FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586c
2006 Silva Júnior, Rogério Dourado.
CASTOR: uma técnica de verificação de conformidade para arquitetura de software baseada em componentes / Rogério Dourado Silva Júnior.─ Campina Grande, 2006.
98f.

Dissertação (Mestrado em Informática) - Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia.
Referências.
Orientadores : Dr. Jorge César Abrantes de Figueiredo, Dr. Dalton Dario Serey Guerrero.

1. Deterioração de software. 2. Componentes. 3. Arquitetura de software. I. Título.

CDU – 004.72(043)

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE CIÊNCIA E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

CASTOR: UMA TÉCNICA DE VERIFICAÇÃO DE CONFORMIDADE PARA ARQUITETURAS DE SOFTWARE BASEADA EM COMPONENTES

ROGÉRIO DOURADO SILVA JÚNIOR

CAMPINA GRANDE – PB

FEVEREIRO DE 2006

CASTOR: Uma Técnica de Verificação de Conformidade para Arquiteturas de Software Baseada em Componentes

Rogério Dourado Silva Júnior

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Jorge César Abrantes de Figueiredo

(Orientador)

Dalton Dario Serey Guerrero

(Orientador)

Campina Grande, Paraíba, Brasil

Resumo

Cada vez mais o software está presente em todos os aspectos do cotidiano das pessoas. Existe uma forte tendência para o crescimento da complexidade, tamanho, e domínios de aplicação dos programas desenvolvidos. Com efeito, o desenvolvimento de software não pode ser mais concebido como uma atividade individual, e sim como um processo essencialmente coletivo. Além disso, são grandes as pressões para que empresas de software sacrifiquem a qualidade de seus produtos para que estes cheguem mais rapidamente ao mercado consumidor. Neste cenário, dois problemas são recorrentes: a deterioração de software e a ausência de integridade conceitual. O primeiro decorre da natural evolução do software diante do surgimento de novos requisitos e restrições. Sem o devido cuidado, as constantes mudanças no projeto podem desfigurá-lo a tal ponto que se torne extremamente dispendioso compreendê-lo. Isso acaba por acarretar na baixa qualidade do código, na alta taxa de erros e no aumento dos custos do projeto. O segundo problema refere-se à dificuldade em se manter, entre a própria equipe de desenvolvimento, a uniformidade dos conceitos e abstrações utilizados ao longo do projeto. Nesse caso, o produto final normalmente é caracterizado por um conjunto não coeso de funcionalidades e aspectos divergentes.

Este trabalho aborda tais problemas através do papel que a arquitetura de software pode exercer como um referencial para a evolução do código (desde sua concepção inicial até ser retirado de uso). Mais especificamente, ela serve como base para avaliar se o código se desviou do que fora projetado. Dessa forma é possível detectar sinais da deterioração do software, bem como qualquer tentativa, por parte de algum desenvolvedor, de violar a integridade resumida na arquitetura consensualmente estabelecida. Diante do exposto, o objetivo deste trabalho é a formulação de uma técnica capaz de automaticamente fornecer as evidências de que certa implementação não está em conformidade com seu modelo arquitetural. Dessa forma, esperamos contribuir na melhoria da qualidade do software no que tange aos problemas levantados.

Abstract

Software plays a critical role in almost every facet of our daily life. While complexity and size are increased by new features and advanced software requirements, the pressure is higher than ever to reduce the time required for designing, prototyping, testing and manufacturing. The results are low-quality products with high software failure rates. In this context, two problems are frequent: software deterioration and the lack of conceptual integrity. The former one is due to the natural evolution of the software since the initial stages of development. Changes are part of the process, but if not well managed they can disfigure the projet to a such point that it becomes very hard to understand and maintain. The latter refers to the difficulty to keep uniform the concepts and abstractions used by the team along the implementation phases. The more lack of conceptual integrity more the system tends to exhibit low cohesion of functionalities and divergent aspects.

This work approaches such problems relying on software architecture as a reference model to the code evolution. More specifically, we investigate how to evaluate such model against the software in order to reveal discrepancies. So it becomes possible to identify signs of software deterioration, as well as violations of the integrity consensually established in the architecutral documents. In summary, the purpose of this work is the development of a technique capable to provide the necessary evidences that certain code does not conform to the intended design. In this way, we hope to contribute improving the software quality concerned to the presented problems.

Agradecimentos

Ainda que tivesse o dom da palavra, não teria a pretensão de resumir neste espaço o meu sentimento de gratidão. Sobretudo, minha gratidão incondicional à vida. Mas diante desta oportunidade, não posso me eximir de tentar. Nesta efêmera passagem pelo mundo, tenho a certeza que nossa mais valia é o que vivenciamos com o outro, pelo outro e para o outro. Qualquer feito não há razão de ser se não considerarmos o papel deste em nossas relações e vice-versa. Na minha trajetória de vida, não são poucos os "outros" a quem neste instante gostaria de dar um abraço de agradecimento:

- Primeiramente em meus pais, **Tina** e **Rogério**, pelo acolhimento, cuidados, ensinamentos e amor a mim dispensados. A vocês que nunca me deixaram faltar nada, saibam que o carinho e amparo que encontro em vocês é o que me fortalece e impulsiona. E que apesar de sermos poucos, aprendemos juntos a constituir o que uma grande família verdadeiramente deve ser.

- Em **Verônica**, minha paixão, amiga e companheira um daqueles bem apertado. Agradeço por todos os momentos que vivemos juntos, em especial aos quase dois anos que passamos longe de nossas famílias. Sem você, tudo teria sido muito mais difícil.

- Nos amigos de longa data que deixei na Bahia e que sempre torceram por mim. Um obrigado em especial à **Alex**, **Carlos**, **Cássio**, **Daniel**, **Fábio**, **Giovanna**, **Grace**, **Marcelo**, **Thiago** e **Victor**.

- Nos amigos que fiz no **GMF** e no inesquecível **Residencial Flamingo** (dentre outros **David**, **Ayslene**, **Valnir**, **Larissinha**, **Luciana** e **Sandrinha**). Espero ter retribuído à altura o companheirismo que fez esta caminhada ser muito mais suave. Sem dúvida, nos reencontraremos ainda.

- Nos professores **Jorge Abrantes** e **Dalton Serey** pela paciente orientação e incentivo oferecidos ao longo deste período.

- E por fim, em **todos** que de alguma forma contribuíram para que chegasse até aqui, mas que não foram mencionados: família, amigos, funcionários da universidade e colegas de curso o meu mais sincero obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

1.1 Contextualização

Passados mais de 50 anos da prática de desenvolvimento de software, uma grande parcela dos sistemas hoje desenvolvidos ainda não consegue superar completamente a tríade: baixa qualidade, altos custos e cronograma atrasado [JAW02]. Apesar dessa incômoda constatação, não se pode negar que houveram avanços significativos. O número de projetos de software concluídos com todas as funcionalidades, sem atrasos e com o orçamento previsto é cada vez maior [The99]. Dentre os diversos fatores responsáveis por estas estatísticas, podemos destacar o papel exercido pela arquitetura de software neste contexto. Cada vez mais é reconhecida a sua importância para o sucesso de projetos de software [SG96; Gar00]. Uma evidência disto é que, apesar de ser uma área relativamente nova, as pesquisas sobre o tema vêm alcançando rapidamente a maturidade devido principalmente ao interesse da indústria [Sha01]. A arquitetura, resumidamente, define a forma como o software é organizado em um alto nível de abstração, considerando seus principais elementos de computação e a interação entre estes. Seu surgimento como uma disciplina explícita de pesquisa para a Engenharia de Software, pode ser compreendido com base nas seguintes tendências [GP95]:

- **Mudança do foco de interesse.** O foco no desenvolvimento tem se voltado mais para aspectos arquiteturais do que para algoritmos e estruturas de dados. É natural que na medida em que o tamanho e a complexidade dos sistemas aumentam, o maior desafio se volte para os aspectos que envolvem a organização geral de sua estrutura. De modo

geral, os problemas passam a se concentrar em níveis mais altos de abstração. Estes incluem, por exemplo: protocolos de comunicação, sincronização, acesso aos dados, performance, atribuição das responsabilidades entre os elementos que compõem o software, etc...

- **Natureza da representação.** Atualmente, é comum observar a descrição de sistemas em termos de grafos compostos por componentes que interagem entre si. Até então, a forma mais usual de representação considerava quase que unicamente a modularização envolvendo o código fonte e sua correspondente dependência com o ambiente.

- **Popularização de estilos arquiteturais.** Uma instância de arquitetura refere-se a um sistema específico. Um estilo arquitetural, por sua vez, estabelece as restrições sobre a forma e estrutura de uma família de instâncias. Tais estilos possibilitam descrever sistemas complexos através de abstrações que tornam o sistema mentalmente tratável. Cliente-servidor, camadas, *pipe-filter* são exemplos de estilos já bastante conhecidos.

Mediante essas tendências, espera-se que práticas voltadas à arquitetura de software causem um impacto positivo em diversos aspectos do desenvolvimento de software [Gar00]. O primeiro, e mais direto, deles refere-se ao entendimento da solução, na medida em que se restringe a representação a um nível de abstração no qual pode ser mais facilmente compreendida. Além disso, a descrição arquitetural tende a ser um veículo comum de comunicação entre a equipe, o que conseqüentemente promove a integridade conceitual[1] do software. Sobretudo, a arquitetura define os limites em que o sistema deve evoluir, além de possibilitar sua análise ainda nos estágios iniciais de desenvolvimento.

No entanto, apesar do considerável progresso em se estabelecer as bases da disciplina de arquitetura de software, um problema ainda persiste [SAG+]: Será que o sistema está sendo implementado conforme foi definido na sua arquitetura? Caso não seja usada uma forma de se avaliar tal consistência entre modelo arquitetural e código, a validade de qualquer análise sobre este modelo estará sob suspeita. Este problema é comumente denominado de conformidade arquitetural, cujo desafio é determinar se um dado sistema foi ou está sendo implementado conforme especifica o projeto de sua arquitetura. Sem a devida confiança

[1] Uniformidade de conceitos e abstrações sobre a solução idealizada. Segundo Brooks [FPB95], uma das mais importantes características de um software.

desta relação de conformidade, a utilidade da descrição arquitetural torna-se quase nula. De maneira geral, existem três abordagens distintas para estabelecer tal relação: consistência por construção, análise estática, e por último, análise em tempo de execução. A primeira visa restringir a implementação, através do uso de linguagens e ferramentas específicas, de maneira a garantir a conformidade arquitetural. Embora possuam maior aplicabilidade, as duas últimas exigem uma forma razoável de mapear as estruturas presentes em dois níves de abstração bastante distintos: o código e a arquitetura. A maior dificuldade desse mapeamento deve-se, em grande parte, à falta de rastreabilidade de conceitos entre as linguagens de programação e a arquitetura. As notações usadas comumente para desenvolver software carecem da expressividade necessária para denotar conceitos usados nos níveis de abstrações mais altos. Conseqüentemente, decisões de design são difíceis de serem identificadas no código.

Uma das formas de minimizar esse problema, é utilizar uma arquitetura baseada em componentes. Todo sistema possui uma arquitetura, que por sua vez pode ser visualizada em termos da decomposição do software em partes. A abordagem baseada em componentes permite que estas partes permaneçam reconhecíveis no código. Assumimos como componente uma unidade de implementação que provê um conjunto de funcionalidades coesas e cuja interação com o mundo exterior se dá através de uma interface bem definida. Existem diferentes categorias de componentes: aqueles desenvolvidos especificamente para o sistema, componentes de propósito geral desenvolvidos internamente, e por último, componentes disponíveis comercialmente (COTS). As vantagens provenientes dessa forma de decomposição incluem:

- Manutenção da integridade da arquitetura ao longo do ciclo de vida do software
- Aumento da manutenibilidade do sistema
- Diminuição do impacto causado por mudanças
- Maior familiaridade do programador à arquitetura
- Reusabilidade de código

Pelo fato da descrição arquitetural ser uma abstração do sistema, nenhum modelo isoladamente é capaz de representar a realidade de uma maneira fidedigna. Por isso, o software

normalmente é descrito por um conjunto de modelos que tentam capturar sua arquitetura sob diferentes perspectivas. À essas perspectivas dá-se o nome de visão arquitetural. Do ponto de vista estrutural, a arquitetura baseada em componentes basicamente possui dois tipos de visões: estática e comportamental. A primeira compreende os componentes, suas interfaces, e as restrições relacionadas à interação entre eles. Já a segunda, representa a semântica do comportamento dos componentes e conseqüentemente da arquitetura como um todo. De fato, a visão estrutural constitui o cerne da arquitetura, motivo pelo qual é o foco da maioria dos trabalhos relacionados à conformidade.

1.2 Declaração do Problema

Ao longo do ciclo de vida de sistemas complexos - considerando desde o início de seu desenvolvimento até o fim de sua utilização - sua implementação tende a divergir do *design* esperado. Tal desvio torna o sistema difícil de entender, modificar e manter. Técnicas que visam avaliar a conformidade arquitetural podem ajudar na detecção de inconsistências entre o modelo e a implementação. Quando o *design* esperado não está documentado, existem diversos trabalhos que exploram o problema da reconstrução e extração arquitetural com o intuito de facilitar a compreensão de sistemas [WMSR00; SSC96a; GH00; JR97; KC99; SAG$^+$]. Alguns se utilizam do conhecimento tácito do desenvolvedor neste processo, outros tentam ser quase que totalmente automáticos. Apesar de não objetivarem a questão da conformidade arquitetural, estas técnicas acabam permitindo que os modelos extraídos sejam confrontados com uma certa arquitetura esperada. No entanto, tais trabalhos geralmente partem da premissa que há pouco domínio prévio sobre a arquitetura do software em questão. E por isso, talvez, a análise da conformidade seja vista como um recurso adicional, passível de ser obtida apenas de maneira manual.

Mesmo entre os trabalhos que abordam diretamente o problema da distância entre modelos de alto nível e código [MNS95; SSC96b], muitos não oferecem um mecanismo automático capaz de decidir quanto a existência ou não da relação de conformidade. Aqueles que o fazem, adotam uma abordagem demasiadamente complexa que acaba por inibir a efetiva utilização da técnica. A forma de descrever a arquitetura, por exemplo, pode ser expressiva o bastante para representar todas as suas características. No entanto, quanto mais

simples ela for menos custosa será a confecção e manutenção dos mesmos ao longo do processo de desenvolvimento. Apesar de tal simplificação reduzir o poder de expressão da notação, por outro lado possibilita que os próprios desenvolvedores sejam os responsáveis por manter a documentação arquitetural. Além da própria conformidade em si, assumimos que este seja um dos principais objetivos dos trabalhos na área. Afinal, dessa forma o desenvolvedor torna-se responsável pela compreensão e coesão do todo, ao invés de ser restringido a apenas uma visão local da solução.

Diante do exposto, o problema tratado neste trabalho é a falta de uma técnica suficientemente simples e eficaz para a verificação da conformidade arquitetural com o intuito de tornar controlável a evolução de sistemas.

O objetivo principal deste trabalho é a formulação de uma técnica para verificação da conformidade arquitetural, tanto na perspectiva estática quanto dinâmica, de sistemas de software. A técnica deve ser capaz de automaticamente fornecer as evidências de que certa implementação não obedece à arquitetura estabelecida. Isso sem perder de vista outros atributos considerados indispensáveis, tais como facilidade e simplicidade de uso.

1.3 Breve Resumo da Técnica

A técnica objeto deste trabalho, é denominada CASTOR (Component bASed archiTecture cOnformance monitoR). Esta basicamente é composta por três elementos:

- Descrição estrutural da arquitetura
- Descrição comportamental da arquitetura
- Processo de verificação da conformidade destas descrições em relação ao código

A descrição estrutural consiste na especificação de como as funcionalidades estão distribuídas entre os componentes da aplicação, e como estes cooperam entre si. Tal cooperação se dá através de canais lógicos de comunicação, aqui denominado de portas, que por sua vez oferecem e demandam o serviço de outros componentes. Para que os componentes possam interagir, é necessário que haja a declaração explicíta, na descrição, da conexão entre suas portas. Para tanto, é imperativo que estas sejam complementares, ou seja, que os serviços demandados por uma porta sejam oferecidos pela outra e vice-versa.

Como a técnica prevê o confronto da especificação com o código, deve existir um mapeamento entre os conceitos abstratos utilizados na descrição com as estruturas concretas existentes na implementação. Em CASTOR, este mapeamento é feito da seguinte forma: os componentes são mapeados para um conjunto de classes, enquanto que os serviços, tanto os oferecidos quanto os demandados pelas portas, são mapeados para um conjunto de métodos. Com isto, torna-se possível verificar se dois componentes quaisquer se comunicam somente através dos serviços previstos na especificação arquitetural.

Para saber se a implementação obedece às regras impostas pela especificação arquitetural, a técnica prevê duas etapas de verificação das descrições estrutural e comportamental, respectivamente:

1. **Análise estática** - Visa identificar comunicações não previstas entre os componentes definidos.

2. **Monitoração** - Em tempo de execução, eventos no sistema sendo verificado são capturados e continuamente reproduzidos no modelo que especifica seu comportamento.

Vale ressaltar, que obter um veredicto absoluto quanto a conformidade comportamental é um problema extremamente complexo, uma vez que demanda a investigação de todos os possíveis caminhos de execução do sistema sob análise. Ao invés disso, CASTOR se propõe a identificar inconsistências entre implementação e arquitetura na medida em que estas acontecem, ou seja, em tempo de execução.

1.4 Resultados e Relevância

Existe uma significativa distância entre a teoria e a prática em relação à arquitetura de software. Isto se deve principalmente à falta de padronização na área e à escassez de suporte ferramental adequado. Sobretudo, não é tão evidente aos desenvolvedores as vantagens concretas associadas ao custo de adotar práticas de cunho arquitetural. Faz-se necessário fornecer motivações objetivas, para tanto. Sendo assim, o principal resultado deste trabalho é a disseminação da importância da arquitetura de software, através de uma técnica que eleve a utilidade de sua documentação. A análise automática da conformidade propicia isto na me-

dida em que possibilita que tais artefatos possam ser confrontados com o código. A descrição arquitetural torna-se, portanto, um elemento ativo ao longo do processo de desenvolvimento.

Um outro resultado, decorrente deste, é a contribuição no sentido de amenizar o problema da deterioração de software. Ele decorre da natural evolução destes diante do surgimento de novos requisitos e restrições. O reflexo desse processo é a baixa qualidade do código, alta taxa de erros e o aumento dos custos do projeto. A literatura aponta para a arquitetura de software como uma abordagem capaz de, se não resolver, ao menos minimizar tais efeitos. Acreditamos que a verificação da conformidade arquitetural é uma condição necessária na tentativa de atacar o problema.

Atualmente, uma parcela significativa dos processos de desenvolvimento são baseados essencialmente no trabalho colaborativo, onde cada desenvolvedor tem autonomia para tomar decisões de projeto. Sendo assim, é preciso tomar cuidado para se manter a coesão do design - ou integridade conceitual - que emerge naturalmente ao longo de projetos sob tal perspectiva. Este é um outro problema no qual este trabalho pode contribuir, uma vez que força os desenvolvedores a compreenderem como o código produzido por eles se encaixa no contexto geral do software. Isto porque, caso haja equívocos no mapeamento entre código e arquitetura, realizados por eles, potencialmente a técnica os irá identificar. Com isso evita-se que se perca de vista a floresta (o software), por causa das árvores (partes do código).

1.5 Estrutura da Dissertação

O restante deste documento foi estruturado da seguinte forma:

Capítulo 2: Motivação do Trabalho Este capítulo tem como objetivo apresentar os resultados da tentativa de verificação do MyGrid, um software com características de concorrência e distribuição, desenvolvido na Universidade Federal de Campina Grande (UFCG). Esse relato foi incluído no texto, pois, tal esforço constituiu uma importante fonte de subsídios para este trabalho. Discutimos as motivações, os experimentos realizados e suas conclusões.

Capítulo 3: Fundamentação Teórica Na Fundamentação teórica apresentamos os conceitos necessários para a melhor compreensão da técnica. Inicialmente, são abordadas as principais definições referentes à componentes e arquitetura de software, e como estas áreas

se relacionam. Em seguida, discutimos brevemente as formas de documentar uma arquitetura e como esta atividade é inserida no processo de desenvolvimento. Por último, discutimos a conformidade arquitetural, sua importância e as principais abordagens para verificá-la.

Capítulo 4: Técnica CASTOR Este é o principal capítulo da dissertação, no qual é apresentada a técnica proposta. Para introduzí-la, utilizamos um pequeno sistema denominado KWIC. A idéia é demonstrar CASTOR através de sua aplicação neste referido software. Inicialmente, é mostrado o formato da descrição arquitetural. Logo em seguida, é discutido como é feito o mapeamento entre modelo e código, e por fim como se dá a verificação da conformidade arquitetural.

Capítulo 5: Estudo de Caso Como forma de fornecer evidências que o objetivo do trabalho foi alcançado, realizamos um estudo de caso junto ao mesmo software com o qual iniciamos a pesquisa (MyGrid). O objetivo é demonstrar que a aplicação da técnica seria suficiente para evitar os problemas ocorridos ou que, uma vez corrigidos, dificilmente eles apareceriam de novo.

Capítulo 6: Considerações Finais Baseado nos resultados obtidos, o último capítulo será destinado à apresentação das conclusões e da perspectiva de trabalhos futuros.

Capítulo 2

Motivação do Trabalho

Este capítulo tem como finalidade relatar as tentativas de verificação realizadas junto ao projeto OurGrid, cujo objetivo é o desenvolvimento de um *middleware* para a execução de aplicações *Bag-Of-Task* (BoT)[1] em grades computacionais [FK99]. O MyGrid, nesse contexto, é parte integrante dessa plataforma e responsável pelo escalonamento das tarefas entre as unidades de processamento distribuídas pela rede. Tais experimentações visavam, em última instância, a melhoria da qualidade do módulo escalonador através principalmente da detecção e conseqüente redução de erros do software. O resultado deste esforço se caracterizou como importante fonte de subsídios para este trabalho, motivo pelo qual dedicaremos um capítulo para discutí-lo. Inicialmente, iremos abordar o contexto do projeto, bem como caracterizar, em linhas gerais, o funcionamento e arquitetura do sistema. Em seguida, serão descritos os problemas que motivaram nossas experimentações, as tentativas de verificação do software e finalmente a solução encontrada pela própria equipe de desenvolvimento.

2.1 Caracterização do MyGrid

Antes de caracterizarmos o MyGrid, faz-se necessário contextualizá-lo dentro do projeto OurGrid. O objetivo do projeto é a criação de uma grade computacional aberta, na qual qualquer um possa ceder ou utilizar processadores ociosos. A grade, na verdade, pretende ser uma grande comunidade *peer-to-peer* baseada na troca de favores entre seus integrantes. Para viabilizar tal objetivo, está sendo desenvolvida uma plataforma, denominada também

[1] Aplicações cujo processamento pode ser dividido em tarefas independentes

de OurGrid, para dar suporte à execução das aplicações bem como à negociação dos recursos computacionais. Apesar de sua simplicidade, aplicações BoT são utilizadas em uma variedade de cenários incluindo mineração de dados, processamento de imagens, buscas exaustivas, biologia computacional, dentre outros. O projeto está sendo desenvolvido desde 2001 no Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG), em parceria com a empresa Hewlett Packard (HP).

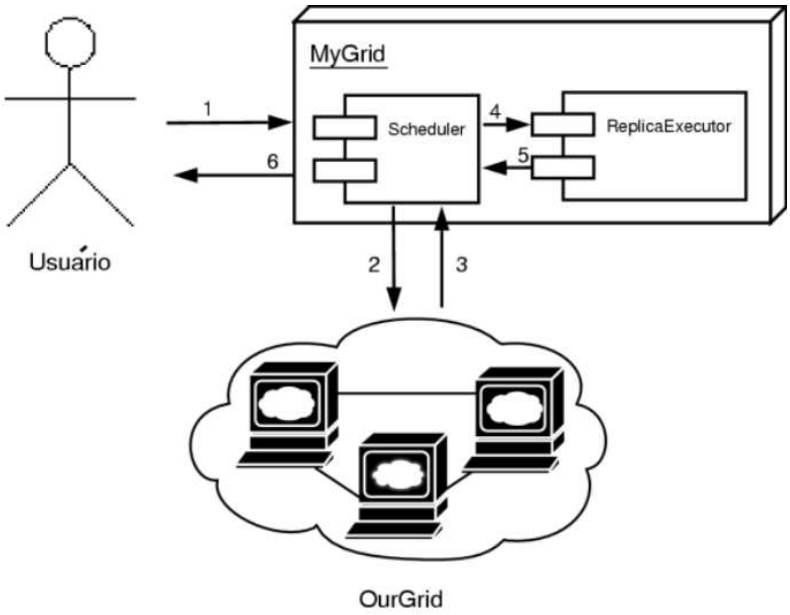

Figura 2.1: Visão geral sobre o funcionamento do MyGrid

Uma vez que o usuário final do OurGrid tenha o interesse de processar algo na grade, é necessário então a figura de um agente escalonador que possa eficientemente alocar processadores às tarefas submetidas. O MyGrid é justamente o módulo responsável em exercer este papel e cuja complexidade está intrinsecamente associada às características típicas de sistemas distribuídos, como paralelismo e concorrência, por exemplo. Na Figura 2.1 é mostrada uma visão simplificada do processo de utilização da grade. Inicialmente, o usuário submete sua aplicação ao MyGrid como um conjunto de tarefas (1), este por sua vez, através do componente *Scheduler*, solicita à comunidade máquinas ociosas que possam ser utilizadas para processamento (2). O OurGrid então cede as máquinas ao *Scheduler* (3), que delega a outro componente do MyGrid, chamado *ReplicaExecutor*, a responsabilidade de efetivamente executar as tarefas (4). Ao término do processamento da tarefa, o *ReplicaExecutor* notifica o

Scheduler (5), que finalmente informa ao usuário (6).

Para dar uma idéia de seu porte, ao longo do tempo, o MyGrid contou em média com uma equipe de 8 pessoas e atualmente possui cerca de 30.000 linhas de código. Inicialmente, nos foi relatado que usuários e desenvolvedores queixavam-se de falhas no funcionamento desse módulo que ninguém conseguia identificar a causa. Em algumas situações o sistema simplesmente parava de responder. Os esforços foram então concentrados na identificação do que parecia se tratar de uma situação de *deadlock*. No entanto, havia uma restrição de que as atividades de verificação deveriam interferir o mínimo possível no ambiente de produção. Ou seja, elas deveriam ser realizadas por pessoas externas ao projeto, e a participação da equipe de desenvolvimento iria se restringir a reuniões para que dúvidas sobre o sistema pudessem ser dissolvidas. Portanto, o fato de *deadlock* ser um alvo constante das técnicas formais de verificação e devido a experiência do grupo nessa área adotamos esta abordagem na tentativa de solucionar o problema. O processo de desenvolvimento que mais se aproxima do adotado no desenvolvimento do MyGrid é XP (Extreme Programming). Não o classificamos propriamente como XP, pois diversas práticas dessa metodologia, como programação em pares, testes de aceitação, estórias do usuário, dentre outras, não são efetivamente utilizadas.

2.2 Verificação

Existem diferentes técnicas para verificação formal de sistemas, Katoen [Kat] destaca as seguintes: simulação, testes, métodos indutivos e verificação de modelos (*model checking*). Com simulação e testes apenas alguns caminhos de execução são observados para identificação de erros. Na simulação a análise é feita sobre um modelo abstrato. No caso de testes, a análise é feita sobre o próprio programa [MS01]. Em ambos os casos, normalmente não é viável analisar todos os caminhos de execução. Dessa forma, é possível constatar somente a presença, mas não a ausência de falhas. Métodos indutivos, por sua vez, tentam utilizar provas matemáticas formais para garantir propriedades especificadas. Mesmo com uso de ferramentas de suporte, este tipo de verificação exige um alto grau de especialização dos engenheiros de software envolvidos e normalmente é bastante custosa. Por fim, verificação de modelos é um processo automatizado para checagem de propriedades sobre o espaço de es-

tados[2] gerado a partir de um modelo (que pode ser extraído a partir do próprio programa, em alguns casos) [CGP99]. Optamos por utilizar a verificação de modelos devido a sua eficácia em indicar a presença ou não de uma determinada propriedade, com um custo relativamente baixo se comparado a métodos indutivos.

Sendo assim, adotamos duas estratégias distintas para aplicação da técnica:

- Utilizar uma linguagem de especificação formal para descrever as características principais do programa, incluindo aspectos comportamentais. O artefato resultante da aplicação desta estratégia é um modelo abstrato, que descreve o comportamento esperado do MyGrid. Dessa forma é possível gerar o espaço de estados e identificar um conjunto de propriedades sobre o mesmo.

- Através de ferramentas apropriadas, aplicar a verificação diretamente sobre o código do programa. Denominaremos está estratégia de verificação automática, uma vez que os princípios são os mesmos de verificação de modelos clássica: geração do espaço de estados (neste caso, a partir do código fonte) e posterior análise de propriedades. A vantagem obtida com a utilização desta estratégia é que elimina-se a etapa de construção manual do modelo abstrato do programa. Por sinal, etapa esta bastante suscetível à introdução de erros.

Antes de efetivamente ser iniciado o trabalho de verificação, houve um período dedicado à compreensão do funcionamento e arquitetura do sistema. A principal dificuldade foi a falta de documentação existente. O único artefato disponível era o código fonte. Foi preciso, então, realizar atividades de engenharia reversa para a construção de alguns diagramas UML. O objetivo era que a documentação gerada facilitasse nossa compreensão sobre o software e servisse, ao mesmo tempo, como um referencial de comunicação com a equipe de desenvolvimento do MyGrid.

2.2.1 Estratégia com Modelagem Manual

A linguagem escolhida para descrever o MyGrid foi RPOO [Gue02] (Redes de Petri Orientadas a Objetos). A escolha foi justificada em função da linguagem ter sido projetada

[2] Conjunto de todos os estados, ou caminhos de execução, possíveis que um sistema pode alcançar

com o objetivo de tornar mais "fácil" o uso de formalismos no desenvolvimento de sistemas orientados a objetos. Neste sentido, RPOO apresenta uma série de vantagens sobre outras linguagens formais para especificação com características afins (ver discussão proposta por Guerrero, em [Gue98]): RPOO foi projetada com base na integração dos conceitos de redes de Petri [Jen92] com orientação a objetos (OO) preservando as características originais de ambos paradigmas. A linguagem permite a construção de diagramas de classes ao estilo convencional de modelagem (podendo mesmo ser utilizado UML), e as redes de Petri são utilizadas para descrever o comportamento interno de cada classe. Um conjunto de regras define o efeito das ações executadas pelos objetos, descritos pelas redes de Petri, sobre o estado global do sistema.

Infelizmente, a ferramenta de suporte para simulação e geração do espaço de estados de modelos RPOO não foi concluída antes do término dos experimentos, conforme havia sido previsto. Isso inviabilizou que tivéssemos resultados mais conclusivos, em relação ao problema original, para esta estratégia de verificação. A seguir, são enumeradas algumas conclusões mais gerais sobre a modelagem do MyGrid utilizando uma linguagem formal, como RPOO.

1. **É custoso modelar comportamento do software na perspectiva do código, mesmo que abstraindo aspectos periféricos.** Foram feitas duas versões do modelo do MyGrid utilizando RPOO. Na primeira delas, muito pouco foi abstraído em relação ao código da aplicação, o que resultou em modelos demasiadamente complexos. O motivo disso foi o receio natural de desconsiderar um aspecto importante para a propriedade de interesse, nesse caso o *deadlock*. Na segunda versão, apesar de continuarmos utilizando o código como base para construção do modelo, apenas o que julgamos ser o cerne da aplicação foi capturado. Mesmo assim, o custo de construção do modelo não foi significativamente reduzido, uma vez que a todo instante era gasto muito tempo na análise do que deveria ou não ser considerado.

2. **É difícil manter os modelos sincronizados com o estado atual do projeto.** Mesmo tendo em vista a segunda versão, o tempo dedicado à construção do modelo não permite mantê-lo consistente com o programa, uma vez que as alterações no código ocorrem muito rapidamente.

2.2.2 Verificação Automática

Outra estratégia utilizada para verificação do MyGrid foi aplicar a técnica de verificação de modelo diretamente sobre o código do programa. A idéia é produzir o espaço de estados a partir do próprio código fonte, ou de um modelo abstrato extraído deste, para identificar propriedades. Os resultados mais expressivos desta abordagem, até então, foram obtidos com as ferramentas Java PathFinder [VHBP00] e Bandera [CDH+00], ambas empregadas neste experimento. Bandera é uma ferramenta que converte código fonte Java para a linguagem de entrada de verificadores, como SMV [K.L92], SPIN [Hol97] e JPF. Tal conversão pode ser acompanhada de uma etapa de abstração, com ou sem intervenção do usuário, que em última instância visa excluir do modelo partes que não interfiram no aspecto sob análise. As propriedades são definidas com ajuda de padrões para especificação de propriedades [DAC98] e de uma linguagem específica que pode ser anotada no próprio código do programa como comentários, ao estilo Javadoc. O JPF, por sua vez, é um verificador cuja linguagem de entrada são *byte-codes* Java. Assim como no Bandera, propriedades usuais como *deadlock*, por exemplo, e checagem de tipos podem ser realizadas sem que seja necessário nenhuma especificação. Propriedades mais complexas requerem a utilização de uma biblioteca própria do JPF e instrumentação do código. Um aspecto bastante positivo é o fato desta abordagem onerar pouco o processo de desenvolvimento, uma vez que nenhum novo artefato precisa ser produzido e mantido. Além disso, há garantia de que os erros descobertos pertencem ao programa e não a um modelo abstrato que, em muitos casos, pode não estar consistente com o código.

As ferramentas funcionavam bem com pequenos exemplos, porém, para sistemas reais como o MyGrid, houveram alguns problemas. No caso específico do Bandera, não havia disponível uma versão estável da ferramenta na ocasião. Fomos então obrigados a utilizar uma versão *beta* que não suportava ainda todos os construtos da linguagem Java. Sendo assim, acabaram ocorrendo vários erros durante a tradução do programa para a linguagem de entrada dos verificadores. A verificação com o JPF apresentou resultados mais consistentes. No entanto, para isso foi necessário um grande esforço para alterar o código da aplicação com o intuito de evitar o problema da explosão do espaço de estados[3]. Em um processo árduo de

[3] Problema clássico de verificação de modelos, na qual o espaço de estados não consegue ser representado devido ao seu tamanho

abstração, muitas classes, chamadas de métodos e atributos tiveram que ser apagados e, ao final, o JPF não detectou qualquer problema. No entanto, devido às drásticas alterações, o código que foi verificado não correspondia mais ao MyGrid. Sendo assim, de pouca utilidade foram os resultados desta tentativa, em particular.

Da experiência concluímos ser interessante a proposição de métodos que nos permitam modularizar o sistema de forma que seja fácil identificar e separar as principais operações de questões "secundárias". Sendo assim, as técnicas de verificação poderiam se ater ao cerne da aplicação, tornando clara as partes do sistemas que deveriam ser abstraídas. Geração de arquivos de log, mensagens para usuários, validação de senhas, etc. são alguns exemplos do que estamos considerando como secundárias, se observarmos as funcionalidades principais do sistema. Uma proposta que tem sido pesquisada e que pode ser considerada na adoção de uma abordagem deste tipo é a utilização da programação orientada a aspectos [KLM+97].

2.3 Solução

Ao final dos experimentos, não fomos capazes de indicar a causa da falha. Sendo assim, diante do crescente número de problemas com o software, a equipe de desenvolvimento resolveu reestruturar completamente a arquitetura do MyGrid. As principais ações foram as seguintes:

1. A interface dos componentes foi melhor definida com o uso do padrão de projeto *Facade*;

2. A comunicação entre componentes passou a ser feita única e exclusivamente através da sua fachada;

3. Para tornar a aplicação *thread-safe*, os componentes ganharam uma fila de mensagens, pela qual as demais linhas de execução (do próprio componente ou de outros) deveriam utilizar para se comunicar;

4. Os métodos da fachada encapsulavam a mensagem e a colocava no final da fila;

5. Existia uma *thread* em cada componente responsável por atender às requisições e tomar as devidas ações;

6. O *Scheduler*, que era a provável fonte de problemas, foi refatorado e teve seu comportamento melhor definido.

Com essas ações, os problemas foram eliminados, e segundo impressões de membros da própria equipe, o código ficou também muito mais organizado e fácil de entender. Esse cenário é bem caracterizado na literatura como sendo uma situação típica de deterioração de software, um problema comum a quase totalidade de sistemas [Par94]. Ele decorre da natural evolução destes diante do surgimento de novos requisitos e restrições. Alguns autores chegam a alegar que não importa o quão criteriosa seja a construção e manutenção do software, seu design tende, com o passar do tempo, a um nível tal de erosão que reprojeta-lo torna-se imperativo [vGB02]. O reflexo desse processo é a baixa qualidade do código, alta taxa de erros e o aumento dos custos do projeto. Algumas causas são apontadas para esse fenômeno, dentre elas temos:

- **Rastreabilidade de conceitos**. As notações (leia-se linguagens) usadas comumente para criar software carecem da expressividade necessária para denotar conceitos usados nos níveis de abstrações mais altos. Conseqüentemente, decisões de design são difíceis de serem identificadas diretamente no código.

- **Integridade conceitual**. Quanto menos os desenvolvedores compreendem a arquitetura e os conceitos envolvidos na solução, mais propensos estão a tomarem decisões ruins no que se refere ao design do software.

- **Consertos rápidos**. A maneira apropriada para corrigir um erro é analisá-lo, projetar uma solução, implementar e testar. Normalmente devido a pressões de tempo ou até mesmo a pouca experiência da equipe, faz com que os desenvolvedores deixem de seguir essas etapas. Pouca atenção é dada ao design nessa tarefa o que geralmente resulta na adição de novos erros.

- **Evaporação das decisões de design**. As decisões de design tomadas ao longo de um projeto interagem de tal forma que constantemente elas devem ser reavaliadas. O problema é que algumas decisões de fases anteriores do processo (e suas motivações) simplesmente são esquecidas ou pouco compreendidas pela equipe. Com isso, aumenta-se

o risco que escolhas futuras se baseiem em decisões que não são mais apropriadas num dado momento.

Uma das características de um bom design, é a sua capacidade de acomodar mudanças futuras esperadas. Isto entra em conflito com a essência iterativa da atual tendência de desenvolvimento. Não significa que tal abordagem seja inadequada, na verdade a prática tem demonstrado justamente o contrário. O que faz-se necessário portanto, é amenizar (e de certa forma acompanhar) a erosão do design diante dessa perspectiva. Um outro fator que torna esse cenário ainda mais evidente é o crescente tamanho e complexidade do software que é atualmente desenvolvido.

2.4 Sumário

Apesar das tentativas de verificação não terem tido sucesso, tendo em vista o objetivo inicial, a experiência como um todo foi bastante proveitosa e acabou por fornecer subsídios importantes para nossa pesquisa. Abaixo seguem algumas constatações, que mais tarde iriam ser utilizadas para justificar a criação da técnica proposta neste trabalho:

- É melhor evitar que erros sejam introduzidos, do que tentar eliminá-los posteriormente;
- Uma arquitetura bem definida, e conhecida por todos é fundamental para a qualidade do software;
- A deterioração de software pode ser amenizada mantendo-se a conformidade arquitetural, que por sua vez implica na integridade conceitual do projeto;
- Uma arquitetura baseada em componentes estabelece uma visão mentalmente tratável do software, ao mesmo tempo que suas estruturas se mantêm reconhecíveis no código;
- Qualquer proposta para atacar o problema deve primar pelo baixo custo de sua aplicação.

O capítulo seguinte se concentra na descrição e definição de alguns conceitos essenciais para a compreensão deste trabalho. São eles: componente de software, arquitetura de software e conformidade arquitetural.

Capítulo 3

Fundamentação Teórica

Neste capítulo, abordaremos o estado da arte da teoria sobre a qual este trabalho está fundamentado. Inicialmente, serão apresentadas as diferentes definições existentes para componente de software e qual destas assumiremos neste trabalho. Discutiremos também as diversas visões sobre arquitetura de software, tanto da academia como na indústria, bem como de que forma estas se relacionam com componentes. Além disso, descreveremos os principais conceitos relativos a arquitetura de software, as formas atualmente disponíveis de descrevê-las e por último, a importância da conformidade arquitetural no que tange a qualidade de software.

3.1 Componente de Software

São inúmeras as definições para o termo componente na literatura. Bachmann et al [Bac00] explora as diferentes perspectivas para o uso de componentes no desenvolvimento de software. Ele inicia a discussão mostrando que com base apenas no significado literal da palavra, todo software engloba componentes. Esse fato advém do uso da técnica padrão para resolver problemas em computação - "dividir para conquistar". No entanto, cada forma de decompor e organizar um sistema resulta em diferentes tipos de componentes. Isso, de certa forma, explica a falta de consenso sobre a definição de um componente.

O todo (software) engloba as partes (componentes), que por sua vez constituem o todo. No entanto, essas partes podem diferir substancialmente dependendo da perspectiva adotada. Devemos diferenciar componentes oriundos da decomposição durante um processo de *de-*

sign, daqueles já disponíveis para a composição em novos sistemas. Nesta dissertação em particular, adotaremos a perspectiva de decomposição, porém, de qualquer forma, é válido explorarmos um pouco esse assunto na visão do que vem sendo chamado de *Engenharia de Software Baseada em Componentes* (ESBC).

ESBC [Crn01] é uma sub-disciplina da Engenharia de Software que trata da construção de sistemas a partir da integração de componentes, do desenvolvimento desses componentes como entidades reusáveis e da manutenção e evolução destes sistemas pela troca e adaptação de suas partes. A proposta é que dessa forma, seria possível desenvolver softwares com menos custos e em menos tempo, preservando ou até mesmo aumentando a qualidade do software se comparado ao seu desenvolvimento por completo. Nesse contexto, a característica mais importante de um componente é a sua reusabilidade, e com ela estão associados alguns riscos que podem comprometer o sucesso dessa abordagem. O principal deles é a dificuldade de integração de componentes de propósito geral, para satisfazer a requisitos específicos de uma aplicação. É provável que a combinação dos componentes não consiga preencher as necessidades do sistema, ou até mesmo que características e funcionalidades não desejadas sejam incorporadas. Além disso, devido a reusabilidade, tais componentes tendem a ser mais complexos (conseqüentemente mais difíceis de desenvolver e usar) do que unidades de propósito específico. O objetivo da ESBC é, portanto, minimizar os riscos levantados de modo a se usufruir as vantagens do Desenvolvimento Baseado em Componente (DBC).

Nesse contexto, componentes disponíveis comercialmente são comumente chamados de COTS (*Commercial Off The Shelf*). Para viabilizar de fato o desenvolvimento baseado em componentes, além da disponibilidade comercial de uma ampla gama de componentes, faz-se necessário a padronização de frameworks que regulem as regras de composição dos mesmos. Com esse intuito, foram concebidos diversos modelos de componentes tais como CORBA, COM, Java 2 Enterprise Edition e .NET. Uma vez que o modelo é obedecido, é possível inclusive que o componente seja construído em qualquer linguagem ou ambiente de desenvolvimento. Szyperski captura a noção de componente discutido até aqui, em uma definição bastante referenciada na literatura [Szy98]:

> Um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas, que pode

ser distribuída independentemente e está sujeita a composição com outras partes.

Um dos aspectos mais importantes de um componente é a clara separação entre sua interface e a implementação. Sua interação com o mundo exterior se dá através de especificações bem definidas acerca do que o componente requer para o seu correto funcionamento e de quais os serviços por ele fornecidos. O usuário de um componente, portanto, não precisa saber sobre o funcionamento interno do mesmo, apenas como usá-lo. A forma de definir tal interface normalmente incluem especificação com semântica formal ou contratos. Diferentes processos de desenvolvimento, já estabelecidos na Engenharia de Software, podem ser aplicados a DBC. No entanto, algumas atividades, como as mostradas abaixo, são comuns a todos eles.

- Elencar uma lista de possíveis componentes a serem utilizados no sistema;
- Selecionar os componentes que atendem aos requisitos do sistema;
- Possivelmente, desenvolver algum componente internamente;
- Adaptar os componentes selecionados;
- Integrar e implantar os componentes utilizando algum *framework*;

Vamos agora explorar a noção de componente como uma unidade de decomposição, ao invés de composição. Para gerenciar melhor a complexidade inerente ao desenvolvimento de software é natural dividí-lo em partes. A forma mais comum de fazer isso é separar as funcionalidades relacionadas em diferentes componentes. Tal separação pode inclusive, ser feita no nível de requisitos. Porém, existem outros motivos, de cunho mais organizacional, que justificam também a componentização de um sistema. Por exemplo, a definição precisa de componentes permite a total paralelização e distribuição dos esforços de desenvolvimento.

É um desafio combinar o processo de decompor um sistema em partes com o de integrar componentes já disponíveis no desenvolvimento de software. Obviamente, não se espera que componentes oriundos de uma abordagem *top-down* estejam automaticamente prontos para serem utilizados em outros contextos. Porém, independente do tipo de componente ou se a perspectiva utilizada é a de composição ou decomposição, componentes não são usados de maneira isolada. Para, de fato, agregarem algum valor, em qualquer que seja o domínio da

aplicação, eles precisam interagir e formar uma estrutura que, de certa forma, irá determinar as propriedades do sistema. Tal estrutura é usualmente chamada de arquitetura de software.

3.2 Arquitetura de Software

Assim como componentes, arquitetura de software possui também uma vasta lista de definições na literatura, como pode ser visto em [SEI05]. No entanto, indiscutivelmente, a mais aceita e comum delas é dada por Bass et al [BCK03]:

> Arquitetura de software de um programa é a estrutura de estruturas do sistema que compreende os elementos de software, as suas propriedades externamente visíveis e o relacionamento entre eles.

Para ser mais preciso, a definição mais citada é a da primeira edição do livro, na qual o trecho "elementos de software" aparece como "componentes de software" [BCK98]. Como podemos ver, arquitetura de software e componentes são duas áreas que estão intimamente ligadas. Todo sistema possui uma arquitetura (quer ela esteja documentada ou não), que por sua vez pode ser visualizada em termos da decomposição do software em componentes. A questão é que tais componentes identificados na arquitetura podem não ser mais reconhecíveis durante a execução do sistema [Crn01]. Por exemplo, em aplicações monolíticas, a arquitetura especificada durante as fases de *design* é transformada, em tempo de execução, em um único bloco de código executável. Por isso, "componente" foi substituído na definição de Bass et al, uma vez que o uso do termo parece exigir que os componentes arquiteturais devem necessariamente ser mantidos íntegros também na implementação. Apesar de que em um sistema baseado em componentes (seja na perspectiva de composição ou decomposição), isto de fato acontece: a arquitetura permanece obrigatoriamente nítida ao longo de todo o processo de desenvolvimento. Um aspecto importante a ser considerado sobre essa definição é que um sistema não possui apenas uma estrutura, mas muitas, superpostas umas sobre as outras [FPB95]. Esta característica é capturada pelo conceito de visão arquitetural (ver Seção 3.2.2). Um segundo ponto, é que somente as propriedades visíveis externamente são relevantes à arquitetura de software. Ou seja, somente aquilo que é observável através da interface do componente.

Uma outra definição, um pouco diferente da anterior, mas não menos referenciada foi concebida por Perry e Wolf em 1992 [PW92], na qual é estabelecida a seguinte fórmula:

Arquitetura de Software = *Elementos, Forma, Argumento*

Nesse contexto, uma arquitetura de software é um conjunto de elementos arquiteturais que possuem uma forma em particular. O ítem *Argumento* da fórmula, refere-se à motivação pela qual uma determinada escolha foi tomada na arquitetura em detrimento de outra. As duas definições, apresentadas até aqui, guardam muitas semelhanças entre si. *Elementos* poderiam ser vistos como componentes e *Forma* como sendo a estrutura citada por Bass et al. A maior diferença, portanto, se encontra no fato de que nessa última definição é incluído explicitamente um terceiro e não menos importante ítem: a lógica que explica as escolhas feitas na definição da arquitetura. De acordo com essa definição, seria impossível portanto, extrair completamente a arquitetura de um sistema a partir exclusivamente do código através de técnicas de engenharia reversa. Um último aspecto que não ficou evidente a partir das definições, é que não somente os componentes, mas também os conectores (elementos que interligam tais componentes) são também tratados como entidades de primeira ordem na arquitetura. O motivo disso é separar claramente as unidades de processamento daquelas responsáveis exclusivamente da comunicação. Conectores exercem uma função essencial, podendo inclusive modificar completamente as características de uma determinada arquitetura de software. Chamadas de métodos, mensagens, dados compartilhados, etc..., são exemplos de conectores que servem para unir os elementos arquiteturais.

Finalmente, podemos estabelecer que a importância da arquitetura de software advém principalmente dos seguintes fatores:

- Reflete as primeiras decisões de design
- Estabelece a essência da solução (aquilo que é crítico)
- Importante veículo de comunicação
- Evidencia os requisitos não-funcionais

3.2.1 Arquitetura de Software no Processo de Desenvolvimento

Arquitetura de software é tradicionalmente associada às primeiras fases do ciclo de desenvolvimento. No entanto, isso vem mudando aos poucos e muitas referências, tanto na indústria como na academia, já reconhecem a importância de atividades voltadas para a arquitetura em outras etapas do processo. O artefato de arquitetura de software tem sido gradualmente desvinculado de uma fase particular do ciclo de vida de um sistema [BGHK02]. De acordo com a recomendação 1471-2000 da IEEE, a arquitetura de software contribui para o desenvolvimento, operação e manutenção de um sistema desde sua concepção inicial até ser retirado de uso. E como tal, ela deve ser entendida dentro do contexto do ciclo de vida do software, e não mais como uma atividade isolada em uma determinada etapa [IEE00]. Há sugestões na literatura, por exemplo, que a gerência de projetos teria muito a ganhar em ser centrada na arquitetura. O produto e o processo afetam um ao outro, e a arquitetura seria o elo de ligação entre eles [PBP01].

Entre os processos de desenvolvimentos mais usados hoje na indústria encontra-se o RUP (Rational Unified Process) [Kru00]. Dentre as suas recomendações, uma delas sugere a adoção de uma arquitetura baseada em componentes, como uma das seis melhores práticas a fim de reduzir os riscos inerentes ao desenvolvimento moderno de software. Já em relação às metodologias ágeis, muito tem se discutido sobre arquitetura. Alguns críticos alegam que uma das causas da dificuldade de tais abordagens serem aplicadas a projetos maiores se deve justamente à ausência de práticas voltadas para arquitetura de software [RME03]. Há uma demasiada importância do código sobre os demais artefatos, inclusive os que tratam de aspectos arquiteturais, e com isso a equipe acaba ficando sem um referencial de comunicação [Wes02]. Em XP (eXtreme Programming) [Bec00] por exemplo, o artefato *System Metaphor* - que visa descrever a solução em função dos seus principais conceitos - foi apontada em pesquisas como a prática menos utilizada e a mais mal entendida entre seus adeptos [TH03]. O próprio Beck em seu livro afirma que: “Arquitetura é tão importante em projetos XP, como em qualquer outro projeto de software” [Bec00]. No entanto, toda discussão sobre *metaphor* e arquitetura ocupa menos de 2 das 190 páginas do livro.

3.2.2 Documentação Arquitetural

Produzir uma documentação de fato representativa do software, que seja utilizada na prática e sobretudo mantê-la atualizada ao longo do desenvolvimento sempre foi um dos desafios da indústria de software. Apesar de sua importância, arquitetura de software ainda é largamente documentada de uma maneira *ad-hoc* e informal. Indício disso, é que até recentemente era comum encontrar modelos arquiteturais representados graficamente por quadrados e linhas, que apesar de fornecerem uma visão geral da arquitetura, deixavam indefinidas diversas questões [MT00]. Nesse contexto, Garlan observa que [GMW97]:

- A arquitetura normalmente é pouco compreendida pelos desenvolvedores devido à sua ambigüidade;
- As restrições arquiteturais não são observadas, durante a evolução do sistema, como deveriam;
- Praticamente não há ferramentas de apoio para os arquitetos de software;
- As decisões arquiteturais não tomam como base princípios sólidos da engenharia;
- Não se consegue analisar a documentação arquitetural quanto à sua consistência e completude.

Em resposta a esses problemas, pesquisadores têm proposto diversas notações formais para representar e analisar arquiteturas de software. Tais notações são chamadas de ADLs (Architectural Description Languages), e normalmente contam também com uma suíte variada de ferramentas. Uma ADL define os elementos básicos que devem ser utilizados para documentar uma arquitetura. Existem inúmeras ADLs, que concebidas sob diferentes critérios resultaram, naturalmente, em linguagens com conceitos e características distintas. Uma análise geral sobre as diferentes ADLs pode ser encontrada em [MT00]. No entanto, pela própria imaturidade ainda dessas linguagens e a conseqüente falta de padronização de conceitos, ADLs ainda não são utilizadas em larga escala pela indústria. Uma outra alternativa é utilizar uma linguagem mais genérica e já estabelecida, como UML ou uma de suas extensões, para documentar arquiteturas. Garlan e Cheng fazem uma análise sobre o quão apropriadas, para descrições arquiteturais, são UML [GCK02] e UML-RT [CG01].

Contudo, vale ressaltar que nenhuma abstração é capaz de representar fielmente a realidade e que nenhum modelo poder ser utilizado para todos os propósitos. Diferentes visões são necessárias para representar aspectos distintos da arquitetura. Por exemplo, uma visão arquitetural pode se restringir em documentar a estrutura do sistema em termos do agrupamento lógico do código, enquanto outra visão se concentra no padrão de interação em tempo de execução. Uma visão é, portanto, a representação do sistema em uma perspectiva relacionada a um conjunto de propriedades de interesse. Alguns autores sugerem um conjunto básico de visões, que em conjunto, seriam capazes de representar a essência de uma arquitetura. A mais conhecida delas talvez seja a proposta por Krutchen [Kru95], denominada de visões 4+1, no qual as visões lógica, de processo, física e de desenvolvimento são complementadas e interconectadas com uma visão de caso de uso. Existem sugestões similares de outros autores, porém abordagens recentes reconhecem o fato de que nenhum modelo fixo de visão é apropriado para todos os sistemas [CGB+02].

3.3 Conformidade Arquitetural

Na última década, devido ao reconhecimento da importância da arquitetura de software para o sucesso de sistemas complexos, considerável esforço foi despendido na pesquisa e desenvolvimento de notações, ferramentas e métodos de suporte ao projeto arquitetural [CGB+02; MT00].

No entanto, o problema em se avaliar a conformidade entre o sistema e seu projeto arquitetural ainda está em aberto [SAG+]. A validade de toda e qualquer análise tendo como base a modelagem arquitetural parte da premissa que este representa fielmente o software. Sem essa confiança, sempre restará dúvidas se a documentação arquitetural é de fato uma representação abstrata do sistema. Atualmente existem três abordagens para determinar ou restringir a relação entre a arquitetura de software e sua implementação [SAG+]:

- **Consistência por construção**. Isto pode ser concretizado embutindo construções arquiteturais na própria linguagem de programação [ACN02] ou através da geração de código a partir de uma definição arquitetural mais abstrata [SG96]. Ambas as técnicas têm aplicabilidade limitada, pois exigem o uso de linguagens, ferramentas e estratégias de implementação específicas.

- **Análise estática do código**. Essa técnica prevê a extração da arquitetura a partir do código para posterior comparação com o modelo idealizado. Para tal, a implementação deve estar suficientemente organizada para que, a partir da sua modularização e padrões de codificação, seja possível identificar os elementos arquiteturais. O problema aqui é que muitos aspectos sobre o comportamento do sistema são ignorados devido a impossibilidade de serem analisados estaticamente.

- **Análise do comportamento em tempo de execução**. Esta técnica é relativamente pouco explorada e se baseia na monitoração do sistema para inferir os aspectos dinâmicos da arquitetura do software. Esta abordagem tem a vantagem de poder ser aplicada a qualquer sistema que seja passível de monitoração. No entanto, sua maior dificuldade é encontrar uma forma razoável de mapear as estruturas presentes em dois níves de abstração bastante distintos: o código e a arquitetura.

Caso nenhuma dessas técnicas seja usada, faz-se necessário introduzir no processo de desenvolvimento atividades que auxiliem a obtenção da conformidade. Basicamente, significa dizer que o software deve ser revisado e reorganizado freqüentemente, a exemplo do que acontece com as práticas de refatoramento e programação em pares de XP. Na prática, conformidade arquitetural é algo difícil de ser alcançado. Em projetos de médio e grande porte, são muitas as pessoas que devem entender e compartilhar uma mesma noção sobre a arquitetura. Essa característica é o que Brooks [FPB95] denomina de integridade conceitual, aspecto que considera como um dos mais importantes no design de um software - “Melhor ter uma única boa idéia do que muitas ruins, divergentes ou não-padronizadas”. Ou seja, é necessário haver, entre a equipe de desenvolvimento, uma uniformidade de conceitos e abstrações sobre a solução idealizada. A prática tem mostrado que publicar a documentação arquitetural em uma Intranet não é suficiente. As pessoas devem conhecer, entender e de certa forma se sentirem comprometidas com a arquitetura. Vale ressaltar que a própria arquitetura pode não permanecer estável durante o desenvolvimento. É comum que surjam problemas não antecipados, que em face à restrições de tempo, são corrigidos somente no nível de código. Nesse caso, esquece-se de adaptar e reavaliar o impacto sobre a arquitetura, o que provavelmente acarretará em mais problemas no futuro. Algo parecido ocorre durante a fase de manutenção do software, quando normalmente pessoas que não participaram da

sua construção são encarregadas de modificá-lo.

Capítulo 4

Técnica CASTOR

Neste capítulo, apresentamos a técnica CASTOR (Component bASed archiTecture cOnformance monitoR), cujo principal objetivo é estabelecer a conformidade arquitetural de sistemas de software. Para tanto, iremos utilizar como exemplo o sistema KWIC (*Key Word in Context*), proposto por Parnas [Par72]. Este mesmo sistema foi utilizado como estudo de caso por Garlan e Shaw, anos mais tarde, em um dos principais trabalhos sobre arquitetura de software [GS94]. Contudo, a escolha foi norteada pela simplicidade do exemplo e pela possibilidade de explorar todas as nuances da técnica. Inicialmente, será mostrado o formato da descrição arquitetural previsto em CASTOR, bem como as razões pelas quais chegamos a ele. Logo em seguida, discutiremos como é feito o mapeamento entre modelo e código, e por fim como se dá a verificação da conformidade arquitetural.

4.1 O Sistema KWIC

Antes de introduzirmos a técnica em si, faz-se necessário apresentar o sistema KWIC, uma vez que o utilizaremos ao longo de todo este capítulo. Abaixo, segue o problema original proposto por Parnas [Par72].

> O sistema indexador KWIC aceita um conjunto ordenado de linhas, sendo cada linha um conjunto ordenado de palavras, e cada palavra um conjunto ordenado de caracteres. Quaisquer dessas linhas pode ser deslocada circularmente. Ou seja, gerar outras cujas posições das palavras estejam modificadas, obedecendo-se a ordem entre elas. O sistema KWIC tem como saída um conjunto de todos

os possíveis deslocamentos circulares das linhas recebidas como entrada, em ordem alfabética.

Apesar de bastante simples, mesmo na época em que foi formulado, o problema tem grande valor didático. Além de ser passível de diferentes formas de modularização, instâncias desse problema e suas devidas variações, são recorrentes em computação [GS94]. Além dos requisitos contidos no problema original, será acrescentada a exigência de interação com o sistema através de uma interface gráfica. Nela, deve ser possível armazenar os textos antes de serem processados. Uma vez que o usuário os tenha confeccionado, ele pode então dar o comando ao sistema que irá processá-los um a um. Durante o processamento, a interface deve continuar respondendo aos possíveis comandos do usuário. O estado do texto submetido indicará o término de seu processamento, e a partir de então o resultado poderá ser visualizado na interface.

Para solucionar o problema, iremos utilizar uma das formas de decomposição analisadas por Parnas, que é a baseada em tipos abstratos de dados. Nesta abordagem, cada módulo encapsula seu estado e interage com outros através de uma interface bem definida. Ou seja, o módulo assume feições de um componente, assim como definido no Capítulo 3. Desta forma, o sistema pode ser decomposto em quatro componentes:

- **Deslocador** - Componente responsável pelo deslocamento circular das linhas.
- **Ordenador** - Componente responsável por ordenar alfabeticamente as linhas.
- **Coordenador** - Coordena a utilização dos demais componentes, a fim de realizar o objetivo do sistema.
- **Interface** - Interface com usuário, que regula a entrada e saída do sistema.

No nível de classes, esses componentes são organizados conforme mostra a Figura 4.1. Para o componente **Interface** foi utilizado o padrão MVC (Modelo-Visão-Controlador), com o intuito de desacoplar a lógica de negócio da interface do usuário. A classe *Controller* é a responsável por transformar eventos gerados pela interface em ações de negócio, e vice-versa. A classe *UI* é a visão, ou seja, é quem efetivamente interage com o usuário. Por sua

vez, a classe *Texto* não pertence a nenhum componente específico, já que sua função é encapsular o texto e o resultado de seu processamento. As demais classes têm a responsabilidade de realizar o objetivo dos componentes dos quais fazem parte.

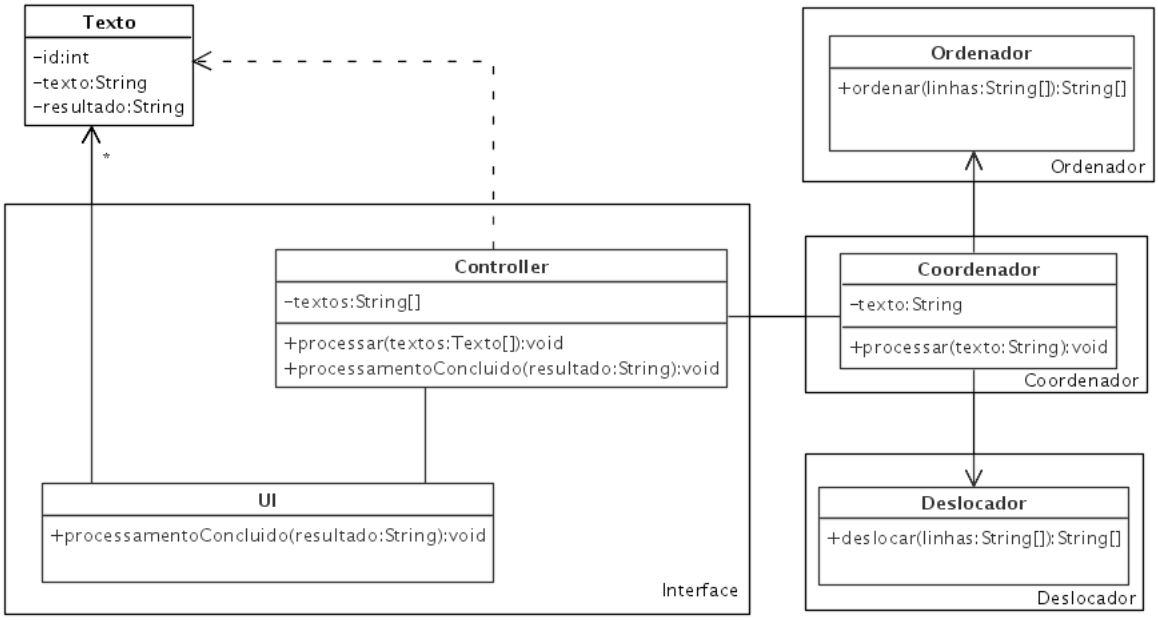

Figura 4.1: Diagrama de classes do sistema KWIC

4.2 Descrição Arquitetural

Em CASTOR, A arquitetura alvo é definida através de dois tipos de especificações:

1. **Especificação estrutural** - Define a estrutura da arquitetura. Contém o nome dos componentes, suas interfaces, os serviços externos que estes precisam e os seus pares na comunicação.

2. **Especificação comportamental** - Expresso como um modelo executável (transição e estados), define como o componente deve evoluir em tempo de execução.

Para cada uma delas, a técnica propõe uma forma particular de descrição. A partir do mapeamento entre as estruturas arquiteturais e de implementação, utiliza-se uma abordagem estática de verificação para estabelecer, em termos absolutos, a conformidade estrutural. No entanto, garantir o mesmo no nível comportamental é, geralmente, uma tarefa difícil. Sendo

assim, CASTOR se compromete em identificar inconsistências, entre arquitetura e implementação, em tempo de execução. A idéia é que na ocorrência de certos eventos no software o modelo comportamental da arquitetura (que deve ser executável) seja devidamente estimulado. Enquanto for possível simular o modelo a partir dos eventos gerados pela execução do software, significa que o último está se comportando como previsto. Uma violação, nesse caso, representa que a partir daquele instante foi atingido um caminho de execução que não pôde ser alcançado no modelo.

Na Figura 4.2, é mostrado o esquema que representa as especificações em CASTOR e como estãs são mapeadas no código. Estruturalmente temos três componentes que interagem entre si, sendo que a mensagem M1 é utilizada na comunicação no sentido A->B, enquanto que M2 é empregada no sentido B->A. Por sua vez, o componente B é mapeado para as classes X, Y e Z, que juntas, compartilham a responsabilidade de prover o serviço descrito em sua interface. Em relação à especificação comportamental, pode-se perceber ainda que para cada tipo de mensagem trocada, tanto o envio, quanto o recebimento devem estar previstos nos componentes correspondentes. Nos itens a seguir, esses pontos são discutidos mais detalhadamente.

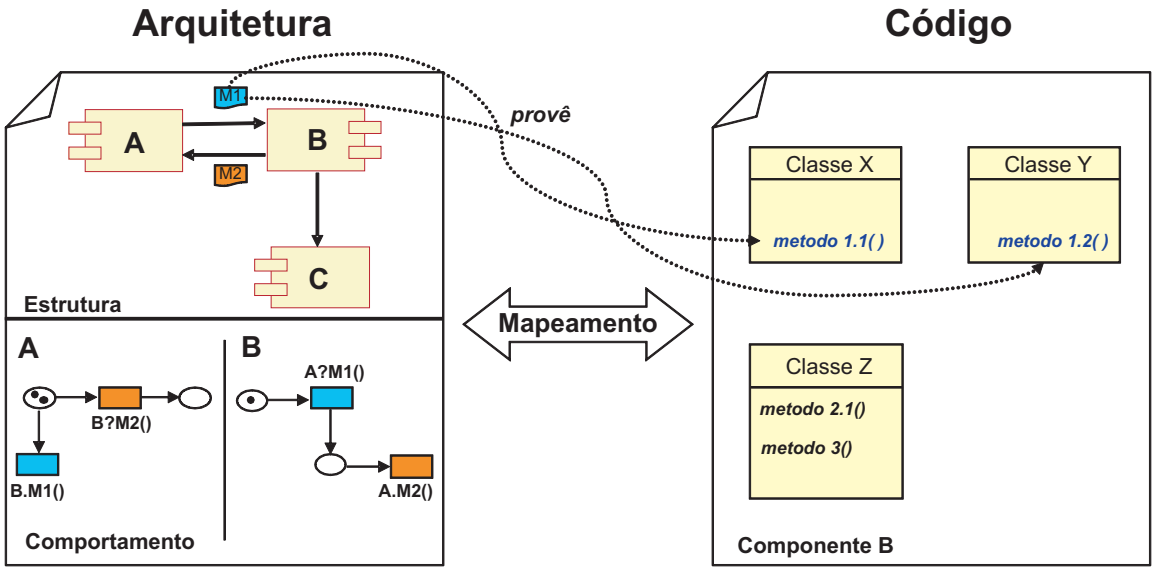

Figura 4.2: Esquema da Descrição Arquitetural em CASTOR e seus mapeamentos

A princípio, os desenvolvedores são os responsáveis por manterem o mapeamento entre arquitetura e implementação. Isso gera um outro benefício indireto do uso da técnica, que é fazer com que os desenvolvedores compreendam como o código produzido por eles

se encaixa no contexto geral do software. Vale lembrar que uma falha no mapeamento, potencialmente, será detectada pela técnica. Além disso, conforme já discutido, um modelo arquitetural estabelece uma visão única sobre a solução vislumbrada e fornece um idioma comum para comunicação. Para ser viável, é imprescindível que a descrição dos artefatos arquiteturais sejam simples de fazer e entender. O formato de tal descrição (estrutural e comportamental) será introduzido, a seguir, com base no sistema KWIC, previamente apresentado.

4.2.1 Especificação Estrutural

Um componente em CASTOR só pode se comunicar com outros componentes através de portas - chamadas de método entre componentes só são permitidas caso sejam explicitamente declaradas. Uma porta representa um canal de comunicação entre componentes conectados e é constituída de dois conjuntos de operações: *provides* e *requires*. O conjunto *provides* define quais operações são passíveis de serem invocadas por outros componentes conectados à porta. De maneira análoga, *requires* estabelece as operações providas por algum outro componente em uma determinada porta. Duas portas só podem ser conectadas se e somente se as operações do conjunto *requires* de um componente estejam presentes no conjunto *provides* do outro, e vice-versa.

```
1  incomplete component Coordenador{
2       port ui{
3            provides processar();
4            requires processamentoConcluido();
5       }

6       port ordenador{
7            requires ordenar();
8       }

9       port deslocador{
10           requires deslocar();
11      }
12 }
```

Código 4.1: Especificação parcial do componente Coordenador

No Código 4.1, é mostrada a descrição parcial do componente **Coordenador**. Existem duas razões para que esta especificação seja considerada parcial: i) não é fornecida a imple-

mentação da operação *processar()* da porta *ui*; ii) não há nenhuma declaração sobre quais classes concretas constituem o componente. Por essas razões o componente é declarado com a palavra reservada `incomplete`. Em concordância com a Figura 4.1, a especificação do componente **Coordenador** possui três portas: *ui*, *ordenador*, *deslocador*. Respectivamente, para permitir a comunicação com os componentes **Interface**, **Ordenador** e **Deslocador**. Portas e componentes são representações abstratas da arquitetura, e como tal, precisam ser mapeadas para as construções concretas do código afim de viabilizar a análise de conformidade. Sendo assim, em CASTOR, componentes são mapeados para um conjunto de classes e suas portas são mapeadas para um ou mais métodos destas classes. No Código 4.2, é mostrada a especificação completa do componente **Coordenador**. A linha 2 define que o componente é constituído somente pela classe `Coordenador`. Na linha 13 temos que a operação *processar()* corresponde a qualquer método público de mesmo nome pertencente à classe `Coordenador`, que tenha como parâmetro o tipo `String`, não importando seu tipo de retorno (expresso através do uso do símbolo *). Vale ressaltar, que um método não pode estar contido em duas operações distintas.

```
1  component Coordenador{
2       class Coordenador;

3       port ui{
4            provides processar();
5            requires processamentoConcluido();
6       }

7       port ordenador{
8            requires ordenar();
9       }

10      port deslocador{
11           requires deslocar();
12      }

13      processar(){
14           public * Coordenador.processar(String);
15      }
16 }
```

Código 4.2: Especificação do componente Coordenador

Ao invés de listar nominalmente cada uma das classes que compõem o componente, é possível usar também símbolos para representar um conjunto de tipos. A sintaxe é a mesma

suportada por AspectJ [Pro05], uma extensão que permite a programação orientada a aspectos [KLM+97] na linguagem Java. No caso da especificação de tipos, o símbolo `*` representa um conjunto quaisquer de caracteres para designar parte de uma classe, interface ou pacote. O símbolo `..` denota todos os sub-pacotes indiretos e diretos de um determinado pacote. E por fim, `+` é usado para especificar sub-tipos. Na Tabela 4.1, são mostrados alguns exemplos simples da utilização de tais símbolos.

Padrão de Tipo	**Significado**
`Coordenador*`	Classes cujo nome inicia com a palavra Coordenador, tais como `CoordenadorKWIC` e `CoordenadorTeste`
`java.*.Date`	Qualquer classe `Date` pertencente a algum sub-pacote `java`. Nesse caso, por exemplo, `java.util.Date` e `java.sql.Date`
`java..*`	Qualquer classe pertencente a algum sub-pacote direto (`java.awt`) ou indireto (`java.awt.event`) de `java`
`java..*List+`	Qualquer classe pertencente a algum sub-pacote de `java` que termina com a palavra List incluindo todos os seus sub-tipos
`!Vector && Collection+`	Qualquer classe que não seja `Vector` e que seja um sub-tipo de `Collection`

Tabela 4.1: Exemplos de padrões para especificação de tipos

O mesmo artifício é utilizado também para designar os métodos concretos que constituem uma operação arquitetural. Os símbolos utilizados para especificar conjunto de tipos são os mesmos dos usados para métodos. A única diferença é que `..` é usado aqui para denotar qualquer tipo e quantidade de argumentos do método. Além disso, caso os modificadores de acesso do método - tais como `public`, `private`, `static` e `final` - não sejam especificados, eles serão ignorados pelo casamento de padrões. Por exemplo, se o padrão não contiver o modificador `final`, tanto os métodos que são e os que não são `final` serão considerados. Os modificadores podem ser usados também com o operador de negação `!` para especificar métodos que não possuam tal modificador. No padrão de assinatura de método, onde é especificado o tipo de retorno, dos parâmetros ou das exceções pode-se utilizar os padrões de tipos mostrados na Tabela 4.1. Obviamente, os métodos designados nas operações devem pertencer ao conjunto de classes que compõem o componente. Além disso, cada operação deve ter ao menos um método concreto associado. Alguns exemplos de especificação de métodos podem ser vistos na Tabela 4.2.

Padrão de Método	Significado
`public void Conta.finalizar()` `throws InvalidOperationException`	Método público `finalizar()` da classe `Conta`, que não recebe nenhum parâmetro e retorna `void`
`* Conta+.*(..)`	Todos os métodos da classe `Conta` e de seus sub-tipos
`public Conta.new(double,..)`	Todos os construtores públicos da classe `Conta` cujo primeiro argumento é do tipo `double`

Tabela 4.2: Exemplos de padrões para especificações de métodos

Após ter especificado cada um dos componentes (ver Código 4.3), é preciso então conectar suas portas para que a estrutura seja efetivamente criada. Como veremos adiante, um sistema pode ter sua arquitetura visualizada sob diferentes perspectivas. Em CASTOR, essas perspectivas são chamadas de visões. No Código 4.4, é mostrada a visão do sistema KWIC conforme apresentado no início deste capítulo. A técnica não contempla configurações dinâmicas, ou seja, a modificação estrutural, em tempo de execução, da arquitetura. No entanto, isso não chega a ser uma limitação significativa, uma vez que a maioria das ADLs existentes também só suportam configurações estáticas e mesmo assim são bastante úteis na descrição arquitetural [MT00]. Podemos concluir, a partir desta observação, que um grande número de sistemas realmente não possui uma arquitetura de caráter dinâmico.

Uma outra visão

Como dito anteriormente, é possível visualizar a arquitetura do sistema KWIC de diversas formas. É possível, por exemplo, representá-lo em termos do padrão MVC, conforme mostra a Figura 4.3. Nesse caso, haveriam três componentes: **Modelo**, **Visão** e **Controlador**. O componente **Modelo** agruparia as classes `Coordenador`, `Ordenador` e `Deslocador`. A **Visão** poderia ser composta pelas classes `UI` e `Texto`, e por último, **Controlador** apenas pela classe `Controller`.

A especificação dos componentes **Modelo**, **Visao** e **Controlador** pode ser vista no Código 4.5. Diferentemente da visão anterior, a classe `Texto` passou a fazer parte de um componente - nesse caso, o componente **Visao**. Perceba que há uma relação de dependência entre a classe `Controlador` e `Texto`. Isso se deve ao fato de que o controlador deve ter acesso aos dados encapsulados - através do método *getTexto()* - para repassá-los ao modelo. Há portanto, uma comunicação entre os componentes **Visao** e **Controlador** que não está

```
component Interface{
     class UI, Controller;

     port processor{
          provides processamentoConcluido();
          requires processar();
     }

     processamentoConcluido(){
          public * UI.processamentoConcluido(String);
     }
}

component Deslocador{
     class Deslocador;

     port cliente{ provides deslocar();}

     deslocar(){ public * Deslocador.deslocarIndiceLinhas(..); }
}

component Ordenador{
     class Ordenador;

     port cliente{ provides ordenar(); }

     ordenar(){ public * Ordenador.ordenarIndiceLinhas(..); }
}
```

Código 4.3: Especificação dos demais componentes (Interface, Deslocador e Ordenador)

```
view KWIC{
     connect Interface.processor, Coordenador.ui;
     connect Coordenador.deslocador, Deslocador.cliente;
     connect Coordenador.ordenador, Ordenador.cliente;
}
```

Código 4.4: Conectando as portas

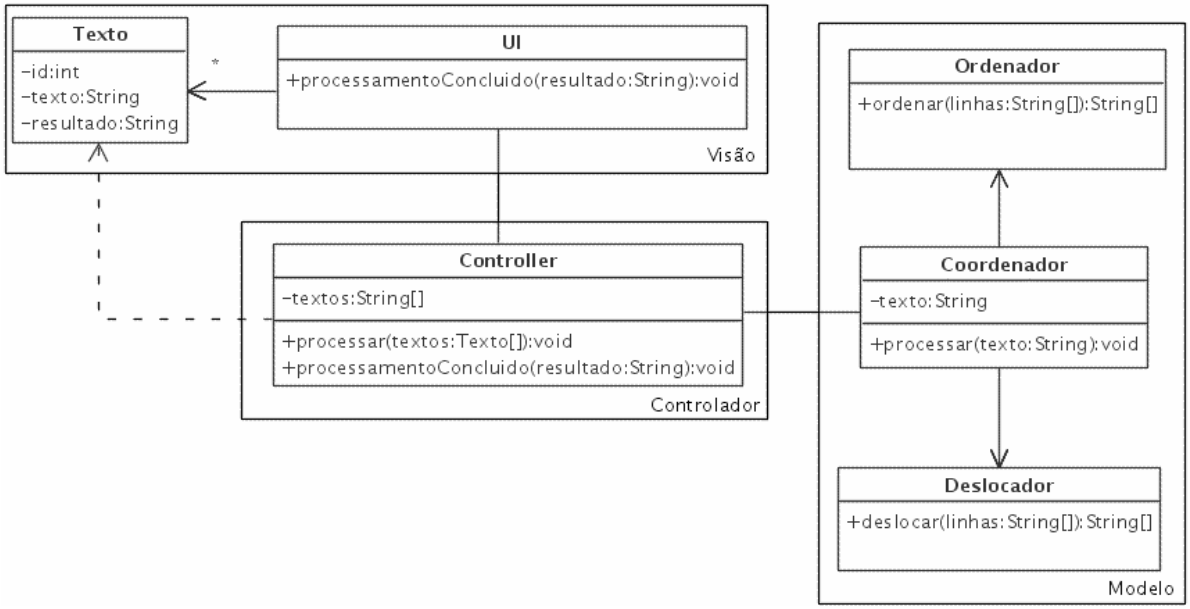

Figura 4.3: Diagrama de classes do sistema KWIC, na visão MVC

especificada em nenhuma porta. Nessa situação, existem três opções:

1. Registrar essa comunicação como um caso excepcional, porém permitido;
2. Incluir na porta dos componentes envolvidos as operações (e seus mapeamentos) que contemplem a comunicação;
3. Redistribuir as classes entre os componentes. Por exemplo, retirando a classe `Texto` do componente **Visao**.

Nessa situação, a terceira opção seria a mais recomendada, uma vez que `Texto` não possui nenhuma regra de negócio. Porém, iremos optar pela primeira, justamente para exemplificar como fazê-la. A linha 4 do Código 4.6, especifica quais chamadas de métodos, do componente **Controlador** para o componente **Visao**, são permitidas sem passar por suas portas. O exemplo ainda estabelece que somente o método *Controller.process(Object[] jobs)*, em particular, pode iniciar uma comunicação direta através da invocação de *Texto.getTexto()*. Como podemos ver, os símbolos, discutidos anteriormente, também podem ser utilizados na descrição dos métodos. Um outro aspecto, ainda não discutido, é a possibilidade das operações receberem parâmetros. Nesse caso, cada método concreto que constitui a operação deve fornecer o mapeamento para tais parâmetros. Na seção seguinte, na qual será abordada

```
component Modelo{
     class Coordenador, Ordenador, Deslocador;

     port controller{ provides processar(); }

     processar(){ public * Coordenador.processar(String); }
}

component Controller{
     class Controller;

     port modelo{ requires processar(); }

     port visao{
          provides processar(int num);
          requires processamentoConcluido();
     }

     processar(int num){
          public void Controller.processar(Texto[] textos){
               num = jobs.length;
          };
     }

}

component Visao{
     class UI;

     port controlador{
          provides processamentoConcluido();
          requires processar();
     }

     processamentoConcluido(){
          public * UI.processamentoConcluido(String);
     }
}
```

Código 4.5: Especificação dos componentes Modelo, Visão e Controlador

a especificação da arquitetura do ponto de vista comportamental, discutiremos melhor esse ponto.

```
1 view MVC{
2      connect Visao.controller, Controlador.view;
3      connect Controlador.model, Modelo.controller;
4      exception Controlador -> Visao{
5           void Controller.processar(..) ->
6                String Texto.getTexto();
7      }
8 }
```

Código 4.6: Especificação da visão MVC contendo uma comunicação excepcional

4.2.2 Especificação Comportamental

Uma ADL, tipicamente, se utiliza da semântica de alguma teoria formal para especificar o comportamento de uma arquitetura. A escolha dessa teoria tem grande influência nos tipos de sistemas - ou nos aspectos de um dado sistema - que podem ser modelados com uma ADL, em particular. É possível classificar os paradigmas de especificação existentes nos seguintes grupos [vL00]:

- **Especificação baseada em História** - O sistema é caracterizado através de um conjunto de propriedades (expressas em lógica temporal) permitidas ao longo do tempo. Essas propriedades são interpretadas sobre uma estrutura de estados. Ex: LTL, CTL
- **Especificação baseada em Estados** - Os estados admissíveis são especificados em termos de asserções (invariantes, pré e pós-condições). Ex: Z, B, VDM
- **Especificação baseada em Transições** - Define as regras de evolução de um estado para outro. Dado um estado inicial qualquer e um evento, o conjunto de funções de transição deve fornecer o estado resultante. Ex: Statecharts, PROMELA
- **Especificação Funcional** - O princípio é especificar o sistema como um conjunto estruturado de funções matemáticas. Ex: OBJ, PVS

- **Especificação Operacional** - Define uma coleção estruturada de processos que pode ser executada por uma máquina abstrata. Ex: Redes de Petri, Álgebra de Processos (CSP, π-calculus)

A técnica CASTOR, não está vinculada a nenhum formalismo específico. Ou seja, qualquer teoria pode ser utilizada, contanto que a especificação possa ser executada. Para isso, faz-se necessário que o formalismo possua os conceitos de estado, transição e regras para o disparo das mesmas. Como veremos na seção seguinte, é essa característica (encontrada nos paradigmas de especificação operacional e baseada em transições) que será utilizada para estabelecer a relação de conformidade entre modelo e código.

Redes de Petri

Para instanciar a técnica proposta, escolhemos o formalismo de rede de Petri como base para a descrição comportamental. Essa escolha deveu-se, principalmente, aos seguintes fatores:

- É uma linguagem gráfica;
- Permite abstrair tipos de dados (Redes de Petri Lugar-Transição);
- Suporta paralelismo e concorrência;
- Possui disponibilidade de ferramentas.

Uma estrutura de Rede de Petri é um grafo bipartido dirigido, cujos arcos possuem um peso (número inteiro positivo) associado e constituído de dois tipos de nós chamados de lugar e transição. Os arcos só podem ligar lugares a transições ou transições a lugares. Graficamente, lugares são representados por círculos e transições por retângulos. Adicionando a esta estrutura uma marcação inicial, temos uma Rede de Petri. Uma marcação associa um inteiro positivo a cada lugar. Graficamente uma marcação é representada por uma quantidade *x* de pontos (chamados fichas ou tokens) dentro de cada lugar, onde *x* é o número inteiro associado ao lugar pela marcação atual. A marcação pode ser representada por num vetor de *n* posições, onde *n* é a quantidade de lugares da rede. O *p-ésimo* componente do vetor, denotado por *M(p)*, é o número de tokens do lugar *p*. Formalmente, uma rede de Petri é expressa como sendo a seguinte 5-tupla:

$PN = \{P, T, F, W, M_0\}$

onde:

$P =$ conjunto finito de lugares;

$T =$ conjunto finito de transições;

$F \subseteq \{P \text{ X } T\} \cup \{T \text{ X } P\}$;

$W = F \rightarrow \{1, 2, 3, ...\}$

$M_0 = P \rightarrow \{0, 1, 2, 3, ...\}$

Lugares representam condições, enquanto transições representam eventos. Uma transição (evento) possui um conjunto de lugares de entrada e de saída representando as pré-condições e pós-condições respectivamente. A quantidade de fichas num lugar indica a quantidade de recursos disponíveis para as transições que o tem como lugar de entrada. Sendo assim temos que:

1. Uma transição t é dita estar habilitada se cada lugar de entrada p possui pelo menos a quantidade de tokens definido pelo arco que conecta *p* a *t*.

2. Uma transição habilitada pode ser ou não disparada (não determinismo).

3. O disparo de uma transição *t* habilitada remove a quantidade de fichas de cada lugar de entrada *p* de *t* (definido pelo inteiro positivo do arco que liga *p* a *t*, e adiciona a quantidade de fichas a cada lugar de saída de *t* (definido pelo inteiro positivo do arco que liga *t* a *p*).

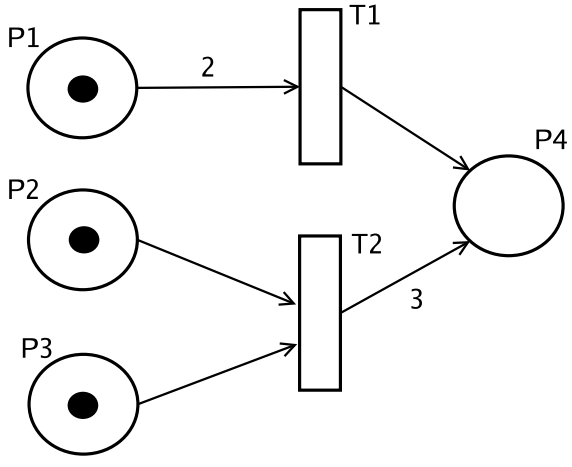

Figura 4.4: Exemplo da representação gráfica de uma Rede de Petri

Uma transição que não possui nenhum lugar de entrada é chamada de transição de origem e incondicionalmente estará sempre habilitada. Uma transição que não possui lugar de saída é chamada de sorvedouro e nunca produz fichas, apenas as consome. Na Figura 4.4, podemos ver a representação gráfica de uma rede de Petri bem simples. Conforme mostrado, apenas a transição *T2* encontra-se habilitada, pois todos os seus lugares de entrada possuem, no mínimo, a quantidade de fichas expressa no arco (por convenção, assume-se o peso 1 quando não há valor no arco). De maneira análoga, a transição *T1* só poderá ser disparada quando o lugar *P1* contiver ao menos duas fichas. Depois de disparado *T2*, o lugar *P2* e *P3* ficará sem nenhuma ficha, enquanto que *P4* passará a ter três.

Modelagem do Sistema KWIC

Em CASTOR, os componentes têm seu comportamento especificado separadamente e a semântica da arquitetura é expressa pela união dos modelos comportamentais de cada componente. Na Figura 4.5, é mostrado o modelo do componente **Coordenador**. Todas as operações arquiteturais, definidas no Código 4.2, devem estar associadas a uma transição na rede de Petri. Essa associação é feita nomeando as transições com a identificação completa da operação (o que inclui a porta a qual pertence). O símbolo de interrogação, empregado entre o nome da porta e o nome da operação, denota uma operação provida pelo componente, enquanto que as operações requeridas utilizam-se de um ponto. Pelo fato de cada transição ter apenas um único lugar de entrada e saída, este modelo, em particular, caracteriza-se como uma máquina de estado. No entanto, a técnica assume que os estados devem ser utilizados apenas para restringir a forma com que as transições são disparadas. Ou seja, o foco durante a modelagem deve se voltar para como o modelo evolui e não como este se apresenta ao longo de sua execução. O modelo deste componente, por exemplo, define a ordem estrita em que as operações ocorrem. Pode-se concluir também, que há apenas um processo ativo no componente, uma vez que nunca duas operações estão habilitadas simultaneamente. A simples exclusão do lugar contendo a única ficha do modelo, faria com que a operação *ui?processar()* sempre estivesse habilitada. Conseqüentemente, a ordem estrita entre as operações deixaria de existir. Dessa forma, a semântica do modelo apenas definiria que para cada operação *ui?processar()* executada, as outras deveriam executar uma única vez na ordem pré-estabelecida. Isso mostra que pequenas modificações na rede, alteram significativamente as

restrições impostas ao comportamento do componente.

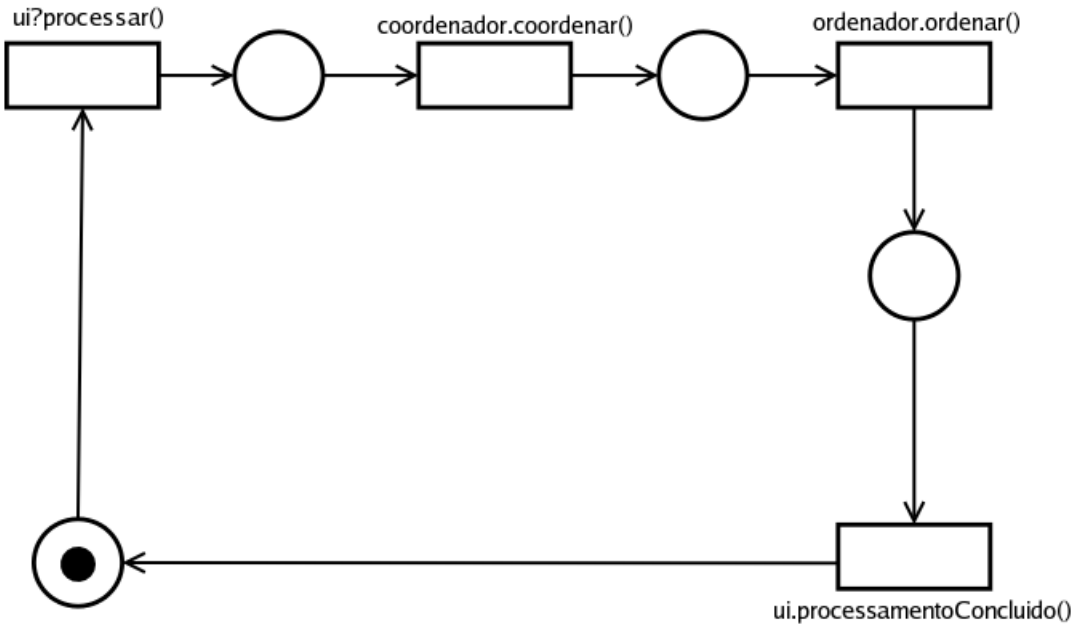

Figura 4.5: Modelo comportamental do componente Coordenador

Na Figura 4.6, é mostrada a rede de Petri que descreve o comportamento do componente **Controlador** (ver Código 4.5). Inicialmente, somente a operação *visao?processar(num)* está habilitada. Ela pode ser disparada consecutivamente infinitas vezes, uma vez que ela retira e ao mesmo tempo repõe a ficha no seu único lugar de entrada. No entanto, no momento em que *modelo.processar()* ocorrer, ela só poderá disparar novamente quando a operação *visao.processamentoConcluido()* recolocar a ficha no lugar *sync*.

Neste modelo, podemos observar também o uso de parâmetros nas operações arquiteturais. A quantidade de fichas criadas pela operação *visao?processar(num)* é variável de acordo com o valor assumido por *num*. Nesse caso, *num* será o tamanho do *array* passado como parâmetro ao método `Controller.processar(Texto[] textos)`, no momento de sua invocação (ver linha 15 do Código 4.5). Isso é necessário, pois, a quantidade de vezes que as operações requeridas devem ser executadas, depende do número de textos submetidos pela interface. Vale lembrar que, de acordo com os requisitos, a interface gráfica submete todos os texto de uma única vez, enquanto que o componente **Coordenador** somente os processa individualmente. Ou seja, é necessário um *buffer* que processe os textos a medida que outros forem sendo finalizados. Essa função é exercida pelo componente **Con-**

trolador. Para o formalismo de rede de Petri, as operações arquiteturais só podem conter parâmetros do tipo inteiro. Isto porque, no modelo, o estado é descrito apenas pela quantidade de fichas nos lugares e no máximo variáveis inteiras podem ser usadas para estabelecer quantas delas uma transição consome ou produz.

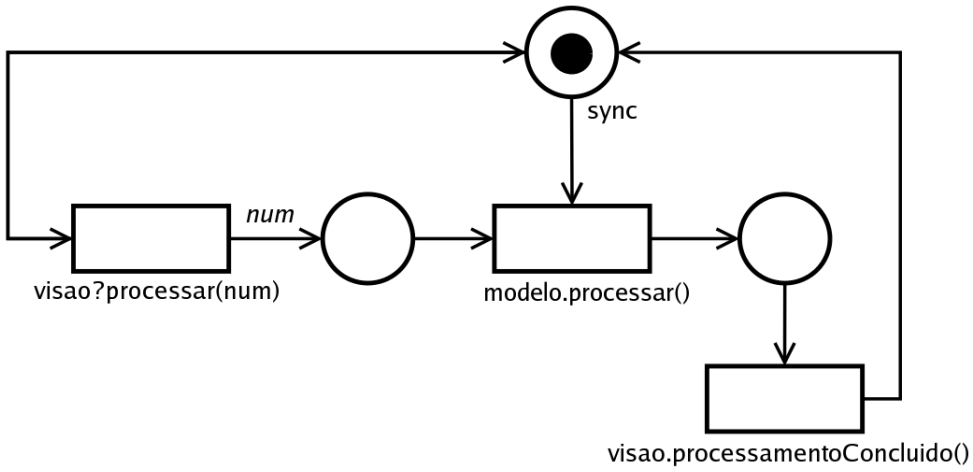

Figura 4.6: Modelo comportamental do componente Controlador

4.3 Processo de Verificação

Após a arquitetura ter sido devidamente especificada, o interesse agora é em saber se a implementação obedece às restrições impostas pelos modelos. Resumidamente, a verificação da conformidade arquitetural se dá em duas etapas:

1. **Análise estática** - Visa identificar comunicações não previstas entre os componentes definidos.

2. **Monitoração** - Em tempo de execução, o comportamento do sistema é continuamente confrontado com sua especificação.

Conforme mostrado na Figura 4.7, para ambas as etapas é utilizado a programação orientada a aspectos como meio de viabilizar a técnica proposta. Para cada componente deriva-se um aspecto e uma classe de simulação, as quais irão interagir em tempo de execução para validar a descrição comportamental frente ao código. Porém, antes disso, a porção de análise estática do aspecto se encarrega pela checagem estrutural do componente, condição

necessária para o início da fase de monitoração. Antes de continuarmos, apresentamos a devida contextualização sobre Aspectos, bem como os principais conceitos envolvidos.

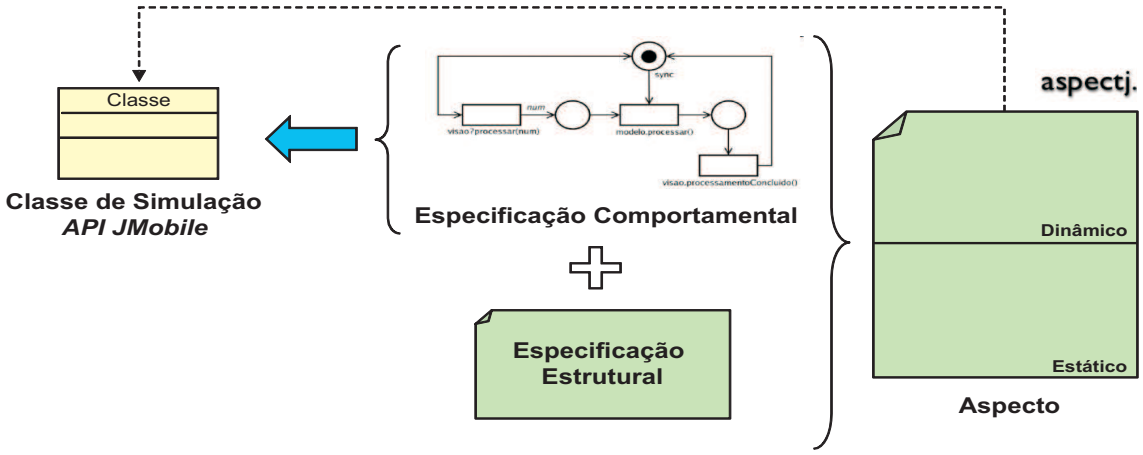

Figura 4.7: Resumo do Processo de Verificação

Aspectos

Os mecanismos de modularização hierárquica das linguagens orientadas a objetos, e até mesmo das procedurais, já demonstraram sua utilidade ao longo do tempo. Essas unidades modulares - que em orientação a objetos são expressas como pacotes, classes, objetos e métodos - concentram-se nos interesses funcionais da aplicação. No entanto, há também outros interesses a serem considerados que podem envolver mais de um componente funcional, tais como: sincronização, interação de componentes, persistência e controle de segurança. Esses interesses transversais à decomposição funcional são geralmente espalhados por diversas partes do código. A programação orientada a aspectos consiste, então, na separação dos interesses em unidades modulares, chamados de aspectos, e posterior composição desses em um único sistema [KLM+97].

POA não substitui os paradigmas e linguagens de programação existentes, ao contrário, ela é utilizada em conjunto com estas. O propósito é justamente tornar possível expressar adequadamente a separação de interesses a fim de melhorar o *design* e conseqüentemente a manutenibilidade dos sistemas desenvolvidos. Nesse contexto, AspectJ [KHH+01] é um dos meios mais difundidos para a implementação de sistemas orientados a aspectos. Ela é, na verdade, uma extensão da linguagem Java que provê suporte à implementação modular de interesses transversais. Existem dois tipos de interesses: dinâmicos e estáticos. Como o

próprio nome denota, interesses dinâmicos definem a implementação adicional que deve ser executada em pontos pré-definidos. Já os interesses estáticos visam modificar a estrutura do programa, o que torna possível, por exemplo, definir novas operações em tipos já existentes de uma maneira não intrusiva.

Em AspectJ, os interesses transversais de cunho dinâmico são definidos em termos de pontos ao longo do fluxo de execução do software. Tais pontos recebem o nome de pontos de junção, e incluem chamadas de métodos, instanciação de objetos, acesso a atributos, dentre outras ações. A composição de diversos pontos de junção denomina-se de ponto de corte. O código do comportamento transversal (*advice*) pode ser entendido como um procedimento que será ativado quando certos pontos de corte forem alcançados durante a execução de um programa. A ativação do *advice* pode acontecer antes, depois ou antes e depois dos pontos de corte, conforme definido pelo desenvolvedor. Por último, o aspecto em si é uma unidade modular que contém a implementação de um interesse transversal, composta por pontos de cortes, *advices* e outras declarações usuais de Java. Na tabela 4.3, são mostrados alguns pontos de junção suportados por AspectJ. A diferença entre `execution` e `call` é que o primeiro é alcançado no momento em que o controle de execução é transferido ao método, para que este possa iniciar sua execução. O segundo, por sua vez, define o instante anterior a este, quando a invocação já foi feita porém o controle ainda não foi transferido.

Ponto de Junção	**Significado**
call	Captura chamadas à métodos e construtores
execution	Momento em que métodos e construtores são executados
within	Limita o escopo do ponto de corte para determinados tipos
withincode	O mesmo que **within**, só que utilizado também para métodos
args	Expõe os argumentos de métodos e construtores ao escopo dos *advices*
get	Captura referências aos atributos de uma classe
set	Captura atribuições aos atributos de uma classe

Tabela 4.3: Principais pontos de junção de AspectJ

4.3.1 Análise Estrutural

Alguns pontos de corte, que usam apenas informações obtidas em tempo de compilação, são possíveis de serem determinados estaticamente. Em AspectJ, os tipos de pontos de

junção que têm essa característica incluem: *call*, *within*, dentre outros. Com o uso de apenas esses dois designadores de pontos de junção é possível verificar se a estrutura definida para arquitetura é obedecida pela implementação. Para tanto, seria gerado automaticamente um aspecto para cada componente a partir de sua especificação. Como exemplo, vejamos como seria essa geração para o componente **Controlador** (ver Códigos 4.5 e 4.6). A descrição arquitetural define que **Controlador** se comunica com os componentes **Visao** e **Modelo**. Com o primeiro, a comunicação é bi-direcional, enquanto que com o segundo **Controlador** assume apenas o papel de cliente (somente invoca operações). A idéia é que para cada porta do componente, seja criado um ponto de corte para cada direção de comunicação (uma para o conjunto de operações *requires* e outra para as operações *provides*). Sendo assim, conforme mostra o código 4.7, o componente Controlador contaria com três pontos de corte:

- ***modeloOutCall()*** - Verifica se as chamadas destinadas ao componente **Modelo**, são feitas somente através do método `* Coordenador.processar(..)`
- ***visaoOutCall()*** - Verifica se as chamadas destinadas ao componente **Visao**, são feitas somente através do método `* UI.processamentoConcluido(..)`
- ***visaoInCall()*** - Verifica se as chamadas oriundas do componente **Visao**, são feitas somente através do método `call(* Controller.processar(Object[]))`

Todos os pontos de corte assumem a forma de uma conjunção entre três conjuntos distintos. O primeiro deles é uma disjunção que captura toda e qualquer chamada de método para algum componente envolvido na comunicação (ou ele próprio ou algum de seus pares). Já o segundo conjunto é um conjunção com todas as chamadas permitidas. E por fim, o terceiro é novamente uma disjunção que restringe as chamadas para um determinado conjunto de classes (que compõem o outro componente da comunicação). Sendo assim, poderíamos, por exemplo, interpretar da seguinte forma o ponto de corte ***modeloOutCall()***:

> Todas as chamadas para o componente **Modelo** (classes Coordenador, Ordenador ou Deslocador), que não seja a única chamada permitida (`Coordenador.processar(..)`) oriundas do componente **Controlador** (classe Controller).

```
1  public aspect ControladorVerifier{
2
3       pointcut modeloOutCall(): (call(* Coordenador.*(..))
4            || call(* Ordenador.*(..)) || call(* Deslocador.*(..)))
5         && !(call(* Coordenador.processar(..)))
6         && within(Controller);
7
8       pointcut visaoOutCall(): call(* UI.*(..))
9         && !(call(* UI.processamentoConcluido(..)))
10           && !(withincode(* Controller.processar(..))
11                && call(String Texto.getTexto()))
12        && within(Controller);
13
14      pointcut visaoInCall(): call(* Controller.*(..))
15        && !(call(* Controller.processar(Object[])))
16        && within(UI);
17
18      declare error: modeloOutCall():
19           "Chamadas permitidas para classes (Coordenador," +
20           "Deslocador, Ordenador):\n" +
21           "* Coordenador.processar(..)";
22
23      declare error: visaoOutCall():
24           "Chamadas permitidas para classes (UI):\n" +
25           "* UI.processamentoConcluido(..)";
26
27      declare error: visaoInCall():
28           "Chamadas permitidas para classes (Controller):\n" +
29           "* Controller.processar(Object[])";
30 }
```

Código 4.7: Aspecto que verifica conformidade estrutural do componente Controlador

Conforme mostrado no Código 4.6, o componente **Controlador** se comunica com **Visao** de uma forma excepcional (sem utilizar sua porta). O aspecto apresentado anteriormente, contempla essa situação justamente nas linhas 8 e 9. Para isso, é acrescentado tal comunicação no segundo conjunto de pontos de junção. Ou seja, além da comunicação normal via porta, também é permitido qualquer chamada oriunda do método `Controller.processar(..)` para `String Texto.getTexto()`.

4.3.2 Análise Comportamental

Em CASTOR, a análise da conformidade comportamental da arquitetura exige o uso de uma simulador de modelo para o formalismo escolhido. Para os modelos em rede de Petri, foi utilizado o *framework* JMobile [dMS05]. Oriundo de um trabalho de dissertação desenvolvido no próprio grupo, seu objetivo é prover suporte à simulação e geração do espaço de estados para modelos RPOO [Gue02]. Apesar de não ter sido desenvolvido especificamente para redes de Petri, com poucas modificações, foi possível adequar o JMobile para que este fosse capaz de simular os modelos em tal formalismo. Sendo assim, cada modelo foi codificado em uma classe Java utilizando-se do *framework* JMobile (ver Apêndice A).

O código do aspecto que captura os eventos do componente **Controlador** e executa as transições do modelo correspondente é mostrado no Código 4.8. Cada operação no nível arquitetural é traduzida para um ponto de corte que engloba os métodos concretos que o compõem, conforme definido em sua especificação. Os pontos de cortes oriundos da especificação arquitetural assumem sempre o seguinte padrão:

`pointcut` *nome*(*argumentos*): *pj1* && *pj2* && *pj3*;

Onde, *pj1*, *pj2* e *pj3* são conjuntos de pontos de junção. O primeiro é uma conjunção que define os métodos que constituem a operação arquitetural. Para as operações providas pelo componente, é utilizado o tipo de ponto de junção `execution`, enquanto que para as operações requeridas, é utilizado `call`. O conjunto *pj2* também é uma conjunção, que se aplica somente quando a operação em questão é do tipo *requires*. Ele é composto por designadores `within` cuja finalidade é definir o componente de origem da chamada de método. Por fim, o último conjunto é opcional, e só é utilizado quando os argumentos de operações arquiteturais, quando existentes, são baseados nos argumentos dos métodos que

constituem tal operação. Esse é, por exemplo, o caso da operação *Visao?processar(num)*. Para cada ponto de corte é definido um *advice* que deve ser executado depois que os eventos sob monitoração aconteçam. Nesse momento, a transição correspondente do modelo deve ser executada através do método `fire(String)`, que recebe como parâmetro o nome da transição em questão. Se a transição for disparada normalmente, significa que até aquele instante o comportamento do sistema está de acordo com o descrito pelo modelo. Caso contrário, a execução do software é interrompida, e uma mensagem indicando a causa do problema é apresentada ao usuário. No entanto, quaisquer outras ações poderiam ser tomadas nessa situação, como por exemplo, a geração de logs de conformidade comportamental sem a necessidade de encerrar a aplicação.

O *advice* definido na linha 16 se distingue dos outros, pois antes de disparar a transição associada é preciso atribuir um valor à variável *num*, presente na Figura 4.6. Para tanto, utiliza-se o mapeamento definido na descrição do componente (ver Código 4.5, assim como é mostrado na linha 18. Uma vez definido este valor, é invocado o método do *framework* JMobile, descrito na linha 19, para que o modelo seja efetivamente atualizado. Para isto, é passado como parâmetro o nome da transição, o nome do lugar de saída, e o novo peso do arco que os conecta.

```
1  JMConfiguration conf = new JMConfiguration();
2  ControllerModel controller = new ControllerModel("Controlador");

3  ControllerObserver(){
4       conf.add(controller);
5  }

6  pointcut visaoProcessar(Object[] textos):
7       execution(* Controller.processar(..)) && args(textos);

8  pointcut modeloProcessar():
9       call(* Coordenador.processar(..)) && within(Controller);

10 pointcut visaoProcessamentoConcluido():
11      call(* UI.processamentoConcluido(..)) && within(Controller);

12 after(): modeloProcessar(){ fire("Modelo.processar()"); }

13 after(): visaoProcessamentoConcluido(){
14      fire("Visao.processamentoConcluido()");
15 }

16 after(Object[] textos): visaoProcessar(textos){
17      try{
18           int num = textos.length;
19           controller.changeOutWeight(
20                "Visao?processar(num)","textos",num);
21      }catch(Exception e){e.printStackTrace();}
22      fire("Visao?processar(num)");
23 }

24 private void fire(String transitionName){
25      try{ conf = conf.execute(transitionName,conf); }
26      catch(Exception e){ e.printStackTrace(); }
27 }
```

Código 4.8: Código responsável pela conformidade comportamental de Controlador

4.4 Sumário da Técnica

Nesta seção é apresentado um breve resumo sobre a técnica CASTOR. Vale ressaltar que alguns aspectos presentes neste sumário não foram utilizados ao longo deste capítulo, a exemplo do conceito de ações internas e da cláusula `import`. Eles serão discutidos no próximo capítulo, dentro do contexto do estudo de caso escolhido.

4.4.1 Descrição do Componente

A seguir, o formato de descrição para componentes é decomposto em elementos, que por sua vez são detalhados na tabela logo abaixo. Os elementos entre chaves são obrigatórios, enquanto que os elementos entre colchetes podem, em alguns casos, serem suprimidos.

```
[import]
[incomplete] component <nome>{
     <classes>
     <portas>
     [ações internas]
     [implementação]
}
```

Elemento	Finalidade	Exemplo	Observações
incomplete	Sua presença indica que a especificação é parcial	`incomplete`	-
nome	Identifica unicamente o componente	-	Somente caracteres alfa-numéricos
import	A mesma da cláusula `import` de Java	`import` java.util.List;	-
classes	Define quais classes compõem o componente	`class` java.util.* && !java.util.Date;	O conjunto de classes não pode ser vazio
portas	Agrupamento lógico de operações (`provides` e `requires`) através das quais dois componentes se comunicam	`port` exemplo { `requires` operacaoA(); `provides` operacaoB(); }	A porta deve conter ao menos uma operação e não pode haver duas portas de mesmo nome no componente
ações internas	Permite que ações internas (aquelas não presentes em nenhuma porta) sejam utilizadas na especificação comportamental	`internal` { operacaoC(); operacaoD(); }	Cada operação deve ser mapeado para um conjunto não vazio de métodos concretos
implementação	Prover o mapeamento das operações (e seus argumentos) providas pelas portas e das ações internas	operacaoB(int arg){ metodoC(int num){ arg = num * 2; } }	Cada método concreto só pode estar vinculado a uma única operação

Tabela 4.4: Detalhamento dos elementos que compõem um componente

4.4.2 Descrição da Visão

Nesta subseção é utilizado o mesmo esquema da anterior para sumarizar a forma de uma visão em CASTOR.

```
[import]
view <nome>{
     <connect>
     [exception]
}
```

Elemento	Finalidade	Exemplo	Observações
nome	Identifica unicamente o componente	-	Somente caracteres alfa-numéricos
import	A mesma da cláusula `import` de Java	`import` java.util.List;	-
connect	Interligar duas portas em componentes distintos	`connect` ComponenteA.portaA, ComponenteB.portaB;	As portas devem possuir operações complementares
exception	Define um canal de comunicação que não passa por nenhuma porta	`exception` CompA -> CompB { * -> Classe.metodoA(..); }	-

Tabela 4.5: Detalhamento dos elementos que compõem uma visão

4.4.3 Descrição Comportamental

Em relação ao uso das rede de Petri na técnica CASTOR, vale ressaltar que:

- Toda transição deve estar associado com alguma operação presente no componente e vice-versa;
- Variáveis presentes nos arcos só assumem algum valor no momento em que a transição vinculada é disparada. Tal mapeamento é descrito na implementação da operação;
- As transições devem conter o nome da porta e a operação separados por um . ou ?. O primeiro símbolo indica que esta é uma operação requerida enquanto o segundo indica que é uma operação provida pelo componente.

Capítulo 5

Estudo de Caso

Neste capítulo, um estudo de caso será apresentado para a aplicação da técnica CASTOR. Para isto, iremos utilizar o sistema MyGrid, apresentado no Capítulo 2. O objetivo é demonstrar a técnica em um software real, além de validá-la em relação aos objetivos inicias. Inicialmente, a descrição arquitetural é discutida detalhadamente, bem como o mapeamento com o código. Em seguida, veremos como a conformidade é de fato verificada e como proceder na presença de alguma violação. Por último, apresentamos um sumário contendo algumas considerações a respeito da experimentação.

5.1 Arquitetura do MyGrid

Na Figura 5.1, é mostrado o diagrama de componentes que captura o cerne da arquitetura sob análise. A notação utilizada refere-se à versão 2.0 de UML [BK03]. Podemos constatar que o MyGrid é composto por três componentes principais: ***UI***, ***Scheduler*** e ***ReplicaExecutor***. O primeiro é o responsável pela interação do usuário com o escalonador (***Scheduler***), que por sua vez delega ao ***ReplicaExecutor*** a execução efetiva das tarefas submetidas. A comunicação entre ***UI*** e ***Scheduler*** se dá apenas de forma unidirecional, do primeiro para o segundo - a porta de UI (representada pelo quadrado em sua borda) requer uma interface (símbolo de semi-círculo), que é provida pelo componente ***Scheduler*** através de uma de suas portas. Já entre ***Scheduler*** e ***ReplicaExecutor*** a comunicação é bi-direcional, ambos componentes invocam operações do outro. Para cumprir sua função, o MyGrid precisa se comunicar com algum *Peer* da grade, possivelmente remoto, para obter as máquinas necessárias ao proces-

samento. O componente ***GuMP***, especificamente, é o responsável por prover tais máquinas.

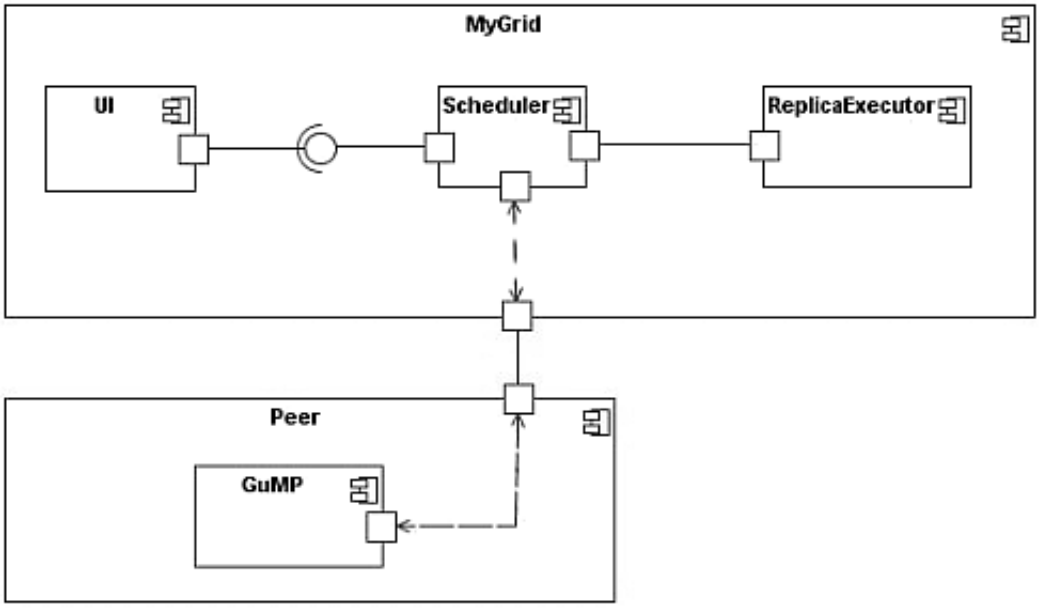

Figura 5.1: Arquitetura MyGrid – Diagrama de componentes

5.1.1 Descrição Estrutural

No Código 5.1, podemos ver a a especificação, em CASTOR, do componente ***UI***. O mapeamento estabelece que todas as classes do pacote `org.ourgrid.mygrid.ui`, exceto aquelas que têm no nome a palavra *Test* ou *Fake* (classes de teste), pertencem logicamente a este componente. A porta denominada `scheduler` requer que o outro par na comunicação forneça a operação através da qual o *job*[1] será submetido ao escalonador. Esta operação, por sua vez, exige um parâmetro inteiro que determina a quantidade de tarefas que compõem o *job*.

Já a especificação do componente ***GuMP*** é mostrada no Código 5.2. Na linha 2, vemos o uso da cláusula `import`, cuja finalidade é a mesma da utilizada na linguagem Java. Ou seja, serve para indicar a localização na hierarquia de pacotes das classes referenciadas na descrição. Este componente recebe requisições por máquinas provenientes do escalonador através da operação `wannaGums(int num)`. O parâmetro representa justamente a quantidade de máquinas solicitadas. Por sua vez, cada uma delas é entregue ao escalonador através

[1] Agrupamento de tarefas independentes a serem processadas

```
1  component UI{
2       class org.ourgrid.mygrid.ui.* && !org.ourgrid.mygrid.ui.*Test*
3            && !org.ourgrid.mygrid.ui.*Fake*;
4       port scheduler{
5            requires addJob(int num);
6       }
7  }
```

Código 5.1: Especificação do componente UI

da invocação da operação `hereIsGum()`, requerida pela porta *scheduler*. Na linha 8, está representado a implementação para a operação `wannaGums(int num)` que estabelece seu mapeamento para com o método de mesmo nome da classe `GridMachineProvider`. Durante a execução, o parâmetro `num`, da operação arquitetural, assume o valor do argumento deste método.

```
1  component GuMP{
2       import org.ourgrid.gump.GridMachineProvider;
3       class org.ourgrid.gump.*;
4       port scheduler{
5            provides wannaGums(int num);
6            requires hereIsGum();
7       }
8       wannaGums(int num){
9            GridMachineProvider.wannaGuMs(..,int numWantedGuMs){
10                num = numWantedGuMs;
11           }
12      }
13 }
```

Código 5.2: Especificação do componente GuMP

No Código 5.3, é mostrado a especificação do componente ***ReplicaExecutor***, composto por todas as classes do pacote `org.ourgrid.mygrid.replicaexecutor` (exceto aquelas que contém a palavra *Fake*). Nela, podemos observar um porta de nome `scheduler`, cuja finalidade é ser utilizada na comunicação com o escalonador. A operação `executeReplica()` é usada para delegar a execução das tarefas ao ***ReplicaExecutor***, e é mapeada para o método homônimo da classe `EBReplicaExecutorFacade`. Enquanto

que a operação `replicaFinished()` notifica o escalonador quanto ao término da execução de uma das tarefas.

```
1  component ReplicaExecutor{
2       import org.ourgrid.mygrid.replicaexecutor.EBReplicaExecutorFacade;
3       class org.ourgrid.mygrid.replicaexecutor.*
4            && !org.ourgrid.mygrid.replicaexecutor.*Fake*;
5       port scheduler{
6            provides executeReplica();
7            requires replicaFinished();
8       }
9       executeReplica(){
10           * EBReplicaExecutorFacade.executeReplica(..);
11      }
12 }
```

Código 5.3: Especificação do componente ReplicaExecutor

Por último, é mostrado a especificação de ***Scheduler***, o principal componente da arquitetura (ver Código 5.4). Todas suas portas são complementares às portas definidas nos componentes anteriores, uma vez que este componente se comunica com todos os outros. Aqui cabe uma ressalva, na linha 8 são excluídas as classes de exceção (cujo nome contém a palavra *Exception*) pois é comum o escalonador lançar exceções que são capturadas e acessadas por outros componentes através de chamadas de métodos - por exemplo, para obter a mensagem de erro. Caso tais classes não fossem excluídas do componente ***Scheduler***, essa situação seria caracterizada como uma violação da integridade de comunicação entre os componentes.

Na linha 19, é apresentado a construção (`internal`) de CASTOR não utilizada no capítulo anterior. Ela é utilizada para permitir que ações internas do componente (aquelas não presentes em nenhuma porta) possam ser utilizadas na especificação comportamental. Ou seja, algumas operações podem ser significativas o suficiente ao ponto de precisarem ser representadas na arquitetura. No entanto, estas podem não estar associadas diretamente à comunicação com nenhum outro componente. Sendo assim, o conceito de operações internas serve justamente para atender a essa necessidade. A sintaxe é a mesma para operações providas, cada uma é mapeada para um conjunto de métodos concretos. Neste caso, são definidas duas operações: `newGum()` e `freeGum()`. A primeira refere-se à criação de uma

```
1  import org.ourgrid.specs.JobSpec;
2  import org.ourgrid.util.config.Configuration;
3  import org.ourgrid.util.config.MyGridConfiguration;
4  import org.ourgrid.mygrid.scheduler.EBSchedulerFacade;
5  import org.ourgrid.mygrid.scheduler.GridMachineConsumer;
6  component Scheduler{
7       class org.ourgrid.mygrid.scheduler.*
8            && !org.ourgrid.mygrid.scheduler.*Exception*
9            && !org.ourgrid.mygrid.scheduler.*Test*;
10
10      port ui{ provides addJob(int num); }
11
11      port peer{
12           requires wannaGums(int num);
13           provides hereIsGum();
14      }
15
15      port executor{
16           requires executeReplica();
17           provides replicaFinished();
18      }
19
19      internal{
20           newGuM();
21           freeGuM();
22      }
23
23      addJob(int num){
24           * Scheduler.addJob(JobSpec jobSpec){
25                int maxReplicas = Integer.parseInt(
26                          Configuration.getInstance().getProperty(
27                               MyGridConfiguration.PROP_MAX_REPLICAS));
28                num = jobSpec.getTaskSpecs().size() * maxReplicas;
29           }
30      }
31
31      hereIsGum(){ * GridMachineConsumer.hereIsGuM(..); }
32
32      replicaFinished(){
33           * EBSchedulerFacade.replicaFinished(..);
34           * EBSchedulerFacade.replicaCanceled(..);
35           * EBSchedulerFacade.replicaAborted(..);
36           * EBSchedulerFacade.replicaFailed(..);
37      }
38
38      newGuM(){ GuMeXEntry.new(..); }
39
39      freeGuM(){
40           * GuMeXEntry.free(..);
41           * GuMeXEntry.fastFree(..);
42      }
43 }
```

Código 5.4: Especificação do componente Scheduler

instância para encapsular a máquina recém entregue pelo ***Gump***. Enquanto que a segunda, captura o evento de liberação da máquina assim que a execução de uma tarefa é finalizada. Veremos na seção seguinte o uso efetivo destes conceitos.

Como forma de maximizar o uso da grade, o MyGrid replica as tarefas submetidas uma certa quantidade de vezes e as aloca para diferentes máquinas. No momento em que a primeira tarefa tem seu processamento finalizado, o usuário é notificado e as outras tarefas são imediatamente canceladas. Na verdade, a própria tarefa original é tratada também como uma réplica. Esse mecanismo aproveita melhor os recursos computacionais disponíveis, uma vez que diminui a degradação de performance causada por máquinas demasiadamente lentas. Na linha 25 à 28 é feito justamente esse cálculo para que o parâmetro `num` possa ser mapeado corretamente. Ou seja, a quantidade real de tarefas corresponde ao índice de replicação vigente multiplicado pela quantidade de tarefas no *job*.

A topologia da arquitetura mostrada na Figura 5.1, bem como as comunicações excepcionais entre componentes são definidas na visão MyGrid (ver Código 5.5). Na linha 10, por exemplo, estabelece-se que é permitida qualquer classe do componente ***UI*** se comunicar com o componente ***Scheduler*** através dos métodos públicos de `Job` e `Task`. Fazem parte também desta lista os métodos `Scheduler.jobList()`, `Scheduler.cancelJob(int)` e `Scheduler.waitForJob(int)`. O primeiro simplesmente consulta a relação de *jobs* submetidos, o segunda cancela a execução de um certo *job*, enquanto que o último encerra as execuções e aguarda até que um *job* específico seja finalizado. Vale ressaltar que todos esses métodos foram abstraídos na representação arquitetural. No entanto, em CASTOR, faz-se necessário definir tal abstração explicitamente, por isso a necessidade de listá-los na especificação.

Entre o ***Scheduler*** e ***Executor*** a única comunicação excepcional refere-se ao cancelamento das réplicas de um determinado *job*. Isso é necessário para cancelar réplicas remanescentes de um *job* já finalizado. Já entre o ***GuMP*** e o ***Scheduler***, algumas mensagens trocadas destinam-se a informar um ao outro sobre uma possível inalcançabilidade das máquinas (`GridMachineConsumer.gumIsDead(..)` e `GridMachineProvider.lostGuM(..)`). O método `GridMachineProvider.disposeOf(..)`, por sua vez, devolve ao ***GuMP*** máquinas que já foram utilizadas pelos escalonador. E finalmente

`GridMachineProvider.noMoreGuMs(..)` notifica ao ***GuMP*** que o escalonador não necessita mais de máquinas.

```
1  import org.ourgrid.mygrid.scheduler.Job;
2  import org.ourgrid.mygrid.scheduler.Task;
3  import org.ourgrid.mygrid.scheduler.Scheduler;
4  import org.ourgrid.mygrid.scheduler.GridMachineConsumer;
5  import org.ourgrid.gump.GridMachineProvider;

6  view MyGrid{
7       connect Scheduler.ui, UI.scheduler;
8       connect Scheduler.executor, ReplicaExceutor.scheduler;
9       connect Scheduler.peer, Gump.scheduler;

10      exception UI -> Scheduler{
11           * -> public * Job.*(..);
12           * -> public * Task.*(..);
13           * -> * Scheduler.jobList();
14           * -> * Scheduler.cancelJob(int);
15           * -> * Scheduler.waitForJob(int);
16      }

17      exception Scheduler -> Executor{
18           * -> * EBReplicaExecutorFacade.cancelReplicasOfJob(..);
19      }

20      exception Gump -> Scheduler{
21           * -> * GridMachineConsumer.gumIsDead(..);
22      }

23      exception Scheduler -> Gump{
24           * -> * GridMachineProvider.lostGuM(..);
25           * -> * GridMachineProvider.disposeOf(..);
26           * -> * GridMachineProvider.noMoreGuMs(..);
27      }
28 }
```

Código 5.5: Descrição da arquitetura do MyGrid

5.1.2 Descrição Comportamental

Dentre todos os componentes mostrados até aqui, o interesse quanto ao comportamento restringe-se ao ***Scheduler*** e ***ReplicaExecutor***. Os demais componentes, portanto, só possuem a descrição estrutural. O motivo na caso do ***GuMP***, se deve ao fato deste não fazer parte do MyGrid, e o ***UI*** por não ter relevância no que tange a especificação de seu comportamento. Na Figura 5.2, é mostrada a rede de Petri que descreve o comportamento do componente ***Scheduler***.

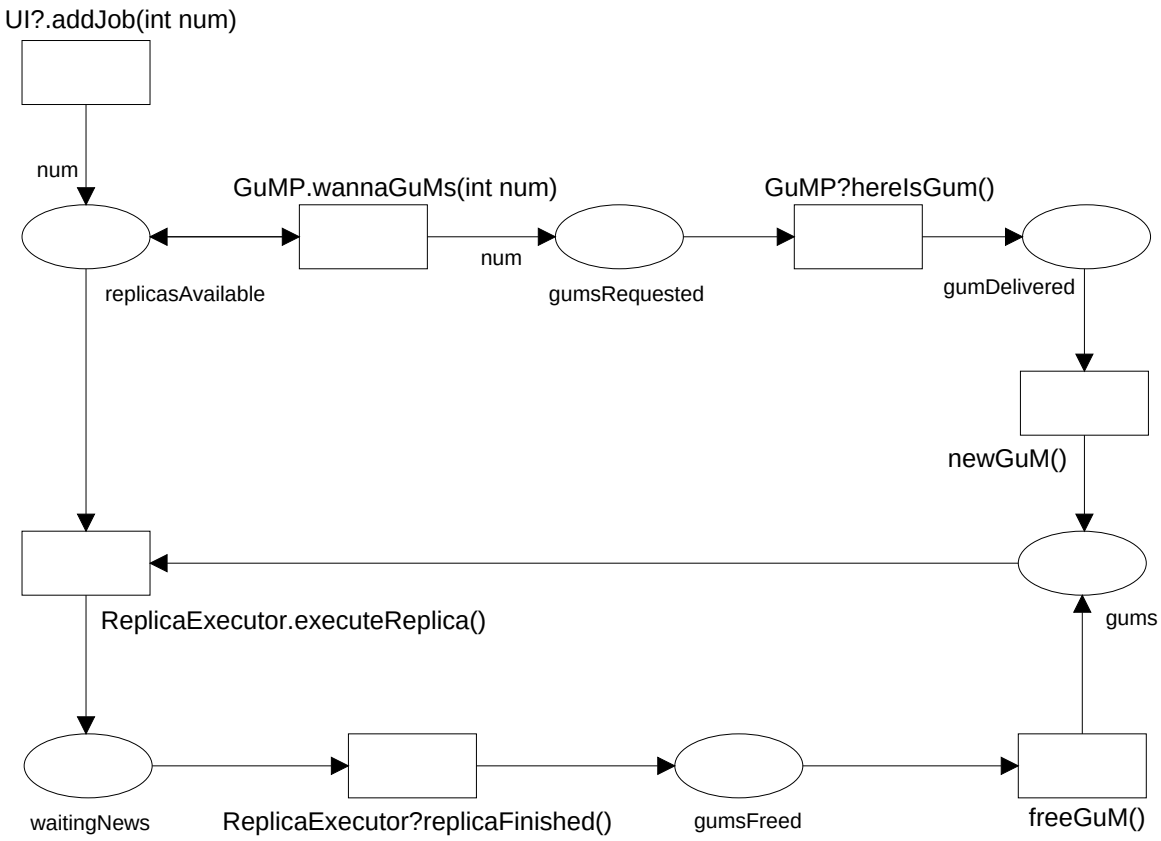

Figura 5.2: Rede de Petri do componente Scheduler

É importante ressaltar alguns aspectos deste modelo:

- O disparo de *UI?addJob(int num)* coloca a quantidade fichas no lugar *replicasAvailable* igual ao número de réplicas criadas no sistema.
- O disparo de *GuMP.wannaGuMs(int num)* não altera o lugar *replicasAvailable* devido ao arco ser bi-direcional.
- Possivelmente, nem todas as fichas do lugar *gumsRequested* sejam consumidas. Para cada réplica é solicitado uma máquina, no entanto a quantidade de máquinas cedidas para a execução pode ser inferior a esse número.
- Para que haja a execução de uma réplica é preciso haver disponível a própria réplica (uma ficha no lugar *replicasAvailable*) e uma máquina cedida pela grade (uma ficha no lugar *gums*).
- Ao final de uma sessão de utilização do MyGrid, espera-se que apenas os lugares *gums* e *gumsRequested* possuam fichas.

O comportamento do ***ReplicaExecutor*** é bem mais simples (ver Figura 5.3). Seu comportamento é abstraído em duas operações: i) o recebimento da solicitação de executar uma certa réplica proveniente do escalonador (`ReplicaExecutor?executeReplica()`); ii) notificação que a execução foi finalizada (`ReplicaExecutor.replicaFinished()`).

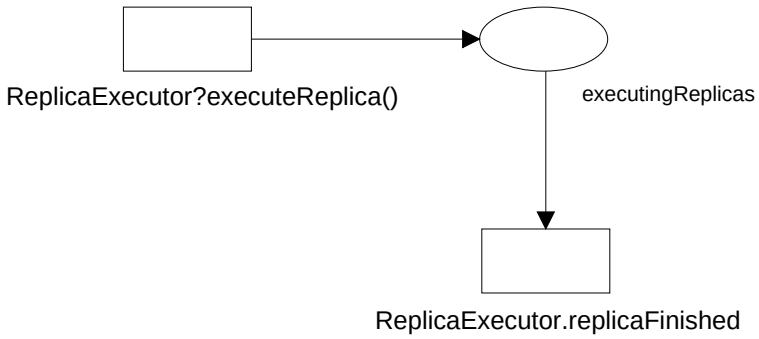

Figura 5.3: Rede de Petri do componente ReplicaExecutor

5.2 Verificação

Os componentes que não possuem especificação comportamental, obviamente, só podem ser verificados quanto à sua estrutura. O ***GuMP*** por exemplo, é um deles. Nas seções anteriores, foi visto que tal componente só se comunica com o ***Scheduler***. No Código 5.2, é mostrado o aspecto que verifica a integridade de comunicação entre eles. Os pontos de corte podem ser interpretados da seguinte forma:

- **schedulerOutCall()** - Toda invocação para métodos do componente ***Scheduler*** (classes do pacote `org.ourgrid.mygrid.scheduler` exceto aquelas que contenham as palavras *Exception* ou *Test* no nome), que não seja através da porta (`GridMachineConsumer.hereIsGuM(..)`) e que não seja uma comunicação excepcional (`GridMachineConsumer.gumIsDead(..)`), proveniente do componente ***GuMP*** deve ser considerada um erro arquitetural.

- **schedulerInCall()** - Toda invocação para métodos do componente ***GuMP*** (classes do pacote `org.ourgrid.gump`), que não seja através da porta (`GridMachineProvider.wannaGuMs(String,int,..)`) e que não seja uma comunicação excepcional (`lostGuM()`, `disposeOf()` e `noMoreGuMs()` da classe `GridMachineProvider`), proveniente do componente ***Scheduler*** deve ser considerada um erro arquitetural.

Todos os demais componentes são verificados da mesma forma. O código completo dos aspectos utilizados neste estudo de caso podem ser vistos no Apêndice A.

No Código 5.7, é mostrado um trecho do código responsável pela verificação comportamental do componente ***Scheduler***. O primeiro ponto de corte refere-se à uma operação pertencente provida pelo próprio componente, a segunda à uma operação requerida e a última é uma operação interna. Todas as informações contidas nos aspectos de verificação são extraídas das especificações do sistema, o que possibilita a geração automática destes. Assim como mostrado no Capítulo 4, o estudo de caso fez uso da API JMobile para simular os modelos construídos. As linhas 7, 13 e 15 tentam disparar a transição associado ao ponto de corte passando o seu nome através do método `fire(String)` (não mostrado aqui). Na linha 11, é feita a ligação entre o parâmetro da operação arquitetural com o valor mapeado no

```
1  import org.ourgrid.mygrid.scheduler.GridMachineConsumer;
2  import org.ourgrid.gump.GridMachineProvider;

3  public aspect GuMPObserver {

4       pointcut schedulerOutCall() :
5            call(* org.ourgrid.mygrid.scheduler.*.*(..))
6            && !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..)))
7            && !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..)))
8            && !(call(* GridMachineConsumer.hereIsGuM(..)))
9            && !(call(* GridMachineConsumer.gumIsDead(..)))
10           && within(org.ourgrid.gump.*);

11      pointcut schedulerInCall() : call(* org.ourgrid.gump.*.*(..))
12           && !(call(* GridMachineProvider.wannaGuMs(String,int,..)))
13           && !(call(* GridMachineProvider.lostGuM(..)))
14           && !(call(* GridMachineProvider.disposeOf(..)))
15           && !(call(* GridMachineProvider.noMoreGuMs(..)))
16           && within(org.ourgrid.mygrid.scheduler.*)
17           && !(within(org.ourgrid.mygrid.scheduler.*Exception*))
18           && !(within(org.ourgrid.mygrid.scheduler.*Test*));

19      declare error: schedulerOutCall(): "Integridade de comunicação\n" +
20           "GuMP -> Scheduler violada.";

21      declare error: schedulerInCall(): "Integridade de comunicação\n" +
22           "Schduler -> GuMP violada.";
23 }
```

Código 5.6: Aspecto de verificação do componente GuMP

sistema. Esses pontos são específicos da representação escolhida para os modelos comportamentais. Ou seja, para cada formalismo, possivelmente, as construções seriam diferentes nessas linhas.

```
...
pointcut hereIsGuM(): execution(* GridMachineConsumer.hereIsGuM(..));

pointcut wannaGuMs(String requirements,int numWantedGuMs):
     call(* GridMachineProvider.wannaGuMs(..))
     && args(requirements, numWantedGuMs,..);

pointcut newGuMeXEntry(): execution(GuMeXEntry.new(..));

after(): hereIsGuM(){ fire("Scheduler.hereIsGum"); }

after(String requirements, int requestedGums):
     wannaGuMs(requirements, requestedGums){
     try{
          sch.changeOutWeight("wannaGums","gumsRequested",requestedGums);
     }catch(Exception e){ e.printStackTrace(); }
     fire("Scheduler.wannaGums");
}

after(): newGuMeXEntry(){ fire("Scheduler.newGumex"); }
...
```

Código 5.7: Trecho de verificação do comportamento de Scheduler

Para ilustrar a relação comportamental entre o modelo e o software, iremos utilizar a Figura 5.4. Ela mostra a árvore de alcançabilidade parcial da rede de Petri que descreve o componente ***Scheduler*** (ver Figura 5.2). Cada nó é representado por uma 6-tupla, na qual cada posição especifica a quantidade de fichas nos lugares *replicasAvailable*, *gumsRequested*, *gumDelivered*, *gums*, *waitingNews* e *gumsFreed*, respectivamente. Além disso, os nós estão rotulados com letras do alfabeto para que possamos referencia-los e os arcos são identificado com o nome da transição que levou à mudança de estado. Para um grau de replicação igual a 1, inicialmente, não há nenhuma ficha em qualquer lugar da rede (estado *A*). De forma mandatória, a primeira transição a ser disparada é `addJob(int)` com um determinado valor como argumento. *A* partir de então, o modelo irá para o estado *B* caso sejam submetidas duas tarefas, e para o estado *C*, caso seja apenas uma. Dessa mesma forma, o restante da árvore é construída.

O que se deseja é que o espaço de estados do software abstraído (considerando os pontos mapeados na arquitetura) esteja contido no espaço de estados do modelo em rede de Petri.

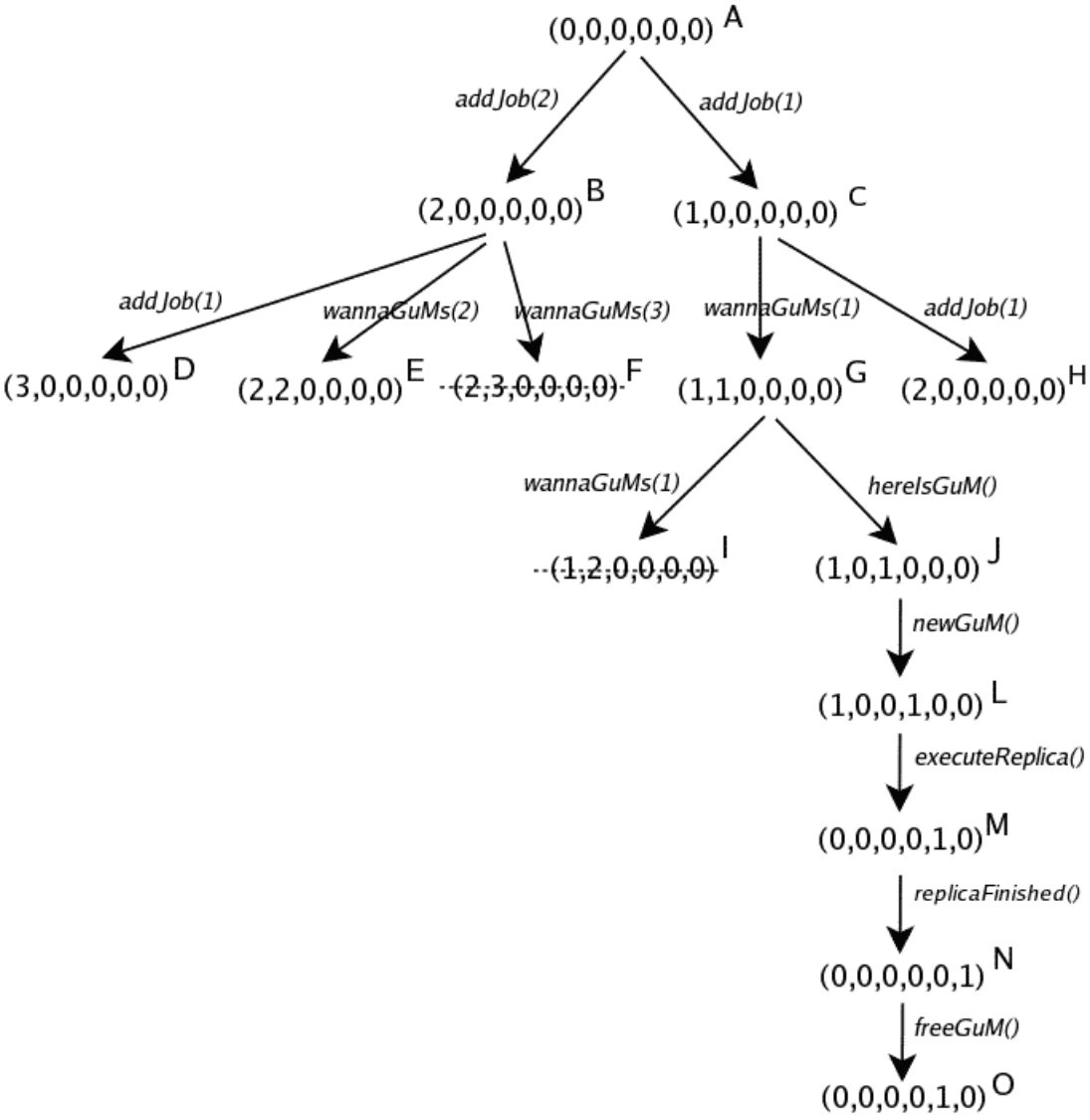

Figura 5.4: Árvore de alcançabilidade parcial do componente Scheduler

Isso não significa dizer que o software, enquanto não houver violação, não possui falhas. Por exemplo, o nó *F* representa um estado indesejado, já que para atingi-lo deve ter havido a solicitação de três máquinas enquanto só existem duas tarefas a serem processadas. Em uma situação parecida, o estado *I* também configura um erro, já que possibilita duas solicitações consecutivas (de uma máquina cada) para apenas uma tarefa disponível. No primeiro caso, o formalismo escolhido não permite restringir que este caminho seja alcançável no modelo. Já no segundo, seria possível resolvê-lo pela adição de mais um lugar de entrada na transição `wannaGuMs(int)` (com a quantidade de fichas igual ao número de *jobs* submetidos). Por outro lado, vale ressaltar que o modelo consegue excluir inúmeros caminhos errôneos, como por exemplo, todos aqueles que não iniciados pela transição `addJob(int)`. Sendo assim, diante da não existência de violações pode-se dizer que o software, até um dado momento, está correto em relação ao modelo. O estado *O* é alcançado pela execução normal de um *job* com uma única tarefa.

5.2.1 Violação

Existem dois tipos de violação: a estrutural e a comportamental. Obviamente, ambas podem gerar falsos positivos caso a própria especificação não esteja correta. No caso da violação estrutural, é uma fonte comum de erros a designação das classes que compõem um determinado componente e a relação das comunicações excepcionais. O primeiro, por exemplo, ficaria caracterizado se não houvéssemos excluído do componente ***Scheduler*** todas as classes de teste e de exceção. Quanto ao segundo, é evidente as conseqüências de não incluir em tal relação alguma comunicação (entre componentes) deliberadamente abstraída.Vale ressaltar que a inexistência de violações estruturais é uma condição necessária para a conformidade comportamental.

Em se tratando do formalismo de redes de Petri, uma transição só não consegue ser disparada se os seus lugares de entrada não possuírem a quantidade de fichas exigidas no arco (todos os arcos de entrada da rede ***Scheduler*** possuem peso 1). Uma violação comportamental é justamente a não possibilidade de reproduzir no modelo, um evento ocorrido no software. Pela lógica então, a transição que representa tal evento só não poderá ser disparada em duas situações:

1. Alguma transição anterior a esta não colocou a quantidade devida de fichas nos lugares de entrada;

2. Alguma outra transição consumiu indevidamente as fichas antes que fosse disparada.

Quando uma violação desta natureza é identificada, a primeira coisa a ser feita é se certificar que o modelo realmente representa o comportamento esperado do software. Uma vez certo disso, restam apenas duas opções: ou o erro é no mapeamento, ou o sistema possui, de fato, uma falha. Erros no mapeamento podem ocasionar ambas as situações descritas acima e são classificados nos seguintes tipos:

- Ausência de certos métodos na operação arquitetural;
- Inclusão de um método que não deveria fazer parte da operação;
- Erro na atribuição de valor do parâmetro utilizado na operação arquitetural.

Vamos agora exemplificar o que foi dito até aqui com algumas possíveis violações comportamentais no contexto deste estudo de caso. Suponha que o modelo se encontra no estado *J* (ver Figura 5.4), e nesse ponto haja a tentativa de disparar a transição `ReplicaExecutor.executeReplica()`. Pelo fato de não haver nenhuma ficha no lugar *gums*, obviamente, ela não poderá ser executada. Considerando que o modelo está correto, a causa da violação foi o não disparo anterior da transição `newGuM()`. Nessa situação, deve-se examinar se a lista de métodos que compõem esta operação está completa. Caso esteja faltando relacionar algum método, a invocação deste não será detectada em tempo de execução e portanto a transição com a qual está associada não será disparada. Este é um caso típico em que a violação se deve a um erro na própria especificação. Um outra situação parecida poderia ocorrer se o disparo de um transição que possui uma variável em algum arco de saída (ex: `UI.addJob(int num)`), produzisse menos fichas do que deveria. Isso seria possível, por exemplo, se na linha 28 do Código 5.4 fosse esquecido de multiplicar a quantidade de tarefas no *job* pelo índice de replicação (variável `maxReplicas`) na atribuição de valor à variável `num`.

5.3 Sumário

O estudo de caso demonstrou a capacidade de CASTOR em descrever arquiteturas de software e analisá-las quanto a conformidade com o código. Apesar do considerável nível de complexidade do MyGrid, consideramos que o custo despendido na aplicação da técnica é mínimo se comparado aos benefícios esperados. É notória também a impossibilidade de, baseado apenas nesse estudo de caso, avaliar de uma forma definitiva a eficácia da técnica quanto aos problemas da integridade conceitual e da deterioração de software. No entanto, espera-se que a argumentação desenvolvida ao longo deste trabalho tenha dado evidências suficientes que, de fato, os objetivos levantados inicialmente foram alcançados. Vale apresentar aqui algumas limitações da técnica, identificadas durante a realização do experimento, a saber:

- **Suporte a configurações dinâmicas** - CASTOR pressupõe que a topologia da arquitetura (a organização dos componentes em relação a comunicação entre eles) é estática. Ou seja, durante toda a execução ela se mantém inalterada não podendo haver mudança nas regras de comunicação.

- **Herança entre componentes** - A relação entre componentes distintos se resume necessariamente à apenas a comunicação através de invocação de métodos. Não deve haver nenhum tipo de herança entre classes pertencentes a componentes diferentes. Isto poderia comprometer a verificação da conformidade estrutural, uma vez que a análise estática não contempla este tipo de relação entre componentes.

- **Múltiplas instâncias de componentes** - Não há suporte para o conceito de instância de componente. Ou seja, a cardinalidade de cada um dos componentes é sempre 1.

Capítulo 6

Conclusão

Nesta dissertação, tratamos do problema de se avaliar os sistemas de software quanto à conformidade arquitetural. Consideramos que a falta de disciplina no desenvolvimento de software aliada a uma certa desvalorização de práticas voltadas para a especificação (principalmente arquitetural) são problemas, de certa forma, comuns nos projetos de pequeno e médio porte. Isso se deve, em grande parte, à alta competitividade e às pressões para a entrega cada vez mais rápida do produto. No entanto, este cenário acaba acarretando ainda mais problemas no futuro, principalmente na fase de manutenção do software devido a baixa qualidade do código produzido. Uma das principais preocupações no desenvolvimento deste trabalho foi a concepção de uma técnica automatizada capaz de ser efetivamente aplicada na indústria. Para tanto, faz-se necessário a disponibilidade de ferramental apropriado para suporte à técnica, o que não fez parte do escopo da dissertação. Contudo, a técnica mostrou ser totalmente passível de ser implementada em uma ferramenta, o que naturalmente constitui um dos seus possíveis desdobramentos.

A arquitetura de software vem progressivamente se consolidando como uma importante área de pesquisa da Engenharia de Software. No entanto, ainda não existe um consenso estabelecido quanto aos conceitos, definições, técnicas e métodos que a envolvem. Isto sem dúvida virá com a natural transferência deste conhecimento para a indústria. Sendo assim, procuramos utilizar como base para esta pesquisa o que havia de comum entre os autores. Tal estratégia foi usada, por exemplo, na formulação do que deveria ser incluído na descrição arquitetural (componentes, portas e conectores). Quanto à especificação comportamental, optamos por utilizar o formalismo de rede de Petri como prova de conceito da técnica. Essa

escolha se deveu principalmente ao conhecimento do grupo sobre o assunto e a disponibilidade de uma API para a simulação de tais modelos. Além disso, consideramos que as redes de Petri Lugar-Transição (no qual as fichas não possuem tipos) eram simples o suficiente e possuíam a expressividade ideal para modelar o comportamento. No entanto, talvez outros formalismos, como por exemplo *statecharts* [Har87], possam ser posteriormente experimentados.

6.1 Contribuições

Após a conclusão deste trabalho, algumas contribuições trazidas pelo mesmo podem ser destacadas. Primeiramente, ele promove a difusão da importância da arquitetura de software na medida em que disponibiliza uma técnica que agrega ainda mais valor à este tipo de documentação. Além dos benefícios já conhecidos que o cuidado com a arquitetura pode propiciar ao projeto, CASTOR acrescenta ainda a possibilidade de avaliar, com um baixo custo, a observância do código em relação a arquitetura concebida.

Outra contribuição importante refere-se ao problema da deterioração de software. É sabido que a arquitetura tem um papel fundamental ao longo da evolução de uma sistema, principalmente em relação a sua manutenibilidade. Identificar desvios da implementação para poder manter atualizada a documentação arquitetural, normalmente, demanda um esforço manual de revisão. Vale lembrar que este tipo de atividade costuma ser dispendioso e suscetível a erros. CASTOR ameniza tais problemas, uma vez que automatiza a tarefa de verificar o quão o código se deteriorou, ou seja, quanto este se afastou do projeto previsto.

Em decorrência também da conformidade arquitetural, podemos considerar que CASTOR também promove a integridade conceitual do software. A coesão do design é um fator de suma importância para a qualidade do código. Até mesmo porque, atualmente os desenvolvedores têm muito mais autonomia para tomar decisões que impactam nos principais atributos de qualidade do projeto. É necessário manter um referencial, a partir do qual seja possível detectar quando um desenvolvedor tome uma ação que divirja do arcabouço obtido de forma consensual entre a equipe.

Por último, este trabalho marca a abertura de uma nova linha de pesquisa do grupo, a qual tenta utilizar métodos formais, sem o grande custo normalmente a eles associados, na

solução de problemas comuns da indústria de software.

6.2 Trabalhos Futuros

Alguns trabalhos futuros podem ser destacados. Conforme já mencionado, o mais direto deles é a implementação da técnica em uma ferramenta de apoio ao processo de descrição, verificação e depuração das possíveis violações encontradas. Além disso, tratar das limitações apresentadas no Capítulo 5. O mapeamento entre os eventos arquiteturais e do código poderia contar com outras ações baixo nível além da invocação de métodos, como acesso a atributos, criação de processos, dentre outros. Isto daria maior poder ao modelador na definição do que deveria ser capturado pela descrição comportamental da arquitetura.

Considerar também na técnica a possibilidade de estabelecer regras de *design* (não arquiteturais), como convenções de nomenclatura e padrões de projeto, a tornaria mais completa na medida em que expande sua capacidade de expressar e verificar propriedades do código. Avaliar a viabilidade de utilizar outros formalismos para descrever o comportamento, ou até mesmo adaptar algum já existente (podendo ser inclusive a próprias redes de Petri) para melhor se adequar aos propósitos da técnica.

Outros trabalhos relevantes que também podem ser realizados são os seguintes:

1. Verificação formal do modelo comportamental: Visa aproveitar a disponibilidade dos modelos que descrevem o comportamento do componente para fazer verificação formal. Uma possibilidade seria gerar o espaço de estados e utilizar a técnica de *model-checking* [CGP99] para analisar a presença ou não de certas propriedades descritas em lógica temporal. Essa atividade poderia ser feita inclusive, antes do início da implementação com a finalidade de ganhar confiança quanto a correção do projeto.

2. Métricas para integridade conceitual e deterioração de software: A literatura consultada não define uma forma de quantificar tais conceitos. Seria interessante a formulação de algum tipo de métrica tendo em vista a técnica CASTOR.

Bibliografia

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM Press.

[Bac00] Bass L. Buhman C. Comella-Dorda S. Long F. Seacord R.J. Wallnau K Bachmann, F. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000. (CMU/SEI-2000-TR-008).

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[BGHK02] Jan Bosch, W. Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors. *Software Architecture: System Design, Development and Maintenance, IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA3), August 25-30, 2002, Montréal, Québec, Canada*, volume 224 of *IFIP Conference Proceedings*. Kluwer, 2002.

[BK03] Morgan Bjorkander and Cris Kobryn. Architecting systems with uml 2.0. *IEEE Softw.*, 20(4):57–61, 2003.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CG01] Shang-Wen Cheng and David Garlan. Mapping architectural concepts to uml-rt. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 2001.

[CGB+02] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[CGP99] Edmund Clarke, Orna Grumber, and Doron Peled. *Model Checking*. MIT Press, 1999.

[Crn01] Ivica Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001.

[DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, , 1998.

[dMS05] Taciano de Morais Silva. Simulação automática e geração de espaço de estados de modelos em redes de petri orientadas a objetos. Master's thesis, COPIN - Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brasil, 2005.

[FK99] Ian Foster and Carl Kesselman. Computational grids. pages 15–51, 1999.

[FPB95] Jr. Frederick P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gar00] David Garlan. Software architecture: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[GCK02] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002.

[GH00] John Grundy and John Hosking. High-level static and dynamic visualization of software architectures. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 5, Washington, DC, USA, 2000. IEEE Computer Society.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*, 21(4):269–274, 1995.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[Gue98] Dalton D. Serey Guerrero. Orientação a objetos e modelos de redes de petri. Technical report, UFPB - COPELE, Abril 1998.

[Gue02] Dalton D. Serey Guerrero. *Redes de Petri Orientadas a Objetos*. PhD thesis, Curso de Pós-gradução em Engenharia Elétrica, Universidade Federal da Paraíba, Campina Grande, Paraíba, Brasil, Abril 2002.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[IEE00] IEEE Architecture Working Group,. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. 2000. report IEEE Std 1471-2000.

[JAW02] Jeffrey M. Voas James A. Whittaker. 50 Years of Software: Key Principles for Quality. *IT Professional*, 4(6):28–35, November/December 2002.

[Jen92] K. Jensen. *Coloured Petri Nets 1: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, Berlin, Alemanha, 1992.

[JR97] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 56, Washington, DC, USA, 1997. IEEE Computer Society.

[Kat] Joost-Pieter Katoen. *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course "Mechanised Validation of Parallel Systems", YEAR = 1999. Friedrich-Alexander Universitat Erlangen-Nurnberg.

[KC99] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engg.*, 6(2):107–138, 1999.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[K.L92] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Mic] Sun Microsystems. Java compiler compiler (javacc). URL: https://javacc.dev.java.net/ (Acessado em Fev/2006).

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18–28, 1995.

[MS01] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par94] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PBP01] Daniel J. Paulish, Len Bass, and D. J. Paulish. *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Pro05] AspectJ Project. The aspectj project at eclipse.org. Web site, November 2005. URL: http://eclipse.org/aspectj/.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RME03] Donald J. Reifer, Frank Maurer, and Hakan Erdogmus. Scaling agile methods. *IEEE Softw.*, 20(4):12–14, 2003.

[SAG+] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems using colored petri nets. November.

[SEI05] SEI. How do you define software architecture?, November 2005. URL: http://www.sei.cmu.edu/architecture/definitions.html, 2003.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Sha01] Mary Shaw. The coming-of-age of software architecture research. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, page 656, Washington, DC, USA, 2001. IEEE Computer Society.

[SSC96a] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-oriented visualization. *SIGPLAN Not.*, 31(10):389–405, 1996.

[SSC96b] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998. SZY c 98:1 1.Ex.

[TH03] James Tomayko and James Herbsleb. How useful is the metaphor component of agile methods? a preliminary study. Technical report, School of Computer Science, Carnegie Mellon, June 2003.

[The99] The Standish Group. Chaos, recipe for success. 1999. http://www.pm2go.com/sample-research/chaos1998.pdf.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.

[Wes02] *Metaphor, Architecture and XP*, In Proceedings of the 2002 XP Conference, 2002.

[WMSR00] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 12. IBM Press, 2000.

Apêndice A

Código do Estudo de Caso

O objetivo dste Apêndice é apresentar o código por completo dos aspectos utilizados no estudo de caso. São ao todo quatro códigos, um para cada componente (***GuMP***, ***ReplicaExecutor***, ***UI*** e ***Scheduler*** respectivamente).

GuMPObserver.aj

```
1  package br.edu.ufcg.gmf.archconform;
2
3  import org.ourgrid.mygrid.scheduler.GridMachineConsumer;
4  import org.ourgrid.gump.GridMachineProvider;
5
6  public aspect GuMPObserver {
7
8       pointcut schedulerOutCall() :
9            call(* org.ourgrid.mygrid.scheduler.*.*(..)) &&
10           !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
11           !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
12           !(call(* GridMachineConsumer.hereIsGuM(..))) &&
13           !(call(* GridMachineConsumer.gumIsDead(..))) &&
14           within(org.ourgrid.gump.*);
15
16      pointcut schedulerInCall() :  call(* org.ourgrid.gump.*.*(..)) &&
17           !(call(* GridMachineProvider.wannaGuMs(String,int,..))) &&
18           !(call(* GridMachineProvider.lostGuM(..))) &&
19           !(call(* GridMachineProvider.disposeOf(..))) &&
20           !(call(* GridMachineProvider.noMoreGuMs(..))) &&
21           within(org.ourgrid.mygrid.scheduler.*) &&
22           !(within(org.ourgrid.mygrid.scheduler.*Exception*)) &&
23           !(within(org.ourgrid.mygrid.scheduler.*Test*));
24
25      declare error: schedulerOutCall(): "Integridade de comunicação\n" +
26           "GuMP -> Scheduler violada.";
27
28      declare error: schedulerInCall(): "Integridade de comunicação\n" +
29           "Scheduler -> GuMP violada.";
30 }
```

ReplicaexecutorObserver.aj

```
1  package br.edu.ufcg.gmf.archconform;
2  import jmobile.examples.mygrid.ReplicaExecutor;
3  import dividuo.dssg.jmobile.rpoo.JMConfiguration;
4  import org.apache.log4j.*;
5  import org.ourgrid.mygrid.replicaexecutor.EBReplicaExecutorFacade;
6  import org.ourgrid.mygrid.scheduler.EBSchedulerFacade;
7  public aspect ReplicaexecutorObserver {
8      private static transient final Logger LOG =
9           Logger.getLogger("ReplicaExecutor");
10     JMConfiguration conf = new JMConfiguration();
11     ReplicaExecutor repexec = new ReplicaExecutor("ReplicaExecutor");
12     ReplicaexecutorObserver(){
13         SimpleLayout layout = new SimpleLayout();
14         FileAppender appender = null;
15         try {
16            appender =
17               new FileAppender(layout,"replicaexcutor.log",false);
18         } catch(Exception e) {}
19         LOG.addAppender(appender);
20          conf.add(repexec);
21     }
22     pointcut executeReplica():
23          call(* EBReplicaExecutorFacade.executeReplica(..));
24     pointcut replicaFinished():
25          call(* EBSchedulerFacade.replicaFinished(..));
26     after(): executeReplica(){
27          fire("ReplicaExecutor.executeReplica");
28     }
29     after(): replicaFinished(){
30          fire("ReplicaExecutor.replicaFinished");
31     }
32     private void fire(String transitionName){
33          try{
34               LOG.info(conf);
35               LOG.info(transitionName);
36               conf = conf.execute(transitionName,conf);
37          }catch(Exception e){
38               LOG.error(e.getMessage(),e);
39          }
40     }
41     pointcut schedulerInCall() :
42          call(* org.ourgrid.mygrid.replicaexecutor.*.*(..)) &&
```

```
43            !(call( * EBReplicaExecutorFacade.executeReplica(..))) &&
44            !(call(* EBReplicaExecutorFacade.cancelReplicasOfJob(..))) &&
45            within(org.ourgrid.mygrid.scheduler.*) &&
46            !(within(org.ourgrid.mygrid.scheduler.*Exception*)) &&
47            !(within(org.ourgrid.mygrid.scheduler.*Test*));

48       pointcut schedulerOutCall() :
49            call(* org.ourgrid.mygrid.scheduler.*.*(..))  &&
50            !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
51            !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
52            !(call(* EBSchedulerFacade.replicaFinished(..))) &&
53            !(call(* EBSchedulerFacade.replicaFailed(..))) &&
54            !(call(* EBSchedulerFacade.replicaCanceled(..))) &&
55            !(call(* EBSchedulerFacade.replicaAborted(..)))     &&
56            within(org.ourgrid.mygrid.replicaexecutor.*) &&
57            !(within(org.ourgrid.mygrid.replicaexecutor.*Fake*));

58       declare error: schedulerInCall(): "Integridade de comunicação\n" +
59            "Scheduler -> ReplicaExecutor violada.";

60       declare error: schedulerOutCall(): "Integridade de comunicação\n" +
61            "ReplicaExecutor -> Scheduler violada.";
62  }
```

UiObserver.aj

```
1  package br.edu.ufcg.gmf.archconform;
2  import org.ourgrid.mygrid.scheduler.Scheduler;
3  import org.ourgrid.mygrid.scheduler.Job;
4  import org.ourgrid.mygrid.scheduler.Task;
5  import org.ourgrid.specs.JobSpec;
6  public aspect UiObserver {
7      pointcut schedulerOutCall() :
8           call(* org.ourgrid.mygrid.scheduler.*.*(..)) &&
9           !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
10          !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
11          !(call( * Scheduler.addJob(JobSpec))) &&
12          !(call( * Scheduler.jobList())) &&
13          !(call( * Scheduler.cancelJob(int))) &&
14          !(call( * Scheduler.waitForJob(int))) &&
15          !(call(public * Job.*(..))) &&
16          !(call(public * Task.*(..))) &&
17          within(org.ourgrid.mygrid.ui.*) &&
18          !(within(org.ourgrid.mygrid.ui.*Test*)) &&
19          !(within(org.ourgrid.mygrid.ui.*Fake*));
20     pointcut schedulerInCall() :
21          call(* org.ourgrid.mygrid.ui.*.*(..)) &&
22          !(call(* org.ourgrid.mygrid.ui.*Test*.*(..))) &&
23          !(call(* org.ourgrid.mygrid.ui.*Fake*.*(..))) &&
24          within(org.ourgrid.mygrid.scheduler.*)  &&
25          !(within(org.ourgrid.mygrid.scheduler.*Exception*)) &&
26          !(within(org.ourgrid.mygrid.scheduler.*Test*));
27     declare error: schedulerInCall(): "Integridade de comunicação\n" +
28          "Scheduler -> UI violada.";
29     declare error: schedulerOutCall(): "Integridade de comunicação\n" +
30          "UI -> Scheduler violada.";
31 }
```

SchedulerObserver.aj

```
1  package br.edu.ufcg.gmf.archconform;
2  import jmobile.examples.mygrid.Scheduler;
3  import dividuo.dssg.jmobile.rpoo.JMConfiguration;
4  import org.apache.log4j.*;

5  import org.ourgrid.mygrid.scheduler.GridMachineConsumer;
6  import org.ourgrid.gump.GridMachineProvider;
7  import org.ourgrid.mygrid.scheduler.Scheduler;
8  import org.ourgrid.mygrid.scheduler.Task;
9  import org.ourgrid.mygrid.scheduler.Job;
10 import org.ourgrid.mygrid.scheduler.EBSchedulerFacade;
11 import org.ourgrid.mygrid.scheduler.GuMeXEntry;
12 import org.ourgrid.mygrid.replicaexecutor.EBReplicaExecutorFacade;
13 import org.ourgrid.specs.JobSpec;
14 import org.ourgrid.util.config.Configuration;
15 import org.ourgrid.util.config.MyGridConfiguration;

16 public aspect SchedulerObserver {

17      private static transient final Logger LOG =
18           Logger.getLogger("Scheduler");

19      JMConfiguration conf = new JMConfiguration();
20      Scheduler sch = new Scheduler("Scheduler");

21      SchedulerObserver(){
22          SimpleLayout layout = new SimpleLayout();
23          FileAppender appender = null;
24          try {
25             appender = new FileAppender(layout,"scheduler.log",false);
26          } catch(Exception e) {}
27          LOG.addAppender(appender);
28          conf.add(sch);
29      }

30      pointcut replicaFinished():
31           execution(* EBSchedulerFacade.replicaFinished(..)) ||
32           execution(* EBSchedulerFacade.replicaFailed(..)) ||
33           execution(* EBSchedulerFacade.replicaAborted(..)) ||
34           execution(* EBSchedulerFacade.replicaCanceled(..));

35      pointcut addJob(JobSpec jobSpec):
36           execution(* Scheduler.addJob(..)) &&
37           args(jobSpec);

38      pointcut hereIsGuM():
39           execution(* GridMachineConsumer.hereIsGuM(..));

40      pointcut wannaGuMs(String requirements,int numWantedGuMs):
41           call(* GridMachineProvider.wannaGuMs(..)) &&
42           args(requirements, numWantedGuMs,..);

43      pointcut executeReplica():
44           call(* EBReplicaExecutorFacade.executeReplica(..));
```

```
45      pointcut newGuMeXEntry(): execution(GuMeXEntry.new(..));

46      pointcut free():
47           execution(* GuMeXEntry.free(..)) ||
48           call(* GuMeXEntry.fastFree(..));

49      after(JobSpec jobSpec): addJob(jobSpec){
50           int maxReplicas = Integer.parseInt(
51                Configuration.getInstance().getProperty(
52                     MyGridConfiguration.PROP_MAX_REPLICAS ) );
53           int replicas = jobSpec.getTaskSpecs().size() * maxReplicas;
54           try{
55                sch.changeOutWeight(
56                     "addJob","replicasAvailable",replicas);
57           }catch(Exception e){ e.printStackTrace(); }
58           fire("Scheduler.addJob");
59      }

60      after(): replicaFinished(){ fire("Scheduler.replicaFinished"); }

61      after(): hereIsGuM(){ fire("Scheduler.hereIsGum"); }

62      after(String requirements, int requestedGums):
63           wannaGuMs(requirements, requestedGums){
64           try{
65                sch.changeOutWeight(
66                     "wannaGums","gumsRequested",requestedGums);
67           }catch(Exception e){ e.printStackTrace(); }
68           fire("Scheduler.wannaGums");
69      }

70      after(): executeReplica(){ fire("Scheduler.executeReplica"); }

71      after(): newGuMeXEntry(){ fire("Scheduler.newGumex"); }

72      after(): free(){ fire("Scheduler.free"); }

73      private void fire(String transitionName){
74           try{
75                LOG.info(conf);
76                LOG.info(transitionName);
77                conf = conf.execute(transitionName,conf);
78           }catch(Exception e){ LOG.error(e.getMessage(),e); }
79      }

80      pointcut uiInCall() :
81           call(* org.ourgrid.mygrid.scheduler.*.*(..))  &&
82           !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
83           !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
84           !(call(* Scheduler.addJob(JobSpec))) &&
85           !(call(* Scheduler.jobList())) &&
86           !(call(* Scheduler.cancelJob(int))) &&
87           !(call(* Scheduler.waitForJob(int))) &&
88           !(call(public * Job.*(..))) &&
89           !(call(public * Task.*(..))) &&
90           within(org.ourgrid.mygrid.ui.*) &&
```

```
91          !(within(org.ourgrid.mygrid.ui.*Test*)) &&
92          !(within(org.ourgrid.mygrid.ui.*Fake*));

93      pointcut executorOutCall() :
94          call(* org.ourgrid.mygrid.replicaexecutor.*.*(..)) &&
95          !(call( * EBReplicaExecutorFacade.executeReplica(..))) &&
96          !(call(* EBReplicaExecutorFacade.cancelReplicasOfJob(..))) &&
97          within(org.ourgrid.mygrid.scheduler.*) &&
98          (within(org.ourgrid.mygrid.scheduler.*Exception*)) &&
99          !(within(org.ourgrid.mygrid.scheduler.*Test*));

100     pointcut executorInCall() :
101         call(* org.ourgrid.mygrid.scheduler.*.*(..))  &&
102         !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
103         !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
104         !(call(* EBSchedulerFacade.replicaFinished(..))) &&
105         !(call(* EBSchedulerFacade.replicaFailed(..))) &&
106         !(call(* EBSchedulerFacade.replicaCanceled(..))) &&
107         !(call(* EBSchedulerFacade.replicaAborted(..))) &&
108         within(org.ourgrid.mygrid.replicaexecutor.*) &&
109         !(within(org.ourgrid.mygrid.replicaexecutor.*Fake*));

110     pointcut peerInCall() :
111         call(* org.ourgrid.mygrid.scheduler.*.*(..)) &&
112         !(call(* org.ourgrid.mygrid.scheduler.*Exception*.*(..))) &&
113         !(call(* org.ourgrid.mygrid.scheduler.*Test*.*(..))) &&
114         !(call(* GridMachineConsumer.hereIsGuM(..))) &&
115         !(call(* GridMachineConsumer.gumIsDead(..))) &&
116         within(org.ourgrid.gump.*);

117     pointcut peerOutCall() :
118         call(* org.ourgrid.gump.*.*(..)) &&
119         !(call(* GridMachineProvider.wannaGuMs(String,int,..))) &&
120         !(call(* GridMachineProvider.lostGuM(..))) &&
121         !(call(* GridMachineProvider.disposeOf(..))) &&
122         !(call(* GridMachineProvider.noMoreGuMs(..))) &&
123         within(org.ourgrid.mygrid.scheduler.*) &&
124         !(within(org.ourgrid.mygrid.scheduler.*Exception*)) &&
125         !(within(org.ourgrid.mygrid.scheduler.*Test*));

126     declare error: peerInCall() : "Integridade de comunicação\n" +
127         "GuMP -> Scheduler violada.";

128     declare error: peerOutCall() : "Integridade de comunicação\n" +
129         "Scheduler -> GuMP violada.";

130     declare error: uiInCall() : "Integridade de comunicação\n" +
131         "UI -> Scheduler violada.";

132     declare error: executorOutCall() : "Integridade de comunicação\n" +
133         "Scheduler -> ReplicaExecutor violada.";

134     declare error: executorInCall() : "Integridade de comunicação\n" +
135         "ReplicaExecutor -> Scheduler violada.";
136 }
```

Apêndice B

Gramática da Linguagem

Neste Apêndice, apresentamos a gramática para descrição dos modelos arquiteturais em CASTOR. O formato da especificação é o mesmo utilizado pela ferramenta JavaCC [Mic], o mais utilizado gerador de *parser* para Java. A sintaxe é muito próxima à EBNF (Extended Backus-Naur Form), sendo que os símbolos não terminais são representados por métodos.

B.1 Gramática de Descrição da Visão

Logo abaixo, é mostrada a especificação da gramática utilizada na descrição de uma visão.

ViewParser

```
options { JAVA_UNICODE_ESCAPE = true; }

PARSER_BEGIN(ViewParser)

public class ViewParser {}

PARSER_END(ViewParser)

SKIP :
{
  " " | "\t" | "\n" | "\r" | "\f"
}

TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
| < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
| < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
```

```
| < #OCTAL_LITERAL: "0" (["0"-"7"])* >
| < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
| < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
| < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

TOKEN :
{
< #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
| < #DIGIT:
      [
       "\u0030"-"\u0039",
```

```
        "\u0660"-"\u0669",
        "\u06f0"-"\u06f9",
        "\u0966"-"\u096f",
        "\u09e6"-"\u09ef",
        "\u0a66"-"\u0a6f",
        "\u0ae6"-"\u0aef",
        "\u0b66"-"\u0b6f",
        "\u0be7"-"\u0bef",
        "\u0c66"-"\u0c6f",
        "\u0ce6"-"\u0cef",
        "\u0d66"-"\u0d6f",
        "\u0e50"-"\u0e59",
        "\u0ed0"-"\u0ed9",
        "\u1040"-"\u1049"
      ]
  >
| <BOOLEAN: "boolean">
| <CHAR: "char">
| <BYTE: "byte">
| <SHORT: "short">
| <LONG: "long">
| <FLOAT: "float">
| <DOUBLE: "double">
| <DOT: ".">
| <SEMICOLON: ";">
| <IMPORT: "import">
| <STAR: "*">
| <INCOMPLETE: "incomplete">
| <COMPONENT: "component">
| <CONNECT: "connect">
| <EXCEPTION: "exception">
| <VIEW: "view">
| <CLASS: "class">
| <LBRACE: "{">
| <RBRACE: "}">
| <NOT: "!">
| <AND: "&&">
| <ONEORMORE: "+">
| <SUBPACKAGE: "..">
| <PORT: "port">
| <REQUIRES: "requires">
| <PROVIDES: "provides">
| <INTERNAL: "internal">
| <LPAREN: "(">
| <RPAREN: ")">
| <INT: "int">
| <PRIVATE: "private">
| <PROTECTED: "protected">
| <PUBLIC: "public">
| <STATIC: "static">
| <VOID: "void">
| <FINAL: "final">
| <SYNCHRONIZED: "synchronized">
| <ABSTRACT: "abstract">
| <NATIVE: "native">
| <STRICTFP: "strictfp">
```

```
| <THROWS: "throws">
| <IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
| <IDENTIFIERPATTERN: (<LETTER>|<DIGIT>)(<LETTER>|<DIGIT>|"*")*
                 |   ("*")(<LETTER>|<DIGIT>|"*")+
  >
}

void ViewModel() : {}
{
     (ImportDeclaration())*
     ViewStructure()
     <EOF>
}

void ImportDeclaration() : {}
{
  <IMPORT> Name() [ <DOT> <STAR> ] <SEMICOLON>
}

void ViewStructure() : {}
{
<VIEW> <IDENTIFIER> <LBRACE>
     (ConnectDeclaration())+
     (ExceptionDeclaration())*
<RBRACE>
}

void ConnectDeclaration() : {}
{ <CONNECT> <IDENTIFIER> <DOT> <IDENTIFIER> ","
     <IDENTIFIER> <DOT> <IDENTIFIER> <SEMICOLON>
}

void ExceptionDeclaration() : {}
{ <EXCEPTION> <IDENTIFIER> "->" <IDENTIFIER> <LBRACE>
     (
          (LOOKAHEAD(MethodPattern())MethodPattern()|TypePattern()) "->"
          (LOOKAHEAD(MethodPattern())MethodPattern()|TypePattern())
          <SEMICOLON>
     )+
  <RBRACE>
}

void OperationDeclaration() : {}
{ SignatureDeclaration() <LBRACE> (MethodPattern())+ <RBRACE> }

void SignatureDeclaration() : {}
{ <IDENTIFIER> <LPAREN> [<INT> <IDENTIFIER>] <RPAREN> }

void MethodPattern() : {}
{
  [ LOOKAHEAD(2) ModifiersPattern() ] TypePattern()
    [ LOOKAHEAD(TypePatternDot()) TypePatternDot() ]
    IdentifierPattern()
    <LPAREN> TypePatternList() <RPAREN>
    [ThrowsPattern()]
}
```

```
177 void ModifiersPattern() : {}
178 {
179   (LOOKAHEAD(2)
180    [ <NOT> ] ( <PUBLIC> | <PROTECTED> | <PRIVATE> | <STATIC>
181           | <ABSTRACT> | <FINAL> | <NATIVE> | <SYNCHRONIZED> |
182              <STRICTFP>))+
183 }

184 void BasicTypePattern() : {}
185 {
186   PrimitiveType() ("[" "]")* | <VOID>
187 | IdentifierPattern() [LOOKAHEAD(1) "+" ] (LOOKAHEAD(1) "[" "]")*
188      (LOOKAHEAD(1) ( <DOT> | ".." ) IdentifierPattern()
189           [LOOKAHEAD(1) "+" ] (LOOKAHEAD(1) "[" "]" )* )*
190 | "!" TypePattern() | "(" TypePattern() ")"
191 }

192 void TypePattern() : {}
193 {
194   LOOKAHEAD(BasicTypePattern() "&&")
195   BasicTypePattern() "&&" TypePattern() | BasicTypePattern()
196 }

197 void BasicTypePatternDot() : {}
198 {
199   (PrimitiveType() ("[" "]")* | <VOID>) "."
200 | ( LOOKAHEAD( IdentifierPattern() [ "+" ] ("[" "]")* (<DOT>|"..") )
201                IdentifierPattern() [ "+" ] ("[" "]")* (<DOT>|"..")  )+
202 | "!" TypePatternDot()
203 | "(" TypePattern() ")" (<DOT>|"..")
204 }

205 void TypePatternDot() : {}
206 {
207   LOOKAHEAD(BasicTypePattern() "&&")
208   BasicTypePattern() "&&" TypePatternDot() | BasicTypePatternDot()
209 }

210 void IdentifierPattern() : {}
211 {
212   (<IDENTIFIER>|<IDENTIFIERPATTERN>|<STAR>)
213 }

214 void TypePatternList() : {}
215 {
216   [ ( TypePattern() | ".." ) ( "," (TypePattern() | ".." ))* ]
217 }

218 void ThrowsPattern() : {}
219 { <THROWS> TypePattern() }


220 void Type() : {}
221 {
222   ( PrimitiveType() | Name() ) ( "[" "]" )*
```

```
}

void PrimitiveType() : {}
{
  <BOOLEAN> | <CHAR> | <BYTE> | <SHORT> | <INT> | <LONG> |
  <FLOAT> | <DOUBLE>
}

void Name() : {}
{
  // A lookahead of 2 is required due to "ImportDeclaration"
  <IDENTIFIER>
  ( LOOKAHEAD(2) <DOT> <IDENTIFIER>
  )*
}
```

B.2 Gramática de Descrição do Componente

Em relação à descrição do componente, vale lembrar que se a operação arquitetural possuir algum argumento, é obrigatório que haja um mapeamento em cada um dos métodos concretos que o compõem. Tal mapeamento é representado na gramática abaixo pelo símbolo não terminal `Block()` (linhas 183 e 192), porém sua especificação foi intencionalmente suprimida por ser demasiadamente extensa. O motivo é que `Block()` representa qualquer bloco válido de comandos Java. No entanto, esta especificação pode ser encontrada facilmente na Internet [Mic].

ComponentParser

```
1  options { JAVA_UNICODE_ESCAPE = true; }
2  PARSER_BEGIN(ComponentParser)
3  public class ComponentParser {}
4  PARSER_END(ComponentParser)
5  SKIP :
6  {
7    " " | "\t" | "\n" | "\r" | "\f"
8  }
9  TOKEN :
10 {
11   < INTEGER_LITERAL:
12         <DECIMAL_LITERAL> (["l","L"])?
13       | <HEX_LITERAL> (["l","L"])?
14       | <OCTAL_LITERAL> (["l","L"])?
15   >
16 | < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
17 | < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
18 | < #OCTAL_LITERAL: "0" (["0"-"7"])* >
19 | < FLOATING_POINT_LITERAL:
20       (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
21       | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
22       | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
23       | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
24   >
25 | < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
26 | < CHARACTER_LITERAL:
27       "'"
28       (   (~["'","\\","\n","\r"])
29         | ("\\"
30             ( ["n","t","b","r","f","\\","'","\""]
31             | ["0"-"7"] ( ["0"-"7"] )?
32             | ["0"-"3"] ["0"-"7"] ["0"-"7"]
33             )
34           )
35       )
```

```
      "'"
  >
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

TOKEN :
{
<#LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
| <#DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
| <DOT: ".">
| <BOOLEAN: "boolean">
```

```
| <CHAR: "char">
| <BYTE: "byte">
| <SHORT: "short">
| <LONG: "long">
| <FLOAT: "float">
| <DOUBLE: "double">
| <SEMICOLON: ";">
| <IMPORT: "import">
| <STAR: "*">
| <INCOMPLETE: "incomplete">
| <COMPONENT: "component">
| <CLASS: "class">
| <NEW: "new">
| <LBRACE: "{">
| <RBRACE: "}">
| <VOID: "void">
| <NOT: "!">
| <PORT: "port">
| <REQUIRES: "requires">
| <PROVIDES: "provides">
| <INTERNAL: "internal">
| <LPAREN: "(">
| <RPAREN: ")">
| <INT: "int">
| <PRIVATE: "private">
| <PROTECTED: "protected">
| <PUBLIC: "public">
| <STATIC: "static">
| <FINAL: "final">
| <SYNCHRONIZED: "synchronized">
| <ABSTRACT: "abstract">
| <NATIVE: "native">
| <STRICTFP: "strictfp">
| <THROWS: "throws">
| <IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|      <IDENTIFIERPATTERN: (<LETTER>|<DIGIT>)(<LETTER>|<DIGIT>|"*")*
          |   ("*")(<LETTER>|<DIGIT>|"*")+
  >
}

void parseComponentModel() : {}
{
     (ImportDeclaration())*
     ComponentStructure()
     <EOF>
}

void ImportDeclaration() : {}
{
  <IMPORT> Name() [ "." "*" ] <SEMICOLON>
}

void ComponentStructure() : {}
{
[<INCOMPLETE>] <COMPONENT> <IDENTIFIER> <LBRACE>
     ClassInComponentDeclaration()
```

```
144      (PortDeclaration())+
145      [InternalDeclaration()]
146      (OperationDeclaration())*
147 <RBRACE>
148 }

149 void ClassInComponentDeclaration() : {}
150 { <CLASS> TypePattern() <SEMICOLON> }

151 void PortDeclaration() : {}
152 {
153      <PORT> <IDENTIFIER> <LBRACE>
154           ( RequiresDeclaration() | ProvidesDeclaration() )
155           ( RequiresDeclaration() | ProvidesDeclaration() )*
156      <RBRACE>
157 }

158 void SignatureDeclaration() : {}
159 { <IDENTIFIER> <LPAREN> [<INT> <IDENTIFIER>] <RPAREN> }

160 void RequiresDeclaration() : {}
161 { <REQUIRES> SignatureDeclaration() <SEMICOLON>}

162 void ProvidesDeclaration() : {}
163 { <PROVIDES> SignatureDeclaration() <SEMICOLON>}

164 void InternalDeclaration() : {}
165 {
166   <INTERNAL> <LBRACE>
167        (SignatureDeclaration() <SEMICOLON>)+
168   <RBRACE>
169 }

170 void OperationDeclaration() : {}
171 {
172   SignatureDeclaration() <LBRACE>
173        (LOOKAHEAD(MethodPattern())MethodPattern()|ConstructorPattern())+
174   <RBRACE>
175 }

176 void MethodPattern() : {}
177 {
178   [ LOOKAHEAD(2) ModifiersPattern() ] TypePattern()
179     [ LOOKAHEAD(TypePatternDot()) TypePatternDot() ]
180     IdentifierPattern()
181     <LPAREN> TypePatternList() <RPAREN>
182     [ThrowsPattern()]
183     (Block()|<SEMICOLON>)
184 }

185 void ConstructorPattern() : {}
186 {
187   [ LOOKAHEAD(1) ModifiersPattern() ]
188     [ LOOKAHEAD(TypePatternDot()) TypePatternDot() ]
189     <NEW>
190     <LPAREN> TypePatternList() <RPAREN>
```

```
191     [ThrowsPattern()]
192     (Block()|<SEMICOLON>)
193 }

194 void ModifiersPattern() : {}
195 {
196   (LOOKAHEAD(2)
197    [ <NOT> ] ( <PUBLIC> | <PROTECTED> | <PRIVATE> | <STATIC>
198      | <ABSTRACT> | <FINAL> | <NATIVE> | <SYNCHRONIZED> | <STRICTFP>))+
199 }

200 void BasicTypePattern() : {}
201 {
202   PrimitiveType() ("[" "]")* | <VOID>
203 | IdentifierPattern() [LOOKAHEAD(1) "+" ] (LOOKAHEAD(1) "[" "]")*
204     (LOOKAHEAD(1) ( "." | ".." ) IdentifierPattern() [LOOKAHEAD(1) "+" ]
205     (LOOKAHEAD(1) "[" "]" )* )*
206 | <NOT> TypePattern() | <LPAREN> TypePattern() <RPAREN>
207 }

208 void TypePattern() : {}
209 {
210   LOOKAHEAD(BasicTypePattern() "&&") BasicTypePattern()
211        "&&" TypePattern() | BasicTypePattern()
212 }

213 void BasicTypePatternDot() : {}
214 {
215   (PrimitiveType() ("[" "]")* | <VOID>) "."
216 | ( LOOKAHEAD( IdentifierPattern() [ "+" ] ("[" "]")* ("."|"..") )
217               IdentifierPattern() [ "+" ] ("[" "]")* ("."|"..")  )+
218 | <NOT> TypePatternDot()
219 | <LPAREN> TypePattern() <RPAREN> ("."|"..")
220 }

221 void TypePatternDot() : {}
222 {
223   LOOKAHEAD(BasicTypePattern() "&&") BasicTypePattern()
224        "&&" TypePatternDot() | BasicTypePatternDot()
225 }

226 void IdentifierPattern() : {}
227 {
228   (<IDENTIFIER>|<IDENTIFIERPATTERN>|<STAR>)
229 }

230 void TypePatternList() : {}
231 {
232   [ ( Type() <IDENTIFIER>| ".." | <STAR>)
233     ( "," (Type() <IDENTIFIER>| ".." | <STAR> ))* ]
234 }

235 void ThrowsPattern() :{}
236 { <THROWS> TypePattern() }

237 void Type() : {}
```

```
238 { ( PrimitiveType() | Name() ) ( "[" "]" )* }

239 void PrimitiveType() : {}
240 {
241   <BOOLEAN> | <CHAR> | <BYTE> | <SHORT> | <INT> | <LONG> |
242   <FLOAT> | <DOUBLE>
243 }

244 void Name() :{}
245 {
246   <IDENTIFIER>
247   ( LOOKAHEAD(2) <DOT> <IDENTIFIER>
248   )*
249 }
```