consistency-Preserving Model Evolution through Transformations. In *5th International Conference on the Unified Modeling Language*, volume 2460 of *LNCS*, pages 212–226. Springer, 2002.

[26] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-Preserving Model Evolution through Transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. 5th International Conference*, volume 2460 of *LNCS*, pages 212–226, Dresden, Germany, October 2002. Springer.

[27] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *8th European Software Engineering Conference*, pages 186–195. ACM Press, 2001.

[28] A. Evans. Reasoning with UML Class Diagrams. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*. IEEE Computer Society, 1998.

[29] A. Evans, R. France, and E. Grant. Towards formal reasoning with uml models. In *OOPSLA'99 Workshop on Behavioral Semantics*, 1999.

[30] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In *First International Workshop on the Unified Modeling Language*, LNCS. Springer, 1999.

[31] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.

[32] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP View on UML-RT Structure Diagrams. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 91–108. Springer, 2001.

[33] M. Fowler. *Refactoring-Improving the design of existing code*. Addison Wesley, 1999.

[34] H. Frank and J. Eder. Equivalence transformations on statecharts. In *12th International Conference on Software Engineering and Knowledge Engineering*, Chicago, July 2000. Knowledge System Institute.

[35] D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44(1):23–49, 2002.

[36] D. Garlan, R. Monroe, and D. Wile. ACME: Architecture Description of Composed-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[37] R. Gheyi. Basic laws of object modeling. Master's thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil, February 2004.

[38] R. Gheyi and P. Borba. Refactoring Alloy Specifications. In *6^{t}h Brazilian Workshop on Formal Methods*, volume 95 of *ENTCS*, pages 227–243. Elsevier Science, May 2004.

[39] W. Gibbs. Software's Chronic Crisis. *Scientific American*, 271(3):86–95, September 1994.

[40] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of SDL-2000: status and perspectives. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 42(3):343–358, 2003.

[41] M. Gogolla and M. Richters. Transformation Rules for UML Class Diagrams. In *First International Workshop on the Unified Modeling Language (UML)'98*, LNCS, pages 92–106. Springer, 1999.

[42] G. Gullekson. Designing for Concurrency and Distribution with Rational Rose RealTime. Technical Report TP-1864/00, Rational Software Corporation, 2000. Rational Software White Paper.

[43] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[44] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical Report MSC00-16, 2000.

[45] C. A. R. Hoare. Programming: Sorcery or science? *IEEE Software*, 1(2):5–16, 1984.

[46] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[47] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[48] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Carnegie Mellon, Software Engineering Institute, 2004.

[49] D. Jackson. A Comparison of Object Modelling Notations: Alloy, UML and Z. Technical report, MIT. Lab for Computer Science, August 1999.

[50] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, Software Design Group. MIT Lab for Computer Science, 2002.

[51] S. Kent. Model Driven Engineering. In *Proceedings of IFM'02*, volume 2335 of *LNCS*, pages 286–298. Springer-Verlag, 2002.

[52] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 2 edition, 2000.

[53] I. Krüger, W. Prenninger, and R. Sandner. Development of an Autonomous Transport System using UML-RT. Technical Report TUM-I0215, Technische Universität, München, 2002.

[54] I. H. Krüger. Towards Precise Service Specification with UML and UML-RT. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 19–34. Technische Universität München, Institut für Informatik, 2002.

[55] K. Lano and J. Bicarregui. Semantics and Transformations for UML Models. In *First International Workshop on the Unified Modeling Language*, volume 1618 of *LNCS*, pages 107–119. Springer, June 1999.

[56] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. pages 483–499, 1996.

[57] M. M. Lehman. Laws of software evolution revisited. In C. Montangero, editor, *5th European Workshop on Software Process Technology*, volume 1149 of *LNCS*, pages 108–124, Nancy, France, October 1996. Springer.

[58] J. Liu, J. S. Dong, B. Mohany, and K. Shi. Linking uml with integrated formal techniques. In *Unified Modeling Language: Systems Analysis, Design, and Development Issues*. IDEA GROUP Publishing, 2000.

[59] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[60] J. Ludewig. Models in software engineering. *Software and System Modeling*, 2(1):5–14, 2003.

[61] J. Magee and J. Kramer. Dynamic structures in software architecture. In *Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[62] M. S. Mahoney. The roots of software engineering. *CWI Quarterly*, 3(4):325–334, February 1990.

[63] T. McClean, F. Bordelau, and J.-P. Corriveau. Scenario-Driven Refactoring in UML-RT. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, Portland, Oregon, USA, May 2003. Held at the International Conference on Software Engineering 2003 - ICSE'03.

[64] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[65] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. OMG document omg/03-06-01.

[66] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking csp-oz with uml and java: A case study. In E. A. Boiten, J. Derrick, and G. Smith, editors, *4th International Conference on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 267–286. Springer, 2004.

[67] C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[68] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Inf.*, 27(6):481–503, 1989.

[69] M. Y. Ng and M. J. Butler. Towards formalizing uml state diagrams in csp. In *1st International Conference on Software Engineering and Formal Methods*, pages 138–147. IEEE Computer Society, September 2003.

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

[71] Object Management Group, Inc. *The OMG's Home page*, 2005. Avaliable at: http://www.omg.org/.

[72] M. Oliveira and A. Cavalcanti. From Circus to JCSP. In *6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 320–340. Springer, 2004.

[73] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, 1986.

[74] R. Ramos, A. Sampaio, and A. Mota. A semantics for uml-rt active classes via mapping into circus. In *To Apear in 7th IFIP international Conference on Formal Methods for Open Object-Based Distributed Systems*, LNCS, Athens, Greece, June 2005. University of Athens, Springer.

[75] H. Rasch. Translating a subset of uml state machines into csp. Technical report, Universitãt Freiburg, May 2002. Mobi-J Meeting.

[76] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th international conference on Software engineering*, pages 209–218. IEEE Computer Society, 1998.

[77] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana Champaign, 1999.

[78] D. Roe, K. Broda, and A. Russo. Mapping uml models incorporating ocl constraints into object-z. Technical Report 2003/9, Imperial College London, 2003.

[79] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[80] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theor. Comput. Sci.*, 60(2):177–229, 1988.

[81] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Mass., 1999.

[82] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schrr. UML + ROOM as a Standard ADL? In F. Titsworth, editor, *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, page 43. IEEE Computer Society, 1999.

[83] A. Sampaio, A. Mota, and R. Ramos. Class and Capsule Refinement in UML for Real Time. In *Proceedings of the Brazilian Workshop on Formal Methods*, volume 95 of *ENTCS*, pages 23–51, 2004.

[84] A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in *Circus*. In *International Symposium of Formal Methods Europe*, volume 2391 of *LNCS*, pages 451–470. Springer, 2002.

[85] R. Sandner. Developing Distributed Systems Step by Step with UML-RT. In *Workshop Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme.* Universität Münster, 2000.

[86] B. Selic. An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. In *11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, volume A-32 of *IFIP Transactions*, pages 335–344, 1993.

[87] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[88] B. Selic. Tutorial: An overview of uml 2.0. In *6th International Conference on Software Engineering*, pages 741–742, Edinburgh, United Kingdom, May 2004. IEEE Computer Society.

[89] B. Selic, G. Gullekson, and P. T. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, Inc., 1994.

[90] B. Selic and J. Rumbaugh. Using UML for Modeling Complex RealTime Systems. Rational Software Corporation, 1998. available at http://www. rational.com.

[91] M. Shaw and D. Garlan. Software Architecture - Perspectives on an Emerging Discipline. *Softwaretechnik-Trends*, 20(2), 2000.

[92] A. J. H. Simons. On the compositional properties of uml statechart diagrams. In *Rigorous Object-Oriented Methods*, Workshops in Computing, York, UK, January 2000. BCS.

[93] G. Smith. *The Object-Z specification language.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[94] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.

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

[96] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML Models. In *4th International Conference on the The Unified Modeling Language*, volume 2185 of *LNCS*, pages 134–148. Springer, October 2001.

[97] P. Welch and J. Martin. A CSP Model for Java Multithreading. In P. Nixon and I. Ritchie, editors, *Software Engineering for Parallel and Distributed Systems*, pages 114–122. ICSE 2000, IEEE Computer Society Press, June 2000.

[98] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, 1993.

[99] J. Woodcock and A. Cavalcanti. Circus: a concurrent refinement language. Technical Report Oxford OX1 3QD UK, Oxford University Computing Laboratory, Wolfson Building, Parks Road, July 2001.

[100] J. Woodcock and A. Cavalcanti. The Semantics of *Circus*. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.

[101] L. Zhang, D. Xie, and W. Zou. Viewing use cases as active objects. *ACM SIGSOFT Software Engineering Notes*, 26(2):44–48, 2001.