Uma Extensão do RUP para Modelagem Rigorosa de Sistemas Concorrentes

Robson Godoi, Rodrigo Ramos, Augusto Sampaio

Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 – CEP 50.740-540 – Recife – PE – Brasil

{rgam,rtr,acas}@cin.ufpe.br

Abstract. *Despite the great success of traditional software development processes on defining guidelines to be used in practice, many development aspects of critical and concurrent systems are still neglected. These problems are even more serious in processes that support emerging paradigms, such as model driven development. In this context, we propose a rigorous development strategy to model concurrent systems in UML-RT, based on the Rational Unified Process. In this strategy, although each analysis and design task is consistent and justified by model transformations that preserve the system behaviour, the underlying formalism is transparent for the developer.*

Resumo. *Apesar do grande sucesso dos processos de desenvolvimento de software tradicionais na criação de guias e boas práticas de desenvolvimento, diversos aspectos do desenvolvimento de sistemas críticos concorrentes são negligenciados. Estes problemas são ainda mais acentuados em processos que suportam paradigmas emergentes, como o desenvolvimento dirigido a modelos. Neste contexto, propomos uma estratégia rigorosa de modelagem de sistemas concorrentes em UML-RT, baseada no Rational Unified Process. Nesta estratégia, apesar de cada passo de análise e projeto ser consistente e ser justificado por transformações de modelos que preservam o comportamento do sistema, todo o formalismo que suporta estas atividades é transparente para o desenvolvedor.*

1. Introdução

Com a crescente demanda por aplicações concorrentes, um grande número de novos princípios e paradigmas de desenvolvimento de software vem surgindo para lidar com a complexidade inerente a estas aplicações. Alguns processos e metodologias têm se destacado por incluírem guias dos passos e de boas práticas a serem seguidos pelos desenvolvedores. Porém, devido à complexidade destes domínios, a execução segura e consistente destes passos constitui um grande desafio na prática.

Estes problemas são ainda mais evidentes quando lidamos com aplicações não triviais, que possuem aspectos de concorrência que devem ser precisamente especificados e verificados. Isto gera a necessidade de estratégias de desenvolvimento com passos que garantam a consistência do desenvolvimento; estes passos devem ser incorporados a um processo bem definido que, preferencialmente, torne transparente ao desenvolvedor formalismos possivelmente utilizados. Na literatura, algumas estratégias de desenvolvimento [7, 11, 12] têm adotado Métodos Formais, porém estas têm se

concentrado, essencialmente, na preservação de comportamento e consistência das diferentes visões da arquitetura durante o desenvolvimento, não incorporando os aspectos formais a um processo de desenvolvimento mais amplo.

Neste trabalho, propomos uma estratégia de desenvolvimento de sistemas concorrentes para UML-RT [15], um *profile* para UML utilizado para descrever sistemas concorrentes e distribuídos, através de uma adaptação e extensão do Rational Unified Process (RUP) [6]. Focamos na disciplina de Análise e Projeto, mas analisamos também os impactos em outras disciplinas do RUP para suportar um processo de desenvolvimento rigoroso baseado em modelos, que garanta a consistência e preservação de propriedades do sistema durante cada passo da modelagem. Cada passo é justificado por transformações de modelos e noções precisas baseadas em uma linguagem formal. Tais noções agem como uma interface formal para práticas de desenvolvimento, fazendo com que o desenvolvedor possa exercer suas atividades de forma consistente, porém sem se preocupar com o formalismo que as suporta.

A base formal da estratégia proposta é construída a partir de resultados com uma base semântica sólida. Em [11], atribuímos uma semântica formal e unificada à UML-RT através de seu mapeamento para *Circus* [17], uma linguagem formal que combina Z [16], CSP [13] e suporta o cálculo de refinamento de Morgan [9]. Em [10, 12, 14] apresentamos e provamos um conjunto abrangente de leis algébricas para UML-RT, que capturam tanto transformações simples no modelo, como *refactorings* precisos, preservando tanto o comportamento estático quanto o dinâmico do modelo.

Na próxima seção, introduzimos brevemente a linguagem de modelagem UML-RT. Nas seções 3 e 4 apresentamos as adaptações necessárias para as disciplinas de Requisitos e Análise e Projeto do RUP, respectivamente. No decorrer destas seções, ilustramos nossa estratégia através da modelagem rigorosa de parte de um sistema de automação industrial. Finalmente, nossas conclusões e trabalhos relacionados são apresentados na Seção 5.

2. Introdução à UML-RT

UML-RT, como outras linguagens de descrição arquitetural (ADLs), modela sistemas reativos com componentes arquiteturais ativos que interagem entre si e operam concorrentemente. A comunicação é modelada através da troca de mensagens de entrada e saída, que podem ser síncronas ou assíncronas. Apesar destes conceitos terem influenciado também o modelo de componentes na recente versão UML 2.0, aqui, utilizamos o *profile* UML-RT porque consideramos que seu modelo de componentes ativos é mais consolidado que o proposto para UML 2.0. Além disto, UML-RT conta com um suporte ferramental mais consolidado.

Os conceitos de UML-RT são adicionados à UML através de quatro novos elementos de modelagem: cápsula, protocolo, porta e conector. Cápsulas descrevem componentes arquiteturais cujos únicos pontos de interação são chamados de portas. Tais portas são interligadas por conectores e comunicam sinais declarados previamente em protocolos.

Para ilustrar a notação de UML-RT e nossa estratégia de desenvolvimento, apresentamos a modelagem simplificada de um sistema de automação industrial, responsável por processar 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 fisicamente distantes, e solicitam peças a um agente autônomo de transporte, chamado *holon*.

Um modelo inicial do sistema é apresentado na Figura 2.1. No diagrama de estrutura Str_M da figura (retângulo inferior), é possível ver que a cápsula Main, que representa todo o sistema, é, na realidade, composta de três outras instâncias de cápsula: sys, sin e son, que interagem entre si; estas instâncias são dos tipos ProdSys e Storage. No diagrama Str_M, Main delega todas mensagens comunicadas pelas portas mi e mo para sin e son.

A declaração das cápsulas (classes ativas) é apresentada no diagrama de classes Cls_M (retângulo superior), mostrando as relações entre cápsulas, protocolos e classes. Por exemplo, a relação son entre Main e Storage indica que existe uma instância da cápsula son dentro da estrutura de Main, e a relação so entre Storage e o protocolo STO indica que a porta so é do tipo do procolo STO. Outra parte importante do comportamento do sistema é descrito através de *Statecharts*. Através destes diagramas (lado direito da figura), protocolos podem estabelecer quais são as seqüências de sinais possíveis na interação entre cápsulas (como o *Statechart* de STO que estabelece que um sinal req deve sempre ser seguido por um output), e cápsulas podem definir como seu comportamento reativo será disparado por sinais externos.

Nos *Statecharts* da Figura 2.1 utilizamos a linguagem de especificação *Circus* para facilitar a verificação dos modelos durante um desenvolvimento rigoroso. Por exemplo, no *Statechart* de Storage, existem duas transições de saída no estado Sa. A transição à direita é disparada caso um sinal req seja recebido através da porta so e buffer não esteja vazio. A ação correspondente declara uma variável x para capturar o resultado do método remove. Esta é a maneira como é feito em *Circus*, já que remove é na realidade interpretado por um esquema em Z. O valor de x é então enviado através da porta so. A sintaxe para escrever estas ações de comunicação é a mesma que em CSP.

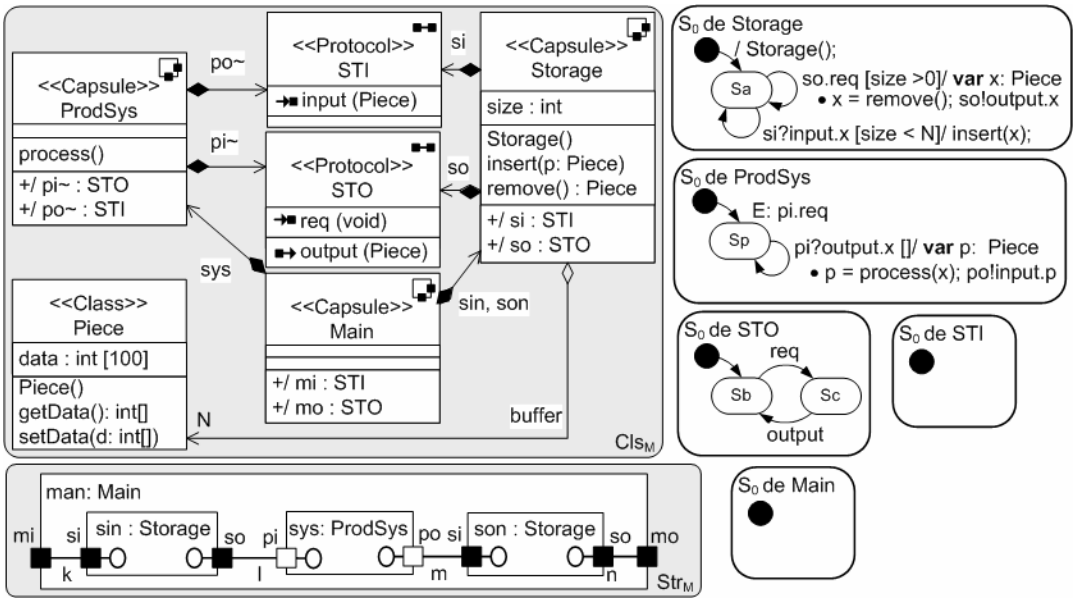


Figura 2.1. Modelo Abstrato de Análise

3. Modelagem de Requisitos

O RUP é um framework para processos incrementais e iterativos de desenvolvimento de software de propósito geral, com características, entre outras, de ser guiado por casos de uso e ser focado na criação de uma arquitetura robusta. Neste processo, atividades são agrupadas nas seguintes disciplinas: Modelagem do Negócio, Requisitos, Análise e Projeto, Implementação, Teste, Distribuição, Gerenciamento de Configuração e Mudanças, Gerenciamento de Projeto e Ambiente.

Apesar de nosso foco ser na disciplina de Análise e Projeto do RUP, mudanças são necessárias em outras disciplinas do processo, visando, principalmente, antecipar e mitigar potenciais riscos no desenvolvimento de aspectos relativos à concorrência. Nossa proposta inclui ou altera detalhes de fluxo e atividades das disciplinas de Requisitos e Análise e Projeto. Outras disciplinas, como Gerenciamento de Projeto e a Implementação, são afetadas somente em termos da organização da execução de suas atividades, considerando principalmente impactos na visão dinâmica (fases e iterações) do RUP; maiores detalhes sobre o impacto nestas disciplinas podem ser encontrados em [5]. No restante desta seção, apresentamos nossa proposta com relação à disciplina de Requisitos.

Na disciplina de Requisitos, no detalhe de fluxo *Definição do Sistema*, inicia-se a convergência dos requisitos de alto nível para um esboço mais detalhado das exigências do sistema. O foco é a identificação completa de atores e casos de uso e a extensão da modelagem dos requisitos não-funcionais, como definido no artefato *Especificações Suplementares*, bem como a criação de matrizes de rastreabilidade constantes no artefato *Atributos de Requisitos*. Neste trabalho, duas matrizes são utilizadas: uma relacionando os *Elementos Ativos* aos *Requisitos* e outra os *Casos de Uso* aos *Requisitos*.

Propomos a extensão do fluxo da disciplina, introduzindo um desvio condicional entre o detalhe de fluxo *Definição do Sistema* e o *Gerenciamento do Escopo do Sistema* (Figura 3.1), onde poderá ser identificada a existência de um comportamento ativo. Esta identificação é de vital importância para antecipar o tratamento de concorrência, conforme ilustrado a seguir. Criamos o detalhe de fluxo de *Identificação de Atividades Autônomas*, com objetivos de: nivelar a compreensão sobre concorrência no sistema entre toda equipe de projeto; detalhar o documento de *Especificações Suplementares* com a representação de concorrência no sistema; e refinar o *Modelo de Casos de Uso* para refletir os casos de uso relacionados à concorrência no sistema. Para tanto, foi criada a atividade *Identificação de Elementos Ativos* dentro deste detalhe de fluxo para se definir que partes do software serão passivas ou ativas, através do exame dos

Figura 3.1 – Contexto com o Fluxo de Requisitos adaptado

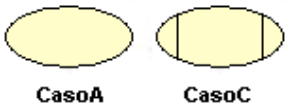


Figura 3.2 – Casos de Uso padrão (A) e com identificação de concorrência (C)

requisitos funcionais (**RF)** e não-funcionais **(RNF**).

Dois passos importantes são descritos nesta atividade: *Descoberta de Elementos Ativos* e *Análise de Requisitos Não-Funcionais de Concorrência*. Estes são responsáveis por analisar as descrições dos **RF** e **RNF**, respectivamente, do sistema, onde possíveis elementos ativos são identificados na descrição destes requisitos. Estes elementos ativos são sinalizados no artefato *Especificações Suplementares*. Consultando as matrizes de rastreabilidade, identificamos os casos de uso relacionados a estes elementos e os sinalizamos no artefato *Modelo de Casos de Uso*, através de uma representação gráfica para expressar concorrência similar à encontrada em [4], como o CasoC na Figura 3.2.

Tabela 3.1 Enumeração dos requisitos do sistema

Requisitos	Descrição
[**RF01**] Inserir Peças	O cliente pode inserir peças no repositório de entrada a qualquer momento.
[**RF02**] Recuperar Peças	O cliente pode recolher peças processadas pelo sistema no repositório de saída.
[**RF03**] Processar Peças	O sistema dever garantir que o processamento das peças ocorrerá obedecendo à seqüência de processadores descrita no programa de produção.
[**RF04**] Programar Produção	O operador está apto a mudar a configuração das máquinas.
[**RNF02**] Controle de Acesso (**RNF** de segurança)	O sistema controla o acesso às funcionalidades mediante a identificação do usuário através de uma senha de acesso e o nível de privilégio que ele tenha.
[**RNF04**] Processamento Simultâneo de Peças (**RNF** de concorrência)	O sistema pode processar diversas peças simultaneamente, obedecendo-se a programação de produção elaborada pelo operador.

Em nosso estudo de caso, nos passos de *Descoberta de Elementos Ativos* e *Análise de Requisitos Não-Funcionais de Concorrência*, a partir da análise dos requisitos do sistema (apresentados na Tabela 3.1), antecipamos a descoberta dos elementos ativos contidos na Tabela 3.2, que representam dispositivos físicos e autônomos neste sistema.

Tabela 3.2 Elementos ativos identificados

Nome	Descrição
Repositório de entrada	Responsável por armazenar peças a serem processadas e encaminha-las ao primeiro processador do programa de produção.
Processadores	Responsáveis pela recepção, processamento e encaminhamento de peças na linha de produção.
Holons	Responsável pelo transporte de peças entre os repositórios e processadores.
Repositório de saída	Responsável por armazenar peças processadas, e entregá-las aos clientes.

Tabela 3.3 - Matriz de rastreabilidade relacionando elementos ativos e requisitos

	RF01	RF02	RF03	RF04	RNF02	RNF04
Repositório de entrada	x		x			
Processadores			x			x
Holons			x			
Repositório de saída		X	x			

Tabela 3.4 - Matriz de rastreabilidade entre requisitos e casos de uso

	RF01	RF02	RF03	RF04	RNF02	RNF04
Inserir Peças	x					
Recuperar Peças		x				
Processar Peças			x			x
Programar Processamento				x	x	

Como saída desta atividade temos a atualização das *Especificações Suplementares*, onde os elementos com comportamento ativo são sinalizados (Tabela 3.2), e dos *Atributos de Requisitos*, onde são registradas as matrizes de rastreabilidade relacionando *Elementos Ativos* aos *Requisitos* (Tabela 3.3) e os *Casos de Uso* aos *Requisitos* (Tabela 3.4).

Consultando-se as matrizes que relacionam os elementos ativos, requisitos e casos de uso (Tabelas 3.3 e 3.4), identificamos os casos de uso (Figura 3.3) relacionados aos elementos ativos e a concorrência, que são: *Inserir*, *Recuperar* e *Processar Peças*.

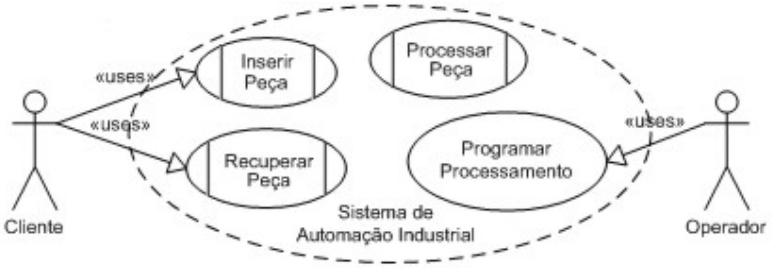


Figura 3.3 Casos de Uso do Sistema de Automação Industrial

4. Uma disciplina para Análise e Projeto (A&P)

Em nossa extensão da disciplina de A&P (Figura 4.1), adicionamos os detalhes de fluxo: *Projetar Componentes Ativos*, onde são refinados os elementos de projeto ativos

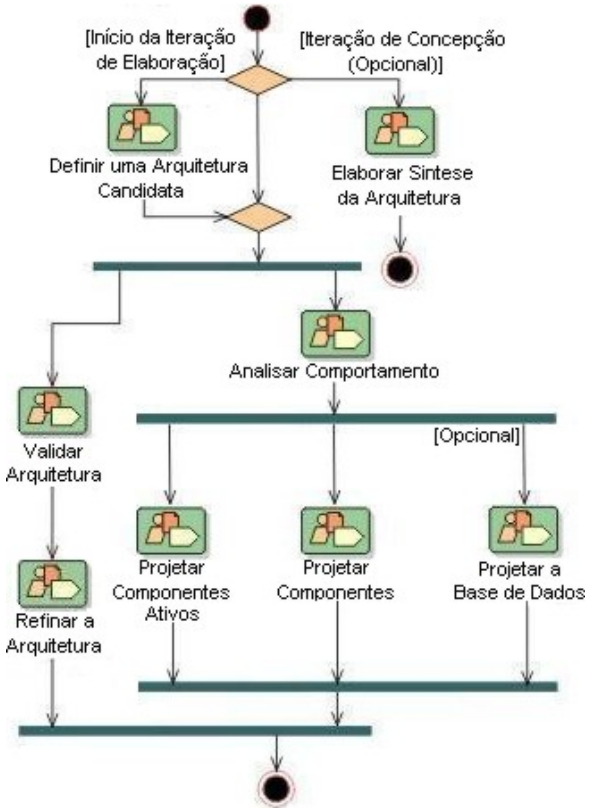


Figura 4.1 – Novo fluxo de Análise e Projeto

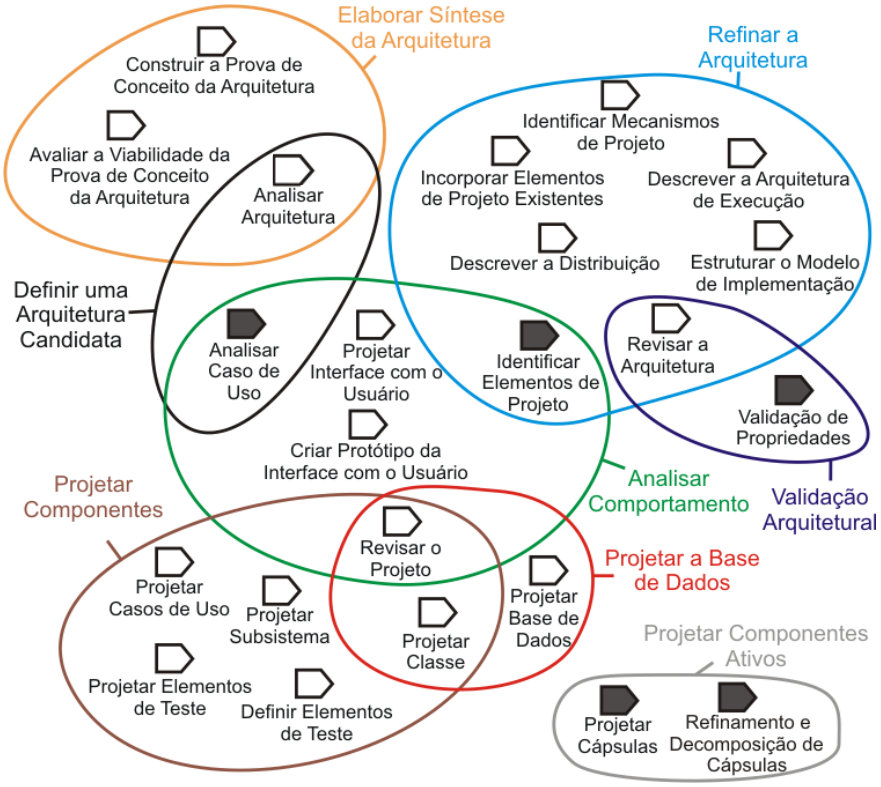


Figura 4.2 – Visão consolidada das Atividades x Detalhe de fluxo

e atualizadas as realizações de caso de uso, com base em leis para transformação de modelos UML-RT [10, 11,14]; e *Validar Arquitetura*, onde a arquitetura do sistema é validada utilizando-se a semântica de UML-RT proposta em [11]. Além disto, os detalhes de fluxo *Definir uma Arquitetura Candidata* e *Analisar Comportamento* foram alterados. Na Figura 4.2, enfatizamos quais atividades são alteradas ou adicionadas em cada um destes detalhes de fluxo, os quais serão detalhados no restante desta seção.

4.1. Definir uma Arquitetura Candidata

Um dos principais objetivos deste detalhe de fluxo, conduzido no início da fase de Elaboração, é criar um esboço inicial da arquitetura do software, onde são identificadas as classes de análise para os casos de uso relevantes da arquitetura. Este detalhe de fluxo é afetado em nossa estratégia pela alteração da atividade *Analisar Caso de Uso*, conforme apresentamos a seguir. O restante deste detalhe de fluxo não foi alterado, entretanto sugerimos a aplicação de leis de transformação de modelos consistentes [10], um dos focos de nosso trabalho em todas as atividades.

4.1.1. Analisar Caso de Uso

Um novo objetivo foi adicionado à atividade original: *Identificar as cápsulas que participam do fluxo de eventos do Caso de Uso*, bem como um novo passo: *Encontrar Cápsulas e Classes de Análise a partir do comportamento de um Caso de Uso*. A identificação de cápsulas deve ocorrer para todo caso de uso com indicação de concorrência, fornecida pela disciplina de Requisitos; à medida que cápsulas são identificadas, protocolos que regem sua interação são também definidos.

Para atingir este objetivo os seguintes artefatos são gerados e atualizados: *Classes de Análise*, representando um modelo inicial das responsabilidades e do comportamento do sistema; *Cápsula* e *Protocolo*, identificando parte dos elementos de projeto criados para representar concorrência; e *Realização de Caso de Uso*, representando a realização dos casos de uso em termos de colaborações de objetos.

Nossa estratégia assume a seguinte premissa: sistemas ativos ou concorrentes necessitam incluir pelo menos uma cápsula, que representa um fluxo de execução no sistema, e, desta forma, é parte inerente de qualquer sistema concorrente. No passo *Encontrar Cápsulas e Classes de Análise a partir do comportamento de um Caso de Uso*, é identificado o conjunto dos possíveis elementos do modelo (cápsulas e classes de análise) que serão capazes de representar o comportamento descrito nos casos de uso. Assim, cada caso de uso com concorrência é representado ao menos por uma cápsula de fronteira, que representa a interface do caso de uso, juntamente com seus protocolos. Caso um elemento de controle seja identificado também como ativo, este também será representado como uma cápsula.

Em nosso estudo de caso, executamos o detalhe de fluxo *Definir uma Arquitetura Candidata* em uma das iterações iniciais na fase de Elaboração; por conseguinte, as atividades de *Analisar Caso de Uso* e *Analisar Arquitetura* são exploradas. Destacamos na primeira atividade, a identificação das cápsulas que participam do fluxo de eventos do caso de uso. De acordo com o *guideline* do RUP, estas são classificadas como cápsulas de fronteira ou controle, dependendo dos elementos ativos identificados. Baseado neste *guideline*, construímos nosso modelo da Figura 2.1 sistematicamente a partir da visão de casos de uso. As instâncias de cápsula sys, sin e son representam as cápsulas de fronteiras dos casos de uso *Inserir*, *Recuperar* e *Processar Peças*, respectivamente; neste contexto interpretamos cada cápsula como subsistemas de todo o sistema, representado aqui pela cápsula Main, e assumimos que possíveis classes de controle estão contidas dentro destes subsistemas. Desta forma deixamos as extrações destes elementos de controle para um passo subseqüente do desenvolvimento.

A partir do modelo de análise na Figura 2.1, prosseguimos na *Definição de uma Arquitetura Candidata*. Adotamos uma arquitetura em camadas simples, onde a manipulação de dados é isolada das regras de negócio. Conseqüentemente, explicitamos a introdução de uma classe de coleção de dados, PieceCollection, para armazenar peças de trabalho; na realidade, esta classe é extraída de Storage utilizando-se uma regra de *Extração de Classes* a partir de cápsulas [10]. O resultado deste passo de projeto é apresentado na Figura 4.3.

4.2. Analisar Comportamento

Neste detalhe de fluxo, transformamos as descrições comportamentais de um caso de uso em elementos nos quais o projeto pode ser baseado. Alteramos a atividade *Identificar Elementos de Projeto.*

4.2.1. Identificar Elementos de Projeto

Nesta atividade, as interações entre as classes de análise são examinadas para identificar elementos de projeto do modelo. Estas classes evoluem para diversos elementos de projeto: classes, cápsulas, subsistemas, interfaces e protocolos. A decisão de quais classes de análise devem ser transformadas em cápsulas pode ser realizada através de

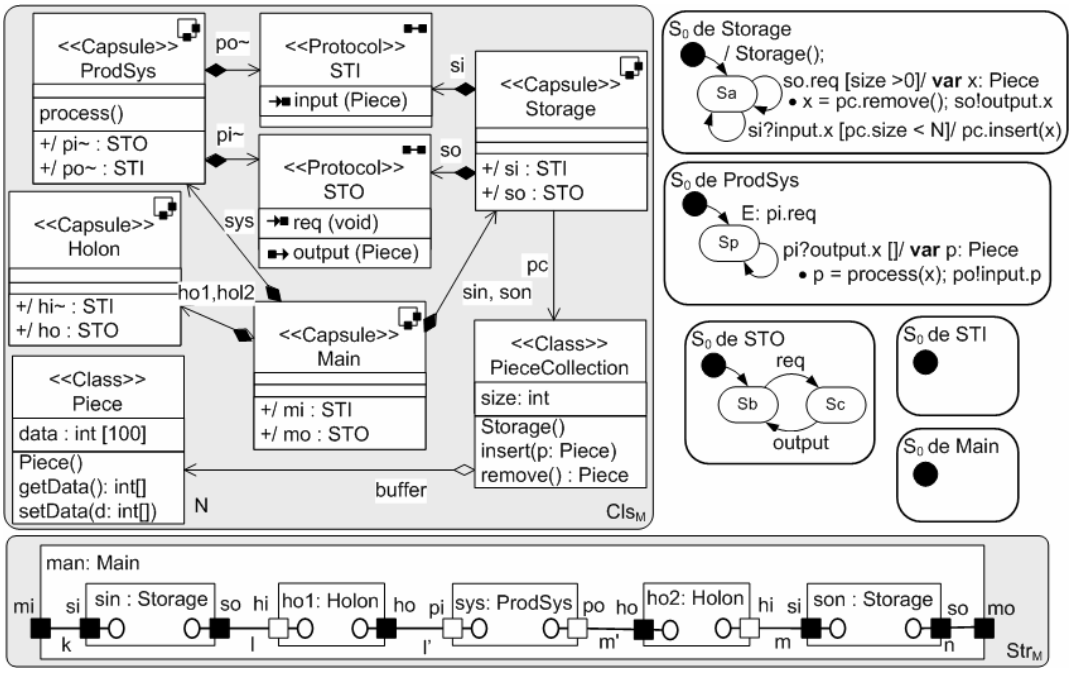


Figura 4.3. Extração de PieceCollection e identificação de agentes de transporte

diversas técnicas. Neste trabalho, utilizamos uma árvore de decisão (Figura 4.4), onde sugerimos que: uma classe de análise, cujo comportamento pode ser disparado por eventos externos, deve ser representada como uma cápsula. Analogamente, se a classe de análise controla ou delega ações a um elemento de projeto já identificado como cápsula, então esta classe deve-se tornar também uma cápsula. A razão é que, conceitualmente, uma classe passiva não pode ter um elemento ativo (cápsula) como atributo e invocar seus serviços diretamente; serviços de uma cápsula devem ser requeridos através de portas que implementam um protocolo de comunicação. Tais serviços devem ser chamados a partir de uma outra cápsula. Durante esta atividade, transformações de modelo consistentes [10], como *Transformar Classe em Cápsula* ou *Extrair Classe*, podem ser aplicadas.

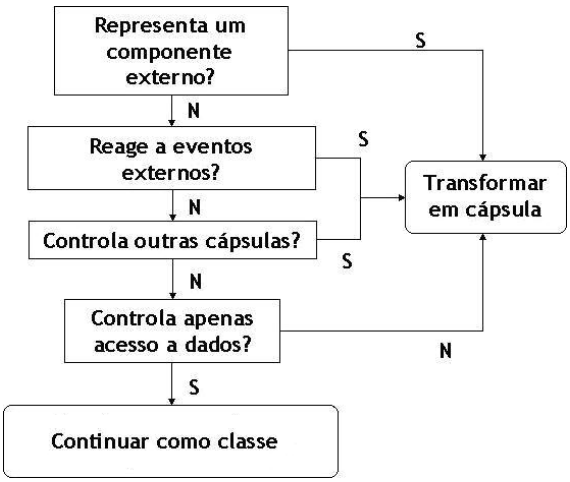


Figura 4.4 – Árvore de decisão para transformar classes em cápsulas

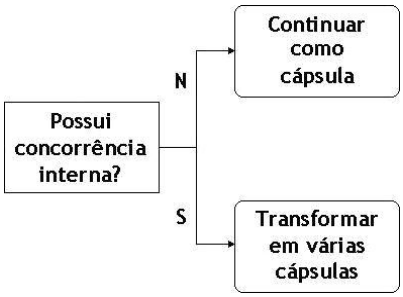


Figura 4.5 – Árvore de decisão sobre cápsulas

Agentes de transporte não são necessários somente para intermediar a comunicação entre processadores fisicamente separados, mas também para liberar processadores de

se preocuparem com o plano de processamento global. Para criar um agente de transporte (*Holon*), precisamos executar as atividades *Identificar Elementos de Projeto.* Para isto, introduzimos uma cápsula intermediaria entre as cápsulas que se comunica com os processadores, como mostra a Figura 4.3.

4.3. Projetar Componentes Ativos

Este novo detalhe de fluxo busca, de maneira complementar e especializada, atingir os mesmos objetivos do detalhe de fluxo original (*Projetar Componentes*), mas enfatizando componentes ativos. O fluxo é conduzido principalmente durante as fases de Elaboração e Construção, onde são refinados os elementos de projeto ativos e atualizadas as realizações de casos de uso, com base em leis de refinamento de modelos apresentados em [10, 12, 14]. Seus objetivos são: elaborar, refinar e decompor a definição dos elementos ativos do projeto, a fim de detalhar como os mesmos realizarão o comportamento no projeto; e refinar e atualizar as realizações de casos de uso com base nos novos elementos ativos de projeto identificados. A seguir apresentamos suas atividades.

4.3.1. Projetar Cápsulas

Nesta atividade, apresentada na Figura 4.6, são elaboradas e refinadas as descrições das cápsulas, devendo ser executada para cada cápsula, incluindo as identificadas no escopo desta atividade. Apesar de ser mencionada no RUP no detalhe de fluxo *Projetar Componentes*, neste trabalho introduzimos novos passos, como exposto a seguir.

Atividade: Projetar Cápsula

Propósito:
- Elaborar e refinar as descrições de uma cápsula.

Papel: Projetista de Cápsula

Freqüência: Quando requerido, tipicamente várias vezes em uma iteração, e mais frequentemente nas iterações de elaboração e construção.

Passos:
- Introdução de Novos Elementos de Projeto
- Criar Portas e Protocolos
- Validar Interações da cápsula
- Definir Máquina de estado da Cápsula
  - Definir Estados
  - Definir Transições
- Definir necessidade de Classes Passivas
- Introduzir Herança à Cápsula
- Validar Comportamento da Cápsula

Artefatos de Entrada	Artefatos de Saída
Cápsula Eventos Protocolos	Cápsula Classes de Projeto Protocolos

Detalhes do Fluxo
- Análise e Projeto
- Projeto de Componentes Ativos

Figura 4.6 – Atividade de Projetar Cápsulas

Atividade: Refinamento e Decomposição de Cápsula

Propósito:
- Refinar e decompor a descrição de uma cápsula.

Papel: Projetista de Cápsula

Freqüência: Quando requerido, tipicamente várias vezes em uma iteração, e mais frequentemente nas iterações de elaboração e construção.

Passos:
- Identificar Oportunidades de Refinamento
- Realizar Refinamento de Dados
- Realizar Refinamento de Controle
- Realizar Decomposição

Artefatos de Entrada:	Artefatos de Saída:
Cápsula Eventos Protocolos	Cápsula Classes de Projeto Protocolos

Detalhes do Fluxo:
- Análise e Projeto
- Projeto de Componentes Ativos

Figura 4.7. Atividade de Refinamento e Decomposição de Cápsulas

No passo *Introdução de Novos Elementos de Projeto*, introduzimos elementos de projeto a uma cápsula, como atributos, cápsulas internas, métodos, classes, portas ou relacionamentos. Tais elementos podem ser introduzidos através de leis básicas de UML-RT, como *Introduzir Método* [10]. No passo *Definir Máquina de Estados da Cápsula*, os aspectos comportamentais das cápsulas são definidos em termos de máquinas de estados, onde são definidas as respostas aos eventos e o ciclo de vida dos objetos; desta forma, podemos atualizar o modelo com a inclusão dos novos elementos. No último passo desta atividade, *Validar Comportamento da Cápsula*, o comportamento da cápsula deve ser avaliado e validado, usando a técnica manual *Walk-Through* ou uma ferramenta de simulação automática. Neste momento, o comportamento da cápsula é validado isoladamente do resto do sistema, de maneira similar à execução de teste unitário. Uma validação de todo o sistema será executada no detalhe de fluxo *Validar Arquitetura*.

4.3.2. Refinamento e Decomposição de Cápsulas

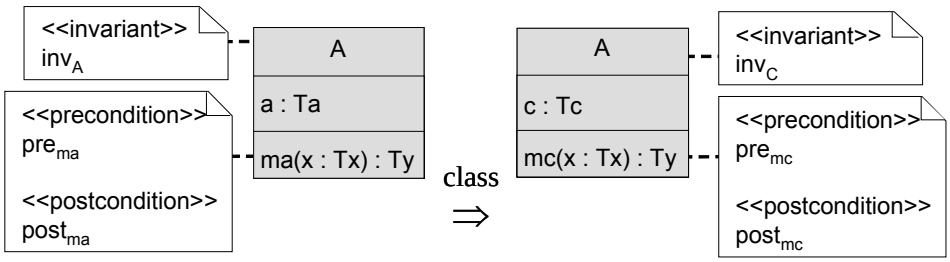
Nesta atividade, apresentada na Figura 4.7, faz-se a evolução das cápsulas através de refinamentos e decomposições, gerando e atualizando classes, cápsulas e protocolos do modelo; este é um processo iterativo que pode incluir, inclusive, cápsulas decompostas anteriormente. Neste processo, cápsulas que representam mais de uma abstração são decompostas (Figura 4.5), quer seja para aumentar a qualidade do modelo, representar entidades identificadas nos requisitos do sistema, ou atender requisitos não funcionais do sistema (como a execução de fluxos de controle concorrentes).

No passo *Identificar Oportunidade de Refinamento*, cada cápsula ou classe deve ser analisada em busca de oportunidades de refinamento. Podemos refinar um componente, realizando: refinamento de dados, onde a estrutura interna dos dados é refinada, buscando uma melhor forma de representação em termos de desempenho ou aderência a padrões arquiteturais e de projeto; um refinamento de controle, onde os elementos comportamentais (máquinas de estados, código, etc.) são refinados, objetivando melhorar o desempenho ou uma melhor distribuição de responsabilidades; e uma decomposição, onde cada componente é decomposto em dois ou mais componentes, buscando aumentar a possibilidade de reuso ou distribuição e diminuir a complexidade. Neste passo, podemos utilizar diversos *guidelines*, dentre eles a análise do comportamento interno das cápsulas para identificar uma possível concorrência interna, indicando a necessidade de transformar em várias cápsulas, conforme Figura 4.5. Este passo é vital para a evolução desta atividade, pois a partir dele identificamos quais passos adicionais serão executados.

Em todos estes passos, o refinamento ou a decomposição deve ser executada através da aplicação de leis de transformação que garantam a preservação do comportamento. Uma vez identificada a necessidade de refinamento de dados, o passo *Realizar Refinamento de Dados* é executado, sendo obtido através da substituição da estrutura interna dos dados, mantendo a interface externa. Para garantir a preservação do comportamento externo observável, propomos a utilização da lei *Refinamento de Dados* [14], Figura 4.8.

Nesta lei, a classe A é refinada mantendo-se a interface original; entretanto, para cada método, é possível enfraquecer a pré-condição ou fortalecer a pós-condição, tornando A mais especializada. Esta lei pode ser aplicada também a cápsulas. Precisamos destacar

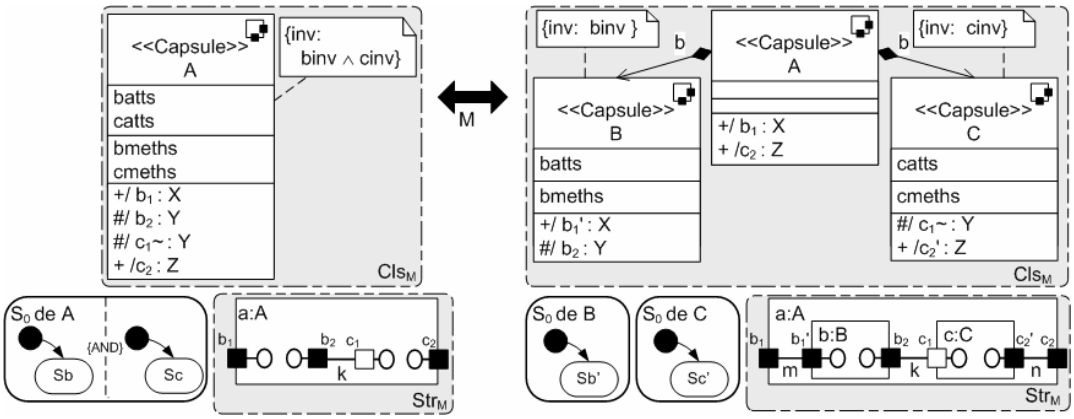
que quando um refinamento de dados ocorre, automaticamente as operações que manipulam os atributos devem ser ajustadas à nova representação.



provided 1. $\forall$ a: Ta; c : Tc; x : Tx | $inv_A \wedge inv_C \bullet pre_{ma} \wedge CI \Rightarrow pre_{mc}$
2. $\forall$ a : Ta; c, c' : Tc; x : Tx; y : Ty | $inv_A \wedge inv_C \wedge inv_C' \bullet$
$pre_{ma} \wedge CI \wedge post_{mc} \Rightarrow (\exists$ a' : Ta | $inv_A' \bullet CI' \wedge post_{ma})$

Figura 4.8 Lei de Refinamento de Dados

Uma vez identificada a necessidade de refinamento de controle, o passo *Realizar Refinamento de Controle* é executado, através do refinamento dos elementos comportamentais, podendo ser utilizada, por exemplo, a lei *Criar Região* [10], que cria uma nova região no *Statechart*. No passo *Realizar Decomposição*, cada elemento é decomposto em dois outros elementos, aplicando-se, por exemplo, a lei *Decomposição Paralela de Cápsula*, conforme a Figura 4.9, onde uma cápsula A é decomposta no paralelismo de instâncias de cápsulas (B e C) com o propósito de diminuir complexidade e de, potencialmente, aumentar reuso.



Condições: (→) { batts, binv, bmeths, (b_1, b_2), Sb} e { catts, cinv, cmeths, (c_1, c_2), Sc} 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 são substituídas respectivamente, por b_1' e c_2'; os protocolo X e Z possuem uma máquina de estados determinística.

Figura 4.9 Lei de Decomposição Paralela de Cápsula

No lado esquerdo da lei da Figura 4.9, existe a condição de que A deve estar *particionada*. Informalmente, uma cápsula *particionada* é formada por dois grupos disjuntos de variáveis, métodos, portas e regiões do *Statechart*, que se comunicam somente através de portas internas de A; cada partição é proibida de acessar diretamente qualquer elemento da outra parte. O efeito da decomposição é criar duas novas cápsulas B e C, uma para cada partição, e reestruturar a cápsula original (A) para agir como um mediador, preservando sua assinatura e comportamento externo.

A arquitetura de nosso estudo de caso é incrementalmente enriquecida por meio da execução dos detalhes de fluxos de *Analisar Comportamento* e *Projetar Componentes Ativos*. Em particular, durante a atividade *Refinamento e Decomposição de Cápsulas*, decompomos o processador ProdSys nas duas cápsulas ProcessorA e ProcessorB, usando principalmente a *Lei de Decomposição Paralela* (Figura 4.9). Com a introdução destas duas cápsulas, as responsabilidades de ProdSys são divididas entre estes dois novos processadores, e o seu comportamento original é representado pela interação deles. O papel final de ProdSys é somente o de mediar a comunicação entre ProcessorA e ProcessorB através de suas portas com o mundo externo. Como este papel de ProdSys passa a ser irrelevante em nossa arquitetura, ela é eliminada através de leis para remoção de cápsulas [10]. O modelo resultante é apresentado na Figura 4.10.

Mostramos também, na Figura 4.10, a reestruturação do agente de transporte Holon para que este se conecte a todos os elementos do processamento (repositórios e processadores), e possa, assim, gerenciar a programação do processamento. Para realizarmos esta reestruturação, primeiramente, observamos a necessidade de um outro agente de transporte entre os processadores, como na atividade *Identificar Elementos de Projeto* e, depois, compomos os agentes dois a dois (*refactoring* através da aplicação inversa da *Lei de Decomposição Paralela*, Figura 4.9). Como, neste momento, *Holon* possui um papel de um simples *proxy* de comunicação, um refinamento de controle e de dados (durante a atividade *Refinamento e Decomposição de Cápsulas*) poderá ser necessário em uma iteração posterior com o objetivo de criar agentes mais complexos e

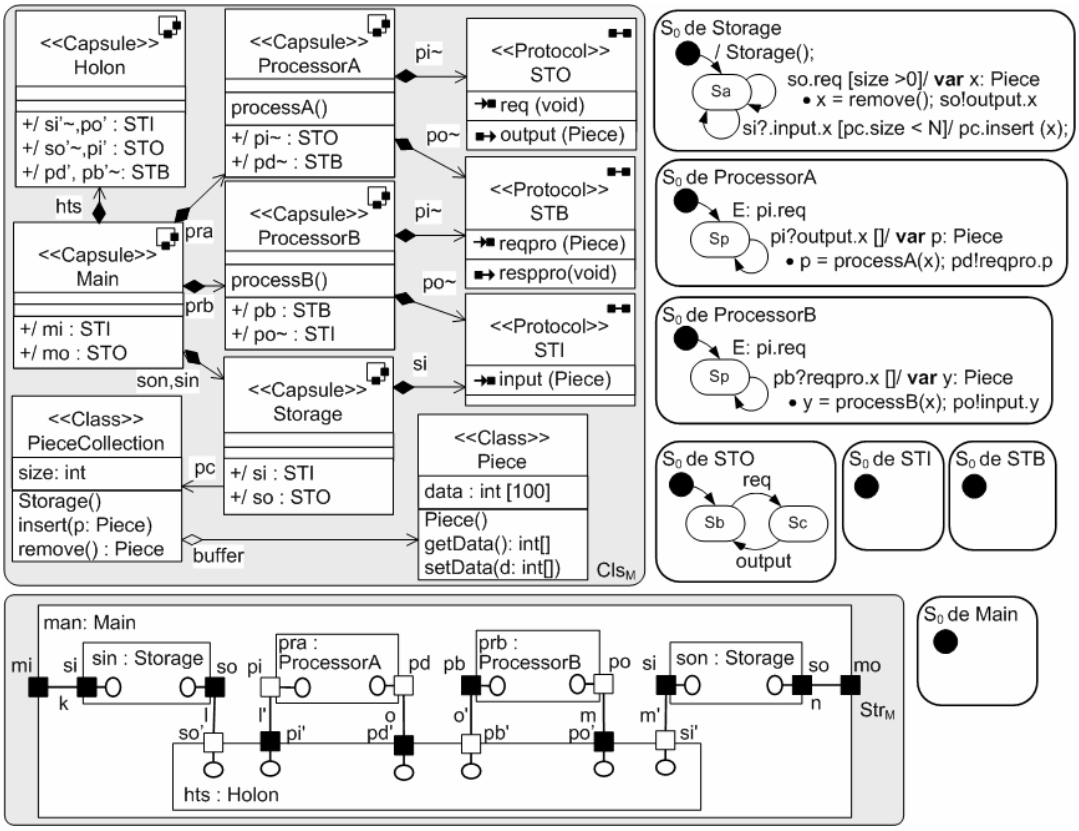


Figura 4.10. Decomposição do processador, projeto de cápsulas e *refactoring* do agente de transporte

que possam lidar com possíveis mudanças na programação do processamento.

Note que a decomposição de cápsulas pode acontecer em diferentes atividades da modelagem, diferenciando-se basicamente pelo papel da cápsula no sistema. Uma cápsula identificada em um **RF**, durante a atividade *Descoberta de Elementos Ativos*, geralmente será considerada na atividade *Identificar Elementos de Projeto*, pois ela representa um elemento necessário na arquitetura para a realização do comportamento do caso de uso. Já uma cápsula identificada em um **RNF**, durante a atividade *Análise de Requisitos Não-Funcionais de Concorrência*, é considerada durante a atividade de *Refinamento e Decomposição de Cápsulas*.

4.5. Validar Arquitetura

Neste novo detalhe de fluxo busca-se garantir que a arquitetura desenvolvida seja robusta, livre de problemas inerentes a sistemas concorrentes como *livelock* e *deadlock*, e que de fato implementa os **RFs** da aplicação. Neste fluxo, todas as cápsulas projetadas são validadas em conjunto, sendo realizado durante as fases Elaboração e Construção. Seus objetivos são: Avaliar e validar a arquitetura considerando propriedades clássicas de concorrência e propriedades específicas da aplicação; e Revisar a arquitetura para refletir possíveis problemas encontrados na validação, possivelmente provocando novas iterações para tratar concorrência.

4.5.1. Validação de Propriedades

Esta atividade foi criada para avaliar o projeto das cápsulas, onde as propriedades clássicas de concorrência são validadas, possibilitando também a validação de propriedades específicas da aplicação, obedecendo ao mesmo formato de outras atividades apresentadas (Figura 4.6 e 4.7).

O RUP indica que o comportamento de cada elemento ativo deve ser avaliado e validado, entretanto, a validação do sistema como um todo (todas as cápsulas), em termos das propriedades clássicas de concorrência não é explorado, muito menos com relação à validação de propriedades específicas da aplicação. Desta forma, consideramos de grande relevância que esta validação ocorra para garantir a robustez da arquitetura, um marco, no próprio RUP, que caracteriza o fim da fase de Elaboração.

Particularmente, propomos o mapeamento dos elementos de UML-RT em uma notação formal, *Circus* [11], uma linguagem formal que combina Z e CSP, e que formaliza todos os passos que envolvem transformações de modelo neste trabalho [10,12,14]. O comportamento das cápsulas de UML-RT, quando desconsideramos ações do *Statechart*, pode ser representado através do modelo de *traces* de CSP, assim como a interação entre cápsulas. Este mapeamento para CSP é uma simplificação do definido para *Circus* em [11].

No passo *Executar a Tradução de Statechart para CSP* são executadas as traduções de todas as máquinas de estados para CSP. No passo *Validar Propriedades Clássicas*, após a tradução de todas as máquinas de estados (de cápsulas e protocolos) para CSP, propriedades clássicas do sistema devem ser validadas, através da ferramenta para verificação de modelos FDR [3], obtendo-se uma verificação parcial da robustez da arquitetura. Para uma verificação mais abrangente, foi criado o passo *Validar Propriedades Específicas*, onde podemos validar todas as regras de negócio que

considerarmos relevantes. Por exemplo, considere uma regra de negócio (propriedade da aplicação) codificada em CSP como um processo RN, assumindo que a aplicação em si está codificada como o processo P, a verificação de que P satisfaz RN pode ser realizada em FDR pelo refinamento: $RN \sqsubseteq P$. Em nosso estudo de caso, por exemplo, pode-se verificar formalmente se toda a peça entregue ao sistema será eventualmente processada ou se obedecerá sempre à programação de processamento.

5. Conclusão e Trabalhos Futuros

Apresentamos uma estratégia rigorosa de modelagem de sistemas concorrentes para UML-RT, através da adaptação e extensão das disciplinas do *Rational Unified Process* (RUP), com foco em Análise e Projeto [5]. Apesar de aspectos de concorrência já serem considerados informalmente no RUP, nossa abordagem propõe e detalha passos consistentes de desenvolvimento que conservam o comportamento do sistema e permite a análise formal de propriedades da aplicação. Estes passos são descritos no contexto de uma estratégia sistemática que integra princípios de um processo de desenvolvimento prático, como o RUP, com resultados consolidados na área de Métodos Formais. Apesar do foco prático deste trabalho, aspectos formais são usados para garantir, por exemplo, que regras de transformação de modelos preservam o significado. Uma vez provadas, estas regras (por exemplo, para decompor cápsulas, Figura 4.7) podem ser usadas pelo desenvolvedor sem necessidade do conhecimento do formalismo utilizado.

Nosso trabalho inclui uma análise de como a concorrência deve ser abordada desde o inicio do desenvolvimento, disciplina de Requisitos, até a disciplina de Análise e Projeto, onde as decisões arquiteturais de um sistema concorrente sofrem seus maiores impactos. Devido ao uso integrado de transformações de modelo consistentes, acreditamos que este trabalho é também uma contribuição original para o desenvolvimento baseado em modelos UML-RT, através da extensão e adaptação de detalhes de fluxo, atividades e *guidelines*.

Relacionado ao desenvolvimento de sistemas concorrentes, podemos citar vários trabalhos [1, 7, 2]. Em [7], uma metodologia de desenvolvimento de modelos UML-RT é apresentada através de transformações de modelo e passos de validação de modelo. Apesar das similaridades deste trabalho com o nosso, ele foca na definição de passos que preservam a integridade entre várias visões do modelo, mas negligenciam a integração destes passos com um processo de desenvolvimento prático, como o RUP [5]. Atividades de decomposição e refinamento de componentes, como definidas na Seção 4, também são relacionadas na literatura em processos baseados em componentes, como Kobra [1], mas este não utiliza uma estratégia formal de refinamentos, como definida em nossa abordagem. Uma extensão do RUP para o desenvolvimento de sistemas baseados em aspectos pode ser encontrada em [2]. Apesar das vantagens deste paradigma para o desenvolvimento de sistemas concorrentes, nenhuma integração com Métodos Formais, capaz de guiar um desenvolvimento rigoroso é apresentada. Além disto, nenhum destes trabalhos apresenta e prova formalmente um conjunto de leis para UML-RT abrangente como o nosso [10].

Um tópico importante para trabalho futuro é a apresentação detalhada de uma atividade de implementação, que integre aspectos práticos e formais, como realizamos para a disciplina de Análise e Projeto. Uma breve discussão desta disciplina é realizada em [5], sugerindo, principalmente, a geração automática de código a partir de diagramas

comportamentais do modelo (por exemplo, *Statecharts*), seguindo a filosofia do desenvolvimento dirigido a modelos. Uma discussão sobre os impactos na visão dinâmica (fases e iterações) do RUP também é realizada em [5], utilizando uma abordagem semelhante a [8].

Referências

[1] Atkinson, C., Bayer, J., Laitenberger O., Zettel, J. Component-Based Software Engineering: The KobrA Approach. International Workshop On Component-Based Software Engineering, 2000.

[2] Cole, L. , Piveta, E., Sampaio, A. RUP Based Analysis and Design with Aspect. XVIII Brazilian Symposium on Software Engineering – SBES, 2004.

[3] Formal Systems (Europe) Ltd. Failures-Divergences Refinement: FDR2 User Manual, 1997. Disponível em http://www.fsel.com.

[4] Fowler, M. UML Essencial - 3ª edição, Porto Alegre, Bookman, 2005.

[5] Godoi, R. Uma disciplina de Análise e Projeto para Aplicações Concorrentes, baseada no RUP. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil, 2005.

[6] Kruchten, P.. The Rational Unified Process - An Introduction. Addison-Wesley, 2000.

[7] Küster, J., Engels, G. Consistency Management within Model-Based Object-Oriented Development of Components. 2nd International Symposium on Formal Methods for Components and Objects, LNCS 3188, p. 157-176, 2004.

[8] Tiago Massoni, Augusto Sampaio and Paulo Borba. A RUP-based Software Process Supporting Progressive Implementation. UML and Unified Process. p. 375-387 IRM Press, April 2003

[9] Morgan, C. Programming From Speci_cations. Second edn. Prentice Hall, 1994.

[10] Ramos, R. Desenvolvimento Rigoroso com UML-RT. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil, 2005.

[11] Ramos, R., Sampaio, A., Mota, A. A Semantics for UML-RT Active Classes via Mapping into Circus. 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Volume 3535 of Lecture Notes in Computer Science, pp. 99-114, Springer (2005)

[12] Ramos, R., Sampaio, A., Mota, A.. Transformation Laws for UML-RT. 8th IFIP international Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, 14 - 16 June (2006). To Appear.

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

[14] Sampaio, A., Mota, A., Ramos, R. Class and Capsule Refinement in UML for Real Time. Invited Paper. Brazilian Workshop on Formal Methods. Volume 95 of Electronic Notes in Theoretical Computer Science, pp. 23–51, Elsevier Science, 2004

[15] Selic, B., Rumbaugh, J.. Using UML For Modeling Complex RealTime Systems. Rational Software Corporation, 1998. Disponível em http://www. rational.com.

[16] Spivey, M.. The Z Notation: A Reference Manual. Second edn. Prentice Hall, 1992.

[17] Woodcock, J., Cavalcanti, A. Semantics of circus, the. In the Formal Specification and Development in Z and B Conference , Volume 2272 of Lecture Notes in Computer Science, pp. 184–203. Springer-Verlag. 2002.