UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

UMA TÉCNICA DE ANÁLISE DE CONFORMIDADE COMPORTAMENTAL PARA SISTEMAS DISTRIBUÍDOS

AMANDA SARAIVA BEZERRA

CAMPINA GRANDE – PB
NOVEMBRO DE 2008

Uma Técnica de Análise de Conformidade Comportamental para Sistemas Distribuídos

Amanda Saraiva Bezerra

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

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

B574t

2008 Bezerra, Amanda Saraiva

Uma técnica de análise de conformidade comportamental para sistemas distribuídos / Amanda Saraiva Bezerra. Campina Grande, 2008.

120 f.: il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientadores: Prof. Dr. Jorge César Abrantes de Figueiredo, Prof. Dr. Dalton Dario Serey Guerrero.

1. Detecção. 2. Propriedades Comportamentais. 3. Sistemas Distribuídos. 4. Conformidade. I. Titulo.

CDU - 004.052.32 (043)

"UMA TÉCNICA DE ANÁLISE DE CONFORMIDADE COMPORTAMENTAL PARA SISTEMAS DISTRIBUÍDOS "

AMANDA SARAIVA BEZERRA

DISSERTAÇÃO APROVADA EM 28.11.2008

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROF. MARCO AURÉLIO SPOHN, Ph.D
Examinador

PROFa. CHRISTINA VON FLACH GARCIA CHAVEZ, Dra.
Examinadora

CAMPINA GRANDE – PB

Resumo

Verificar o comportamento de Sistemas Distribuídos é difícil. Algumas técnicas, como Model Checking, tentam garantir a corretude comportamental de um sistema analisando todos os seus caminhos de execução. Entretanto, esta técnica é limitada para a análise de sistemas consideravelmente pequenos.

A técnica de *Design by Contract* (DbC) sugere a construção de um conjunto de asserções que devem ser verificadas em tempo de execução. Estas asserções são checagens instantâneas, podendo estas serem atendidas ou violadas. Como DbC não dá suporte à verificação de asserções temporais, cuja checagem deve ser realizada sobre vários instantes, então construímos uma técnica que possibilita a verificação comportamental de Sistemas Distribuídos, verificando se a execução apresentada está de acordo com os requisitos comportamentais definidos para o sistema.

Inicialmente é necessário definir os requisitos, que devem descrever o comportamento desejado para o sistema. Como estes requisitos devem ser checagens contínuas utilizamos a Lógica Temporal Linear (LTL) para descrever a evolução do comportamento ao longo do tempo. Usamos então LTL para definir o conjunto de propriedades comportamentais especificadas para o sistema distribuído implementado em Java.

Cada propriedade comportamental deve ser verificada sobre um conjunto específico de objetos do sistema, são os chamados pontos de interesse. Este pontos têm relação direta e devem obedecer a uma determinada propriedade comportamental. Durante a execução é realizada a captura de informações dos pontos de interesse a serem analisados, esta captura é feita usando a Programação Orientada a Aspectos (POA). A linguagem AspectJ, implementação de POA, possibilita a captura de informações sem a necessidade de modificar o código fonte do sistema a ser analisado.

Durante a execução do sistema é realizada a monitoração, além da captura da informação, sendo necessária uma ordenação parcial dos eventos provocados pelos processos que compõem o sistema distribuído. A ordenação é feita com base na relação de causa e efeito entre os eventos, assim usamos o conceito de Relógios Lógicos, que possibilita uma ordenação parcial dos eventos gerados durante a execução do sistema.

Após a captura e organização dos eventos, é realizada a verificação comportamental apresentada pelos eventos de acordo com o conjunto de propriedades definidas para tal. Se em algum momento desta verificação alguma propriedade for violada, então dizemos que foi encontrado um comportamento inesperado ou anômalo, de acordo com o especificado. Esta violação é apresentada para o desenvolvedor ou testador do sistema, que tomará as devidas providências para localizar e corrigir a violação detectada.

Como prova de conceito, implementamos a ferramenta *DistributedDesignMonitor* (DDM), realizando a monitoração de Sistemas Distribuídos implementados na linguagem Java. O DDM é apresentado e discutido através de experimentos realizados com a ferramenta, mostrando como fazer uso da técnica de monitoração.

Abstract

Verify Distributed Systems is very hard. Some techniques, Model Checking for example, trying to grant the behavioral correctness analyzing all the execution paths for these systems. However, this technique is limited to check small systems.

The Design by Contract technique (DbC) suggests makes a set of assertions that should be verified at runtime. These assertions are instant checks, which could be satisfied or violated. As DbC does not support the verification of temporal assertions, which check should be done over a lot of instant time for the execution, so we built a technique that verifies the Distributed Systems' behavior, verifying if the system execution is in accordance with the behavior requirements for it.

Initially is necessary the requirement definition, which should describe the desired system behavior. These requirements should be continuous checks, so we use the Linear Temporal Logic (LTL) to describe the evolution of the behavior over the time. The LTL is used to define set of behavioral properties specified for a Java Distributed System.

Each behavioral property should be verified over defined objects of the system, they are the points of interest. These points should obey to the behavioral property defined for them. During the execution the information catch of each point of interest is done by the Aspect Oriented Programming (AOP). The language AspectJ makes possible the capture of runtime information without intrusiveness.

During the system execution starts the monitoring, being necessary a partial order of the events of the distributed system processes. The ordering is done based on the relation of cause and effect between events, so we use the Logical Clocks to do this partial order.

After the capture and the organization of the events, the behavioral verification is done in accordance with the specified properties defined for this system. If some violation was detected during the system verification, so we found an inconsistent or unexpected behavior, in accordance with the specification. This violation is showed to the system developer or tester, which will correct the analyzed system.

In this work we implemented the DistributedDesignMonitor (DDM) tool to prove our concept, monitoring Distributed Systems implemented in Java language. O DDM is presented and discussed using experiments, showed how use the monitoring technique.

Agradecimentos

Inicialmente, toda minha gratidão a Deus, por me dar forças e perseverança para continuar a jornada de aprendizagem a cada nova etapa da minha vida.

Aos meus pais José Adailson e Lucinha, pelo esforço e confiança dedicados a mim, pelo apoio sempre fornecido mesmo sem esperar por retorno. Com a presença constante, mesmo que à distância, me sinto como em um porto seguro, amparada nos momentos de angústia e acarinhada com um sentimento incondicional de amor. Aos meus irmãos, Andrezza e Júnior, que sempre demonstraram um sentimento de carinho e esperança que mais essa etapa seria conquistada.

A minha família "extra-oficial", meus amigos mais próximos: Danilo, Danillo, Larissa, Netuh e Verla, que compartilharam comigo vários momentos marcantes, dos momentos de trabalhos aos de lazer, dos felizes aos tristes, dos sérios aos hilários, enfim, sempre deixando marcas de uma turma quase família mesmo, até com suas discussões e destemperos instantâneos, posteriormente agraciados com brincadeiras e alegrias.

Meus sinceros agradecimentos:

Aos professores Jorge Abrantes e Dalton Serey pela orientação, pelo incentivo e pelo constante esforço em construir o conceito de um mestrado junto comigo.

A Cássio e Fubica, que colaboraram com idéias e sugestões durante o andamento deste trabalho.

A Adauto, Gustavo, Ana Emília, Lauro, Ayla, que contribuíram fortemente para que este trabalho sempre seguisse evoluindo.

A Yuri e Paulo Rômulo, pessoas que dedicaram tempo e esforços na explicação de conceitos e idéias para a experimentação deste trabalho.

A eterna turma do(s) PET(s): Adauto, Ana Esther, Anderson, Clerton, Danilo, Diego, Edigley, Elíbia, Elloá, Giselle, Ianna, Ighor, Karlla, Larissa, Lorena, Luíza, Mariana, Michelly, Netuh, Prof. Joseana, Saulo, Tabira, Théo. Pessoas que em vários momentos foram fonte de alegria, incentivo e instantes simples, mas também inesquecíveis, que sempre contribuíram (mesmo inconscientemente) com a construção da estrada desta caminhada.

Ao pessoal do GMF: Adauto, Anderson, Ana Emília, Cássio, Diego, Emanuela, Francisco Neto, Gustavo, João, João Arthur, José Filho, Laísa, Lile, Makelli, Mirna, Netuh,

Pablo, Paulo Eduardo, Roberto, Vinicius, Wilkerson. Um grupo heterogêneo, que se mostra em constante evolução justamente pelas múltiplas visões e experiências de cada um dos seus integrantes, confundindo momentos de alegria com trabalho sério e de qualidade.

A Aninha (COPIN), sempre atenta e disponível para dirimir nossas dúvidas junto à coordenação do curso do Mestrado desta instituição.

Aos Professores, amigos e colegas desta Universidade, que foram os principais responsáveis pela minha formação acadêmica.

A CAPES e a FINEP pelo apoio financeiro.

Enfim, agradeço a todos aqueles, mesmo que não citados, mas que têm a mesma importância e que, de uma forma ou de outra, contribuíram na luta de contornar as barreiras desta jornada. Aos vários amigos feitos na Universidade, colegas de graduação, mestrado, funcionários da UFCG. A todos o meu Muito Obrigada!

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

Dentre outros fatores, a necessidade de processamento de uma quantidade cada vez maior de informações tem impulsionado o desenvolvimento de sistemas distribuídos. No entanto, algumas características inerentes a esses sistemas tornam a garantia da corretude mais complexa, tais como: não-determinismo, falha de comunicação, condições de corrida (*race conditions*), dentre outras [Reynolds et al., 2006; Tanenbaum and van Steen, 2007].

Tradicionalmente o desenvolvimento de sistemas computacionais requer a realização de testes para verificar a corretude dos mesmos. Os tipos de testes usados atualmente foram desenvolvidos para suprir a verificação de sistemas seqüenciais, tais como: teste de unidade, teste de aceitação, testes estruturais, testes funcionais, testes de integração, entre outros [Jorgensen, 2002; Massol and Husted, 2003]. O teste de um sistema seqüencial é feito avaliando-se a saída real com a saída esperada para uma determinada entrada. Sendo o programa determinístico, ele sempre irá repetir a mesma execução para uma determinada entrada, então a avaliação do comportamento do sistema durante a sua execução é considerada correta de acordo com a saída respondida pelo sistema.

Contudo, em sistemas distribuídos, quando um valor de entrada é submetido, inicia-se um processamento concorrente das tarefas a serem realizadas para computar a entrada recebida. Este processamento aleatório gera caminhos de execução diferentes mesmo para o processamento de uma mesma entrada de dados. Além da preocupação em retornar a resposta esperada para a respectiva entrada, existe a necessidade de verificar se durante o processamento o sistema apresentou algum estado inconsistente. Por exemplo, se dois processos acessaram concorrentemente um mesmo recurso do sistema, este acesso pode ser gerar um

estado inconsistente, inválido ou ainda incorreto dos elementos do sistema distribuído.

A tarefa de verificar se um sistema distribuído está retornando o resultado de acordo com a resposta esperada para uma determinada entrada é realizada usando os vários tipos de testes exitentes [Jorgensen, 2002]. Informalmente, dizemos que um sistema está processando corretamente "o que" foi especificado. No entanto, verificar "como" o sistema distribuído executou as tarefas não é viável de ser realizado usando somente os testes já conhecidos.

Uma forma de cobrir todos os possíveis caminhos de um sistema distribuído seria utilizando a técnica de checagem de modelos (*Model Checking*). *Model Checking* usa um modelo que descreve o sistema a ser verificado e gera todos os possíveis caminhos para a execução do mesmo. Entretanto, esta técnica é limitada a sistemas que têm um conjunto pequeno de variáveis, pois para cada variável acrescida ao sistema o número de caminhos possíveis cresce exponencialmente [Katoen, 1999]. No contexto de sistemas distribuídos temos muitas variáveis que trabalham concorrentemente com o objetivo de realizar uma determinada tarefa. Considerando a ação aleatória das muitas variáveis de um sistema distribuído, teremos uma quantidade exponencial de possibilidades diferentes para se executar um determinado comando. Então, verificar cada uma dessas possibilidades pode se tornar oneroso, dependendo do tamanho do sistema a ser verificado. Atualmente o poder computacional disponível não consegue realizar a verificação de sistemas com grande número de variáveis, indo de encontro ao problema da explosão de estados, que é a limitação do número de estados gerados por um modelo de sistema e a capacidade computacional de verificar todos estes estados [Katoen, 1999].

Dada a aleatoriedade de um sistema distribuído, não se tem uma forma confiável de avaliar o comportamento do mesmo durante a sua execução. Uma atividade informal realizada é a análise manual de rastros (*traces*) de execução. Essa atividade é feita por desenvolvedores ou testadores usando *traces* produzidos durante sua execução, sendo uma atividade enfadonha e passível de erros [Reynolds et al., 2006].

Uma técnica desenvolvida para a verificação comportamental de sistemas é *Design by Contratc* (DbC) [Mitchell et al., 2002]. A conceito foi introduzido por Bertrand Meyer com o propósito de trazer maior confiança sobre o sistema implementado. DbC realiza a verificação de asserções (verificações booleanas) em determinados pontos do sistema, durante a sua execução; são os chamados contratos de execução. Se alguma incoerência for detectada

durante a execução, esta informação é identificada. Entretanto, nenhuma implementação de DbC suporta a verificação do comportamento de um sistema considerando a análise de asserções temporais, que devem ser analisadas ao longo da execução e não somente em um instante específico.

Na tentativa de aproximar especificações formais e a implementação de sistemas, algumas técnicas foram propostas. Uma delas é a programação orientada a monitoração (*monitoring-oriented programming* - MOP) [Chen and Rosu, 2007]. Um monitor é uma parte do sistema que acompanha e cataloga as informações durante a execução de um sistema sob análise. A monitoração de sistemas compreende os atos de depurar, testar e analisar o desempenho de um sistema [Joyce et al., 1987]. Um monitor é um sistema que observa o comportamento de outro sistema e determina se o mesmo está consistente com uma especificação dada. Monitores podem ser usados para verificar o comportamento de um ponto específico do sistema de forma concorrente à execução do mesmo ou a posteriori, usando alguma forma de armazenamento do comportamento de maneira a possibilitar a análise após o término da execução do sistema [Peters, 1999].

Então, com inspiração na idéia de especificar asserções, assim como em DbC, para verificar o comportamento de um sistema e destacar quando da ocorrência de uma "quebra de contrato", informando o que provocou a quebra, foi implementada a técnica de DbC para sistemas distribuídos e concorrentes. A técnica baseia-se na construção automática de monitores a partir de uma especificação formal, possibilitando a notificação de violação da especificação durante sua execução [Chen et al., 2004; Chen and Rosu, 2007].

A técnica de monitoração de sistemas é utilizada para sistemas distribuídos. Um deles é apresentado em [Reynolds et al., 2006], que permite ao desenvolvedor de um sistema distribuído especificar propriedades que servem para avaliar o desempenho do mesmo, quanto a rapidez de processamento. O foco na análise comportamental quanto a interação entre os processos de sistemas distribuídos ainda não foi coberto pelas propostas de monitoração existentes.

Então, como garantir a coerência entre o comportamento do sistema distribuído durante a sua execução e a especificação comportamental do mesmo? Excluindo as técnicas tradicionais de testes que não contemplam a aleatoriedade desses sistemas [Jorgensen, 2002], e dada a inviabilidade computacional de *Model Checking* [Katoen, 1999], além de Design

by Contract ainda não implementado para sistemas distribuídos [Mitchell et al., 2002], precisamos de uma técnica que confronte o comportamento real corrente de um sistema distribuído com o comportamento esperado para o mesmo. Nessas condições, propomos

uma técnica que auxilie na análise da corretude comportamental de um sistema distribuído, informando quando da ocorrência de algum comportamento não esperado e aumentando a confiabilidade do sistema distribuído implementado.

Como forma de contornar as várias possibilidades de execução de um sistema distribuído, propomos o uso da técnica de Monitoração de Sistemas Distribuídos [Joyce et al., 1987; Mansouri-Samani and Sloman, 1993; Sen et al., 2004], avaliando a execução de um sistema a partir de sua especificação comportamental. Com o intuito de certificar a corretude do comportamento de um sistema distribuído, chamamos esta técnica de **análise de conformidade comportamental** para sistemas distribuídos.

A idéia geral é, de posse de algumas propriedades comportamentais que especificam o comportamento de um sistema distribuído, acompanhar a execução corrente do mesmo e verificar se o caminho de execução por ele traçado está de acordo com as regras comportamentais estabelecidas para o mesmo.

Para verificar a implementação correta de um sistema precisamos primeiramente definir quais proprieddes o mesmo deve obedecer durante a sua execução. Estas propriedades dizem respeito a como o sistema **não** deve se comportar para realizar um determinado conjunto de tarefas. Neste trabalho chamamos cada regra como uma **propriedade comportamental** do sistema. Uma propriedade comportamental deve então descrever o comportamento que o sistema distribuído monitorado não deve percorrer, mostrando assim que se o sistema não passou por caminhos inadequados então comportou-se corretamente quanto à sua especificação. Este formato de propriedade comportamental possibilitará o uso da análise da execução de um sistema distribuído, apontando para os pontos que apresentaram um comportamento incoerente com aquele especificado. Se algum comportamento inesperado for detectado é mostrado o que aconteceu de errado, qual elemento provocou a inconsistência e quando a mesma ocorreu.

As especificações comportamentais de um sistema normalmente estão descritas na forma de requisitos que o sistema deve implementar; estes requisitos são descritos pelo especifi-

cador do software a ser desenvolvido. Assim como os requisitos do sistema, o conjunto de propriedades comportamentais deve ser definido por um especialista do software a ser implementado, pois é ele que tem o conhecimento das regras que o sistema distribuído deve obedecer durante a sua execução.

A monitoração de um sistema requer o conhecimento de informações referentes ao sistema monitorado. Existem algumas técnicas para capturar a informação necessária, sendo uma das mais usadas a anotação no código (*code annotation*) para a monitoração do sistema [Mitchell et al., 2002; Reynolds et al., 2006]. No entanto, uma desvantagem de utilizarmos anotação no código é a necessidade de realizar várias modificações no código fonte, inserindo informações que não são de interesse do sistema monitorado e dificultando a manutenção do código. Como alternativa a essa forma sugerimos o uso da Programação Orientada a Aspectos, que possibilita a captura de informações em tempo de execução usando arquivos independentes para definir os pontos que devem ser observados e deles coletar informação relevante para a realização da análise de conformidade pelo monitor [Laddad, 2003].

Durante a captura dos eventos trocados entre os processos de um sistema distribuído é realizada a instrumentação, possibilitando uma ordenação parcial dos eventos. A instrumentação do código é realizada para termos um conhecimento consistente da ordem dos eventos durante a execução do sistema distribuído monitorado. Essa instrumentação é realizada com base no algoritmo de relógios lógicos com o intuito de chegarmos a uma ordenação parcial dos eventos de todo o sistema [Lamport, 1978]. A realização da instrumentação temporal durante a captura dos eventos possibilitará a construção de uma ordenação parcial dos eventos de forma prática posteriormente.

Depois de um conjunto de informações consistentes à execução global do sistema distribuído, dá-se início à análise do seu comportamento, utilizando o conceito de máquinas de estados, executando os eventos do sistema distribuído sob investigação, e apresentando a execução correta ou inadequada para o conjunto de eventos recebido.

Com o intuito de aplicarmos o conceito de monitoração sugerido, desenvolvemos a ferramenta *DistributedDesignMonitor* (*DDM*). Durante a implementação do *DDM* realizamos a monitoração de alguns sistemas distribuídos que tinham seu conjunto de especificações já definido e fizemos a transformação dessas especificações em propriedades comportamentais

para realizar a monitoração dos mesmo. Para cada exemplo de sistema distribuído, implementamos uma versão correta e outra com erros inseridos propositadamente. Passamos, então, a monitorar as duas versões de cada exemplo para encontrarmos inconsistências durante a execução de cada um. Os resultados mostraram que o *DDM* reportou inconsistências dos exemplos incorretos e acompanhou o comportamento correto dos exemplos que estavam implementados corretamente. Um ponto a se destacar: em um dos exemplos que achávamos que estava implementado corretamente, a monitoração do mesmo reportou uma inconsistência durante a sua execução, nos chamando a atenção para seu erro e possibilitando a correção do mesmo e a posterior monitoração de um sistema finalmente correto.

Nossos experimentos mostraram que podemos acompanhar a coerência de um sistema distribuído de acordo com os requisitos comportamentais especificados. Para isso utilizamos uma técnica que realiza a verificação do comportamento de um sistema distribuído, detectando um comportamento não desejado e reportando este comportamento para os desenvolvedores. Esta técnica pode ser vista como uma nova forma de detectar inconsistências no software implementado, não descartando as técnicas já usadas. A monitoração de sistemas distribuídos aqui sugerida tem o intuito de reforçar o conjunto de ferramentas que verificam a corretude dos mesmos.

O restante do documento está organizado da seguinte forma. No capítulo 2 temos um embasamento teórico dos assuntos relevantes para o entendimento do nosso trabalho. Inicialmente apresentamos os conceitos gerais de *Design by Contrat*, mostrando os pontos principais utilizados neste trabalho. O formato das propriedades comportamentais usadas para a realização da monitoração de sistemas é apresentado e é detalhada qual a idéia de funcionamento destas. Em seguida temos uma introdução geral à Programação Orientada a Aspectos, ajudando a entendermos como funciona a linguagem usada para a captura de informações em tempo de execução do sistema distribuído monitorado. Posteriormente, veremos como instrumentar um sistema distribuído para possibilitar uma ordenação parcial dos eventos por ele gerados usando relógios lógicos. O capítulo 3 descreve a idéia geral da nossa solução para a verificação da corretude comportamental de Sistemas Distribuídos, mostrando como utilizamos e unimos os conceitos apresentados na fundamentação para prover uma solução viável para a monitoração destes sistemas. O capítulo 4 descreve a arquitetura geral da ferramenta *DistributedDesignMonitor* (DDM) desenvolvida neste trabalho e como foi realizada

a implementação da mesma. No capítulo 5 mostramos os resultados obtidos a partir da monitoração de sistemas distribuídos e avaliamos a nossa ferramenta. O capítulo 6 apresenta alguns trabalhos relacionados com a nossa proposta e mostramos a necessidade de uso da mesma em detrimentos das demais. Finalmente, no capítulo 7 temos um apanhado das considerações finais e sugestão para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são abordados os assuntos utilizados durante o desenvolvimento da técnica de monitoração de sistemas distribuídos proposta neste trabalho.

2.1 *Design by Contract*

2.1.1 Introdução

O conceito inicial de *Design by Contract*(DbC) foi inicialmente sugerido por Bertrand Meyer com o intuito de prover uma prática de desenvolvimento de software que traga mais confiança sobre os novos sistemas a serem desenvolvidos do que os sistemas criados atualmente. Para tanto, a prática de DbC necessita da adição de asserções em código, para possibilitar a realização de uma certificação maior do funcionamento geral do sistema [Meyer, 1992; Mitchell et al., 2002].

A prática de DbC é utilizada em diversas atividades de processos de produção de software, dentre elas análise e projeto (*design*) de implementação, documentação, depuração e até gerenciamento do projeto em desenvolvimento.

Contudo, neste trabalho daremos enfoque ao DbC utilizado como técnica para auxiliar na verificação de implementação de sistemas. A idéia geral é fazer a verificação comportamental de um sistema de acordo com um conjunto de asserções que o mesmo deve obedecer durante a sua execução.

A prática de DbC durante o desenvolvimento de sistemas, considerando sistemas ori-

entados a objetos, exige que cada classe seja anotada com asserções (compostas de pré-condições, pós-condições e invariantes) de forma a montar um contrato especificado diretamente no corpo da classe. Este contrato estabelece as regras que a classe deve obedecer durante sua execução.

Esta idéia de contrato em tempo de execução nos leva a uma maneira de forçar a execução correta, a partir de algumas regras definidas pelo contrato do sistema em desenvolvimento e até mesmo em produção. Se a execução não obedecer a alguma dessas regras, os pontos de inconsistência são exibidos, caracterizando quebras de contrato, ou seja, código que contém falhas e que precisa ser corrigido.

A seguir veremos como especificar um contrato que pode ser utilizado para a verificação comportamental de sistemas em desenvolvimento.

2.1.2 Contratos

No mundo dos negócios, um contrato é um conjunto de regras que entidades devem obedecer para melhor estabelecer uma troca de serviços entre as entidades envolvidas. Na técnica DbC esta idéia é equivalente, cada contrato tem o intuito de limitar as ações, tanto do software em desenvolvimento quanto do uso do mesmo, através das regras definidas a serem obedecidas [Meyer, 1992].

Para isso, a prática de Dbc faz uso de algumas definições como pré-condições, pós-condições e invariantes, definições estas originárias de definições formais para construção de software.

O conceito de **pré-condição** está relacionado a uma tentativa de forçar uma condição de ocorrência do sistema, condição esta que deve ser obedecida no instante especificado. Como exemplo, podemos pensar na leitura de uma determinada variável de um sistema. Somente o acesso a esta variável pode não necessitar de uma pré-condição, pois a leitura não modificaria a mesma, porém quando da escrita e modificação da variável, algumas condições podem ser verificadas antes de possibilitar a alteração da mesma, tais como limite de tamanho, limite de valor, entre outros.

A **pós-condição**, como o próprio nome diz, é verificada após a execução de um determinado ítem do sistema, avaliando os resultados dessa execução, como por exemplo, a alteração de variáveis em um sistema.

Uma **invariante** em um sistema é uma propriedade que pode ser observada durante a execução do mesmo e que não deve mudar. A identificação de invariantes em um sistema possibilita a verificação da não violação de uma regra que o sistema deve obedecer. Uma invariante é uma propriedade que deve ter o mesmo valor sempre que esta propriedade for acessada [Mitchell et al., 2002].

Apesar de ter uma base formal, o uso de DbC não necessita de uma completa e total integração com o uso de formalidade, facilitando o uso da formalidade matemática fornecida por esta prática.

Para facilitar o aprendizado do paradigma DbC, Mitchell and McKim sugerem em [Mitchell et al., 2002] uma lista de princípios que devem ser seguidos, instruindo o programador a como fazer uso de DbC da forma mais simples possível.

Os seis princípios para introdução em *Design by Contract* são:

1. **Separe consultas de comandos** Consultas retornam um resultado, mas não modificam objetos. Comandos podem mudar um objeto sem precisar retornar um valor. Ao separar os tipos de acesso aos dados, podemos definir regras de execução dependendo do tipo de acesso.

2. **Separe consultas básicas de consultas derivadas** Consultas derivadas podem ser especificadas em termos de consultas básicas.

3. **Para cada consulta derivada, escreva uma pós-condição que especifica qual resultado será retornado em termos de uma ou mais consultas básicas** Assim, se conhecemos o valor de consultas básicas, então teremos o valor de consultas derivadas.

4. **Para cada comando, escreva uma pós-condição que especifica o valor de toda consulta básica** Em conjunto com o princípio anterior, o resultado é que podemos saber o efeito total de cada comando.

5. **Para toda consulta e comando, defina uma pré-condição adequada** Pré-condições forçam restrições quando um cliente chamar por uma consulta ou um comando.

6. **Escreva invariantes para definir propriedades imutáveis** Definindo uma propriedade que o sistema deve sempre obedecer facilita o entendimento através de uma abstração do modelo conceitual do sistema e as classes que o implementam.

Estes princípios possibilitam a construção de um contrato de maneira mais completa e que tente cobrir todos os pontos a serem verificados em um sistema.

2.1.3 Verificação de Contratos

A verificação do contrato é a confirmação de que tudo está de acordo com as regras ou asserções definidas previamente. Assim, é acompanhada a execução de um sistema e são verificadas se todas as pré-condições, pós-condições e invariantes foram obedecidas. Se alguma dessas variáveis não for obedecida, dizemos então que ocorreu uma violação do contrato.

Uma detecção de violação causada pela quebra de contrato de uma pré-condição identifica um erro provocado por quem invocou o método. Quando uma pós-condição é violada, dizemos que o erro foi do próprio método. Então, para cada entrada e saída de um método, se o mesmo estiver com seu contrato definido, será verificada cada chamada para este método, sabendo se o mesmo está recebendo valores dentro dos que ele espera receber e, em contrapartida, se toda a computação daquele método devolveu resultado que estava no intervalo permitido no seu contrato.

É importante destacar que as regras definidas que compõem o contrato não são regras referentes a casos especiais (conhecidas como exceções na orientação a objetos), mas devem sim verificar situações esperadas em um contexto específico do sistema, cuja situação não é válida. Por exemplo, se tivermos uma fila de espera que tenha limite de integrantes com valor dez, temos uma grande chance de requisição de adição de um décimo primeiro integrante para a mesma. Se esta fila é implementada usando uma coleção qualquer, então precisamos ter uma pós-condição que não irá permitir a inserção do décimo primeiro integrante naquela fila, mantendo assim a fila sempre com valor menor ou igual a dez.

2.2 Especificação de Sistemas Distribuídos

2.2.1 Introdução

Verificação de Sistemas é a uma técnica usada para garantir a concordância entre os requisitos e a implementação de um sistema. A verificação de um sistema depende de duas

informações, a saber: (i) o conjunto de fatos que devem ser verificados; e (ii) os aspectos relevantes relacionados ao fatos. Toda verificação implica na análise da corretude entre um sistema em detrimento do conjunto de fatos a serem verificados [Holzmann, 2003].

A verificação de sistemas implica na definição e especificação do sistema a ser analisado, de forma que possibilite a concordância entre dois modelos distintos de um mesmo sistema. A verificação de sistemas distribuídos, assim como qualquer outro tipo de sistema, depende da especificação dos requistos do sistema. No entanto, algumas características inerentes à tais sistemas diferenciam a especificação comportamental destes.

Sistemas Distribuídos têm características que dificultam atividades de análise e verificação, tais como não determinismo e paralelismo de execução de processos. Portanto, a especificação de sistemas distribuídos precisa considerar tais características [Mcdowell and Helmbold, 1989; Liang and Xu, 2005].

Um requisito importante em sistemas distribuídos é a temporalidade das ações. Por exemplo, sabe-se que um determinado conjunto de comandos deve ser executado por um sistema distribuído, mas não se sabe quanto tempo será necessário para que este conjunto de comandos seja executado. Algumas especificações em sistemas distribuídos verificam se um comando foi executado antes ou depois da execução de outro comando. Por isso, faz-se necessário a formulação de especificações de acordo com a temporalidade dos acontecimentos.

Um formalismo que possibilita a especificação de propriedades temporais é a Lógica Temporal Linear (*Linear Temporal Logic*). LTL é uma lógica temporal que foi definida como linguagem de especificação de propriedades de sistemas reativos [1].

2.2.2 Especificação Temporal

Para sistemas reativos, a corretude depende da execução do sistema, não somente considera a coerência entre entradas e saídas, como também *"como"* as operações que são realizadas durante a execução. Então, Pnueli em [Pnueli, 1977] definiu um formalismo para expressar propriedades características deste tipo de sistemas, propondo o conceito de lógica temporal. A Lógica Temporal foi definida como um sistema formal para especificação e descrição de sistemas reativos concorrentes, provendo um conjunto de operadores para descrever um

[1] Sistemas que têm a característica de contínua iteração de atividades [Katoen, 1999].

sistema em diferentes níveis de abstração [Lamport, 1983]. Esta definição de Pnueli foi muito bem aceita e é amplamente utilizada como técnica de especificação para expressar propriedades de execução de sistemas reativos em um alto nível de abstração [Katoen, 1999].

Lógica temporal é uma das formas de lógica modal que, além da lógica proposicional (ou lógica de predicado), controla operadores modais. Um exemplo típico em lógica temporal é o operador "em algum momento ϕ", que será verdadeira se a fórmula ϕ acontecer em algum momento do futuro. Esta lógica temporal é baseada em operadores que realizam a avaliação para uma determinada fórmula como `verdadeiro` ou `falso` em pontos específicos do tempo.

Sistemas distribuídos possuem características semelhantes aos sistemas reativos e concorrentes no que diz respeito à especificação de comportamento. Assim como em sistemas concorrentes, sistemas distribuídos muitas vezes precisam de verificação comportamental quanto a propriedades de vivacidade (*liveness*) e de segurança (*safety*). A primeira pretende verificar se o sistema continua ativo na execução de suas atividades, enquanto a segunda tenta verificar se o sistema não apresenta um estado indesejado, como por exemplo *deadlock* [Lamport, 1979].

Essas propriedades foram originalmente propostas por Lamport em [Lamport, 1977], de onde também surgiu os termos da ocorrência de *coisas desejáveis* ou *indesejáveis*. Em sistemas concorrentes e/ou distribuídos a sua especificação inclui, em regra geral, uma propriedade de segurança e uma de vivacidade. Além disso, qualquer propriedade temporal sobre o comportamento de um sistema pode ser expressa como uma combinação de uma propriedade de cada um dos dois tipos referidos [Lamport, 1977].

2.2.3 Lógica Temporal Linear

Linear Temporal Logic (LTL) é uma lógica temporal modal que vem sendo comumente utilizada para especificar comportamentos de sistemas reativos. Semelhantemente, o funcionamento de sistemas concorrentes e distribuídos é relacionado à ordem dos eventos destes sistemas, ou seja, o comportamento do sistema durante o tempo de execução. LTL provê um formalismo de especificação de propriedades de seqüências de execução de um sistema de forma a expressar suas propriedades [Katoen, 1999].

Nas próximas seções vamos entender como descrever especificações e o que interpretar

das mesmas, através da sintaxe e semântica de LTL.

2.2.3.1 Sintaxe

Os elementos básicos da sintaxe LTL são proposições atômicas. Uma proposição atômica *p* é a sentença que expressa alguma informação sobre algum estado do sistema. Por exemplo, podemos descrever proposições do tipo "*x* é igual a zero ", "o recurso *r* está alocado ". A definição formal da sintaxe LTL é definida a seguir:

Definição 2.1 (Sintaxe de LTL [Katoen, 1999]) *Seja* PA *um conjunto de proposições atômicas, então:*

1. *Cada proposição atômica* p *é uma fórmula LTL;*
2. *Se* ϕ *é uma fórmula, então* $\neg\phi$ *é uma fórmula;*
3. *Se* ϕ *e* ψ *são fórmulas, então* $\phi \vee \psi$ *é uma fórmula;*
4. *Se* ϕ *é uma fórmula, então* $X\phi$ *é uma fórmula (lê-se "próximo fi");*
5. *Se* ϕ *e* ψ *são fórmulas, então* $\phi \cup \psi$ *é uma fórmula (lê-se "fi até que psi");*
6. *nada mais é uma fórmula.*

Para negação e disjunção temos os operadores $\neg$ and $\vee$, respectivamente. Os operadores booleanos $\wedge$ (conjunção), $\Rightarrow$ (implicação) e $\Leftrightarrow$ (equivalência), assim como a definição de *true* e *false*, são usadas para expressar semântica em formulas menores, e são derivadas dos operadores de negação e disjunção, como podemos ver a seguir:

- $\phi \wedge \psi \equiv \neg(\neg\ \phi \vee \neg\ \psi)$
- $\phi \Rightarrow \psi \equiv \neg\ \phi \vee \psi$
- $\phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
- $true \equiv \phi \vee \neg\phi$
- $false \equiv \neg true$

Os operadores temporais G (lê-se “sempre"ou “globalmente") e F (lê-se “futuramente") são definidos a partir dos operadores definidos anteriormente:

$$G\phi \equiv \neg F \neg \phi$$

$$F\phi \equiv true \cup \phi$$

Para melhor entendimento de como descrever um comportamento de um determinado sistema, considere o Exemplo 2.1 descrito a seguir.

Exemplo 2.1 *Considere a seguinte propriedade para um determinado sistema distribuído: “quando um processo entrar numa região crítica, o número de processos nessa região crítica deve ser exatamente um”. Assim temos que os processos nunca estão simultaneamente na região crítica. Então, seja os processos $p1$ e $p2$, logo:*

$$G\neg(p1 \wedge p2)$$

2.2.3.2 Semântica

A sintaxe nos fornece a maneira correta para a construção das fórmulas LTL, mas não dá uma interpretação aos operadores. Formalmente, uma fórmula LTL é interpretada como uma seqüência infinita de estados. Intuitivamente, temos que:

- $X\phi$ significa que a fórmula ϕ é válida no próximo estado;
- $F\phi$ significa que a fórmula será válida em algum momento futuro;
- $G\phi$ significa que a fórmula é sempre válida;
- $\phi \cup \psi$ expressa que ϕ é válida ao longo de toda uma seqüência de estados consecutivos até a ocorrência de ψ.

As fórmulas LTL também podem ser representadas através de uma espécie de grafo de acessibilidade, denominado Estrutura de Kripke [Katoen, 1999].

Definição 2.2 (Estrutura de Kripke) *Uma estrutura de Kripke $\mathcal{M}$ é uma tupla $(S, I, R, Label)$, onde:*

1. S *é um conjunto finito de estados;*
2. $I \subseteq S$ *é um conjunto de estados iniciais;*
3. $R \subseteq S \times S$ *é uma relação de transição satisfazendo* $\forall s \in S.(\exists s' \in S.(s, s') \in R)$*;*
4. $Label : S \rightarrow 2^{PA}$*, associando a cada estado* s *de* S*, proposições atômicas* $Label(s)$ *que são válidas em* s*.*

Uma estrutura de Kripke é uma máquina de estados finita que representa o comportamento de um sistema. Cada estado do sistema é rotulado com proposições atômicas que são verdadeiras no estado correspondente.

Exemplo 2.2 *Considere um sistema tolerante a falhas, ilustrado na Figura 2.1, formado por três processadores que geram resultados para um quarto que é capaz de eleger majoritariamente qual resposta utilizar [Katoen, 1999].*

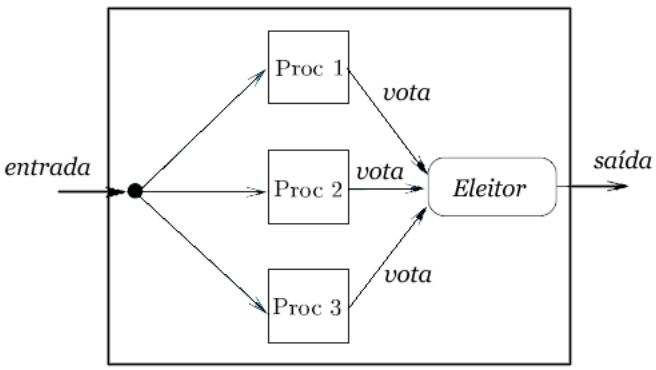

Figura 2.1: Um sistema redundante de eleição por processos.

Inicialmente todos os componentes estão operacionais, porém sujeitos a falhas durante uma execução. Assim, o estado $S_{i,j}$ modela que i processadores ($0 \leq i < 4$) e j eleitores majoritários ($0 \leq j \leq 1$) estão operacionais. Quando um componente falha ele pode ser reparado e voltar a funcionar. Considere que apenas um componente pode ser reparado por vez. Quando o eleitor falha, todo o sistema pára de funcionar. O conjunto de proposições atômicas deste problema é $AP = \{up_i | 0 \leq i < 4\} \cup \{down\}$. A proposição $\{up_0$ denota que apenas o processador eleitor está operacional, $\{up_1$ denota que além do processador eleitor,

um outro também está operacional e assim por diante. A proposição $down$ denota que todo o sistema não está funcionando.

Uma estrutura de Kripke para este sistema tem os seguintes componentes:

- $S = \{S_{i,1} | 0 \leq i < 4\} \cup \{S_{0,0}\}$;
- $I = \{S_{3,1}\}$;
- $R = \{(S_{i,1}, S_{0,0} | 0 \leq i < 4)\} \cup \{(S_{0,0}, S_{3,1})\} \cup \{(S_{i,1}, S_{i,1} | 0 \leq i < 4)\} \cup \{(S_{i,1}, S_{i+1,1} | 0 \leq i < 3)\} \cup \{(S_{i+1,1}, S_{i,1} | 0 \leq i < 3)\}$;
- $Label(S_{0,0}) = \{down\}$ e $Label(S_{i,1}) = \{up_i\}$, para $0 \leq i < 4$.

Graficamente, a estrutura de Kripke para esse problema é ilustrado pela Figura 2.2.

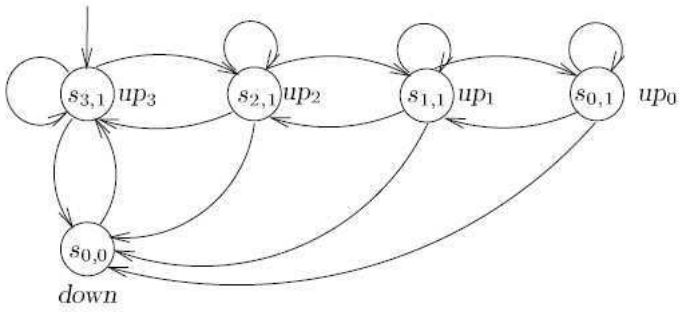

Figura 2.2: Estrutura de Kripke do Exemplo 2.2.

Para se definir formalmente a semântica de LTL, o conceito de caminho também deve ser formalizado.

Definição 2.3 (Caminho) *Um caminho em* $\mathcal{M}$ *é uma seqüência infinita de estados* $s_0,\ s_1,\ s_2,\ \ldots$ *tal que* $s_0 \in I$ *e* $(s_i, s_{i+1}) \in R$ *para todo* $i \geq 0$.

Portanto, um caminho é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. $\sigma[i]$ denota o $(i+1)$-ésimo estado de σ e σ^1 representa o sufixo de σ obtido pela remoção do(s) i-primeiro(s) estados de σ. A função $Caminhos(s)$ determina todos os possíveis caminhos da estrutura $\mathcal{M}$ que se iniciam no estado s.

Uma vez definida a estrutura na qual uma fórmula LTL é interpretada, sua semântica pode ser então formalmente definida através da relação de satisfação, denotada por $\models$, e definida formalmente a seguir.

Definição 2.4 (Semântica de LTL) *Sejam $p \in PA$ uma proposição atômica, σ caminho infinito e ϕ, ψ fórmulas LTL, a relação de satisfação, denotada por $\models$, é definida por:*

- $\sigma \models p \Leftrightarrow p \in Label(\sigma[0])$
- $\sigma \models \neg\phi \Leftrightarrow not(\sigma \models \phi)$
- $\sigma \models \phi \wedge \psi \Leftrightarrow (\sigma \models \phi)\ e\ (\sigma \models \psi)$
- $\sigma \models X\phi \Leftrightarrow \sigma^1 \models \phi$
- $\sigma \models \phi \cup \psi \Leftrightarrow \exists j \geq 0, (\sigma^j \models \psi\ e\ (\forall 0 \leq k < j, \sigma^k \models \phi))$

2.2.4 Autômatos de Büchi

Um autômato de Büchi (LBA) [Büchi, 1962] é uma extensão de um autômato de estados finito para entradas infinitas. Os autômatos finitos podem ser vistos como reconhecedores de palavras. As palavras são definidas como seqüências finitas de elementos de um alfabeto Σ. Denotamos por Σ^* como o conjunto de todas as palavras finitas formadas a partir do alfabeto Σ. Já os autômatos de Büchi são reconhecedores de palavras infinitas. O conjunto das palavras infinitas formadas a partir de elementos de Σ é denotado por Σ^ω.

Definição 2.5 (Autômato de Büchi) *Um autômato de Büchi A é uma 6-tupla $(\Sigma, S, S^0, \rho, F, \ell)$, onde:*

1. *Σ é um conjunto finito e não-vazio de símbolos;*
2. *S é um conjunto finito e não-vazio de estados;*
3. *$S^0 \subseteq S$ é um conjunto não-vazio de estados iniciais;*
4. *$\rho : S \rightarrow 2^S$ uma função de transição;*
5. *$F \subseteq S$ conjunto de estados de aceitação;*

6. $\ell : S \rightarrow \Sigma$ *função de rotulação.*

$\rho(s)$ é um conjunto de estados do autômato A que podem ser alcançados a partir de s, ou seja, $s \rightarrow s^{'}$ se e somente se $s^{'} \in \rho(s)$.

Definição 2.6 (Execução de um autômato de Büchi rotulado) *Seja o autômato de Büchi rotulado A, temos que uma execução π é uma seqüência de estados $\pi = s_0 s_1 ...$ tal que $s_0 \in S^0$ e $s_i \rightarrow s_{i+1}$ para todo $i \geq 0$. Seja $\lim(\pi)$ o conjunto de estados que ocorrem em σ freqüentemente infinita vezes. Uma execução π é chamada aceita, se e somente se, $\lim(\pi) \cap F \neq \varnothing$. Uma palavra $\omega = a_0 a_1 ... \in \Sigma^{\omega}$ é aceita se existe uma execução aceita $\pi = s_0 s_1 ... s_n$tal que $\ell(s_i = a_i)$ para todo $i \geq 0$.*

A idéia do autômato de Büchi é que uma palavra seja aceita, se e somente se, ao ser processada o autômato passa infinitas vezes por algum estado de aceitação. Observe que como o conjunto de estados é finito, então em qualquer seqüência infinita de estados deve haver pelo menos um estado que se repetirá infinitamente.

Formalmente, definimos $lim(\pi)$ como o conjunto dos estados que se repete infitamente em uma execução π do autômato. Dizemos que a execução π é aceitável se e somente se $lim(\pi) \cap F \neq \emptyset$.

Dizemos que uma palavra $\omega = a_0 a_1 ... \in \Sigma^{\omega}$ é reconhecida por um autômato de Büchi A se existe alguma execução aceitável do autômato $s_0, s_1, ...$ tal que $l(s_i) = a_i$ para todo $i \geq 0$.

Como as seqüências são infinitas não podemos definir aceitação em função de um estado final. De acordo com o critério de aceitação de Büchi, uma execução é aceita quando alguns estados de aceitação são freqüentemente visitados infinitas vezes. A linguagem aceita pelo autômato de Büchi A é denotada da seguinte forma:

$\mathcal{L}_{\omega}(A) = \{w \in \Sigma \parallel w$ aceito por $A\}$, se F é vazio, então $\mathcal{L}_{\omega}(A)$ também é vazio.

Exemplo 2.3 *Considere o seguinte autômato de Büchi A:*

1. $\Sigma = a, b$;
2. $S = q0, q1$;
3. $S^0 = q0$;

4. $\rho : (q0, a, q0), (q0, b, q1), (q1, a, q0), (q1, b, q0)$;

5. $F = q1$;

A representação gráfica para este autômato descrito acima é ilustrado na Figura 2.3.

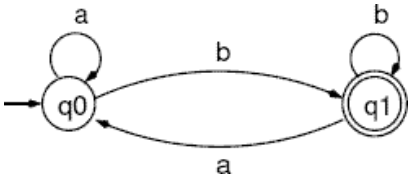

Figura 2.3: Representação gráfica do autômato do Exemplo 2.3.

O autômato A aceita a seguinte linguagem de palavras infinitas:

$$\mathcal{L}_\omega(A) = (a^*b^*)^\omega$$

2.2.5 Transformando fórmulas LTL em autômatos de Büchi

Temos que para cada fórmula LTL (em proposições atômica AP) existe um autômato de Büchi correspondente.

Teorema 2.1 *Para uma fórmula LTL ψ existe um autômato de Büchi A que pode ser construído com o alfabeto $\Sigma = 2^{AP}$ tal que $\mathcal{L}_w(A)$ é igual as sucessões de conjunto de proposições atômicas que satisfazem Σ.*

O algoritmo que trata da associação de uma fórmula LTL com um autômato de Büchi foi definido por Wolper, Vardi e Sistla (1983) e consiste nos passos ilustrados na Figura 2.4 [Katoen, 1999]. O passo principal nesta transformação é a construção do grafo a partir da fórmula na forma-normal.

Figura 2.4: Visão semântica do algoritmo de transformação de fórmulas LTL para autômatos de Büchi.

Fórmulas na forma-normal

O primeiro passo executado pelo algoritmo, ilustrado na Figura 2.4, dada uma fórmula LTL ϕ converter para *fórmula normal* equivalente. Para isto, consideramos inicialmente que ϕ não contém F e G (que podem ser transformados utilizando as seguintes definições: $F\psi \equiv true \cup \psi$ e $G\psi \equiv \neg F \neg \psi$), e todas as negações $\neg\phi$ são adjacentes as proposições atômicas. Considere também que $true$ e $false$ são substituídos por suas definições. A fim de permitir transformar a negação da fórmula até (*until*), um operador temporal auxiliar $\overline{\cup}$ é introduzido e definido como:

$$(\neg\phi)\ \overline{\cup}\ (\neg\psi) \equiv \neg(\phi \cup \psi).$$

Definição 2.7 (Fórmula LTL na forma-normal) *Para $p \in AP$, uma proposição atômica, o conjunto de fórmulas LTL na forma-normal é definido por:*

$$\phi := p \mid \neg p \mid \phi \vee \psi \mid \phi \wedge \psi \mid X\phi \mid \phi \cup \phi \mid \phi\ \overline{\cup}\ \phi.$$

As seguintes equações são usadas na transformação da fórmula LTL ϕ na forma-normal:

- $\neg(\phi \vee \psi) \equiv (\neg\phi) \wedge (\neg\psi)$
- $\neg(\phi \wedge \psi) \equiv (\neg\phi) \vee (\neg\psi)$
- $\neg X\phi \equiv X(\neg\phi)$
- $\neg(\phi \cup \psi) \equiv (\neg\phi)\ \overline{\cup}\ (\neg\psi)$
- $\neg(\phi\ \overline{\cup}\ \psi) \equiv (\neg\phi) \cup (\neg\psi)$

Construção do grafo

A partir das fórmulas na forma-normal obtemos o grafo com a aplicação do algoritmo *CreateGraph*, descrito em [Katoen, 1999] ilustrado na Figura 2.5. A saída da apliucação do algoritmo *CreateGraph* é o grafo $\mathcal{G}_\phi = (V, E)$, onde V é o conjunto de vértices e E o conjunto de arestas, tal que $E \subseteq V \times V$.

```
function CreateGraph (φ : Formula): set of Vertex;
(* pre-condition: φ is a PLTL-formula in normal form *)
begin var S : sequence of Vertex,
          Z : set of Vertex, (* already explored vertices *)
          v, w1, w2 : Vertex;
      S, Z := ⟨({ init }, { φ }, ∅, ∅)⟩, ∅;
      do S ≠ ⟨⟩ ⟶ (* let S = ⟨v⟩⌢S' *)
        if N(v) = ∅ ⟶ (* all proof obligations of v have been checked *)
          if (∃ w ∈ Z. Sc(v) = Sc(w) ∧ O(v) = O(w)) ⟶
            P(w), S := P(w) ∪ P(v), S'  (* w is a copy of v *)
          [] ¬ (∃ w ∈ Z. ...) ⟶
            S, Z := ⟨({ v }, Sc(v), ∅, ∅)⟩⌢S', Z ∪ { v };
          fi
        [] N(v) ≠ ∅ ⟶ (* some proof obligations of v are left *)
          let ψ in N(v);
          N(v) := N(v) \ { ψ };
          if ψ ∈ AP ∨ (¬ψ) ∈ AP ⟶
            if (¬ψ) ∈ O(v) ⟶ S := S' (* discard v *)
            [] ¬ψ ∉ O(v) ⟶ skip
            fi
          [] ψ = (ψ1 ∧ ψ2) ⟶ N(v) := N(v) ∪ ({ ψ1, ψ2 } \ O(v))
          [] ψ = X φ ⟶ Sc(v) := Sc(v) ∪ { φ }
          [] ψ ∈ { ψ1 U ψ2, ψ1 R ψ2, ψ1 ∨ ψ2 } ⟶ (* split v *)
            w1, w2 := v, v;
            N(w1) := N(w1) ∪ (F1(ψ) \ O(w1));
            N(w2) := N(w2) ∪ (F2(ψ) \ O(w2));
            O(w1), O(w2) := O(w1) ∪ { ψ }, O(w2) ∪ { ψ };
            S := ⟨w1⟩⌢(⟨w2⟩⌢S')
          fi
          O(v) := O(v) ∪ { ψ }
        fi
      od;
      return Z;
(* post-condition: Z is the set of vertices of the graph G_φ *)
(* where the initial vertices are vertices in Z with init ∈ P *)
(* and the edges are given by the P-components of vertices in Z *)
end
```

Figura 2.5: Algoritmo de construção de um grafo a partir de uma fórmula LTL ϕ.

Exemplo 2.4 *O grafo resultante da aplicação do algoritmo CreateGraph para a fórmula* $\phi = p \cup q$ *é ilustrado na Figura 2.6 [Katoen, 1999].*

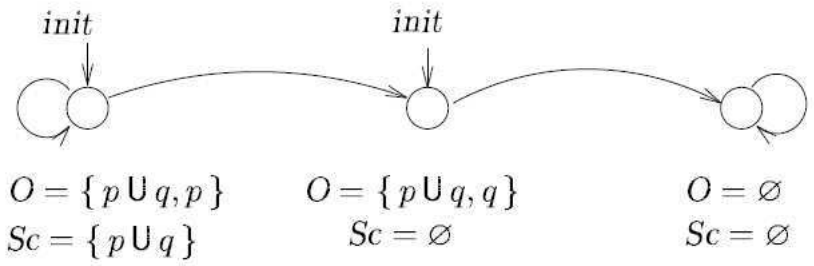

Figura 2.6: Resultado da aplicação do algoritmo na fórmula $p \cup q$.

Transformando grafo em autômato de Büchi generalizado

Definição 2.8 (Autômato de Büchi generalizado) *Um autômato de Büchi generalizado (GLBA)* A *é uma tupla* $(\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ *onde todos os componentes são os mesmos para o LBA, exceto* $\mathcal{F}$ *que é o conjunto dos conjuntos de aceitação* $\{F_1, \ldots, F_k\}$ *para* $k \geq 0$ *com* $F_i \subseteq S$*, isto é* $\mathcal{F} \subseteq 2^S$.

A conversão uma fórmula LTL ϕ na forma-normal para o GLBA $A = (\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ é definida por:

- $\Sigma = 2^{AP}$
- $S = CreateGraph(\phi)$
- $S^0 = \{s \in S | init \in P(s)\}$
- $s \longrightarrow s'$ sss $s \in P(s')$ e $s \neq init$
- $\mathcal{F} = \{\{s \in S | \phi_1 \cup \phi_2 \notin O(s) \vee \phi_2 \in O(s)\} | \phi_1 \cup \phi_2 \in Sub(\phi)\}$
- $\ell(s) = \{\mathcal{P} \subseteq AP | Pos(s) \subseteq \mathcal{P} \wedge \mathcal{P} \cap Neg(s) = \oslash$

$Sub(\phi)$ denota o conjunto de sub-fórmulas de ϕ. $Pos(s) = O(s) \cap AP$, as proposições atômicas válidas em s , e $Neg(s) = \{p \in AP | \neg p \in O(s)\}$, o conjunto das proposições atômicas negativas que são válidas em s.

Exemplo 2.5 *O GLBA que correspondente ao grafo do Exemplo 2.4 é ilustrado na Figura 2.7 [Katoen, 1999].*

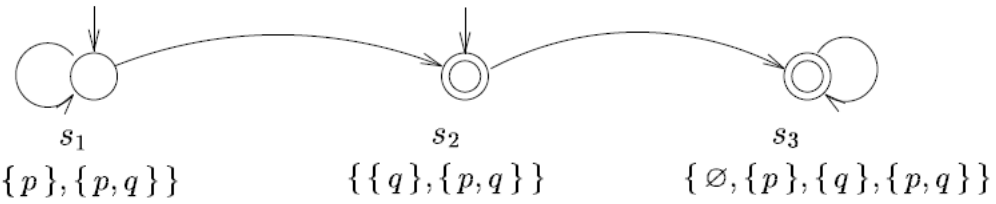

Figura 2.7: GLBA para o grafo do Exemplo 2.4.

Transformando autômato de Büchi generalizado em um autômato de Büchi

Definição 2.9 (GLBA para um LBA) *Seja* $A = (\Sigma, S, S^0, \rho, \mathcal{F}, \ell)$ *um autômato de Büchi generalizado (GLBA) com* $\mathcal{F} = \{F_1, \ldots, F_k\}$. *O autômato de Büchi equivalente* $A' = (\Sigma, S', S^0{}', \rho', \mathcal{F}', \ell')$ *tal que* $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$ *é obtido da seguinte maneira:*

- $S' = S \times \{i | 0 < i \leq k\}$
- $S^0{}' = S^0 \times \{i\}$ *para algum* $0 < i \leq k$
- $(s, i) \longrightarrow' (s', i)$ *sss* $s \longrightarrow s'$ *e* $s \notin F_i$
- $(s, i) \longrightarrow' (s', (i \bmod k) + 1)$ *sss* $s \longrightarrow s'$ *e* $s \in F_i$
- $F' = F_i \times \{i\}$ *para algum* $0 < i \leq k$
- $l'(s, i) = l(s)$.

Exemplo 2.6 *Considere o seguinte autômato de Büchi generalizado (Figura 2.8 [Katoen, 1999]):*

Este autômato contém dois conjuntos de aceitação $F_1 = \{s_1\}$ *e* $F_2 = \{s_2\}$. *Os estados que correspondem ao autômato de Büchi simples são*

$$\{s_0, s_1, s_2\} \times \{1, 2\}$$

Algumas das transições, por exemplo, são:

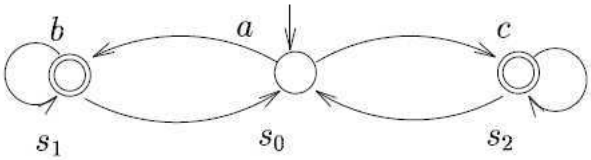

Figura 2.8: GLBA para o grafo do Exemplo 2.6.

- $(s_0, 1) \longrightarrow (s_1, 1)$ *desde que* $s_0 \longrightarrow s_1$ *e* $s_0 \notin F_1$
- $(s_0, 1) \longrightarrow (s_2, 1)$ *desde que* $s_0 \longrightarrow s_2$ *e* $s_0 \notin F_1$
- $(s_1, 1) \longrightarrow (s_0, 2)$ *desde que* $s_1 \longrightarrow s_0$ *e* $s_1 \in F_1$
- $(s_1, 2) \longrightarrow (s_1, 2)$ *desde que* $s_1 \longrightarrow s_1$ *e* $s_1 \notin F_2$
- $(s_2, 2) \longrightarrow (s_2, 1)$ *desde que* $s_2 \longrightarrow s_2$ *e* $s_2 \in F_2$

O autômato de Büchi simples equivalente ao grafo da Figura 2.8 é ilustrado na Figura 2.9 [Katoen, 1999].

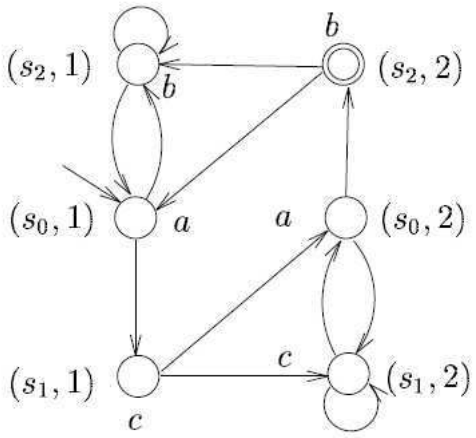

Figura 2.9: LBA para o grafo da Figura 2.8.

2.3 Programação Orientada a Aspectos

2.3.1 Introdução

A Programação Orientada a Aspectos (POA) [Kiczales, 2005] foi proposta com o objetivo de facilitar a modularização dos interesses transversais de um sistema. Interesses transversais são funcionalidades que precisam atingir várias partes do sistema de forma independente da implementação das suas funcionalidades. A POA complementa o uso da Programação Orientada a Objetos (POO) de maneira a contemplar os módulos de um sistema que atravessam toda a sua implementação.

Em POO cada interesse de um sistema é modularizado em objetos, que contém as informações referentes a este e concentra toda esta informação em um único local. Na POA há a inserção de um novo mecanismo para abstração e composição da informação de interesse, facilitando a modularização de interesses transversais. Este mecanismos é chamado de aspecto.

2.3.2 Conceitos básicos

Na tentativa de separar as funcionalidades do sistema dos interesses transversais devemos focar em cada interesse individualmente, reduzindo a complexidade global do projeto (*design*) e implementação do sistema [Laddad, 2003]. Esta separação tem o propósito de decompor os requisitos, sendo identificados em preocupações individuais dentro do sistema.

A Figura 2.10 [Laddad, 2003] mostra como os requisitos de um sistema pode ser decomposto em interesses através da analogia da luz através de um prisma. Enquanto um requisito aparece como uma única preocupação, após passar por um mecanismo identificador de interesses podemos ver cada um deles separadamente.

O processo de desenvolvimento de um sistema usando POA é semelhante ao desenvolvimento baseado em qualquer outra metodologia: identificar os interesses, implementá-los e finalmente juntá-los para compor o sistema. Em [Laddad, 2003] temos os três passos sugeridos pela comunidade para o desenvolvimento utilizando a POA:

1. *Decomposição de Aspectos.* Esta é a etapa de decomposição dos interesses, separando os requisitos principais dos interesses transversais do sistema. Por exemplo, na

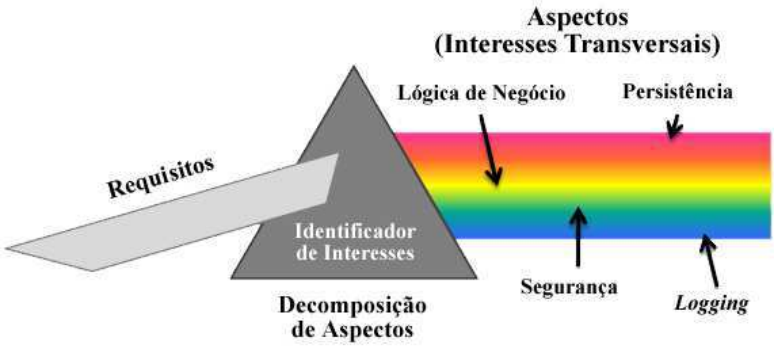

Figura 2.10: Separação de requisitos em interesses na Programação Orientada a Aspectos

Figura 2.10 podemos identificar os seguintes interesses: a lógica de negócio, persistência, segurança e *logging*. A lógica do negócio é o sistema propriamente dito, enquanto que os demais são interesses transversais do mesmo.

2. *Implementação de Interesses.* É a fase de implementação dos interesses de forma independente. Ainda usando o exemplo da Figura 2.10, cada interesse deve ser implementado compondo um módulo, o módulo da lógica de negócio, da persistência, da segurança e de *logging*.

3. *Recomposição de Aspectos.* Nesta fase serão especificadas as regras de recomposição criando as unidades de modularização, ou seja, os aspectos. Este processo é chamado de composição ou integração, usando todos os itens (módulos e aspectos) para compor o sistema final.

O conjunto de passos descrito acima podem ser visualizados na Figura 2.11.

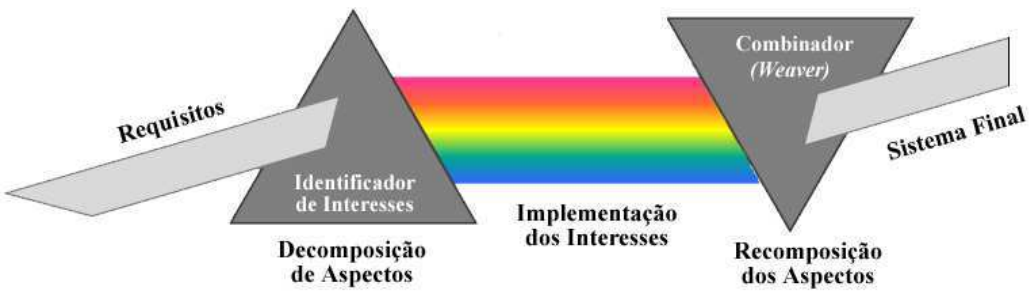

Figura 2.11: Fases de desenvolvimento na Programação Orientada a Aspectos

Desta forma então é realizada a combinação de regras que especificam "qual" e "quando" determinada ação deve ser executada.

Uma implementação de AOP é o AspectJ, uma linguagem que provê a orientação a aspectos para Java. O compilador AspectJ usa os módulos contendo as regras de combinação, identificando cada interesse transversal, introduzindo um novo comportamento para os módulos que contém os principais interesses do sistema.

Na Seção 2.3.3 a seguir vemos como definir um aspecto e inserir regras no mesmo, possibilitando a agregação dos módulos principais do sistema com seus respectivos interesses transversais.

2.3.3 Definindo um aspecto

Um aspecto é composto por um conjunto de definições que especificam as regras de combinação para interesses estáticos e dinâmicos. Um *aspecto* é a unidade central de AspectJ, assim como uma classe é a unidade central em Java [Soares and Borba, 2002].

Em AspectJ, o aspecto é composto por blocos de construção que expressam os interesses transversais da implementação de um sistema, especificando as regras de combinação. Estes blocos são: ponto de junção (*joinpoint*), conjunto de junção (*pointcut*), adendo (*advice*) e declaração intertipos (*inter-type declaration*). Nas próximas seções cada um destes blocos é descrito para entendimento da construção dentro do aspecto.

2.3.3.1 Ponto de junção (*joinpoint*)

Um ponto de junção é um ponto identificável na execução de um programa, pode ser a chamada de um método, a inicialização de um objeto ou a modificação de um membro deste. Em AspectJ, tudo remete a pontos de junção, pois estes são os lugares que as ações transversais acontecem.

A seguir temos a lista de possíveis pontos de junção em AspectJ [Gradecki and Lesiecki, 2003]:

Chamada de método É definido quando qualquer chamada de método é realizada por um objeto ou por um método estático, como o método `main`;

Chamada de construtor Quando um construtor é chamado durante a criação de um novo objeto;

Execução de método Este ponto é definido quando uma chamada de método é realizada por um objeto e o controle é transferido para o método invocado;

Execução de construtor Semelhante ao anterior, quando da entrega do controle para a execução do construtor. O ponto de junção é ativado antes do início da execução do construtor;

Acesso a campo É definido quando um atributo associado a um objeto é lido;

Modificação de campo Acontece quando um atributo associado a um objeto é modificado;

Lançamento de exceção O ponto de junção é definido quando o lançamento de uma exceção é executada;

Inicialização de classe É definido quando qualquer inicializador estático é executado para uma determinada classe. Se não existir nenhum campo estático então não terá este tipo de ponto de junção;

Inicialização de objeto É definido quando um inicializador dinâmico é executado para uma determinada classe. Este ponto de junção é definido depois da construção do objeto e antes de retornar o controle para o criador do objeto.

A mudança de paradigma de Orientação a Objetos para Orientação a Aspectos normalmente confunde o entendimento do fluxo de controle geral de execução do sistema. Quando introduzimos aspectos na implementação de módulos precisamos ter conhecimento total deste fluxo.

Considere o contexto de um banco que faz a transferência entre duas contas. A Figura 2.12 [Filman et al., 2004] mostra o diagrama da execução do método de transferência entre contas. Um cliente precisa realizar a transferência monetária para outro cliente do mesmo banco. De fato, a transferência entre as contas é convertida em duas operações já conhecidas: saque e depósito. No entanto, estas duas ações devem ser realizadas em conjunto e de forma unívoca, caracterizando assim o ato da transferência do valor entre as contas.

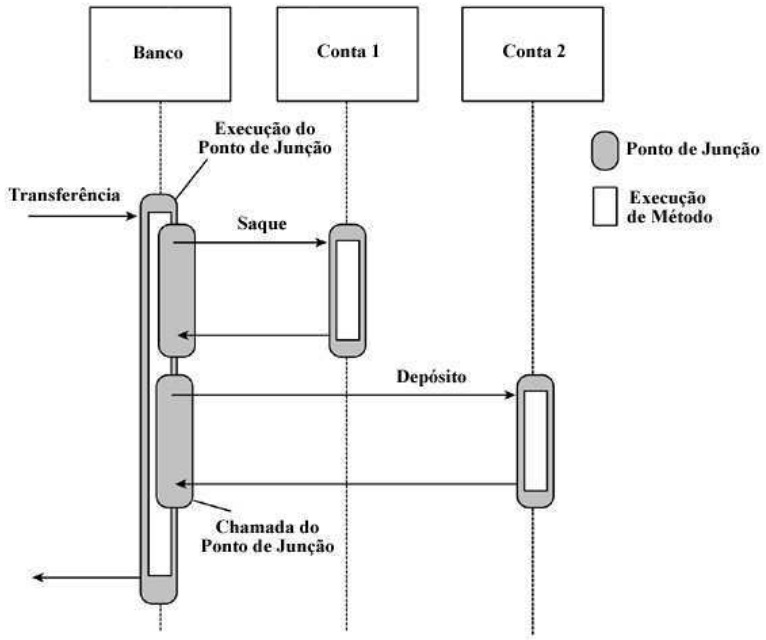

Figura 2.12: Diagrama de execução com pontos de junção.

O fluxo de execução é modificado quando inserimos o controle com aspectos. A Figura 2.13 [Kiczales et al., 2001] facilita o entendimento do fluxo na execução com os aspectos.

2.3.3.2 Conjunto de junção (*pointcut*)

O conjunto de junção consiste na seleção dos pontos de junção e na coleta do contexto de execução destes pontos. Por exemplo, o conjunto de junção seleciona o ponto de junção, que é a chamada para um método, podendo capturar o contexto deste método, como o valor de campos do objeto que está executando o método invocado ou os argumentos do mesmo [Laddad, 2003].

Os conjuntos de junção podem ser agrupados por pontos de junção através de operadores lógicos, tais como conjunção(`&&`), disjunção(`||`) e negação(`!`). O conjunto de junção pode ainda ser *anônimo* ou *nomeado*, com especificação de nível de acesso (publico, privado ou *default*). Um conjunto de junção nomeado pode ser definido no seguinte formato:

```
[especificador de acesso] pointcut <nome> ([argumentos]):
<definição do conjunto de junção>;
```

A definição para um conjunto de junção anônimo é de forma direta, já no ponto de

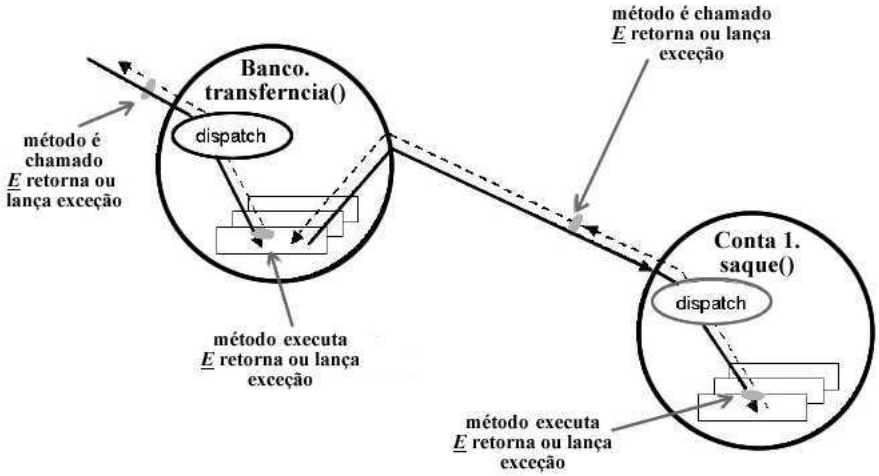

Figura 2.13: Fluxo de execução com pontos de junção.

atuação do mesmo. A definição de um conjunto de junção anônimo implica no uso direto de um adendo, que é a especificação de qual o momento certo de atuar sobre um determinado ponto(s) de junção(ões). O adendo será descrito em maiores detalhes na próxima seção. O conjunto de junção tem a seguinte forma:

```
<tipo de adendo>:  <definição do conjunto de junção>
```

Para definir um conjunto de junção utiliza-se construtores de AspectJ nomeados de designadores. Um designador identifica o conjunto de junção por nome ou por uma expressão. Os principais designadores estão listados na Tabela 2.1 [Gradecki and Lesiecki, 2003]:

Tabela 2.1: Listagem dos designadores em AspectJ.

Designador	**Características**
`call(Signature)`	Invocação do método/construtor identificado pela assinatura
`execution(Signature)`	Execução do método/construtor identificado pela assinaturas
`get(FieldSignature)`	Acesso ao atributo identificado pela assinatura
`set(FieldSignature)`	Atribuição ao atributo identificado pela assinatura
`this(Type pattern)`	Objeto em execução é a instância do tipo
`target(Type pattern)`	Objeto de destino é a instância do tipo
`args(Type pattern)`	Os argumentos são instâncias do tipo
`within(Type pattern)`	Limita o escopo do conjunto de junção para determinados tipos

Durante a especificação de pontos de junção, muitas vezes se faz necessária a especificação de um subconjunto total de um determinado tipo de elementos. AspectJ provê uma

forma de defini-los sem ter que especificar cada um separadamente. A definição de *wildcard*, um caracter usado para representar um conjunto de combinações, é utilizada para generalizar alguns pontos de junção, capturando pontos de junção que compartilham das mesmas características.

Em AspectJ temos os seguintes *wildcards* [Laddad, 2003]:

- `*` qualquer seqüência de caracteres não contendo pontos;
- `..` qualquer seqüência de caracteres, inclusive contendo pontos;
- + qualquer subclasse de uma classe.

2.3.3.3 Adendo (*advice*)

O adendo é semelhante a um método que provê uma forma de expressar a ação transversal nos pontos de junção capturados pelo conjunto de junção. Os três tipos de adendos são:

- Antes (`Before`) executa antes do ponto de junção;
- Depois (`After`) executa depois do ponto de junção;
- Durante (`Around`) executa durante a execução do ponto de junção.

O adendo é estruturado da seguinte forma:

```
<tipo do adendo>:  <nome de um conjunto de junção> ||
<expressão de pontos de junção>
```

Pontos de junção são os únicos pontos que um adendo pode ser aplicado. Na Figura 2.14 podemos ver a identificação dos adendos no fluxo de execução do exemplo da transferência entre contas em um banco.

2.3.3.4 Declaração de Intertipos (*inter-type declaration*)

Além de identificação e modificação em aspectos dinâmicos, o AspectJ possibilita a definição e alteração de tipos estaticamente. A declaração intertipos descreve e pode modificar os interesses estáticos (*static crosscuting*).

AspectJ possibilita a modificação de forma externa em classes, interfaces e aspectos do sistema. Esta modificação aparentemente pode ser considerada inútil, mas se considerarmos

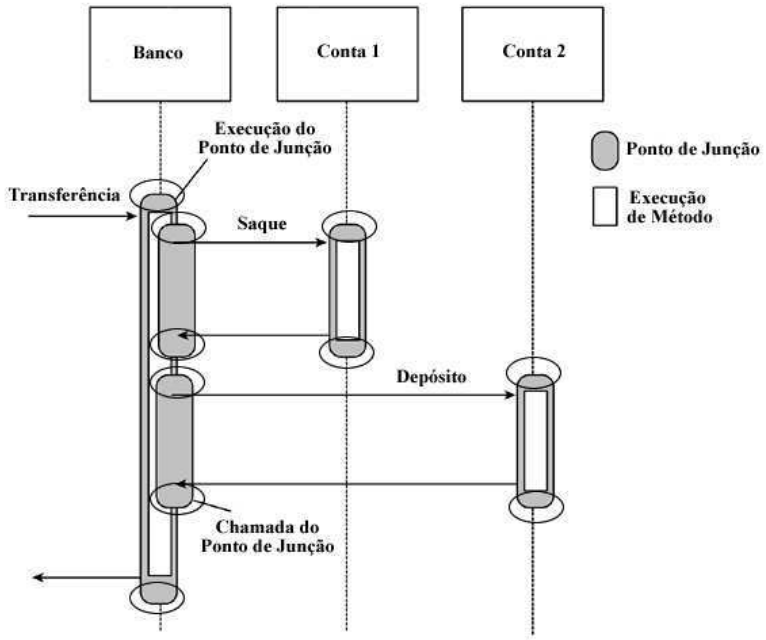

Figura 2.14: Identificação de adendos em um fluxo de execução.

a modificação de uma classe em conjunto com a ação de um aspecto, a declaração intertipos pode se tornar surpreendente [Gradecki and Lesiecki, 2003].

No entanto, a modificação estática de um sistema não afeta diretamente no seu comportamento, não sendo utilizado para avaliar ou observar o controle do comportamento de um sistema desnecessário para uso neste trabalho.

2.3.4 Exemplo de uso de AspectJ

Após a identificação do comportamento transversal, são definidos os pontos de junção referentes a ele, definindo o novo comportamento que os mesmos devem executar. Em seguida, os aspectos que irão modificar o comportamento de sistema devem ser definidos, contendo os conjuntos de junção que devem capturar os pontos de junção previamente definidos. Finalmente, os adendos devem ser definidos para os respectivos pontos de junção, contendo as ações preteridas para cada um deles [Laddad, 2003].

Neste trabalho, o propósito do uso de AspectJ é a possibilidade de realizar *logging* sem ser intrusivo, definindo de forma fácil a captura de eventos em vários pontos de um sistema em execução. A seguir entenderemos o mecanismo de realizar *logging* com AspectJ.

2.3.4.1 *Logging* e *Tracing*

Logging é uma técnica que pode ser usada para proporcionar o entendimento do comportamento de um software. *Logging* é a captura e o armazenamento de informações sobre parte ou todos os processos de um sistema, fazendo a exibição de mensagens com o intuito de descrever as operações executadas em um sistema. Seguindo a mesma linha, o *tracing* ou rastreamento de operações adiciona à atividade de registro a capacidade de vincular uma determinada execução a um usuário do sistema [Winck, 2006; Laddad, 2003].

Estas duas técnicas são comumente utilizadas pelos desenvolvedores para fazer depuração de software. A forma tradicional que a técnica de *logging* é usada exige que cada entidade que precise *logar* informação tenha o controle e a ação sobre a atividade de guardar a informação requerida. Esta exigência implica na modificação de cada entidade de forma a possibilitar a gravação da informação.

De maneira geral, podemos entender a técnica tradicional de *logging* ilustrada na Figura 2.15 [Laddad, 2003].

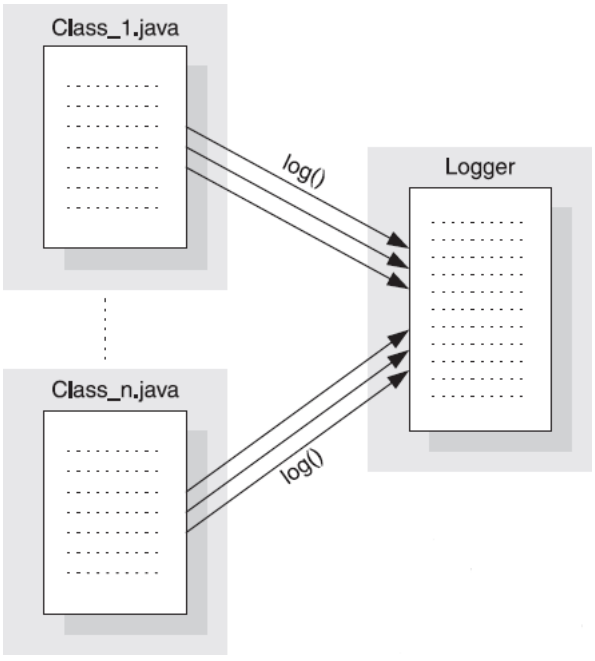

Figura 2.15: *Logging* de informação sem aspectos.

Como podemos ver, cada classe é a responsável por armazenar todas as informações referente a ela, propiciando mais uma responsabilidade para uma única entidade. Outra

desvantagem é a necessidade de inserir código da técnica de *logging* em todas as entidades que precisam guardar informações; isto dificulta a manutenção do software com um interesse que atravessa grande parte ou todos os componentes de um software.

Uma forma de separar este interesse transversal para a técnica de *logging* é usando o conceito da orientação a aspectos [Laddad, 2003]. Usando aspectos é possível inserir a técnica de *logging* sem modificar as classes das quais precisamos extrair informação. Para tanto, defini-se aspectos que fazem a ligação entre o código em execução a ser *logado* e a classe que armazena todas as informações de *logging*.

Podemos entender o mecanismo usando a orientação a aspectos como mostrado na Figura 2.16.

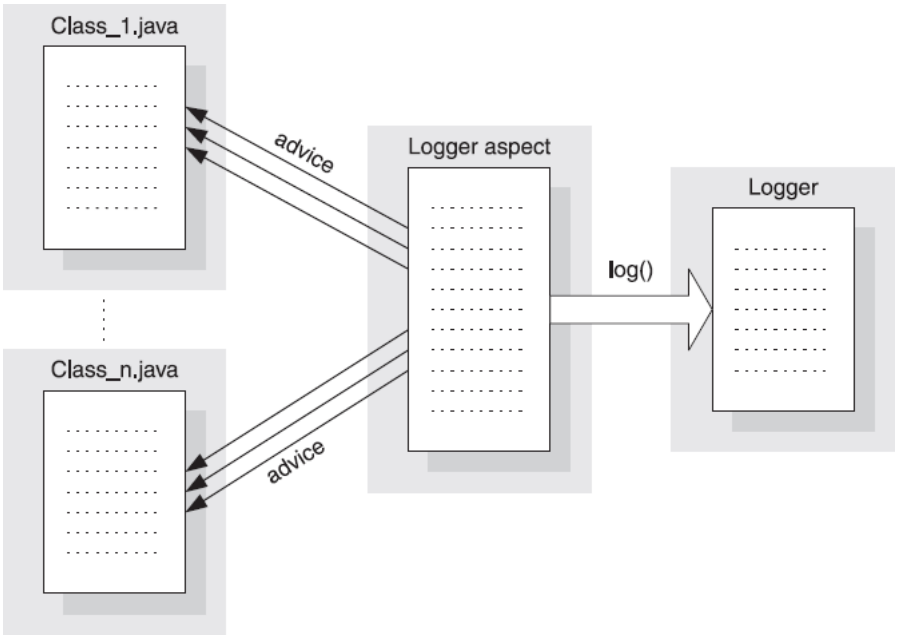

Figura 2.16: *Logging* de informação utilizando o conceito de aspectos.

Usando o conceito de orientação a aspectos conseguimos separar o código do sistema implementado do interesse de armazenar as informações por ele executadas.

A realização do *logging* de informações é possível usando os aspectos. No exemplo de aspecto no Código 2.1 podemos ver um exemplo de aspecto que pode ser utilizado para o armazenamento de informações durante a execução de uma aplicação.

```
1  import org.aspectj.lang.*;
2  import logging.*;

3  public aspect LoggingAspect {

4    protected pointcut loggedOperations()
5      : (execution(* *.*(..))
6          || execution(*.new(..)));

7    before() : loggedOperations() {
8      Signature sig = thisJoinPointStaticPart.getSignature();
9      System.out.println("Entering ["
10        + sig.getDeclaringType().getName() + "."
11        + sig.getName() + "]");
12   }
13 }
```

Código 2.1: Código aspecto de *logging* das execuções de uma aplicação.

2.4 Ordenação de Eventos

2.4.1 Introdução

O conceito de tempo é um ponto fundamental para nossa forma de pensar, possibilitando ordenar a ocorrência de eventos [Lamport, 1978]. Com isso, o conceito de ordenação temporal dos eventos está diretamente associada à forma humana de pensar sobre sistemas.

Considerar que um evento ocorrido em um determinado instante t no tempo forçou a ocorrência de outro, que ocorreu no instante $t + i$ (sendo $i > 0$), nos passa a idéia de sincronismo entre os eventos, que neste caso são dependentes um do outro na relação de causa e efeito em função do tempo.

Fatores como concorrência e não-determinismo são intrínsecos aos sistemas distribuídos, caracterizando-os como sistemas desprovidos de sincronização. A conjunção desses fatores e a falta de um relógio global para todas as máquinas pertencentes a um sistema distribuído faz com que não tenhamos uma relação de ordenação temporal entre a ocorrência de eventos em máquinas distintas [Schwarz and Mattern, 1994; Nett and Gergeleit, 1997].

O assincronismo característico de sistemas distribuídos faz com que a relação causa e efeito, ou a relação “aconteceu antes” (*happened before*), não seja estabelecida e conhecida naturalmente. E esta ordenação se faz necessária para a construção de uma linha (*trace*) de execução de um sistema distribuído. Sendo que esta linha tem que obedecer à relação causa

e efeito entre os eventos ocorridos [Babaoglu and Marzullo, 1993].

Contudo, podemos instrumentar o sistema distribuído com o conceito de relógios lógicos (*logical clocks*) [Lamport, 1978] para fornecer um sistema equivalente à sincronização dos relógios existentes no sistema distribuído, e assim possibilitando a ordenação dos eventos ocorridos.

Na Seção 2.4.2 a seguir entenderemos o funcionamento dos relógios lógicos, que neste trabalho foi utilizado para instrumentar os evento de sistemas distribuídos de forma a possibilitar a ordenação das ações de seus processos.

2.4.2 Relógios Lógicos

Em um sistema assíncrono, que não contém um relógio global, podemos adicionar um simples mecanismo de relógio para marcar o instante em que cada evento ocorreu baseado em uma ordem, incrementando o valor de cada relógio, garantindo assim uma consistente relação de causa entre os eventos em função do tempo [Babaoglu and Marzullo, 1993]. Assim, em um sistema assíncrono podemos satisfazer a **condição de relógio** (*clock condition*).

Consideremos a seguinte definição de condição de relógio: dados dois eventos *e* e *e'*, e seja *RG* o relógio global de um sistema, então:

Condição de relógio: $e > e' \Rightarrow RG(e) < RG(e')$.

Para muitas aplicações, qualquer mecanismo que satisfaça a condição de relógio pode ser tida como suficiente ao usar os valores produzidos por ele, se estes valores foram produzidos por um relógio global de tempo real.

Vamos considerar um sistema distribuído simples contendo três processos. Cada processo p_1, p_2 e p_3 tem a sua computação do histórico local dos eventos, dependentes do seu relógio local em função do tempo, veja Figura 2.17.

Vamos entender primeiramente o funcionamento de relógios lógicos. Cada processo mantém uma variável local chamada *relógio lógico RL*, que mapeia cada evento para um número positivo natural. O valor do relógio lógico quando um evento e_i é executado pelo processo p_i é denotado $RL(e_i)$. A variável *RL* denota o valor corrente do relógio lógico do processo implícito ao contexto. Cada mensagem *m* que é enviada contém um *timestamp TS(m)* que se refere ao valor do relógio lógico associado com o evento de envio. Antes de qualquer evento ser executado, todos os processos iniciam seus relógios lógicos com zero. A

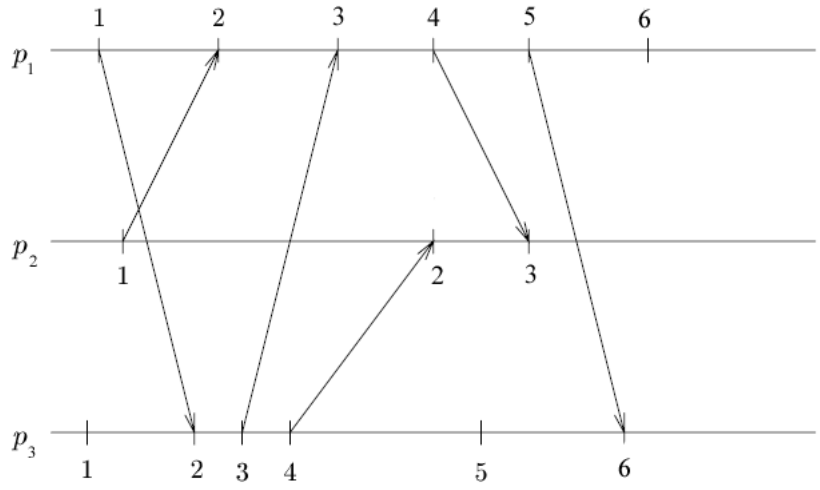

Figura 2.17: Comunicação entre processos em um sistema distribuído simples

partir daí, quando um evento de recebimento é executado, o relógio lógico é atualizado para o maior valor entre o seu relógio lógico local e o valor do *timestamp* da mensagem recebida. Se um evento interno ou envio de mensagem é executado, o relógio lógico é simplesmente incrementado [Babaoglu and Marzullo, 1993].

Formalmente, podemos ver este funcionamento a seguir:

$$RL(e_i) := \begin{cases} LR + 1 \text{Se e}_i \text{ for um evento interno ou envio de mensagem,} \\ max((RL,TS(m)) + 1) \text{Se e}_i \text{ for uma mensagem recebida.} \end{cases} \quad (2.1)$$

A figura abaixo mostra o funcionamento de relógios lógicos aplicados à figura 2.17.

Note que a interação entre processos sendo demarcados com relógios lógicos acima produz valores que incrementam para um valor dependente da sua precedência causal de ocorrência. Esta afirmação possibilita dizer que para dois eventos $e \rightarrow e'$, o valor dos relógios lógicos associados a estes também têm uma relação de precedência $RL(e) < RL(e')$. Respeitando assim a condição de relógio definida previamente nesta seção.

Com isso, podemos instrumentar um sistema distribuído de forma a possibilitar uma construção de precedência entre os eventos ocorridos no mesmo.

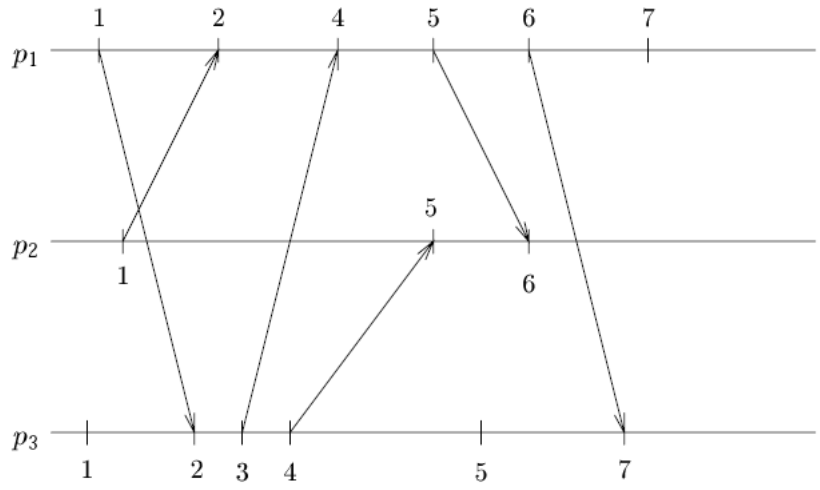

Figura 2.18: Comunicação entre processos usando Relógios Lógicos

2.5 Considerações Finais

O embasamento teórico apresentado neste capítulo mostra os conceitos usados no nosso trabalho, dando subsídios para entendimento de como foi desenvolvida a técnica aqui sugerida. Sistemas distribuídos exibem um comportamento aleatório, de difícil entendimento durante a execução. Então, para certificar que estes sistemas estão sendo corretamente implementados precisamos utilizar um mecanismo para verificação de sua corretude comportamental, dado que estes sistemas não conseguem ser verificados com base somente nos seus conjuntos de entrada e saída de dados. No entanto, a verificação de sistemas implica na prévia especificação de como um determinado sistema deve se comportar. Para isso sugerimos a descrição comportamental utilizando a Lógica Temporal Linear (LTL), como mecanismo que provê operadores que suportam a descrição de propriedades que devem ser averiguadas ao longo do tempo de execução destes sistemas.

Após a especificação, durante a execução do sistema distribuído, é necessária a captura das informações da execução do sistema. Fazemos uso do conceito da Programação Orientada a Aspectos, utilizando a linguagem AspectJ, sendo uma forma não intrusiva de capturar a informação em tempo de execução. De posse das informações capturadas durante a execução do sistema distribuído, uma ordenação dos eventos se faz necessária, possibilitando a verificação do comportamento do sistema com base em um histórico de execução consis-

tente. Usamos então o conceito de relógios lógicos, uma técnica que viabiliza a inferência da ordenação dos eventos de um sistema distribuído baseada na precedência da troca de mensagens entre processos do mesmo.

Capítulo 3

Técnica de Monitoração de Sistemas Distribuídos

Neste capítulo apresentamos a técnica de monitoração comportamental de sistemas concorrentes e distribuídos, que tem como principal objetivo a detecção de comportamentos que não estão de acordo com propriedades especificadas para o sistema em execução. Inicialmente apresentamos uma visão geral da técnica, seus requisitos e a especificação de propriedades a serem verificadas em tempo de execução. Em seguida, mostramos as etapas necessárias para monitorar um sistema em execução e verificar a corretude comportamental destes sistemas.

3.1 Visão Geral

Neste trabalho temos como foco a conformidade de código em relação ao comportamento apresentado pelo sistema em execução. A técnica de monitoração de sistemas concorrentes e distribuídos realiza a verificação da conformidade de código de um sistema. A monitoração auxilia os desenvolvedores de sistemas a detectar comportamentos que não estão de acordo com as especificações comportamentais definidas para o software em desenvolvimento. A principal motivação para o uso desta técnica é a necessidade de verificar se o código produzido está de acordo com as especificações e requisitos que o sistema deve apresentar. Se alguma violação de requisito for encontrada o desenvolvedor é informado e ajudado a encontrar o motivo desta violação.

A Figura 3.1 ilustra a visão geral da técnica de monitoração de sistemas distribuídos.

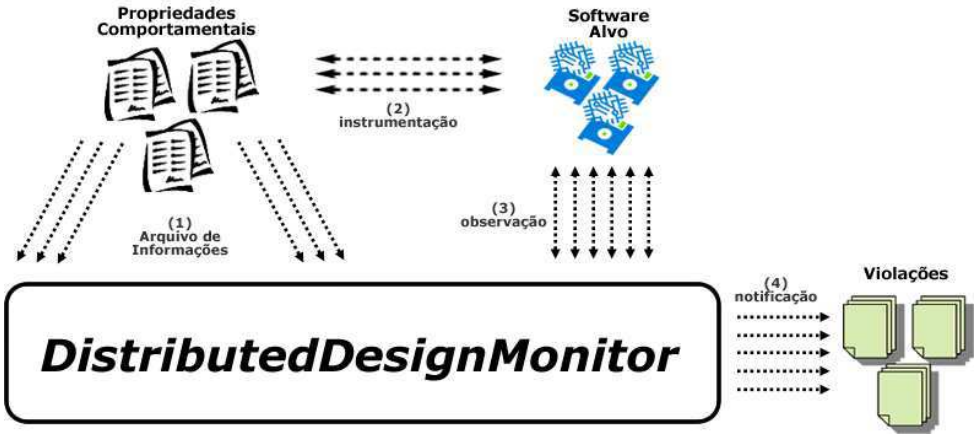

Figura 3.1: Visão geral da técnica *DistributedDesignMonitor* - DDM

Primeiramente o conjunto de propriedades comportamentais é definido e este conjunto de arquivos é submetido para o DDM (1). A partir destas propriedades é gerado automaticamente o código de instrumentação (2), que será responsável por observar (3) as informações em tempo de execução. Durante a execução do sistema monitorado, se alguma inconsistência for detectada então o DDM notifica a ocorrência do comportamento inesperado do sistema (4).

Nas próximas seções entenderemos como é feita esta verificação e quais os artefatos necessários para usarmos a técnica de monitoração de sistemas aqui apresentada.

3.2 Monitoração de Sistemas Distribuídos

Com o propósito de construir uma técnica que detecte comportamentos não esperados durante a execução de um sistema distribuído sob monitoração, os seguintes requisitos devem ser atendidos pela técnica proposta:

Não intrusividade Instrumentar o código possibilitando a captura de informações durante a execução do sistema a ser monitorado deve impactar minimamente no desenvolvimento do sistema. A possibilidade de modularizar a especificação comportamental, separando do sistema originalmente produzido, fornece maior organização e separação de interesses de cada módulo.

Transparência Com a modularização entre a especificação e o código fonte a ser analisado,

o desconhecimento do mecanismo utilizado na monitoração facilita a utilização da técnica.

Automatização A instrumentação do sistema a ser monitorado deve ser mais automátizado possível, diminuindo a necessidade de trabalho manual constante do desenvolvedor que pretende utilizar a monitoração de elementos espalhados em um rede que contém o sistema distribuído em questão.

A técnica de monitoração para sistemas distribuídos, ou (*DistributedDesignMonitor (DDM)*, consiste no acompanhamento da execução do sistema monitorado, analisando se o comportamento apresentado está de acordo com a especificação definida. No entanto, para possibilitar esta análise é necessário primeiramente definir qual o conjunto de propriedades comportamentais que o sistema monitorado deve apresentar.

Antes de detalharmos como é realizada a análise comportamental do sistema distribuído sob monitoração, vamos definir qual a semântica da palavra **Verificação** neste trabalho. A Verificação de Sistemas é usada para garantir se um sistema em análise processa determinadas propriedades. Ainda neste contexto, o conceito de verificação é amplamente utilizado para denotar a checagem formal de todas as possíveis execuções de um determinado sistema [Baier and Katoen, 2008; Katoen, 1999].

A monitoração de Sistemas Distribuídos aqui descrita se propõe a realizar a verificação no sentido de certificar se o sistema sob análise está processando um determinado conjunto de propriedades, propriedades estas definidas na sua especificação. No entanto, esta verificação é feita sobre o conjunto de reais execuções, e não realiza a verificação sobre todas as possíveis execuções daquele sistema. Esta definição e decisão de abordagem foram tomadas, entre outras razões, em detrimento de: um número exponencial, possivelmente intratável para o poder computacional hoje disponível, de caminhos diferentes a serem traçados por um determinado sistema distribuído; e a possível economia no processo de verificação de caminhos que dificilmente serão traçados pelo sistema sob análise. Então, diferentemente do contexto formal definido em *Model Checking* [Baier and Katoen, 2008], o termo Verificação tem a conotação de garantir a concordância entre uma execução e a sua especificação comportamental, ou seja, garantindo se o sistema processa o conjunto de propriedades para ele especificadas.

Propriedades comportamentais definem o comportamento desejado e considerado correto para aquela implementação. Uma propriedade comportamental pode definir o comportamento para todos os itens, mas também pode ser especificada somente para alguns subitens do sistema. A idéia do conjunto de passos necessários para implementar a técnica do DDM pode ser visualizada na Figura 3.2.

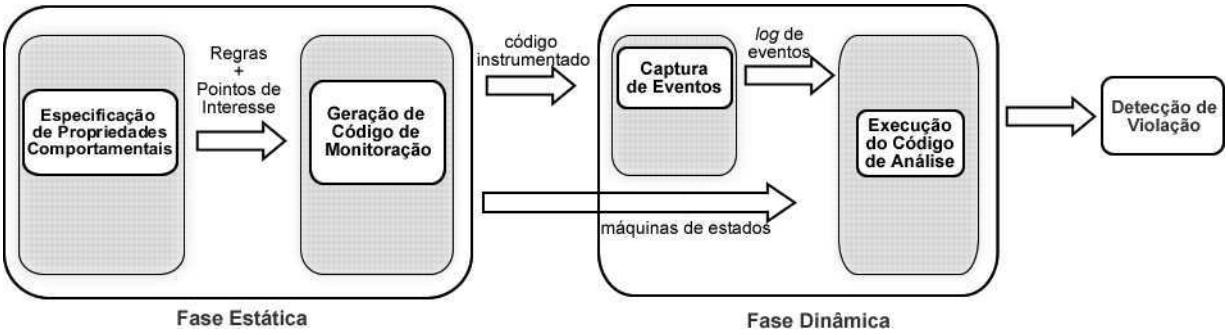

Figura 3.2: Visão geral da execução do *DistributedDesignMonitor* - DDM

As propriedades comportamentais do sistema precisam ser descritas e passadas a utilização na técnica de monitoração DDM. As propriedades são a base para a geração automática dos códigos de instrumentação e de análise, formando o código de monitoração, compondo a Fase Estática.

Na Fase Dinâmica são realizadas a captura dos eventos, a organização destes eventos e a análise comportamental do Sistema Distribuído sob monitoração. Se alguma inconsistência for detectada então o DDM notifica a ocorrência do comportamento inesperado do sistema, através do conjunto de violações.

A partir das propriedades comportamentais extraímos informações que possibilitam a geração automática dos códigos de instrumentação e de análise comportamental. Por este motivo, a especificação de cada propriedade comportamental é crucial para a correta geração e verificação da conformidade do comportamento do sistema.

O código de instrumentação foi desenvolvido de maneira que não seja necessária a modificação do código fonte a ser monitorado, deixando a técnica não intrusiva ao código que vai ser analisado.

O código de análise comportamental também é automaticamente gerado a partir das demais informações contidas na(s) propriedade(s) comportamental(is).

Na Seção 3.2.1 a seguir podemos entender o formato de uma propriedade comportamental e como esta deve ser especificada. A especificação das propriedades comportamentais representa a fase inicial do processo de monitoração, pois a partir destas especificações são extraídas as informações que serão utilizadas durante a monitoração do sistema distribuído.

3.2.1 Especificação de Propriedades Comportamentais

Sistemas Distribuídos são compostos por processos que executam tarefas de forma concorrente e aleatória. Com a continuidade na execução de tarefas, e na comunicação entre estas tarefas, o comportamento traz a idéia de temporalidade relacionada aos processos de um sistema distribuído. Por exemplo, a finalização da execução de um processo dá a possibilidade de execução do seguinte, ou ainda, a liberação de um recurso por um processo disponibilizado para ser usado por outro processo. Esses tipos de comunicação entre processos são típicos de sistemas concorrentes e distribuídos. Estes exemplos nos remetem a especificação de propriedades comportamentais chamadas de propriedades de segurança (*safety properties*) [Lamport, 1977].

No entanto, para especificar o comportamento desejado para um conjunto de processos em um sistema distribuído é necessária a descrição do que **não** pode ocorrer de errado. Considere o Exemplo 3.1 de comportamento desejado:

Exemplo 3.1 *Um processo deve acessar exclusivamente um conjunto de dados.*

Em algum momento precisa-se dar uma notificação se o processo em questão está acessando o conjunto de dados de forma exclusiva. Se até aquele momento a afirmação for verdadeira, poderemos dizer que o sistema está correto, mas nada garante que em um passo seguinte alguma inconsistência ocorra. Então a questão é saber quanto tempo devemos monitorar um sistema de forma a garantir a cobertura da verificação da execução deste. Questão esta que é inviável de ser respondida com grande margem de acerto.

Desta forma, especificamos aquilo que esperamos que **não** ocorra durante a execução do sistema distribuído, ou seja, o comportamento é descrito através da negação de um requisito. Este formato de propriedade é denominado *never claim* [Ireland, 2007]. Este formato foi adotado pelo fato de ser mais fácil solicitar ao especialista do sistema distribuído sob investigação que defina o que deve caracterizar uma violação de comportamento, em vez

de perguntar qual é o comportamento desejado. Durante a análise comportamental poderemos afirmar se o sistema quebrou ou não uma determinada especificação se utilizarmos especificação do tipo *never claim*. Se usássemos a especificação de como o sistema deve se comportar, poderíamos afirmar em algum instante que o sistema está correto (coerente com a especificação) e no instante seguinte violar alguma de suas especificações.

Podemos usar o Exemplo 3.1, citado anteriormente, e transformá-lo em uma propriedade do tipo *never claim*, mostrado no Exemplo 3.2.

Exemplo 3.2 *Nenhum conjunto de dados pode ser acessado instantaneamente por dois processos distintos.*

A estrutura de uma propriedade comportamental foi projetada com o intuito de verificar a execução correta de processos independentes (*threads*) agindo sobre objetos. A execução de uma *thread* apresenta grande dificuldade de rastreamento. Mas a estrutura apresentada na Seção 3.2.1.1 pode ser estendida para outros tipos de especificações, como, por exemplo, a especificação de interação entre objetos.

3.2.1.1 Estrutura da Propriedade Comportamental

Cada propriedade comportamental contém uma identificação (nome), uma especificação de comportamento e os pontos do sistema que devem obedecer a tal comportamento. Estes são os elementos que compõem a especificação de uma propriedade comportamental.

Formalmente, uma propriedade comportamental S de um sistema de software pode ser expressa como uma 3-tupla:

$S = \{P, C, R\}$

onde:

$P =$ conjunto finito dos pontos de interesse;

$C =$ conjunto finito de expressões do comportamento desejado;

$R \subseteq \{P \text{ X } C\} =$ a relação entre ponto e expressão de comportamento;

A definição de uma propriedade comportamental é formada pela relação R entre comportamento desejado e os pontos de interesse do sistema de software alvo. Para cada expressão comportamental (fórmula LTL) de C é preciso definir quais os pontos no código do software

alvo que devem obedecer tal comportamento, formando o conjunto P. O relacionamento entre o comportamento C e os pontos de interesse P, são definidos como R. Cada ponto em P deve obedecer ao comportamento especificado em C.

Em um sistema podemos ter mais de uma especificação comportamental, assim, podemos definir um conjunto de especificações comportamentais, composto por uma ou mais propriedades comportamentais. Então, formalmente temos:

$SC = \{S_1, S_2, ..., S_n\}$

onde:

$SC =$ conjunto de propriedades comportamentais;

$S_1, ..., S_n =$ propriedades comportamentais de uma especificação;

Geralmente, a especificação de uma propriedade comportamental parte de uma descrição em linguagem natural. Entretanto, linguagens naturais são ambíguas, o que possibilita a interpretação errônea de especificações de software e, conseqüentemente, pode gerar uma implementação incorreta. Então, a transformação da especificação inicial em linguagem formal provê uma maneira de descrever o comportamento do sistema com completude, consistência, precisão e concisão. Como o propósito da técnica DDM é realizar a monitoração do comportamento ao longo do tempo, de sistemas concorrentes e distribuídos, a linguagem escolhida para a especificação do comportamento desejado (propriedade comportamental) foi a linguagem LTL, seguindo a sintaxe e a semântica apresentadas na Seção 2.2.

A sintaxe utilizada para especificar os pontos de interesse de P de uma especificação S é a mesma suportada por AspectJ [Laddad, 2003]. O AspectJ dá suporte à generalização de termos para a identificação de vários elementos de um mesmo conjunto ou subconjunto de objetos. Assim, em vez de listar nominalmente cada um dos pontos de interesse, é possível usar símbolos para representar um conjunto de tipo. Temos os seguintes símbolos:

- `*` - representa um conjunto qualquer de caracteres, para designar parte de um método, classe, interface ou pacote;
- `..` - denota todos os sub-pacotes indiretos e diretos de um determinado pacote. Para métodos, é usado para denotar qualquer tipo e quantidade de argumentos do método;
- + - denota qualquer sub-classe ou sub-interface de um determinado tipo.

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 quanto os que não são `final` serão considerados para o acompanhamento da execução, capturando a informação relevante.

Os modificadores podem ser usados também em conjunto 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 tipo. Na Tabela 3.1 são apresentados alguns exemplos do uso de tais símbolos para especificação dos pontos de interesse.

Tabela 3.1: Exemplos de padrões para especificação de pontos de interesse.

Padrão de Tipo	**Significado**
`public void List.clear()`	Método público `clear()` da classe `List`, que retorna `void` e não recebe nenhum parâmetro.
`public void List.clear() throws UnsupportedOperationException`	Método público `clear()` da classe `List`, que retorna `void`, não recebe nenhum parâmetro e tenha declarado como exceção `UnsupportedOperationException`.
`public boolean List.add*(*)`	Todos os métodos públicos da classe `List` que têm seu nome iniciado com `add`, que retornam `boolean` e recebem apenas um único parâmetro de entrada de qualquer tipo.
`public void List.*()`	Todos os métodos públicos da classe `List` que retornam `void` e não recebem nenhum parâmetro.
`public * List.*()`	Todos os métodos públicos da classe `List` que retornam qualquer tipo e não recebem nenhum parâmetro.
`public * List.*(..)`	Todos os métodos públicos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros.
`* List.*(..)`	Todos os métodos da classe `List`.
`!public * List.*(..)`	Todos os métodos não-públicos da classe `List`.
`* List+.*(..)`	Todos os métodos da classe e sub-classes de `List`.
`* List.*(int,..)`	Todos os métodos da classe `List` que retornam qualquer tipo e recebem qualquer número e tipos de parâmetros, mas cujo o primeiro parâmetro deve ser do tipo `int`.
`public List.new(Collection)`	Todos os construtores públicos da classe `List` cujo parâmetro é do tipo `Collection`.

O formato genérico de uma propriedade comportamental pode ser visto no exemplo do Código 3.1. O campo `label` define o nome da propriedade especificada. O campo `rule` define a propriedade comportamental a ser obedecida pelos pontos descriminados no campo `points`. Em `points` esperamos uma expressão AspectJ válida, ou seja, obedecendo às restrições e sintaxe da linguagem.

```
label := BehaviorPropertyName;
rule := Formula LTL;
points := * ObjectName.*(..);
```

Código 3.1: Propriedade Comportamental em formato genérico.

3.2.1.2 Exemplo de Especificação Propriedades Comportamentais

Nesta subseção, apresentaremos como descrever e transformar um requisito de comportamento em propriedade comportamental. Veja os Exemplos 3.3 e 3.4.

Exemplo 3.3 *Suponha que um sistema `S` é composto por dois módulos `A` e `B`, onde cada módulo possui as threads `ThreadA` e `ThreadB`, respectivamente. O módulo `A` é formado por objetos do tipo `ObjetoA`. Durante a execução do sistema `S` a partir do momento em que forem acessados pela `ThreadA`, apenas essa thread é que quem deve acessar os objetos do tipo `ObjetoA`. Comportamento semelhante deve ser observado nos objetos do tipo `ObjetoB` e para a `ThreadB`.*

Especificando essa característica como uma propriedade comportamental para a técnica *DDM*, temos o código 3.2:

```
label := BehaviorPropertyModuleA;
rule := (!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB);
points := * ObjectA.*(..);
```

Código 3.2: Exemplo de Propriedade Comportamental.

Ao transformar esta característica no formato *never claim*, temos: `ThreadA` e `ThreadB` **não** podem acessar objetos do módulo `A` até que uma das duas *threads* acessar, só esta poderá acessar o objeto de `A`.

Exemplo 3.4 *Considerando as mesmas variáveis do exemplo anterior, queremos agora acompanhar o acesso concorrente das threads* `ThreadA` *e* `ThreadB` *sobre o mesmo tipo* `ObjetoA`*. Para isto, define-se a propriedade através da descrição da ocorrência de* `ThreadA` *e* `ThreadB` *acessando concorrentemente o tipo* `ObjetoA`*.*

A propriedade de verificação da ocorrência de acesso concorrente de duas entidades sobre o mesmo objeto deve ser transformada para o formato que viabilize a detecção do momento que apresentou o acesso concorrente, ou seja, para o formato *never claim*, atendendo ao formato de monitoração aplicada nesta técnica. Uma vez que o formato de propriedade desejado descreve a ação que não deve ser executada, a propriedade de detecção de acesso concorrente deve ter o formato apresentado no Código 3.3:

```
label := ConcurrentBehaviorPropertyModuleA;
rule := !(ThreadA && ThreadB) U ([] ThreadA || [] ThreadB);
points := * ObjetoA.*(..);
```

Código 3.3: Propriedade Comportamental de uma verificação de concorrência.

O campo `rule` descreve que `ThreadA` e `ThreadB` não podem acessar `ObjetoA` ao mesmo tempo, até que uma das *threads* passe a acessar isoladamente o `ObjetoA`.

Cada propriedade comportamental é a fonte dos códigos de instrumentação e de análise da execução do sistema distribuído a ser monitorado na técnica DDM.

3.2.2 Geração Automática de Código de Instrumentação e de Análise

De posse da especificação comportamental de um sistema distribuído, iniciamos a geração automática dos códigos para a realização da monitoração e análise do sistema sob investigação.

3.2.2.1 Código de instrumentação

Nem sempre cada propriedade comportamental de um sistema está relacionada a todos os elementos que o compõem. Normalmente, cada uma delas está relacionada a um determinado subconjunto dos pontos de interesse do sistema a ser analisado.

Para realizarmos a monitoração de sistemas distribuídos, relacionamos cada ponto a um objeto (objeto no conceito do paradigma de Orientação a Objetos) [Gamma et al., 1995].

Cada um dos pontos (objetos) especificados com uma propriedade comportamental será monitorado e analisado de acordo com o comportamento desejado para aquele objeto. O mesmo é válido para um subconjunto de objetos que deve obedecer à uma única especificação comportamental descrita, relacionando este comportamento ao subconjunto de objetos.

Uma vez que a captura de informações sobre os objetos deve interferir minimamente na execução do sistema monitorado, usamos o paradigma de Orientação a Aspectos [Kiczales, 2005; Filman et al., 2004], através da linguagem AspectJ [Laddad, 2003], possibilitando a captura de informações de sistemas distribuídos implementados em Java sem a necessidade de modificar o código fonte do mesmo, sendo este um dos maiores benefícios da técnica DDM de monitoração: uma forma não intrusiva de capturar as informações necessárias para realizar a análise comportamental do sistema monitorado.

Além de ser não intrusiva, esta técnica permite o isolamento das definições de propriedades comportamentais, de um ou mais objetos, em um único local. Esta característica possibilita a especificação de uma propriedade que está relacionada a vários objetos, estes podem estar localizados em máquinas diferentes de um sistema distribuído. No entanto a definição da regra é feita uma única vez, em um único local, eliminando a necessidade de distribuir o código sobre todo o sistema a ser monitorado.

A geração do aspecto é automatizada com o intuito de facilitar o processo de monitoração. O ponto, ou conjunto de pontos, contido no campo `points` de cada propriedade comportamental é inserido em um aspecto, sendo este responsável pela captura de informações referente à execução destes objetos.

Usando o Exemplo 3.4 anteriormente descrito, o código aspecto gerado a partir da propriedade comportamental com nome `ConcurrentBehaviorPropertyModuleA` é mostrado no código aspecto 3.4.

O aspect `DistributedDesignMonitorAspect` é uma superclasse que realiza a captura das informações desejadas. Cada aspecto que herda desta superclasse só precisa conter os pontos que devem ser observados. O código 3.5.

Após a geração do aspecto com os pontos de monitoração de cada regra, os aspectos gerados precisam ser inseridos junto ao código fonte do sistema que será monitorado. Antes da execução do sistema distribuído é necessária a compilação do código fonte juntamente

```
1  public aspect ConcurrentBehaviorProperty
2       extends DistributedDesignMonitorAspect{

3     public ConcurrentBehaviorProperty() throws Exception{
4             setNameAspect("ConcurrentBehaviorPropertyModuleA");
5       }

6     public pointcut monitoringPoints()
7      :(execution(* ObjectA.*.*(..)));
8  }
```

Código 3.4: Código aspecto de uma propriedade comportamental de concorrência.

```
1  public abstract aspect DistributedDesignMonitorAspect
2            extends DesignMonitorAspect {

3       public DistributedDesignMonitorAspect() {
4            super();
5       }

6       after() : monitoringPoints() {
7            sendMesg( thisJoinPoint );
8       }

9       protected void sendMesg(JoinPoint joinPoint) {
10           // getInformation Captura as informações referents ao
11           // JoinPoint e o evento recebido.
12           Event e = getInformation(joinPoint, Event.SEND_EVENT);
13           try {
14                RemoteMonitorImpl.getLocalInstance().sendEvent(e);
15           } catch (RemoteException e1) {
16                System.err.println(e1.getMessage());
17           }
18      }

19      protected String getNameAspect() {
20           return nameAspect;
21      }
22      ...
23 }
```

Código 3.5: Código da superclasse aspecto usado na monitoração de objetos.

com os aspectos, possibilitando a construção do código fonte instrumentado com o bytecode dos aspectos inseridos a fim de capturar as informações em tempo de execução.

3.2.2.2 Código de Análise

De posse das propriedades comportamentais definidas, inicia-se o processo de construção automática do código de análise do comportamento do sistema.

O campo `rule` de uma propriedade comportamental contém a especificação no formato da lógica LTL, apresentada na Seção 2.2, que descreve como o sistema distribuído não deve se comportar. Então, esta especificação LTL é convertida em um autômato de Büchi equivalente. Esta transformação é automaticamente realizada utilizando o algoritmo apresentado na Seção 2.2.5.

O autômato de Büchi processa palavras de tamanho infinito [Katoen, 1999], mas como a computação é limitada quanto à quantidade de informação a ser processada, devemos converter o autômato de Büchi gerado em uma máquina de estados equivalente. Esta nova transformação é necessária para adaptar o processamento finito da computação de sistemas distribuídos.

A máquina de estados gerada não possui o conceito de *estado final*. Todos os estados são considerados estados *aceitáveis*, ou seja, se a computação parar estando em qualquer um dos estados aceitáveis da máquina de estados, a computação é considerada íntegra. Entretanto, em cada maquina de estados é introduzido um estado de marcação, chamado de **estado inválido**. Se, em algum momento da monitoração, a máquina de estados alcançar o estado inválido é detectado então um comportamento anômalo e conflitante com o esperado para aquele sistema.

Todo o processo da conversão da propriedade comportamental em LTL para o respectivo autômato de Büchi é auxiliado com o uso da ferramenta LTL2BA4J, que faz a transformação de fórmulas LTL para o autômato respectivo utilizando a linguagem Java como código resultante do autômato gerado [Bodden, 2005]. A Figura 3.3 ilustra como acontece todo o processo de criação do código de análise na técnica *DistributedDesignMonitor*, mostrando a transformação da propriedade do Exemplo 3.3.

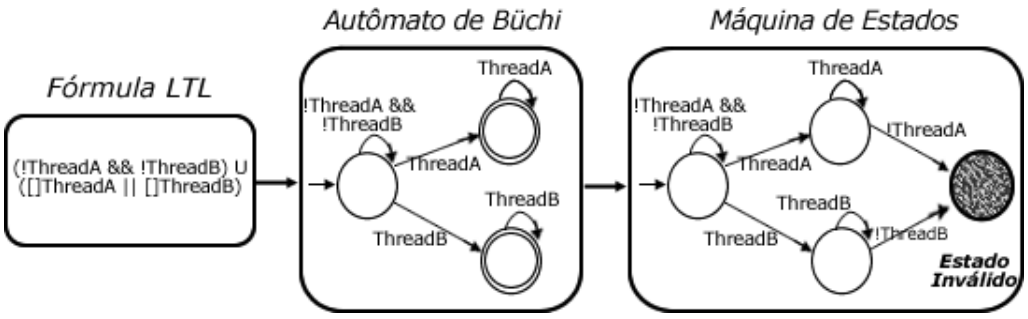

Figura 3.3: Transformação de propriedade comportamental em Autômato de Büchi.

3.2.3 Instrumentação do Código e Captura de Eventos

Após a geração automática do código de instrumentação é preciso capturar as informações em tempo de execução. Nesta subseção, apresentamos como ocorre esta captura.

Durante a execução do sistema distribuído, o código instrumentado, resultante da combinação do aspecto com o código fonte do sistema, observa todo o comportamento e identifica a execução dos objetos que estão sob investigação. Estes objetos estão descritos no campo `points` das propriedades comportamentais.

A cada execução de qualquer comando de um dos objetos sob investigação o código instrumentado realiza em paralelo a captura das informações instantâneas referentes à execução daquele respectivo comando. Estas informações são capturadas para serem posteriormente usadas para a análise da corretude comportamental do sistema, conforme será apresentado na Seção 3.4.

A captura das informações durante a execução do sistema é feita localmente, em cada máquina do sistema distribuído. Cada máquina componente da rede do sistema distribuído contém um monitor local é responsável por capturar as informações referentes aos objetos a serem monitorados. A ocorrência de uma ação sobre um destes objetos do sistema distribuído é chamada de **evento**.

Após a captura dos eventos na máquina local, cada um dos eventos é modificado de forma a agregar informações que serão ordenados. E em seguida, os eventos são enviados remotamente para um monitor central. Este envio é necessário para termos uma visão global da execução do sistema distribuído. A Figura 3.4 ilustra o processo da captura dos eventos e o envio remoto para o monitor central.

Um evento é composto de informações referentes ao conjunto de dados que es-

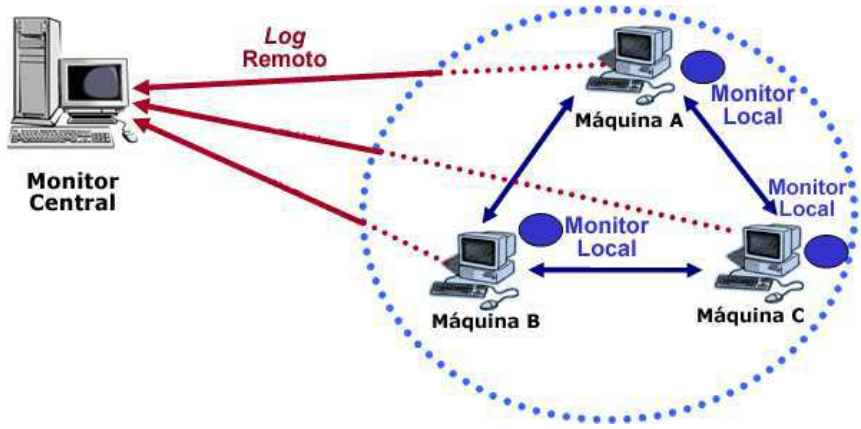

Figura 3.4: Captura e envio remoto de eventos em um Sistema Distribuído.

tão disponíveis e acessíveis aos componentes do sistema em execução. Como a preocupação principal é acompanhar a execução das *threads* sobre os objetos compartilhados captura-se informações como `Nome do Aspecto`, `Nome do Objeto`, `Instância do Objeto`, `Nome da Thread`, `Instância da Thread`, `Nome do método`, `Linha do Método`, `Host local`, `Host do Servidor Central`, `Ordem do evento`. Este conjunto de informações é necessário para utilizarmos como informação decisiva na análise da execução correta dos componentes do sistema distribuído que estão sendo verificadas.

Uma informação importante é o número que indica qual a `Ordem do evento` de acordo com a execução global do sistema distribuído. Este valor é obtido a partir do uso do algoritmo dos Relógios Lógicos [Lamport, 1978; Lamport, 1979]. Os detalhes do algoritmo utilizado na captura dos eventos de um sistema distribuído são exibidos na Seção 3.3.1.

3.3 Organização dos Dados Coletados

Alguns algoritmos foram desenvolvidos com o intuito de construir um conjunto de informações que pudessem ser usadas para inferir o comportamento durante a execução de sistemas distribuídos. Um dos algoritmos pensados para o entendimento do comportamento de tais sistemas foi o algoritmo dos *Snapshots* [Chandy and Lamport, 1985], que utiliza o conceito de tirar "fotos" instantâneas do sistema e depois montar uma conjunção de quadros mostrando como o sistema evoluiu durante a sua execução. Este algoritmo foi modelado para verificarmos a possibilidade do uso do mesmo para inferir o comportamento global de

um sistema distribuído, mas viu-se que se tornaria inviável para a capacidade computacional atualmente utilizada. No Anexo A temos uma explicação mais detalhada do algoritmo e a descrição da modelagem deste sistema, que serviu como tomada de decisão para a utilização de outro algoritmo.

Para possibilitar a verificação ou análise consistente de um sistema distribuído precisamos inferir uma linha de execução consistente, ou seja, que se assemelhe à execução real daquele sistema. A construção de uma linha de execução consistente diz respeito à necessidade de realizarmos uma ordenação global dos eventos do sistema distribuído, possibilitando uma contextualização geral, e viabilizando a análise comportamental de um componente de acordo com toda a execução do sistema [Babaoglu and Marzullo, 1993].

Na Seção 3.3.1 temos a descrição do algoritmo utilizado para a inferência do comportamento de um sistema distribuído, levando em consideração a ordenação global dos eventos por ele executados.

3.3.1 Inferência da Ordenação dos Eventos em um Sistema Distribuído

Como falado anteriormente, para acompanharmos a execução de todo um sistema distribuído é necessário um artifício auxiliar que viabilize a união dos eventos de todos os componentes espalhados pelo sistema de forma a termos uma visão global consistente da execução do mesmo.

A existência de várias máquinas trabalhando em conjunto para resolver um determinado problema também insere a característica de assincronismo para os sistemas distribuídos. Por não existir o conceito de um relógio global compartilhado com todos os componentes do sistema, o conjunto de processos executados em máquinas diferentes não é naturalmente ordenado quanto ao instante da sua ocorrência.

A estratégia adotada para contornar o assincronismo de sistemas distribuídos foi o algoritmo chamado de Relógios Lógicos [Lamport, 1978], que realiza uma correlação de tempo para cada evento no sistema distribuído entre os vários processos espalhados pelas várias máquinas de um sistema. Os detalhes de funcionamento do algoritmo puro sugerido por Lamport estão descritos na Seção 2.4. Nesta seção descrevemos como o algoritmo dos Relógios Lógicos foi inserido na nossa técnica de monitoração de sistemas distribuídos.

Durante a execução do sistema distribuído a ser monitorado, cada monitor local mantém

uma variável chamada `relógio lógico (RL)`, que será a variável que terá o controle da temporização dos processos de uma determinada máquina do sistema. Cada evento proveniente desta máquina é marcado com o valor atual contido na variável `RL`. Após esta marcação, o valor desta variável é incrementado, construindo uma idéia de ordenação temporal dos eventos da máquina. A Figura 3.5 ilustra a idéia de um relógio lógico para cada Monitor Local no sistema distribuído.

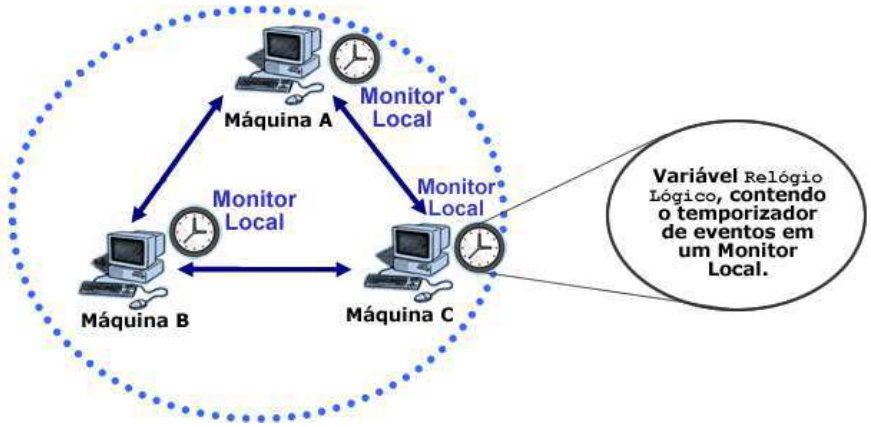

Figura 3.5: Monitores Locais contendo um Relógio Lógico local.

No entanto, só a marcação dos eventos locais em cada máquina não viabiliza uma relação de ordem global entre todos os eventos do sistema distribuído. A correlação entre o funcionamento das variáveis `RL` de cada máquina é realizada quando ocorrem a troca de eventos entre as máquinas, ou seja, é realizada o envio e/ou recebimento de mensagens entre os processos de um sistema distribuído. Usando o algoritmo de Relógios Lógicos, cada mensagem enviada é marcada com o instante que a mesma ocorreu, instante referente ao valor atual da variável `RL` na máquina que envia a mensagem. Quando a mensagem é recebida pelo destinatário, o monitor local faz a comparação entre o *timestamp* da mensagem recebida com o valor atual contido na variável `RL` local, atualizando a variável para o valor maior incrementado de um.

Com a organização de todos os eventos de um sistema distribuído sendo realizada com este algoritmo, o resultado é um conjunto de eventos ordenados parcialmente. A ordenação é parcial por considerar somente a relação de envio e recebimento de mensagens entre os processos do sistema. Os eventos independentes, aqueles que não têm uma relação de ordem e que podem ocorrer paralelamente, não precisam ser considerados na ordenação global do

sistema.

Então, após a centralização do conjunto de eventos ordenados parcialmente, podemos utilizá-lo como base de informação consistente para realizarmos a análise do comportamento do sistema distribuído durante a sua execução, analisando-o quanto às suas especificações comportamentais.

3.3.2 Centralização dos Dados Gerados durante Monitoração

Como podemos ver na Figura 3.6, após a captura local dos eventos, cada evento é enviado remotamente para o `Monitor Central`, que será responsável por centralizar todos os eventos do sistema distribuído, organizando a informação a ser utilizada durante a análise comportamental.

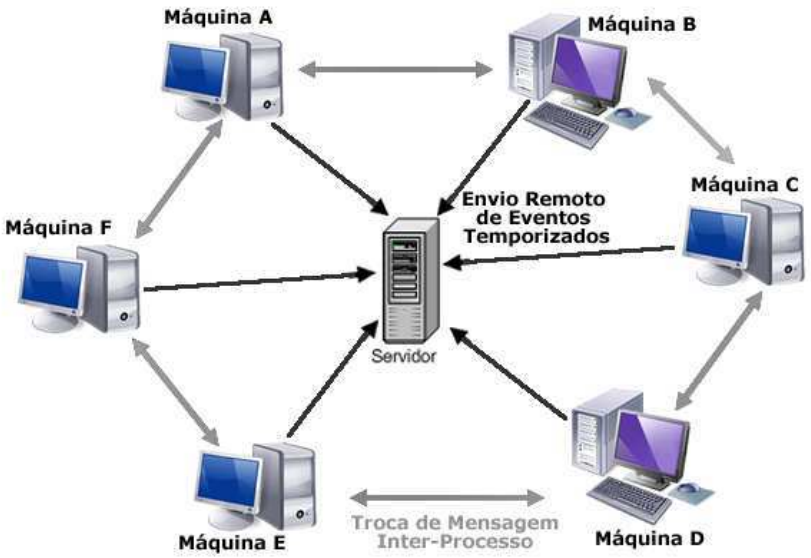

Figura 3.6: Centralização dos eventos gerados por um Sistema Distribuído.

Como todos os eventos estão marcados com o *timestamp* referente à ordem em que cada evento ocorreu, levando em consideração a ordenação parcial entre os eventos inter-processo, então a ordenação é realizada tomando como referência o valor da marcação temporal de cada evento.

De posse de todos os eventos gerados pelo sistema distribuído, o `Monitor Central` realiza a ordenação de todos os eventos recebidos até aquele momento, montando um conjunto de eventos ordenados pelo critério de instante de ocorrência. Este conjunto ordenado

será usado como entrada para o módulo de Análise Comportamental, a ser analisada de acordo com as especificações do sistema.

3.4 Análise da Execução do Sistema Distribuído

Continuando o processo da técnica de monitoração, após a captura e centralização dos eventos descritas nas Seções 3.3.1 e 3.3.2 respectivamente, o sistema distribuído começa a ser analisado pelo conjunto de eventos que apresentou durante sua execução.

O conjunto de eventos ordenados é analisado de acordo com as especificações descritas para o sistema sob verificação. Cada evento é analisado de acordo com a(s) propriedade(s) que o mesmo deve obedecer. No Monitor Central, para cada propriedade comportamental existe um código de análise comportamental correspondente, baseado em autômatos de Büchi.

Os eventos são passados para seus respectivos códigos de análise, códigos que são representados por máquinas de estados. A respectiva máquina de estados então executa e analisa o evento recebido como entrada. Se aquele evento é um evento esperado, então a máquina passa para o próximo estado e continua o seu processamento. Se aquele evento não é esperado naquele momento de execução, então a máquina de estados vai para o estado inválido, identificando assim que o sistema distribuído apresentou um comportamento inesperado. Este processamento é ilustrado na Figura 3.7.

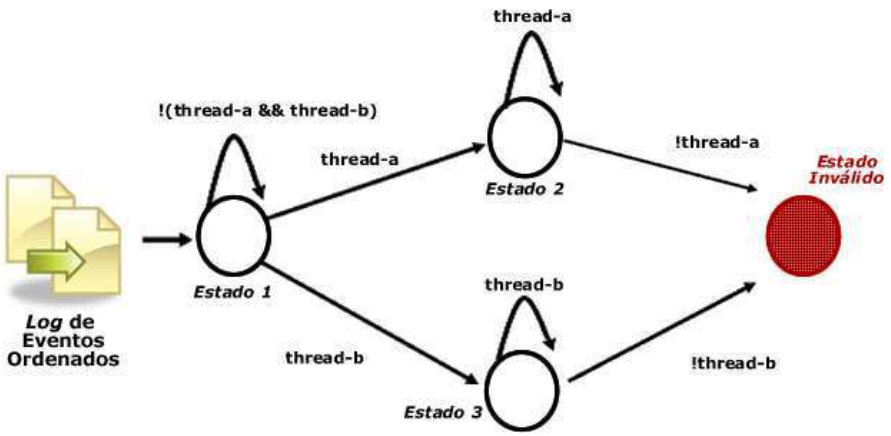

Figura 3.7: Máquina de estados realizando a análise comportamental dos eventos do sistema.

Após a detecção de um comportamento inesperado uma notificação é lançada para o

desenvolvedor, mostrando qual comportamento inesperado foi detectado.

Capítulo 4

A Ferramenta *DistributedDesignMonitor*

Neste capítulo mostramos como o *DistributedDesignMonitor* (DDM) foi implementado, detalhando cada módulo do sistema, que foi projetado e desenvolvido com o intuito de atender às necessidades de monitoração de sistemas distribuídos e concorrentes.

4.1 Visão Geral

A ferramenta DDM é um Sistema Distribuído que realiza a monitoração do comportamento de outro sistema sob investigação. Monitorar um sistema significa acompanhar a execução do mesmo, verificando se o comportamento por ele descrito está de acordo com o comportamento especificado. Para tanto, precisamos inserir código de monitoração em cada máquina de um sistema distribuído, acompanhando e capturando as informações em tempo de execução. Como podemos ver, a Figura 4.1 ilustra como o DDM está organizado, separado em módulos que realizam a monitoração de sistemas distribuídos.

Baseados na especificação comportamental são executados os seguintes módulos: Gerador de Código (1 e 2), o Observador (3, 4 e 5) e o de Analisador (6), que são partes integrantes do DDM e realizam automaticamente as responsabilidades delegadas para cada um.

A Geração de Código constrói automaticamente o Código de Monitoração, que produz o código de captura de informações e o código de análise comportamental. O primeiro é composto de código aspecto que captura as informações de interesse durante a execução do sistema distribuído (1). O segundo é composto por máquinas de estados, sendo cada máquina

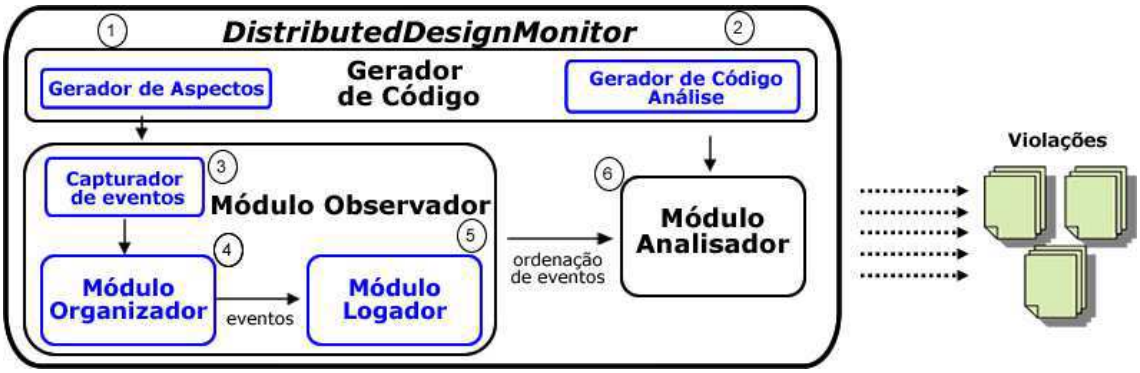

Figura 4.1: Arquitetura da ferramenta *DistributedDesignMonitor* (DDM)

referente a uma especificação comportamental, que serão executadas para a verificação da corretude comportamental do sistema (2).

Antes de iniciar a monitoração propriamente dita, é necessária a inserção do código de captura, o conjunto de código(s) aspecto(s), junto ao código do sistema distribuído a ser monitorado, formando o Capturador de Eventos (3). Isto implica na produção do código fonte do sistema distribuído com o código de monitoração, compilados em conjunto, possibilitando a captura de informações durante a execução do sistema. A captura dos eventos ativa a organização destes, sendo realizada pelo Módulo Organizador em (4). Em seguida estes eventos são enviados para um monitor central através do Módulo Logador (5), onde será realizada toda a análise sobre todos os eventos capturados.

Depois do início da execução do DDM e posteriormente do sistema distribuído sob monitoração, começa a Análise Comportamental. Com a captura dos eventos gerados pelo sistema e a organização da ordem de ocorrência destes, passamos então cada evento para a respectiva máquina de estados, quando é verificada a consistência de cada um pelo Módulo Analisador (6). Se algum evento desconhecido ou inesperado acontecer em um determinado instante, então é lançada a informação da ocorrência da inconsistência para o desenvolvedor. Esta informação vai ajudar a equipe de desenvolvimento a entender o que aconteceu de errado e localizar a parte do código que gerou a inconsistência.

Existe uma interação contínua entre os módulos do DDM, viabilizando a monitoração de sistemas distribuídos. O módulo de Geração de Código de Monitoração produz a informações que serão utilizadas tanto pelo Módulo de Captura de Eventos quanto pelo Módulo de Análise Comportamental. Este último ainda faz uso de informações geradas pelo Módulo

de captura, para poder fazer a comparação entre a execução do sistema e o comportamento desejado para o mesmo.

Nas próximas seções temos detalhes da implementação de cada um dos módulos, como estes interagem entre si trabalhando em conjunto para realizar a monitoração em tempo de execução.

4.2 Geração Automática de Código de Monitoração

Este módulo é responsável pela geração dos códigos usados na monitoração. Tomemos como base o exemplo utilizado para a descrição de uma propriedade comportamental definida na Seção 3.2.1.2. Considere então a propriedade comportamental descrita no Código 4.1. Esta propriedade será usada como exemplo para o entendimento da geração automática dos códigos de monitoração do DDM.

```
label : BehaviorPropertyModuleA;
rule : (!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB);
points : ObjectA;
```

Código 4.1: Exemplo de Propriedade Comportamental definida previamente no Exemplo 3.3.

A partir da especificação das propriedades comportamentais o DDM gera os códigos de Captura de Eventos e o Código de Análise Comportamental, que são detalhados nas seções posteriores.

4.2.1 Geração do Código de Análise Comportamental

Considere o campo `rule` da propriedade comportamental definida no Código 4.1. A especificação de ações das threads `ThreadA` e `ThreadB` é descrita através da fórmula `(!ThreadA && !ThreadB) U ([]ThreadA || []ThreadB)`, que dará origem ao código de análise comportamental.

A ferramenta DDM faz uso de um framework que tem como propósito a conversão de fórmulas LTL em autômatos de Büchi equivalentes. O LTL2BA4J [Bodden, 2005] é um framework que faz a conversão das regras descritas no campo `rule` para autômatos na

linguagem de programação Java, a linguagem utilizada no desenvolvimento do DDM e dos sistemas distribuídos a serem monitorados.

Após a obtenção do autômato equivalente à fórmula LTL, reorganizamos o autômato, desconsiderando a existência de estados finais, e inserindo um estado chamado de *estado inválido*. Cada estado previamente existente terá uma transição que o ligará ao *estado inválido*. Esta modificação é o ponto crucial para a detecção de um evento que ocorreu em um instante inadequado. Esta descrição de conversões de fórmula LTL em autômato de Büchi e posteriormente em máquina de estados é ilustrada na Figura 4.2.

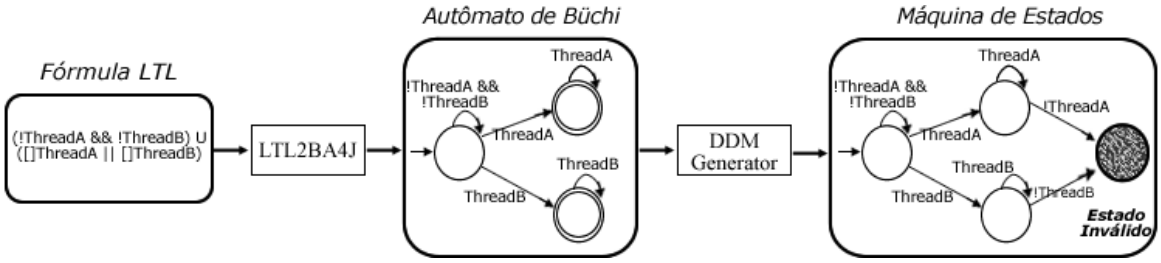

Figura 4.2: Transformação de fórmulas LTL em máquina de estados equivalentes.

Outra forma de especificar o comportamento desejado para um sistema distribuído é montando diretamente o autômato que deve executar corretamente quando receber os eventos gerados pelo sistema distribuído. Esta alternativa de especificação elimina uma das etapas de conversão descritas anteriormente, minimizando assim possíveis perdas de detalhes comportamentais.

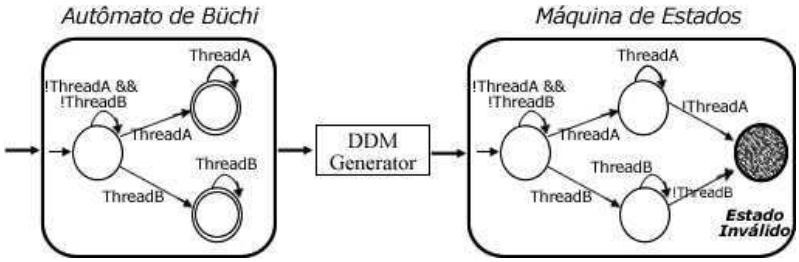

Figura 4.3: Transformação de Autômato de Büchi em máquina de estados equivalente.

A Figura 4.3 ilustra como se dá o processo da transformação de forma direta de autômato para a máquina de estados, máquina que será executada para reportar possíveis inconsistências comportamentais.

4.2.2 Geração do Código Aspecto

Ainda baseando-se na propriedade comportamental descrita pelo Código 4.1, passamos agora a entender como acontece a geração automática do código de Captura de Eventos.

A captura de eventos acontece sobre os pontos de interesse do Sistema Distribuído. Esses pontos devem obedecer à regra definida na respectiva propriedade comportamental onde os mesmos foram indicados, ou seja, existe uma relação `um-para-vários` na propriedade comportamental. Cada regra de uma propriedade comportamental pode ser descrita para um ou vários pontos de interesse do sistema.

Cada objeto definido como ponto de interesse em uma propriedade comportamental deve ter o seu nome completo [1] definido no campo `point` da propriedade.

Na propriedade `BehaviorPropertyModuleA` temos o campo `points`[2] definido genericamente pelo objeto `ObjectA`. Para iniciar a geração do código de captura, o(s) ponto(s) definido(s) em cada propriedade comportamental é(são) extraído(s), sendo utilizado(s) como para a construção dos respectivos código aspecto.

A criação do código aspecto é realizada a partir de um código *template* que define a estrutura padrão para as classes aspectos que serão executadas durante a captura das informações de cada ponto de interesse. O Código 4.2 mostra como foi definido este *template*.

Cada aspecto é gerada para um pacote comum `aspects` (definido na linha 1 Código 4.2), pacote este ao qual o aspecto produzida deve ser inserido no código fonte do sistema a ser monitorado. O nome do aspecto `#AspectClassName#` é definido a partir do nome da propriedade comportamental, definido no campo `label` adicionando o nome Aspect, alterando então as linhas 3 e 5. O padrão `#RuleLabel#` na linha 7 é removido e no seu local é inserido o nome da propriedade comportamental sem modificações. Este passo deve ser enfatizado, pois a definição do nome do aspecto igual ao nome da propriedade comportamental é o que possibilita a interligação da informação capturada por um determinado aspecto à propriedade que o evento capturado deve obedecer. Nas linhas 13 a 15 devem ser definidos os os devidos pontos de interesse, dos quais serão capturadas as informações em tempo de execução. O padrão `#monitoredPoint#` é então sub-

[1] Na linguagem Java o nome completo de um objeto é definido pelo nome da classe que o define junto com todo o caminho de pacotes que a classe faz parte.

[2] Campo de especificação dos objetos a serem monitorados.

```
1  package aspects;
2  import br.edu.ufcg.designmonitor.client.aspects.*;
3  public aspect #AspectClassName# extends DistributedDesignMonitorAspect {
4       public String nameAspect;
5       public #AspectClassName#() {
6            super();
7            setNameAspect("#RuleLabel#");
8       }
9       //Ponto de monitoração do sistema
10      //Para mais de um ponto a ser monitorado,
11      //adicionar após execution a seguinte linha:
12      //&& execution(* #ponto a ser monitorado#)
13      public pointcut monitoredPoints()
14      :   execution(* #monitoredPoint#.*(..))
15           && !cflow(execution(java.lang.String *.toString()));
16      public void setNameAspect(String newName){
17           this.nameAspect = newName;
18      }
19      public String getNameAspect(){
20           return this.nameAspect;
21      }
22 }
```

Código 4.2: *Template* do código aspecto utilizado para a geração automática das classes aspectos com os pontos de monitoração.

stituído pelo nome do ponto definido no campo `points` da propriedade comportamental `BehaviorPropertyModuleA` definida no Código 4.1.

Utilizando a propriedade `BehaviorPropertyModuleA` e o *template* definido no Código 4.2 é gerado automaticamente o código aspecto que realizará a captura de informações para a realização da análise comportamental desta regra. O Código 4.3 é o resultado desta geração automática. O código referente ao aspecto `DistributedDesignMonitorAspect` foi apresentado na Figura 3.5.

```
1  package aspects;
2  import br.edu.ufcg.designmonitor.client.aspects.*;
3  public aspect BehaviorPropertyModuleAAspect extends
4                 DistributedDesignMonitorAspect {
5       public String nameAspect;
6       public BehaviorPropertyModuleAAspect () {
7            super();
8            setNameAspect("BehaviorPropertyModuleA");
9       }
10      public pointcut monitoringPoints()
11      :   execution(* ObjectA.*(..))
12           && !cflow(execution(java.lang.String *.toString()));
13      public void setNameAspect(String newName){
14           this.nameAspect = newName;
15      }
16      public String getNameAspect(){
17           return this.nameAspect;
18      }
19 }
```

Código 4.3: Código aspecto gerado a partir da propriedade comportamental `BehaviorPropertyModuleA` definida no Código 4.1.

4.2.3 Módulo de Observação e Organização

Após a geração automática do código de monitoração inicia-se o processo de execução, realizando a captura e organização dos eventos de um sistema distribuído.

Depois da inserção das classes geradas é necessária a compilação destas junto com as classes do sistema distribuído. A Figura 4.4 ilustra como isso acontece.

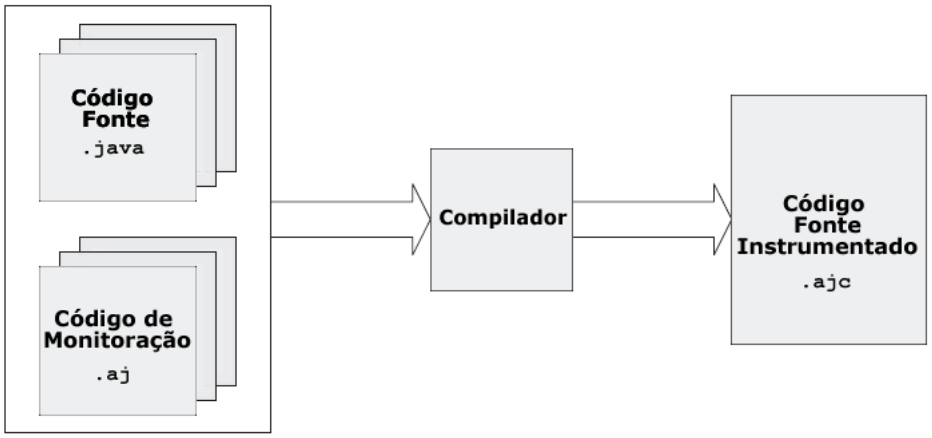

Figura 4.4: Compilação do código Java e AspectJ produzindo o código fonte instrumentado.

O *bytecode* gerado da instrumentação do código fonte original com as classes aspecto produzidas pelo DDM devem ser introduzidas em todas as máquinas que fazem parte da execução do sistema distribuído. Para a identificação das máquinas integrantes do sistema precisamos definir para o DDM através de um arquivo de configurações, quais são estas máquinas, sendo estas nomeadas e identificadas na rede interna do sistema distribuído. O código de monitoração inserido em cada máquina é chamado de Monitor Local.

Além da definição de cada máquina, é preciso também definir qual máquina será a responsável por centralizar a informação trocada durante toda a execução. No DDM chamamos esta máquina de Monitor Central, a máquina que receberá remotamente o conjunto de eventos marcados com o instante temporal de cada maquina integrante da execução. A partir deste instante começa a execução dos sistemas e conseqüentemente o ato da monitoração.

4.2.3.1 Observando o sistema Monitorado

Durante a execução do sistema distribuído sob investigação, o *bytecode* instrumentado é executado e à medida que executa as tarefas requisitos do sistema em desenvolvimento, ele também realiza a captura das informações dos pontos de interesse necessárias para a monitoração. Esta captura é feita em cada máquina do sistema, e no instante do acontecimento deste evento é anexado ao mesmo o valor equivalente ao instante temporal, valor este extraído da variável `Relógio Lógico` contida em cada máquina informada como integrante do sistema.

Quando acontecer troca de mensagens entre máquinas do sistema distribuído cada Monitor Local executa o Código 4.4, realizando a demarcação temporal do evento (na linha 10) e a atualização do relógio lógico local de cada máquina.

```
1  package br.edu.ufcg.designmonitor.client.monitor;
2  ...
3  public class RemoteMonitorImpl ... {
4  ...
5       public void sendEvent(Event event) throws RemoteException {
6            synchronized (logicalClock) {
7                 incrementClock();
8            }
9            long time = getTime();
10           Event timedMesg = timestamp(event);
11           String host = event.getHostName();

12           //Envia evento para o monitor da respectiva máquina
13           ...
14           //Envia remotamente o evento temporizado para
15           //o Monitor Central
16           DesignMonitorLogger.getInstance().log(timedMesg.toString());
17      }

18      public void receivedEvent(Event event, long timestamp)
19                     throws RemoteException {
20           synchronized (logicalClock) {
21                updateClock(timestamp);
22           }
23      }
24 ...
25 }
```

Código 4.4: *Template* do código aspecto utilizado para a geração automática das classes aspectos com os pontos de monitoração.

Com a captura do evento e posteriormente a temporização deste evento, é realizado o envio remoto deste evento temporizado para o Monitor Central, que começa a receber os eventos executados por cada máquina e, utilizando um algoritmo de ordenação, passa a organizar a informação aleatória que recebe durante a execução.

A Figura 4.5 ilustra como acontece o processo de captura e marcação dada a ocorrência de cada evento do sistema distribuído monitorado. Cada evento Xe_i caracteriza um evento originado pela máquina X no instante *i*.

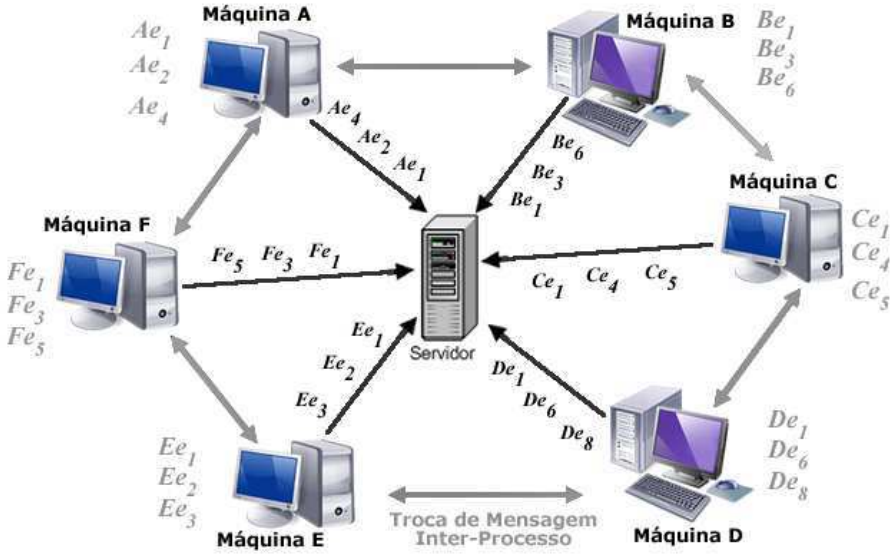

Figura 4.5: Centralização dos eventos temporizados de um Sistema Distribuído.

4.2.3.2 Ordenação dos Eventos Capturados

A organização dos eventos capturados é realizada pela máquina definida como Monitor Central à medida que este recebe remotamente os eventos do sistema. Esta organização é realizada ordenando-se os eventos de acordo com o instante do tempo em que este foi demarcado por cada Monitor Local nas máquinas do sistema.

Durante o recebimento remoto de cada evento pelo Monitor Central, este armazena os eventos em uma estrutura já de forma ordenada, o que agiliza o processo de ordenação por inserção de cada evento na coleção.

Após a captura e ordenação de uma parte dos eventos começa a etapa de análise comportamental, que acontece em paralelo com a execução do sistema. Mais detalhes como a análise é realizada são apresentados na Seção 4.2.4 a seguir.

4.2.4 Módulo de Análise Comportamental

De posse do conjunto de eventos ordenados do sistema distribuído em execução e utilizando o Código de Análise, gerado automaticamente e apresentado na Seção 4.2.1, começamos de fato a etapa de análise comportamental.

Como a análise comportamental é feita no mesmo instante da execução do sistema distribuído sob investigação, então utilizamos o algoritmo de janela deslizante [Harada, 2003]

para começar a analisar um intervalo de eventos válidos e consistentemente ordenados de acordo com a execução do sistema monitorado.

A janela deslizante possibilita que, se algum evento for enviado para o Monitor Central com atraso, o mesmo possa ser recebido e ainda assim o DDM poderá realizar a sua análise consistentemente.

Os eventos são então passados, um a um, para a execução da máquina de estados referente ao comportamento desejado para cada evento. Os eventos são entrada para a execução da respectiva máquina de estados que descreve como o conjunto de eventos possíveis pode interagir entre si.

Durante a execução do código de análise comportamental, se alguma máquina de estados alcançar o estado inválido, ilustrado na Figura 4.2 apresentada, então dizemos que uma violação comportamental ocorreu.

4.3 Considerações Finais

Neste capítulo, apresentamos a arquitetura e as principais características da implementação do protótipo da ferramenta *DistributedDesignMonitor*. Vimos como sua arquitetura foi definida a partir da técnica apresentada no Capítulo 3. A entrada dos módulos de verificação e análise são as propriedades comportamentais especificadas conforme descrito. Como visto, cada propriedade comportamental é formada pela relação entre os pontos de interesse e a especificação do comportamento. Os pontos de interesse dão origem ao código de monitoração e a especificação do comportamento através de uma fórmula LTL dá origem ao código de análise comportamental.

O módulo de geração automática de código de monitoração produz os códigos que serão usados para avaliação do sistema. A captura de eventos e a implementação do algoritmo de ordenação dos Relógios Lógicos definido por Lamport em [Lamport, 1978] possibilita a marcação dos eventos para posterior ordenação parcial dos mesmos. De base na captura de eventos, o módulo de Análise Comportamental inicia sua execução, analisando a corretude de cada evento, atestando o comportamento coerente ou informando quando da ocorrência de uma violação comportamental.

Capítulo 5

Estudo de Caso

Neste capítulo temos a apresentação da avaliação dos resultados obtidos a partir da execução dos estudos de caso realizados para a ferramenta *DistributedDesignMonitor* (DDM). Inicialmente precisamos entender a especificação de cada sistema distribuído analisado. Posteriormente passamos a descrição do comportamento de cada sistema, definindo propriedades comportamentais. Por último temos os cenários de monitoração e os resultados obtidos durante a análise do comportamento de cada um dos sistemas investigados.

5.1 Preâmbulo

A avaliação do DDM precisa ser feita realizando-se a monitoração de alguns sistemas distribuídos. No entanto, não é qualquer sistema que pode ser utilizado para utilizarmos na monitoração e avaliação do DDM.

Um dos sistemas distribuídos selecionados é o OurGrid [OurGrid,], um *middleware* para a execução de aplicações *Bag-Of-Task* (BoT)[1] em grades computacionais [Foster and Kesselman, 1999]. Antes de especificarmos o sistema OurGrid, vamos entender e avaliar um sistema amplamente conhecido no contexto de sistemas concorrentes. O problema do Jantar dos Filósofos é uma forma de entendermos como acontece a interação entre processos executando em paralelo [Tanenbaum and van Steen, 2007].

Utilizamos uma implementação deste problema, trazendo-o para o contexto também de sistemas distribuídos, para podermos avaliá-lo. Após a avaliação deste sistema passaremos a

[1] Conjunto de tarefas que podem ser executadas de forma independente, em paralelo.

entender o OurGrid e avaliar seu comportamento.

5.2 O Jantar dos Filósofos

O Jantar dos Filósofos constitui em um conjunto de filósofos ao redor de uma mesa de jantar, que podem estar pensando ou comendo. Cada filósofo tem a sua disposição um prato de macarrão, no entanto, compartilha os talheres ou rachi (pauzinhos japoneses) com os filósofos adjacentes para se alimentarem. Para comer, cada filósofo poder ter acesso aos talheres somente se os dois filósofos adjacentes não estiverem comendo. A Figura 5.1 ilustra a situação do jantar dos filósofos descrita.

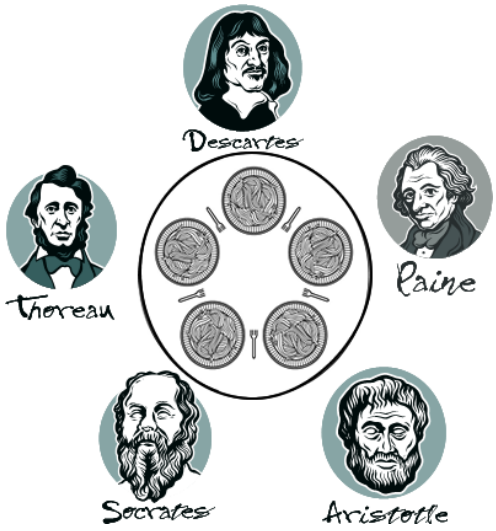

Figura 5.1: O problema do Jantar dos Filósofos.

Cada filósofo é modelado como um processo concorrente que executa aleatoriamente, tentando pegar os talheres, se estiver disponíveis, para poder comer.

Como a interação entre filósofos e talheres é aleatória, é necessária a verificação comportamental de cada filósofo sobre os talheres que são compartilhados. Antes de iniciar a monitoração deste problema, implementamos uma versão do jantar dos filósofos como sistema concorrente e distribuído, onde cada filósofo está em uma máquina diferente de uma rede interna, onde o jantar está sendo executado.

Passamos agora para a especificação do comportamento do jantar dos filósofos distribuídos. Com o intuito de garantir a corretude comportamental do jantar precisamos verificar se

dois filósofos *estão acessando ao mesmo tempo um mesmo talher*. Esta descrição de comportamento em linguagem natural pode ser transcrita para uma especificação de propriedade comportamental, como descrita genericamente no Código 5.1.

```
label : PhilosopherProperty;
rule : ! Philosopher(i)_pegarTalher U  Philosopher(i + 1)_soltarTalher;
point: philosopher.distributed.DistributedChopstickImpl01
```

Código 5.1: Propriedade Comportamental do Jantar dos Filósofos.

A partir desta propriedade comportamental iniciamos o processo de monitoração do Jantar dos Filósofos. O DDM usa então esta propriedade para gerar de forma automática os códigos de monitoração para este sistema.

Primeiramente, a regra definida no campo `rule` é convertida em uma máquina de estados equivalente. Em paralelo, o ponto de interesse definido no campo `point` é identificado e com ele é gerado o código aspecto que será utilizado para a captura de informações em tempo de execução. O Código 5.2 contém o código aspecto gerado pelo DDM a partir da propriedade `PhilosopherProperty`.

Depois da criação dos códigos de monitoração, inserimos o código de captura de eventos junto do sistema que implementa o Jantar dos Filósofos, compilando os dois códigos e gerando o código fonte instrumentado. Inicia-se então a execução do DDM e posteriormente o Jantar dos Filósofos. Enquanto o jantar acontece, o DDM realiza a captura de eventos e analisa um a um os eventos passados como interesse para a monitoração do mesmo. Se alguma violação comportamental for detectada, a mesma é reportada para o desenvolvedor através de um arquivo de *log*, como mostrado no Código 5.3.

Como o Jantar dos Filósofos não apresenta uma complexidade alta para elaborarmos muitas propriedades comportamentais, nos concentramos na verificação do instante de cada processamento entre os objetos que compõem o conceito do jantar, como o acesso concorrente a um garfo. Então, para cada garfo realizamos a monitoração dos objetos que o acessavam. Durante a monitoração detectamos violações comportamentais no instante em que os acessos eram realizados. Para cada violação, um código semelhante ao Código 5.3 será gerado.

A identificação de violações comportamentais da nossa implementação distribuída para o Jantar dos Filósofos nos apontou para o fato da implementação não estar correta quanto ao

```
1  package aspects;
2  import br.edu.ufcg.designmonitor.client.aspects.*;
3  public aspect DistributedChopstickImpl01Aspect
4            extends DistributedDesignMonitorAspect {
5       public String nameAspect;
6       public DistributedChopstickImpl01Aspect() {
7            super();
8            setNameAspect("PhilosopherProperty");
9       }
10      public pointcut monitoringPoints()
11      :execution(* philosopher.distributed.DistributedChopstickImpl01.*(..))
12        && !cflow(execution(java.lang.String *.toString()));
13      public void setNameAspect(String newName){
14           this.nameAspect = newName;
15      }
16      public String getNameAspect(){
17           return this.nameAspect;
18      }
19 }
```

Código 5.2: Código de monitoração da propriedade comportamental `PhilosopherProperty`.

```
RULE VIOLATED: PhilosopherProperty
Expected: (!distributedphilosopherimpl03_soltarTalher_distributedchopstickimpl02
     && !distributedphilosopherimpl03_pegarTalher_distributedchopstickimpl02
     && distributedphilosopherimpl02_soltarTalher_distributedchopstickimpl02)
But was:  distributedphilosopherimpl03_pegarTalher_distributedchopstickimpl02
Object: DistributedChopstickImpl02@3ecfff#cavaquinho/150.165.98.59
Method: pegarTalher(DistributedPhilosopherImpl03)
Method line:15
Time: 6
```

Código 5.3: Notificação de Violação de Comportamento do Jantar dos Filósofos, produzido para um arquivo de *log*.

comportamento desejado, mostrando assim a importância do uso da técnica para a detecção de comportamentos anômalos.

5.3 Projeto *OurGrid*

O software *OurGrid* é uma plataforma para a execução de aplicações *Bag-Of-Task* (BoT). Aplicações BoT são aquelas que podem ser divididas em sub-tarefas independentes, que podem ser executadas paralelamente em computadores diferentes. Com isso, o tempo de processamento da aplicação é reduzido, aumentando conseqüentemente o poder computacional. Esses tipos de aplicações são utilizadas em diversos cenários, incluindo mineração de dados, processamento de imagens, buscas exaustivas, biologia computacional, dentre outros. O objetivo principal do software *OurGrid* é oferecer para o usuário de sua comunidade uma grade computacional aberta. Essa grade é baseada na troca de favores entre seus integrantes, com o compartilhamento de recursos computacionais no processamento de aplicações de grande escala que podem ser subdivididas em tarefas [Cirne et al., 2004]. O *OurGrid* está sendo desenvolvido em Java desde 2001 no Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG), em parceria com a empresa *Hewlett Packard* (HP). A versão do *OurGrid* considerada neste trabalho é a 3.3.1 [2].

Uma visão geral da solução *OurGrid* é ilustrada na Figura 5.2 [OurGrid,]. O *OurGrid* é composto por três componentes principais: *MyGrid*, *Peer*, e *UserAgent*:

- ***MyGrid*** - interface entre o usuário e a grade. O *MyGrid* e o componente responsável por receber dos usuários as solicitações para execução das tarefas, solicitar recursos disponíveis na grade e gerenciar o escalonamento das tarefas entre as unidades de processamento distribuídas pela rede. A máquina onde o *MyGrid* é executado é chamada de *home machine*, que é o ponto central de uma grade. O *MyGrid* é o responsável por distribuir as tarefas para as máquinas disponíveis na grade e, por fim, unir os resultados obtidos pela execução das tarefas e apresentar o resultado final ao usuário solicitante;

- ***Peer*** - é o componente que identifica quais são as máquinas (que pertencem ao seu domínio administrativo) disponíveis e como elas poderão ser utilizadas pelo *MyGrid*

[2] http://www.ourgrid.org/

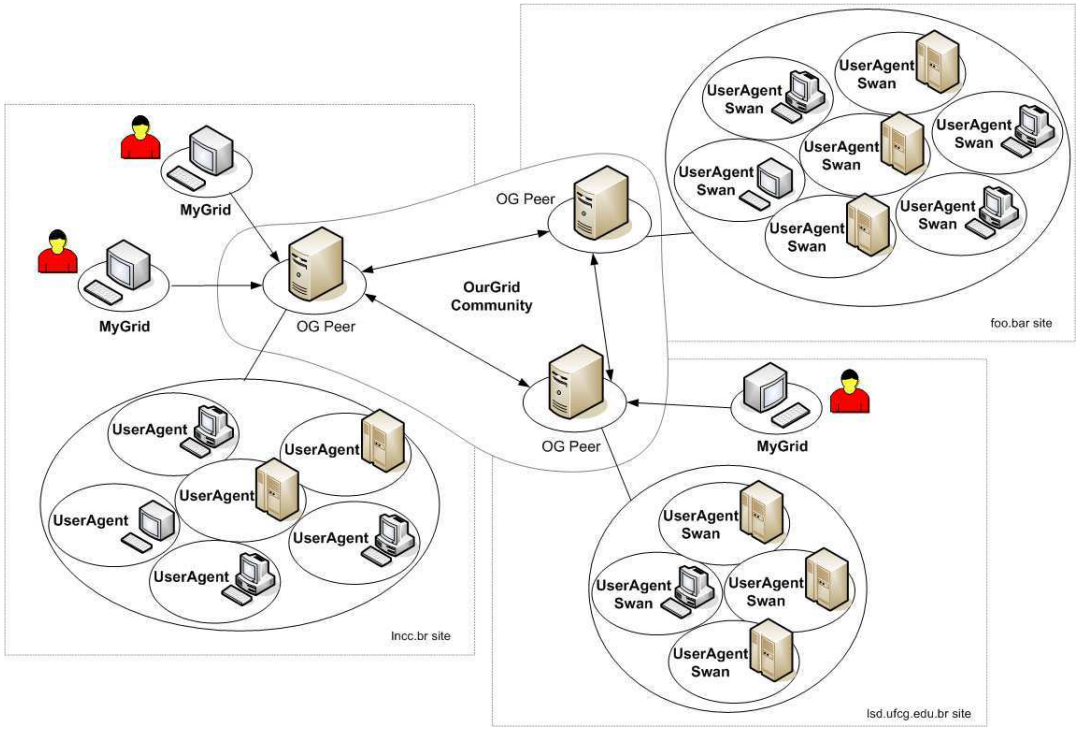

Figura 5.2: Visão geral da solução *OurGrid*.

para executar tarefas;

- ***UserAgent*** - componente responsável por executar as tarefas na grade. As máquinas que possuem o *UserAgent* instalado são chamadas de *gums*, ou seja, são as máquinas reais que executam as tarefas no *OurGrid*. Elas fornecem as funcionalidades necessárias para a comunicação entre a *home machine* e as *gums*.

Ao longo da evolução do software *OurGrid* chegou-se a um ponto em que entender, modificar, manter e evoluir tornavam-se tarefam cada vez mais difíceis. Os usuários e desenvolvedores queixavam-se de falhas no funcionamento do *OurGrid*, as quais não se conseguia identificar a causa. Após avaliação, se constatou que a implementação havia divergido do projeto (*design*) esperado. Diante desse quadro, a solução adotada foi a de se refazer o projeto do *OurGrid*, mas mantendo a idéia da solução *OurGrid* original. Adicionalmente, foram definidas algumas propriedades estruturais e comportamentais que os desenvoldores devem obedecer durante a implementação do software. Entretanto, a atividade de detecção de violações dessas propriedades deve interferir o mínimo possível no ambiente de produção. Dessa maneira, surgiu a necessidade de um mecanismo automático capaz de detectar vio-

lações de propriedades do projeto de software.

5.4 Propriedades comportamentais

As propriedades comportamentais do sistema *OurGrid* estão diretamente relacionadas com o comportamento das *threads* durante a sua execução. Dentre os componentes que compõem o *OurGrid*, o*MyGrid* e *Peer* são componentes modularizados. Cada módulo que compõem esses componentes possuem múltiplas *threads*, dentre as quais existe um "*thread principal*" que possui características especiais, como de ser a única *thread* que pode acessar determinadas instâncias de objetos em um dado momento. Assim, para detectar violações dessa propriedade comportamental durante a execução do software é necessário observar o comportamento das *threads* do sistema *OurGrid*.

Para exemplificar a especificação e como é feita a detectção de violações das propriedades comportamentais na aplicação da técnica *DesignMonitor*, iremos utilizar o componente *MyGrid*. Este componente é composto por dois módulos:

- ***Scheduler*** - módulo responsável por escalonar de maneira eficiente a alocação de processadores disponíveis na grade para a execução das tarefas submetidas e, por fim, unir os resultados obtidos pela execução das tarefas e apresentar o resultado final ao usuário solicitante;

- ***ReplicaExecutor***- módulo responsável por efetivamente executar as tarefas.

5.4.1 Especificação das propriedades comportamentais

Considerando o componente *MyGrid*, uma das propriedades comportamentais que o compõem possui as seguintes características:

- cada módulo que compõe o componente *MyGrid* é *multithreaded*;

- dentre as múltiplas *threads* de um módulo, existe uma "*thread principal*" que possui características especiais.

Uma maneira de exemplificar essa propriedade comportamental é caracterizar as "*threads principais*" com a definição de uma cor específica para essas *threads*, o que irá identificar cada módulo. Dessa forma, temos que:

- os objetos são inicialmente criados sem cor;
- um objeto sem cor ao ser acessado por uma "*thread principal*" passa a ter a cor da *thread* que o acessou;
- caso o objeto já possua uma cor, este só pode ser acessado pela "*thread principal*" de mesma cor, ou seja, pela *thread* do módulo ao qual o objeto pertence.

Para a técnica *DesignMonitor* o que irá representar essa cor é o nome das *threads*. Como visto anteriormente, o componente *MyGrid* é composto pelos módulos *Scheduler* e *ReplicaExecutor*, que possuem respectivamente as seguintes "*threads principais*": *SchedulerEventProcessorThread* e *ReplicaExecutorEventProcessorThread*, conforme ilustramos na Figura 5.3.

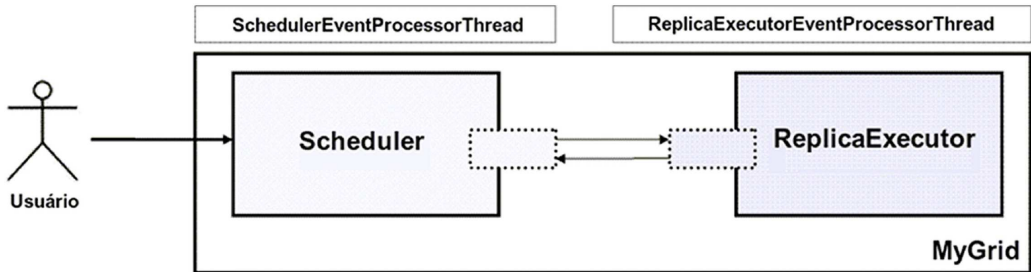

Figura 5.3: Componente *MyGrid*: propriedade comportamental.

Essa propriedade comportamental é então especificada como mostra o Código 5.4.

No Código 5.5 é apresentado o código aspecto de monitoração gerado automaticamente a partir dos pontos de interesse descritos na especificação da propriedades `MyGridThreadsBehaviorProperty` descrita no Código 5.4. Este código deve ser adicionado ao código-fonte do software alvo, no caso o *OurGrid*. Além disso, outras modificações são necessárias no projeto *OurGrid* com respeito a configuração. Neste foram adicionamos o arquivo `designmonitor.jar` ao projeto *OurGrid*, alterando o arquivo de configuração do Ant[3], especificamente os arquivos `build.xml` e

[3]É uma ferramenta escrita em Java que auxilia no desenvolvimento, testes e *deployment* de aplicações Java, permitindo executar automaticamente tarefas rotineiras.

```
label := MyGridThreadsBehaviorProperty;

rule := (!schedulereventprocessorthread
         && !replicaexecutoreventprocessorthread)
            U ([]schedulereventprocessorthread
               || []replicaexecutoreventprocessorthread);

points := (execution(* org.ourgrid.mygrid..*.*(..)))
       && !within(org.ourgrid.mygrid..*Facade)
       && !within(org.ourgrid.mygrid.scheduler.BlackListEntry)
       && !within(org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager)
       && !within(org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager)
       && !within(org.ourgrid.mygrid..*Configuration)
       && !within(org.ourgrid.mygrid..*Impl)
       && !within(org.ourgrid.common.spec.PeerSpec)
       && !within(org.ourgrid.mygrid..*Test)
       && !within(org.ourgrid.mygrid..ui..*)
       && !within(org.ourgrid.mygrid..test..*)
       && !within(org.ourgrid.common.id.*)
       && !within(org.ourgrid.threadServicesAspects..*);
```

Código 5.4: Especificação da propriedade comportamental `MyGridThreadsBehaviorProperty`.

`build.properties` [Ant, 2000].

Na Figura 5.4 temos a representação gráfica do autômato de Büchi equivalente a fórmula LTL para especificação do comportamento das *threads*, que é gerado pela ferramenta LTL2BA4J [Bodden, 2005]. Por restrições da ferramenta LTL2BA4J a especificação do comportamento em LTL deve ser em letras minúsculas. Baseada no autômato de Büchi gerado é criada uma máquina de estados, definida pelo código de análise comportamental gerado automaticamente, onde é realizada a verificação do comportamento observado.

Durante a execução do *OurGrid* para cada instância de objetos pertencentes aos pontos de interesse especificados é gerada uma máquina de estados e, em seguida, adicionada ao módulo analisador do *DesignMonitor*. O módulo observador monitora cada instância de objetos pertencentes aos pontos de interesse especificados e captura o comportamento das *threads* nesses pontos à medida que o *OurGrid* é executado. Esse comportamento capturado é enviado para o módulo analisador que paralelamente analisa na máquina de estados equivalente àquele objeto observado se é um comportamento esperado ou não. A captura e identificação de um comportamento inesperado auxiliam o desenvolvedor a verificar se o sistema implementado está de acordo com o esperado. No entanto, não é fácil descrever uma regra que

```
1  package org.ourgrid.designmonitor.mygrid;

2  import br.edu.ufcg.designmonitor.aspects.DesignMonitorAspect;
3  import java.net.InetAddress;
4  import java.net.UnknownHostException;

5  public aspect MyGridThreadsBehaviorProperty extends DesignMonitorAspect{

6     public MyGridThreadsBehaviorProperty() throws Exception{
7        super("MyGridThreadsBehaviorProperty");
8     }

9     public pointcut monitoringPoints()
10     : (execution(* org.ourgrid.mygrid..*.*(..)))
11       && !within(org.ourgrid.mygrid..*Facade)
12       && !within(org.ourgrid.mygrid.scheduler.BlackListEntry)
13       && !within(org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager)
14       && !within(org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager)
15       && !within(org.ourgrid.mygrid..*Configuration)
16       && !within(org.ourgrid.mygrid..*Impl)
17       && !within(org.ourgrid.common.spec.PeerSpec)
18       && !within(org.ourgrid.mygrid..*Test)
19       && !within(org.ourgrid.mygrid..ui..*)
20       && !within(org.ourgrid.mygrid..test..*)
21       && !within(org.ourgrid.common.id.*)
22       && !within(org.ourgrid.designmonitor..*)
23       && !within(org.ourgrid.threadServicesAspects..*)
24       && !cflow(execution(int *.hashCode()))
25       && !cflow(execution(java.lang.String *.toString()));

26 }
```

Código 5.5: Código de monitoração da propriedade comportamental `MyGridThreadsBehaviorProperty` (P3).

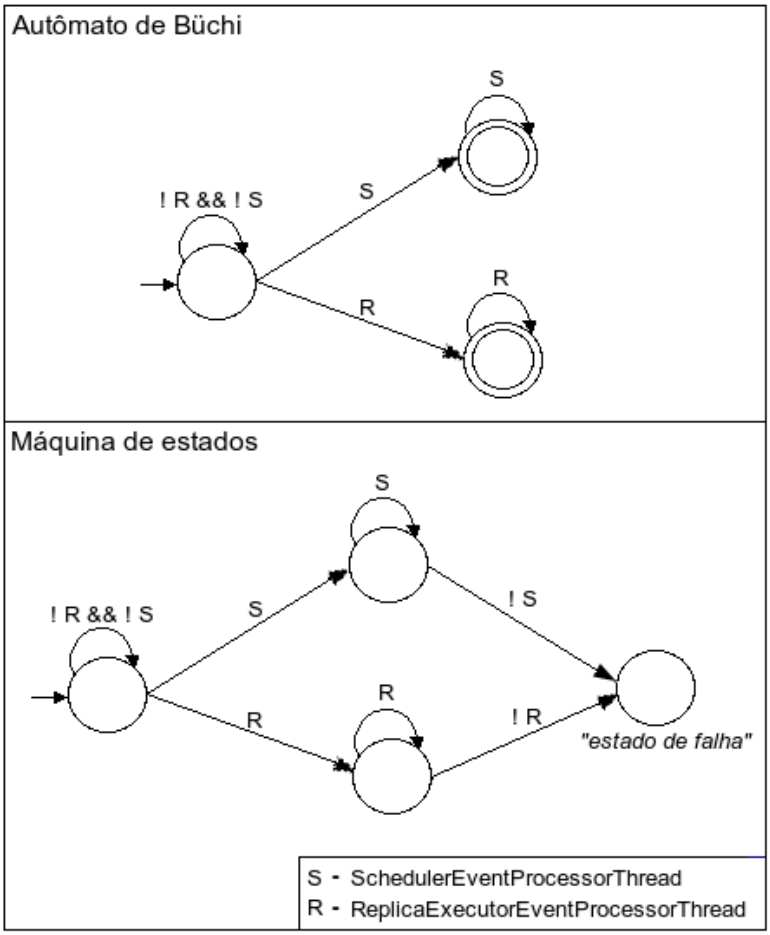

Figura 5.4: Representação gráfica da máquina de estados que verifica o comportamento observado.

descreva um comportamento e manter esta regra coerente com a evolução do sistema, estas são tarefas delegadas à equipe de desenvolvimento do sistema sob análise.

Outras duas regras foram definidas para o *OurGrid*, as mesmas estão descritas nos Códigos 5.6 e 5.7, respectivamente.

```
label: EBGridManagerBehaviorProperty
rule: !Scheduler U []Scheduler
point: org.ourgrid.mygrid.scheduler.gridmanager.EBGridManager
```

Código 5.6: Especificação da propriedade comportamental `EBGridManagerBehaviorProperty` (P1).

```
label: EBJobManagerBehaviorProperty
rule: !Scheduler U []Scheduler
point: org.ourgrid.mygrid.scheduler.jobmanager.EBJobManager
```

Código 5.7: Especificação da propriedade comportamental `EBJobManagerBehaviorProperty` (P2).

Cada uma das propriedades vistas anteriormente descreve comportamento desejado para duas *threads*. A *thread* Scheduler não pode iniciar execução sobre os objetos até que esta inicie, mas somente a *thread* Scheduler pode acessar até o fim do processamento.

No contexto do estudo de caso, de posse das propriedades comportamentais e dos artefatos gerados a partir destas, para analisar o comportamento durante a execução do sistema *OurGrid* foi necessário definir os cenários de monitoração, que serão descritos na Seção 5.4.2.

5.4.2 Cenários de monitoração

O estudo de caso foi realizado considerando três diferentes cenários. No primeiro cenário, foi considerado o OurGrid código-fonte sem qualquer modificação. Os dois últimos cenários inserimos erros propositadamente. Em um deles, o nome foi mudado da Scheduler para SchedulerError. No último cenário, nós modificamos o nome de thread ReplicaExecutor para ReplicaExecutorError.

O OurGrid têm três propriedades comportamentais especificadas para diferentes tipos de objetos. Para cada propriedade foram executadas dez vezes considerando a análise de cada

propriedade, um número estabelecido como suficiente para capturar as falhas aleatórias, já que o sistema distribuído apresentava uma aleatoriedade repetitiva a partir de um conjunto mínimo de três repetições.

5.4.3 Análise das propriedades comportamentais

À medida que o *OurGrid* é executado, o *DistributedDesignMonitor* analisa cada cenário monitorado. Quando algum dos pontos de interesse é acessado o código aspecto de monitoração captura as informações relevantes no módulo de monitoração e envia para o módulo analisador. Este por sua vez, identifica qual a máquina de estado referente a este comportamento, então executa a função de transição considerando o estado corrente e o comportamento observado. Caso seja o primeiro acesso o estado corrente da máquina de transição é o estado inicial da mesma. Essa rotina se repete até que o cenário de monitoração seja completamente executado ou alguma violação seja detectada. Sendo assim, diante da não existência de violações pode-se dizer que o software, para uma execução, está correto em relação às propriedades comportamentais especificadas.

Ao executarmos o *OurGrid*, juntamente com o DDM, algumas violações comportamentais foram detectadas. Essas foram então avaliadas com relação a sua veracidade, como por exemplo, se o ponto de interesse onde a violação ocorreu faz mesmo parte da propriedade comportamental violada. A notificação das violações é feita através de um arquivo de *log*, que identifica o início e o fim de cada cenário de monitoração, bem como as informações das violações detectadas, conforme mostrado no fragmento de Código 5.8.

```
RULE VIOLATED: MyGridThreadsBehaviorProperty
Expected: (!replicaexecutoreventprocessorthread && main && !schedulereventprocessorthread)
 But was: schedulereventprocessorthreaderror
Object: org.ourgrid.mygrid.scheduler.ReplicaExecutorResult@1655d71#cavaquinho/150.165.98.59
Method: getReplica()
Method line: 114
Time: 52053
```

Código 5.8: Notificação de Violação de Comportamento produzido para um arquivo de *log*.

Na Tabela 5.1 podemos ver o conjunto de execuções que temos realizado e os resultados reportados pela DDM ferramenta.

Durante a execução, observamos o número de execuções que apresentou um comportamento inesperado. Também foi observado o número de diferentes tipos de erros que possa levar o sistema a um estado inconsistente.

Tabela 5.1: Resultados do Estudo de Caso do *OurGrid*.

Execuções com Violação				**Numero de Tipos de Erros**			
	Propriedades				Propriedades		
Cenário	**P1**	**P2**	**P3**	**Cenário**	**P1**	**P2**	**P3**
1	0	1	*10*	**1**	-	1	*4*
2	10	10	*10*	**2**	1	1	*1*
3	0	0	*10*	**3**	-	-	*3*

A coluna P3 são refere-se às execuções da propriedade `MyGridThreadsBehaviorProperty` descrita anteriormente. As propriedades comportamentais 1 e 2 são propriedades que descrevem regras apenas para o segmento `Scheduler`, descritas nos Códigos 5.6 e 5.7, respectivamente. A terceira propriedade é referente às threads Scheduler e ReplicaExecutor. No cenário 1, nenhuma inconsistência foi encontrada durante as dez execuções para a propriedade comportamental 1.

O acompanhamento de execuções da propriedade 2 apresentou apenas um comportamento incoerente, mas a terceira propriedade encontrou uma inconsistência para cada execução. O cenário 1 é avaliado de acordo com os relatos de como as informações devem ser processadas, descritas pelos especialistas (*designers*) do sistema. No cenário 2, todas as execuções produziram inconsistências de comportamento, assim como este cenário sugere que devemos encontrar uma inconsistência referente ao erro inserido para o thread Scheduler.

O cenário 3 necessita de mais explicações sobre o log de erro apresentado. Neste cenário mudamos o nome da thread ReplicaExecutor e as propriedades comportamentais 1 e 2 são encaminhados apenas para o Scheduler. Assim, quando acompanhamos a execução avaliando as propriedades 1 e 2 não encontramos qualquer inconsistência, como é esperado. Para a propriedade 3 todas as execuções apresentaram alguma incoerência. Os cenários 2 e 3 dão suporte para avaliar a nossa ferramenta. Para as execuções que esperam por alguma inconsistência, o DDM apresentou o relatório contendo o conjunto de violações ocorridas. Quando a monitoração não deve apresentar nenhuma incoerência, nenhuma incoerência foi detectada.

Esta experiência mostra que a nossa ferramenta é confiável e pode ser utilizada para ajudar desenvolvedores na certificação do comportamento correto dos seus sistemas distribuídos.

Capítulo 6

Trabalhos Relacionados

Várias técnicas têm sido propostas para ajudar os programadores a encontrar defeitos em sistemas concorrentes e distribuídos. Dentre estas técnicas, podemos comparar a eficiência de cada de acordo com algumas características, como qual tipo de especificação de linguagem é necessário, como é feita a captura de eventos, e como as informações capturadas são analisadas e verificadas em conformidade com sua especificação. Este capítulo tem como propósito apresentar alguns desses trabalhos, com a realização de uma análise crítica e comparativa com relação ao *DistributedDesignMonitor*.

Em [Delgado et al., 2004], são destacados alguns pontos de monitoração de sistemas em tempo de execução que devem ser especificados e utilizados na monitoração. Alguns desses pontos são: a linguagem de especificação, o monitor, o controlador de eventos (*event-handler*) e outros assuntos operacionais (que eventualmente são tratados em alguns sistemas de monitoração).

A linguagem de especificação é a linguagem que será utilizada para especificar propriedades de monitoração, bem como o poder de expressão: que tipo de propriedade poderá ser especificado e qual o nível de monitoração realizada no sistema. O monitor é quem observa e analisa o estado do sistema, verificando a partir de uma comparação das informações coletadas com as regras especificadas previamente. Já o controlador de eventos determina como serão notificados os eventos detectados pelo monitor, ou seja, a partir da observação do sistema em execução o monitor pode reagir à alguma violação de regra de especificação, como é feita essa notificação desses eventos é determinada pelo controlador de eventos. Esta idéia de monitoração definida em [Delgado et al., 2004] é também utilizada neste trabalho,

formalizando como devem acontecer cada uma das partes de monitoração de um sistema distribuído.

6.1 Pip

O Pip [Reynolds et al., 2006] proporciona uma infra-estrutura para comparar o comportamento real de sistemas concorrentes e distribuídos com um comportamento esperado previamente especificado. O comportamento desses sistemas é especificado através de uma linguagem de `script` definida, com a possibilidade de identificar propriedades estruturais e problemas de desempenho em sistemas distribuídos. Pip permite a expressão de expectativas com relação à estrutura de comunicação, sincronismo e o consumo de recursos do sistema.

O sistema de verificação comportamental Pip é, dos trabalhos identificados na nossa revisão bibliográfica, o que mais se assemelha com este trabalho. Construindo uma linguagem de especificação para descrever o comportamento desejado, embora utilize de anotações para capturar as informações em tempo de execução do sistema analisado.

A infra-estrutura do Pip é baseada na modificação dos *bytecodes* para possibilitar a observação do sistema durante a sua execução, verificando a corretude comportamental do mesmo. Ver Figura 6.1 [Reynolds et al., 2006].

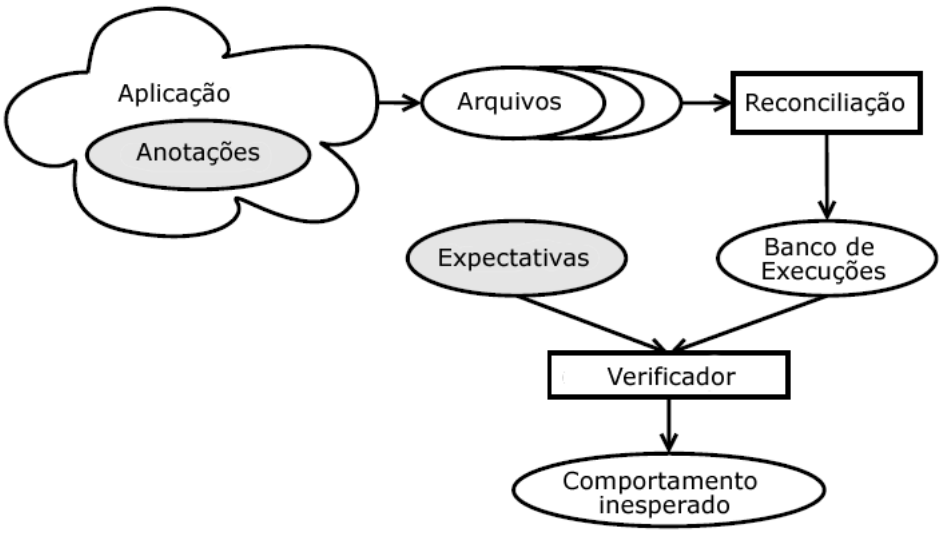

Figura 6.1: Estrutura geral do trabalho Pip.

Embora a verificação seja realizada ao longo da execução das *threads* em sistemas concorrentes e distribuídos, a verificação realizada trata todos os eventos, advindos de todos as máquinas componentes do sistema distribuído, como uma única thread de execução, sendo que esta linearização não se preocupa com o verdadeiro instante de acontecimento dos eventos do sistema analisado.

Durante a execução do sistema, Pip inicia a captura das informações e armazena em arquivos. Depois do fim da execução começa a realização da verificação do comportamento real com o esperado. No entanto, propriedades relacionadas com o design desejado do sistema, e as regras comportamentais que especificam como os componentes precisam se relacionar, não são consideradas pelo Pip.

6.2 Teste de Conformidade com Autômatos Temporizados

A técnica de Teste de Conformidade apresentada em [Cardell-Oliver, 2002] tem como propósito de verificar se um Sistema Distribuído de Tempo Real satisfaz uma dada especificação formal de autômatos temporizados definidos para a ferramenta Uppaal [Bengtsson et al., 1996]. O *Designer* ou Especificador do sistema define como um sistema deve se comportar, usando autômatos temporizados para isso [Cardell-Oliver, 2002].

O sistema de verificação de conformidade então gera casos de teste a partir de especificações formais dos autômatos e executa-os para determinar se o comportamento do chamado SUT (*system under test*) coincide com o comportamento desejado.

O principal objetivo de testar o sistema é encontrar defeitos no sistema, e assim os casos de teste são escolhidos para exercitar, na medida do possível, as partes do sistema, que poderiam apresentar defeitos.

Utilizando de sensores para identificar os eventos do sistema e passando esta detecção para ser analisada pelo autômato temporizado, o comportamento do sistema é observado e analisado quanto à sua corretude. No entanto, o foco de análise e verificação desta técnica é sobre Sistemas Distribuídos em Tempo Real, cuja base da informação a ser analisada é o instante em que as tarefas são executadas.

Além de considerar a avaliação de eventos do sistema distribuído sob investigação, a técnica se propõe a analisar também ações externas ao sistema, como botões pressionados,

luzes a serem ligadas/desligadas, em um determinado instante.

6.3 JavaMOP

A Programação Orientada à Monitoração (*Monitoring-Oriented Programming* - MOP) é uma proposta feita inicialmente em [Chen et al., 2004] e atualizado em [Chen and Rosu, 2007], sugerindo que seja usado como um dos princípios da programação, assim como é Orientação a Objetos hoje.

Assim como Design by Contract (DbC) [Meyer, 1992], Java Modeling Language (JML) [Leavens and Cheon, 2003], Runtime Verification e Aspect Oriented Programming (AOP) [Laddad, 2003; Kiczales et al., 2001] foram sugeridos na tentativa de prover alguma forma de validação de uma implementação realizada para suprir uma determinada especificação, MOP tenta propor outra forma de realizar esta monitoração.

A Programação Orientada a Monitoração foi proposta como uma forma de analisar o desenvolvimento de software baseado na verificação em tempo de execução. A execução controlada a partir do caminho de execução capturado do programa como uma seqüência de eventos, de acordo com propriedades de comportamento, e esta seqüência de eventos tem estados instantâneos. Em MOP especificações podem ser vistas como aspectos formais ou lógicos são automaticamente convertidos em código aspectos. Um monitor (no código aspecto) é criado para cada instância a ser verificado. Tal como no DDM, o MOP é baseado na lógica formal e na verificação das propriedades de segurança, mas MOP gera um grande número de monitores relacionados com cada caso. No entanto, MOP não dá ênfase em sistemas distribuídos, caracterizando um diferencial para esses tipos de sistemas.

Os requisitos do sistema são expressos formalmente, através de anotações em pontos específicos no código-fonte dos programas. Com base nessas anotações é gerado automaticamente o código de monitoração, que funciona como um verificador lógico dos requisitos. Na arquitetura de MOP, apresentada na Figura 6.2 [Chen and Rosu, 2007], a linguagem lógica utilizada para especificar os requisitos é independente da linguagem do código de monitoração. Dessa maneira, MOP pode ser estendida para diversas linguagens lógicas e de codificação.

A especificação descrita no código 6.1 ilustra a especificação de uma propriedade com-

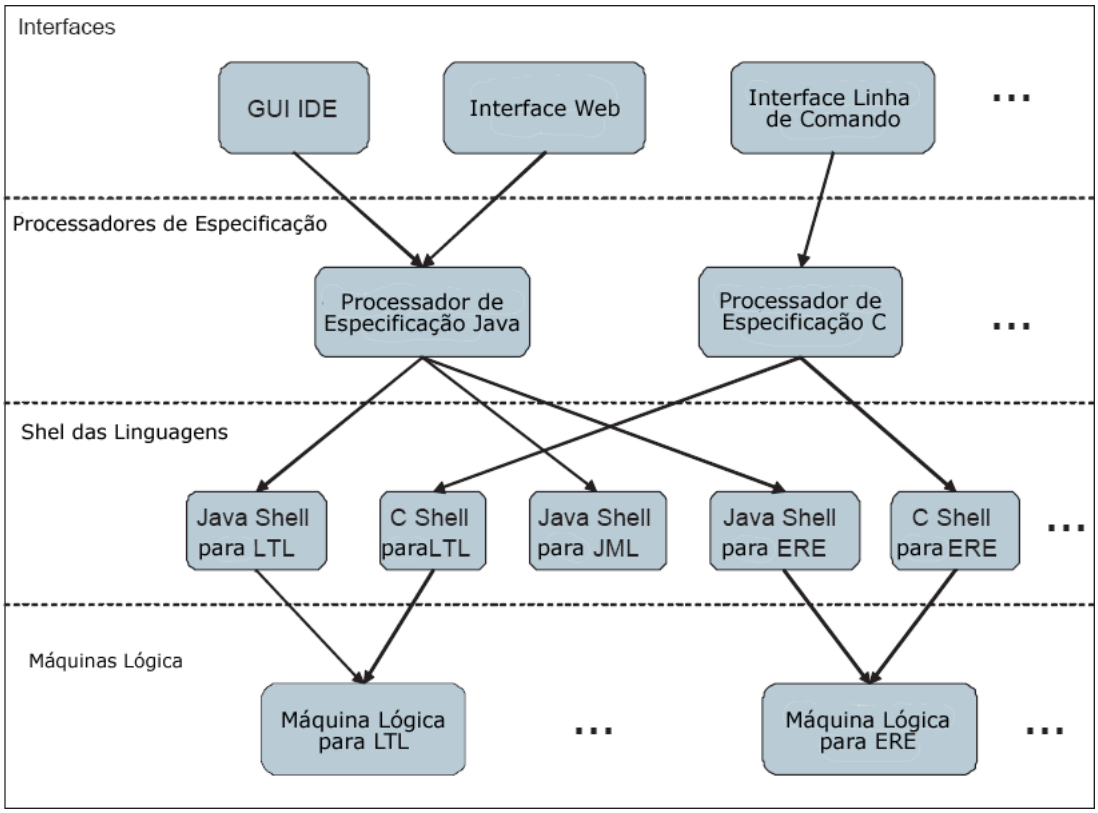

Figura 6.2: Arquitetura do trabalho MOP.

portamental em LTL tempo-passado (PTLTL), uma das possibilidades de especificação definidas pelo JavaMOP. Nesta especificação, é definida a necessidade de autenticação de um usuário antes de liberar acesso de um determinado login.

Com base no código de especificação definido no Código 6.1, é gerado o código de monitoração, que será responsável por capturar as informações durante a execução do sistema monitorado. O Código 6.2 mostra um trecho do código de monitoração gerado a partir do código de especificação definido em 6.1.

Como podemos ver no Código 6.2, ele é construído utilizando a linguagem AspectJ [Laddad, 2003], o qual realiza a captura das informações relevantes para a monitoração do sistema.

JavaMOP permite expressar e verificar em tempo de execução se o comportamento dos sistemas está de acordo com o comportamento especificado. Contudo, para verificação de propriedades de sistemas concorrentes e distribuídos este trabalho não suporta. A especificação e verificação dos requisitos em JavaMOP não permite descrever o comportamento de vários processos na execução de um sistema distribuído, espalhado por várias máquinas e .

```
class Resource {
/*@
scope = class
logic = PTLTL
{
event authenticate: end(exec(* authenticate()));
event access: begin(exec(* access()));
formula: access -> <*> authenticate;
}
violation handler { @this.authenticate(); }
@*/
void authenticate() {...}
void access() {...}
...
}
```

Código 6.1: Especificação de propriedade comportamental JavaMOP.

```
/*+MonitorAspect+*/
public aspect MonitorAspect {
/*+ Generated by JavaMOP for javamop.monitor PTLTL_0 */
...
after(Resource thisObject): PTLTL_0_Init(thisObject) {
boolean authenticate = false;
boolean access = false;
thisObject.PTLTL_0_now[0] = authenticate;
}

pointcut PTLTL_0_authenticate0(Resource thisObject):
target(thisObject) && execution(* Resource.authenticate());
after (Resource thisObject) returning:
PTLTL_0_authenticate0(thisObject) {
boolean authenticate = false;
boolean access = false;
authenticate = true;
thisObject.PTLTL_0_pre[0] = thisObject.PTLTL_0_now[0];
thisObject.PTLTL_0_now[0] = authenticate ||
thisObject.PTLTL_0_pre[0];
if (access && ! thisObject.PTLTL_0_now[0]){
thisObject.authenticate(); }
}
pointcut PTLTL_0_access0(Resource thisObject):
...
/* Generated code ends +*/
}
```

Código 6.2: Código de Monitoração JavaMOP para a especificação comportamental do Código 6.1.

6.4 Java-MaC

A Monitoração e Verificação de programas em Java (*Monitoring and Checking* - Java-MaC) é uma arquitetura proposta em [Kim et al., 2001; Kim et al., 2002]. Java-MaC é um *framework* para a monitoração em tempo de execução de sistemas, verificando se o sistema está executando corretamente de acordo com uma especificação formal de requisitos.

A técnica proposta em [Kim et al., 2001] apresenta duas linguagens de especificação de requisitos, são elas: *Primitive Event Definition Language* (PEDL) e *Meta Event Definition Language* (MEDL). A PEDL é usada para especificar qual informação deverá ser enviada para ser analisada. E a MEDL é a linguagem de especificação de alto nível. A justificativa da existência das duas linguagens de especificação é a separação entre como o código está implementado para capturar esta informação e como o sistema deve se comportar durante a sua execução, ou seja, a linguagem de especificação de alto nível ou de comportamento desejado.

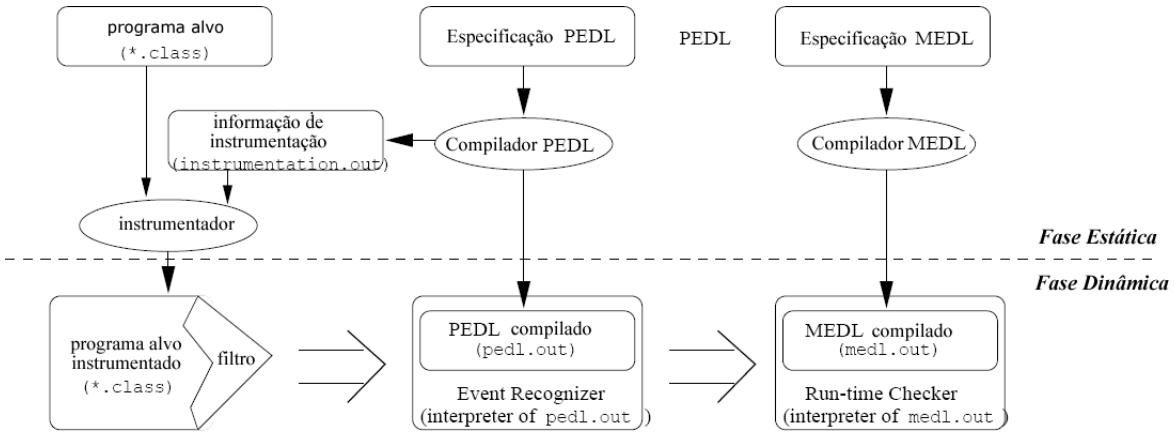

Figura 6.3: Arquitetura do trabalho JavaMaC.

Como podemos ver na Figura 6.3, a Fase Estática é composta pelo código *bytecode* instrumentado, contendo a lista de variáveis e métodos a serem monitorados, gerados automaticamente da especificação PEDL. Baseada nas duas entradas, o `instrumentador` Java-MaC insere um filtro no programa alvo. O compilador PEDL compila a especificação PEDL para uma linguagem abstrata, passada e melhorada pelo *Event Recognizer* em tempo de execução. Ao mesmo tempo, o compilador PEDL gera informação instrumentada que é utilizada pelo `instrumentador`. Equivalentemente, o compilador MEDL compila a

especificação MEDL, que será também evoluída pelo *Run-time Checker* em tempo de execução.

Durante a execução (Fase Dinâmica), o `filtro` extrai *snapshots*[1] a partir do `programa alvo` e envia estes *snapshots* para o *Event Recognizer*. Quando este recebe os *snapshots* avalia os eventos e as condições de acordo com as expressões em PEDL. Neste momento, se o *Event Recognizer* detecta algum erro, este é enviado para o *Run-time Checker*. Por fim, o *Run-time Checker* avalia a execução e verificação recebida de acordo com as expressões descritas em MEDL. Novamente, se alguma identificação de violação for detectada, está é externada através de um sinal.

Assim como os outros trabalhos já citados anteriormente, a técnica provida por MaC e implementada através da ferramenta Java-MaC se concentra na verificação de acesso e modificação de variáveis compartilhadas entre objetos, bem como concentrando-se em sistemas concorrentes. Com isso, sistemas distribuídos não é o foco de verificação da técnica Mac [Kim et al., 2002].

6.5 Diana

A ferramenta Diana (*DIstributed ANAlysis tool*) é a implementação de uma técnica de monitoração descentralizada definida em [Sen et al., 2003; Sen et al., 2004], que monitora a execução de um programa distribuído para verificar a violação de propriedades de segurança (*safety properties*).

A especificação das propriedades é feita usando PT-DTL, uma variante de LTL em tempo-passado definida pela técnica para sistemas distribuídos, e realiza a monitoração a partir dos eventos capturados do sistema com base nessas fórmulas PT-DTL. A estrutura da ferramenta DiAna é apresentada na Figura 6.4.

A realização da verificação comportamental do sistema distribuído monitorado necessita inicialmente da especificação formal do sistema, usando para isto a lógica baseada em LTL PT-DTL. Esta linguagem PT-DTL suporta a especificação de propriedades temporais considerando o acesso remoto e/ou distribuído dos elementos envolvidos em cada propriedade comportamental.

[1] O algoritmo dos Snapshots é apresentado e detalhado no Apendice A.

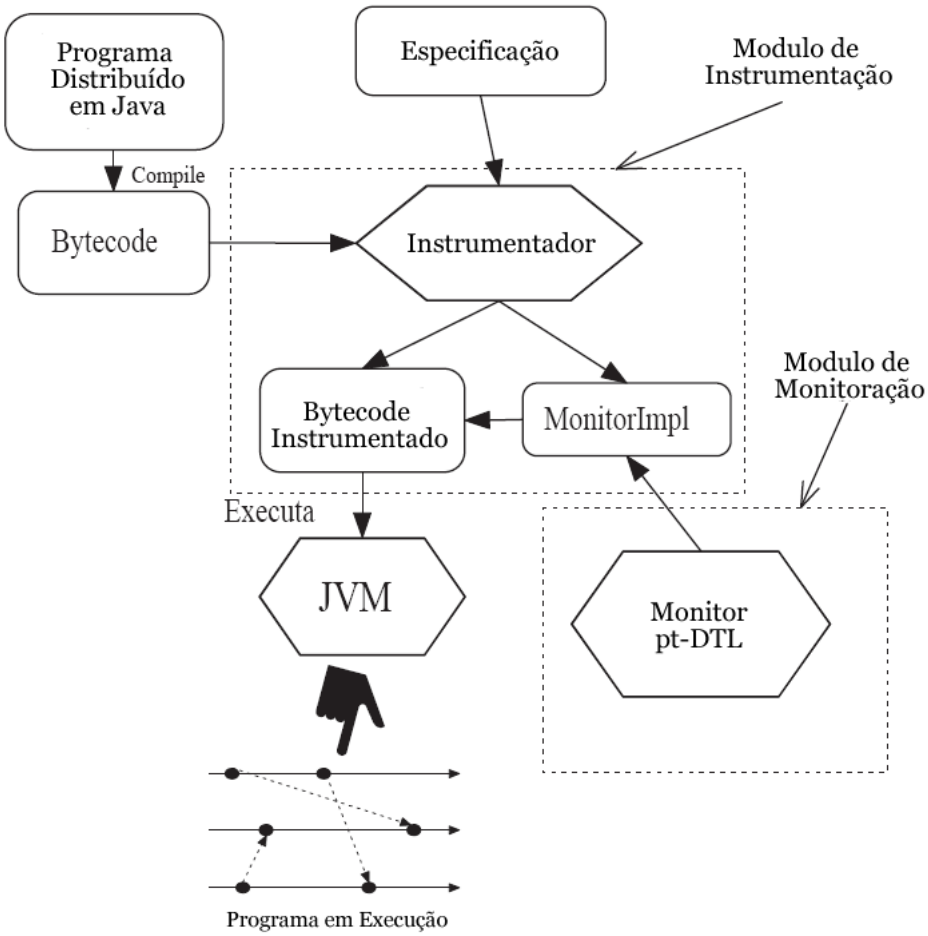

Figura 6.4: Arquitetura da ferramenta DiAna.

Considere a propriedade definida na Figura 6.5 em PT-DTL. Esta propriedade expressa a seguinte situação: "se *a* recebeu um valor então, no passado deve ter acontecido em *b*: *b* computou o valor e em *a* um pedido por este valor foi realizado no passado".

$$@_a(\texttt{receivedValue} \rightarrow @_b(\Diamond(\texttt{computedValue} \wedge @_a(\Diamond\texttt{requestedValue}))))$$

Figura 6.5: Propriedade comportamental usando a lógica PT-DTL.

Durante a execução é realizado um conjunto de *snapshots* do sistema como um todo para possibilitar uma avaliação consistente do sistema distribuído. No entanto, com um conjunto de formalismos de especificação e monitoração de informações locais e distribuídas, em [Sen et al., 2004] não é apresentado nenhum conjunto de resultados de experimentação da ferramenta Diana.

Existe o site com *link* para *download* da ferramenta DiAna, mas não está disponibilizado para utilizarmos em comparação com o *DistributedDesignMonitor* - DDM.

Capítulo 7

Conclusão

Neste trabalho de dissertação, apresentamos uma técnica de monitoração e verificação comportamental para sistemas concorrentes e distribuídos. Em face da dificuldade de avaliar o comportamento apresentado durante a execução destes tipos de sistemas, sugerimos a técnica de monitoração como maneira de facilitar a identificação de anomalias e comportamentos não desejados para sistemas distribuídos.

A técnica de monitoração aqui sugerida compara o código implementado de sistemas concorrentes e distribuídos com uma especificação formal do comportamento desejado para o sistema sob monitoração. Algumas técnicas já tradicionalmente e amplamente utilizadas como Testes [Jorgensen, 2002], *Model Checking* [Baier and Katoen, 2008] e *Design by Contract* [Mitchell et al., 2002] são utilizadas para detectar erros durante a execução de sistemas. No entanto, a utilização de Testes foca na identificação de erros para sistemas seqüenciais, que têm a característica de serem determinísticos. A verificação utilizando *Model Checking* é inviável para sistemas de grande porte, pois atinge-se o problema da explosão de estados, impossibilitando a verificação de todos os estados possíveis. *Design by Contract* é uma técnica que mais se aproxima com o propósito de verificação de assertivas durante a execução, mas esta ainda não foi apresentada para sistemas distribuídos e concorrentes.

Contornando o problema de analisar todas as possibilidades de execução de um sistema distribuído, passamos a analisar somente o conjunto de execuções reais do mesmo, minimizando assim o número execuções a serem analisadas. Informando se ocorreu alguma quebra de contrato durante a execução do sistema avaliado, inspirados no conceito de DbC.

Inicialmente, esta técnica de monitoração proposta foi projetada para sistemas concor-

rentes, implementada pela ferramenta *DesignMonitor* [Ana E. V. Barbosa, 2007], com o mesmo propósito de verificar o comportamento de sistemas concorrentes em tempo de execução. Evoluímos esta técnica para abranger a verificação comportamental de sistemas distribuídos. Contudo, para realizar a verificação de sistemas distribuídos é necessária a construção de um caminho consistente de execução, com base na execução real dos mesmos. Deste modo, a ferramenta *DistributedDesignMonitor* (DDM) é capaz de realizar a monitoração de sistemas concorrentes e distribuídos.

A monitoração inicia com a especificação formal do comportamento desejado para sistemas distribuídos. A especificação do comportamento é feita utilizando uma linguagem baseada na Lógica Temporal Linear (LTL) [Katoen, 1999]. Chamamos esta especificação de propriedades comportamentais, que descrevem como determinados pontos do sistema (pontos de interesse) devem se comportar durante a execução.

A especificação do conjunto de propriedades comportamentais é a base de funcionamento do DDM, pois este utiliza a especificação para gerar o código de captura dos eventos de cada ponto de interesse e o código de monitoração, que irá verificar se os pontos monitorados se comportaram como esperado.

A partir dos pontos de interesse são gerados os códigos AspectJ [Laddad, 2003], códigos estes responsáveis por capturar a informação em tempo de execução dos pontos de interesse (objetos, em Java) descritos nas propriedades comportamentais. E a partir da regra LTL da propriedade comportamental, são geradas as máquinas de estados equivalentes ao comportamento descrito formalmente. As máquinas de estados resultantes são base do código de análise, verificando a corretude comportamental do sistema.

Durante a execução de cada máquina de um sistema distribuído, um Monitor Local é responsável por capturar os eventos daquela máquina, e paralelamente realiza o envio destes eventos para um Monitor Central, responsável por organizar toda a informação capturada. Antes de enviar os eventos, estes são marcados com o instante lógico que o mesmo ocorreu. A utilização de um algoritmo de ordenação parcial é necessária para construir um caminho consistente em detrimento da execução real e aleatória de sistemas distribuídos.

Inicialmente analisamos a utilização do algoritmo de *Snapshots* Distribuídos, detalhado e modelado no Apêndice A, mas o algoritmo tornaria inviável a execução do sistema distribuído analisado, pois o número de mensagens enviadas para realizar a captura dos estados

temporários do sistema analisado poderia causar pausa na execução do sistema distribuído. A solução viável encontrada foi o algoritmo dos Relógios Lógicos, introduzido por Lamport em [Lamport, 1978].

Após a captura local, o Monitor Central organiza toda a informação capturada e repassa as informações para a Análise Comportamental do sistema. Cada máquina de estados executa o conjunto de eventos relacionados a ela, verificando se o comportamento está de acordo com o especificado. Se alguma inconsistência for detectada, a mesma é notificada para o usuário ou desenvolvedor do sistema analisado.

Neste documento também apresentamos um experimento com seus respectivos resultados, mostrando a viabilidade de uso da técnica de monitoração de sistemas distribuídos. Além do mais, a identificação de comportamentos indesejados é realizada, sendo localizados e detalhados, facilitando uma possível correção por parte dos desenvolvedores do sistema distribuído sob análise.

7.1 Contribuições

Neste trabalho apresentamos uma técnica de verificação comportamental de sistemas distribuídos, possibilitando analisar se o comportamento do sistema está de acordo com o desejado para o sistema em desenvolvimento. Algumas outras técnicas propõem a análise de sistemas concorrentes [Kim et al., 2002; Chen and Rosu, 2007; Ana E. V. Barbosa, 2007], mas não abrangem sistemas distribuídos. Outros propõem a avaliação de sistemas distribuídos [Reynolds et al., 2006; Sen et al., 2004], mas não realizam o devido cuidado com a ordenação dos eventos ou não apresentam seus resultados.

A monitoração comportamental de sistemas distribuídos realiza a verificação entre o projeto/especificação do software e a implementação do software. A constante necessidade de manutenção destes sistemas contribui para uma maior divergência entre o software implementado e seu projeto inicial. A técnica DDM aproxima e facilita a verificação entre a especificação e o comportamento real do sistema em desenvolvimento, ou na análise do comportamento de um sistema já em produção, identificando pontos para provável manutenção.

A detecção e identificação de comportamentos inesperados durante a execução de sistemas caracterizados pela aleatoriedade, e consequentemente pela dificuldade de repetir um

mesmo caminho de execução, facilitam e agilizam o trabalho de correção e atualização do software em desenvolvimento.

7.2 Sugestões de Trabalhos Futuros

Após a apresentação da viabilidade e funcionalidade da técnica de monitoração de sistemas distribuídos, temos um conjunto de pontos que podem ser melhorados e realizados em trabalhos futuros.

Generalização da linguagem de especificação comportamental. Apesar de ter sido projetada para aproximar a especificação comportamental de um sistema com a sua implementação, ainda há uma resistência quanto ao aprendizado da Lógica Temporal por parte dos usuários desta técnica. Uma forma de contornar este problema seria implementando uma maneira de possibilitar a especificação comportamental a partir de uma interface programacional, ou seja, mais próxima do contexto de especificação de Testes, assim como as já tradicionalmente utilizadas [Jorgensen, 2002].

Ajustes na ferramenta DDM. Melhorias no formato da interface de saída, que facilitem a identificação das violações comportamentais.

Análise de Sistemas Distribuídos não-locais. A técnica definida neste trabalho foi idealizada e testada para sistemas distribuídos que rodam em uma rede local. No entanto, sabemos da existência de sistemas distribuídos que são implementados (e que executam) entre redes. Este tipo de sistemas devem ultrapassar barreiras como *firewalls* para realizarem suas tarefas. A verificação da corretude comportamental destes tipos de sistemas poderia ser também avaliada.

Bibliografia

[Ana E. V. Barbosa, 2007] Ana E. V. Barbosa, Dalton D. S. Guerrero, J. C. A. d. F. (2007). Detecção automática de violações de propriedades de sistemas concorrentes em tempo de execução. Master's thesis.

[Ant, 2000] Ant, P. A. (2000). Apache ant. Web site. Disponível em: <http://ant.apache.org/>. Acessado em: outubro de 2008.

[Babaoglu and Marzullo, 1993] Babaoglu, O. and Marzullo, K. (1993). Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Mullender, S., editor, *Distributed Systems*, pages 55–96. Addison-Wesley.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[Büchi, 1962] Büchi, J. R. (1962). On a decision method in restricted second-order arithmetic. In *Proceedings of the International Logic, Methodology and Philosophy of Science*, pages 1–12, Stanford, CA, USA. Stanford University Press.

[Bengtsson et al., 1996] Bengtsson, J., Larsson, F., Pettersson, P., Yi, W., Christensen, P., Jensen, J., Jensen, P., Larsen, K., and Sorensen, T. (1996). Uppaal: A tool suite for validation and verification of realtime systems.

[Bodden, 2005] Bodden, E. (2005). Ltl2ba4j. Web site. Disponível em: <http://www-i2.informatik.rwth-aachen.de/Research/RV/ltl2ba4j/>. Acessado em: Setembro de 2008.

[Cardell-Oliver, 2002] Cardell-Oliver, R. (2002). Conformance test experiments for distributed real-time systems. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT in-*

ternational symposium on Software testing and analysis, pages 159–163, New York, NY, USA. ACM.

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

[Chen et al., 2004] Chen, F., D'Amorim, M., and Roşu, G. (2004). A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, Lecture Notes in Computer Science. Springer-Verlag.

[Chen and Rosu, 2007] Chen, F. and Rosu, G. (2007). Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588.

[Cirne et al., 2004] Cirne, W., Brasileiro, F., Costa, L., Paranhos, D., Santos-Neto, E., Andrade, N., Rose, C. D., Ferreto, T., Mowbray, M., Scheer, R., and Jornada, J. (2004). Scheduling in bag-of-task grids: The pauÁ case. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 124–131, Washington, DC, USA. IEEE Computer Society.

[Delgado et al., 2004] Delgado, N., Gates, A. Q., and Roach, S. (2004). A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872.

[Filman et al., 2004] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., editors (2004). *Aspect-Oriented Software Development*. Addison-Wesley, Boston.

[Foster and Kesselman, 1999] Foster, I. and Kesselman, C., editors (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

[Gradecki and Lesiecki, 2003] Gradecki, J. D. and Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA.

[Harada, 2003] Harada, L. (2003). An efficient sliding window algorithm for detection of sequential patterns. *Database Systems for Advanced Applications, International Conference on*, 0:73.

[Holzmann, 2003] Holzmann, G. J. (2003). *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional.

[Ireland, 2007] Ireland, A. (2007). Ltl reasoning: How it works. Technical report, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Ger. Disponível em: <www.macs.hw.ac.uk/ air/spin/lectures/lec-8-how-it-works.ps>. Acessado em: Junho de 2008.

[Jorgensen, 2002] Jorgensen, P. C. (2002). *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA.

[Joyce et al., 1987] Joyce, J., Lomow, G., Slind, K., and Unger, B. (1987). Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150.

[Katoen, 1999] Katoen, J.-P. (1999). *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course Mechanised Validation of Parallel Systems. Friedrich-Alexander Universitat Erlangen-Nurnberg.

[Kiczales, 2005] Kiczales, G. (2005). Aspect-oriented programming. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 730–730, New York, NY, USA. ACM.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Getting started with aspectj. *Commun. ACM*, 44(10):59–65.

[Kim et al., 2001] Kim, M., Kannan, S., Lee, I., Sokolsky, O., and Viswanathan, M. (2001). Java-mac: a run-time assurance tool for java programs.

[Kim et al., 2002] Kim, M., Kannan, S., Lee, I., Sokolsky, O., and Viswanathan, M. (2002). Computational analysis of run-time monitoring - fundamentals of java-mac. *Electronic Notes in Theoretical Computer Science*, 70(4).

[Laddad, 2003] Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA.

[Lamport, 1977] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Lamport, 1979] Lamport, L. (1979). A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97.

[Lamport, 1983] Lamport, L. (1983). What good is temporal logic? In Mason, R., editor, *Information Processing 83*, pages 657–668.

[Leavens and Cheon, 2003] Leavens, G. and Cheon, Y. (2003). Design by contract with jml.

[Liang and Xu, 2005] Liang, D. and Xu, K. (2005). Monitoring with behavior view diagrams for debugging. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 637–644, Washington, DC, USA. IEEE Computer Society.

[Mansouri-Samani and Sloman, 1993] Mansouri-Samani, M. and Sloman, M. (1993). Monitoring distributed systems.

[Massol and Husted, 2003] Massol, V. and Husted, T. (2003). *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA.

[Mcdowell and Helmbold, 1989] Mcdowell, C. E. and Helmbold, D. P. (1989). Debugging concurrent programs. *ACM Computing Surveys*, 21:593–622.

[Meyer, 1992] Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

[Mitchell et al., 2002] Mitchell, R., McKim, J., and Meyer, B. (2002). *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Nett and Gergeleit, 1997] Nett, E. and Gergeleit, M. (1997). Preserving real-time behavior in dynamic distributed systems. In *Proc. of the Int. Conf. on Intelligent Information Systems, The Bahamas*, pages 8–10.

[OurGrid,] OurGrid, P. Ourgrid. Web site. Disponível em: <http://www.ourgrid.org/>. Acessado em: 10 setembro 2008.

[Peters, 1999] Peters, D. K. (1999). Automated testing of real-time systems.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *In Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, New York, NY, USA. IEEE Computer Society.

[Reynolds et al., 2006] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. (2006). Pip: Detecting the unexpected in distributed systems.

[Schwarz and Mattern, 1994] Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations. In *In search of the holy grail. Distributed Computing*, pages 149–174.

[Sen et al., 2003] Sen, K., Rosu, G., and Agha, G. (2003). Runtime safety analysis of multithreaded programs. In *In Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03). ACM*, pages 337–346. ACM.

[Sen et al., 2004] Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2004). Efficient decentralized monitoring of safety in distributed systems.

[Soares and Borba, 2002] Soares, S. and Borba, P. (2002). Aspectj - programação orientada a aspectos em java. In *VI Simpósio Brasileiro de Linguagens de Programação*, Rio de Janeiro, RJ, BRA. SBC.

[Tanenbaum and van Steen, 2007] Tanenbaum, A. and van Steen, M. (2007). *Distributed Systems: Principles and Paradigms (2nd ed.)*. Prentice Hall.

[Winck, 2006] Winck, D.V. e Junior, V. (2006). *AspectJ - Programação Orientada a Aspectos com Java*. Novatec, Brasil.

Apêndice A

Estado Global em Sistemas Distribuídos

Para possibilitar a monitoração de um sistema distribuído precisamos primeiramente conhecer o estado do mesmo em um determinado instante no tempo. Então, pensamos na implementação de um algoritmo que nos possibilitasse encontrar o estado de um sistema distribuído durante a sua execução. O processo de capturar "fotos" em determinados momentos durante a execução de um sistema distribuído é chamado de *snapshots* distribuídos. Esse algoritmo é definido em [Chandy and Lamport, 1985]. No entanto, como veremos a seguir, a implementação deste algoritmo para encontrarmos o estado global de um sistema distribuído seria inviável quanto ao consumo computacional produzido pelo mesmo.

A.1 Estados Globais

Em um sistema distribuído, é comum termos paralelismo entre processos, atraso de comunicação aleatória, falhas parciais e tempo global não sincronizado. Todas estas características podem resultar em diferentes ordenações de eventos, o que torna o sistema não-determinístico. Nestes sistemas é difícil de dizer o que ocorreu em primeiro lugar, pode ser melhor usar da definição de "aconteceu antes" ("*happened before*") para definir uma ordenação parcial de eventos [Chandy and Lamport, 1985]. Temos que definir se um evento a aconteceu antes de um evento b (denotado por $a \rightarrow b$) se uma das seguintes condições: (a) se a e b são eventos no mesmo processo, e a vem antes de b, então $a \rightarrow b$; (b) se a é o envio de uma mensagem por um processo e b é a recepção da mesma mensagem por um outro processo, então $a \rightarrow b$; (c) se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

Com o intuito de preservar a causalidade do sistema e na falta de um relógio global, é possível determinar não exatamente um estado global, mas um estado global aproximado chamado de corte consistente. Um corte consistente é retirado de um estado global significante, um estado que o sistema poderia ter tido, respeitando a causalidade dos acontecimentos.

Um *snapshot* global de um corte consistente pode ser capturado usando o algoritmo de *snapshots* distribuídos. Neste algoritmo, um processo começa o *snapshot* do sistema, gravando o seu estado e solicitando que os outros processos gravem também o seus estados. O pedido é difundido através da rede, atingindo todos os processos [Chandy and Lamport, 1985].

A.2 *Snapshots* Distribuídos

No algoritmo *snapshot* distribuído, o estado global do sistema é encontrado através da sobreposição dos estados dos processos locais e os estados de seus canais de comunicação.

O modelo representando um sistema distribuído que utiliza essa técnica é uma coleção de processos que formam um grafo dirigido de vértices conectados entre si, onde cada vértice é um processo e cada aresta é um canal. Considerando os canais livres de erros, com *buffers* de comprimento infinito e a entrega de mensagens é realizada na ordem de recebimento. Um estado de um processo depende exclusivamente das suas operações internas e as mensagens recebidas. Um estado de um canal é a seqüência de mensagens enviadas ao longo do canal, excluindo as mensagens já recebidas.

Para registrar um estado global, um processo inicia o algoritmo gravando seu estado e entra em comunicação com os outros processos para que registrem também os seus estados. Para fazer isso, após gravar o seu estado, o processo envia mensagens especiais chamados marcadores através do seus canais de saída. Após o recebimento de um marcador, um processo pode registar o seu estado (se for o primeiro marcador recebido) ou atualizar seus estados dos canais de entrada (para os marcadores seguintes). Posteriormente, este processo também envia marcadores através dos seus canais de saída. Cada processo termina sua parte quando receber um marcador por cada um dos seus canais de entrada. O pseudo-código do algoritmo é mostrado a seguir.

Para iniciar o processo de um *snapshot*:

```
p registra o seu estado
     FOR EACH canal que c é um processo de saída p DO
          antes p envia mais mensagens através cp envia um marcador ao
          longo c

Para todos os outros processos:

IF q já recebeu marcadores através de todos os canais incoming
     enviar o instantâneo para iniciar processo p
     Q acabamento da parte do algoritmo, ignorando cada novo marcador
ELSE
     recebendo um marcador ao longo de um canal c fazer
     IF q se não tiver gravado o seu estado THEN
          q registra o seu estado
          q registos do estado de c como vazio
     ELSE
          q registra o estado de c como a seqüência das mensagens
          recebidas c depois q através do estado foi gravada e antes
          q recebeu o marcador através de c
```

A Figura A.1 mostra as fases do *snapshot* de um processo q receber um marcador.

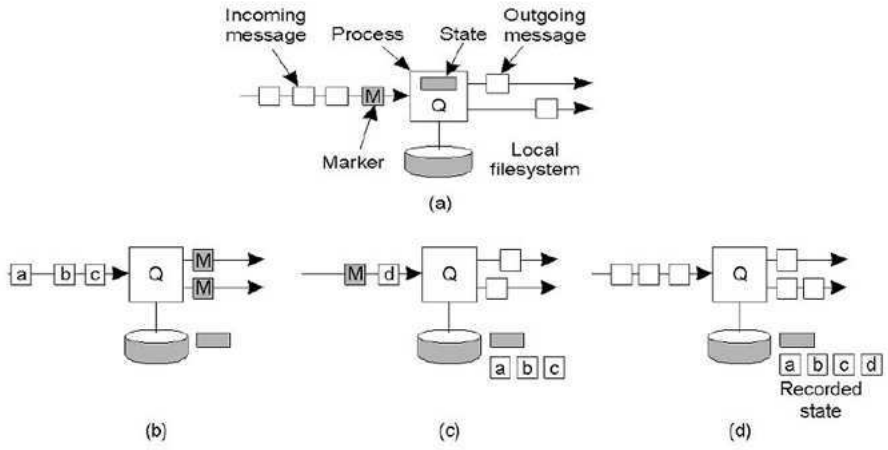

Figura A.1: *Snapshots* distribuídos - Recebendo marcadores

Monitorar utilisando software é normalmente superposto sobre um pedido de computação distribuída. Assim, pode ser importante conhecer o *overhead* o algoritmo impõe sobre o sistema subjacente. Alguns pontos então foram analisados, tais como:

- Qual o máximo de memória necessária para armazenar um estado global.
- Qual a sobrecarga causada por marcadores enviado através dos canais.
- Quanto tempo leva para registrar um estado global.

Estes são exemplos típicos questões relacionadas com software de controle. Para responder a este tipo de questão, que desenvolvemos modelos de sistemas distribuídos com monitoração concorrente usando *snapshots* distribuídos.

A.3 Modelagem de *Snapshots* Distribuídos em Redes de Petri Coloridas

Modelos de Redes de Petri Coloridas (*Colored Petri Nets* - CPN) são modelos executáveis que permitem uma simulação e uma análise automatizada do sistema modelado. Usando a ferramenta de Redes de Petri *CPN Tools*, é possível criar modelos CPN e simular determinados cenários usando estes modelos. Então, um modelo de Redes de Petri Colorida foi desenvolvido para simular um sistema distribuído em execução usando o algoritmo de *snapshots* distribuídos para encontrar o seu estado global.

Os modelos podem ser usados para entender melhor idéias de maneira abstrata. O algoritmo de *snapshot* à primeira vista parece ser de difícil compreensão. Daí, um modelo gráfico de um cenário simples que possa ser utilizado e simulado pode ser útil para a sua compreensão. Decidimos utilizar um cenário de um sistema distribuído de três processos, com conectividade total entre si, o que constou de seis canais confiáveis FIFO (*First In Firt Out*). Simulando este modelo através da ferramenta CPN Tools podemos entender cada passo do algoritmo. O modelo consiste de uma hierarquia coloridas Petri Net e algumas declarações globais e funções.

Considerando o pior caso em um sistema distribuído em funcionamento, também foi desenvolvida a modelagem de um sistema distribuído genérico, com n processos e m canais de comunicação entre todos os processos existentes. Essa generalização no número de processos e canais envolvidos se deu para possibilitar a execução de diferentes cenários durante a análise, determinando a sobrecarga a qual o modelo é submetido. Considerando a descrição

do algoritmo, a conectividade entre cada um dos processos é direta, cujo cada processo é conectado com cada um dos demais no modelo.

A.3.1 Estrutura Hierárquica em Redes de Petri Colorida

Modelos de Redes de Petri Colorida, ou simplesmente CPN, podem ser muito grandes. Tornar esses modelos menos enfadonho pode ser alcançado quebrando-se a lógica de uma solução em sub-módulos. É possível desenvolver módulos em CPN e combiná-los com outros módulos de uma hierarquia. Isso pode ajudar na reutilização de estruturas que se repetem em um modelo e também melhorar a visualização, uma vez que se pode analisar toda a estrutura hierárquica ou somente um determinado sub-módulo. Um módulo de Hierárquico no *CPN Tools* é chamado de uma página. Cada página de nível mais baixo pode ser ligado a uma página Transição de Substituição de um nível mais alto. A página `Top` define todo o sistema.

Nosso sistema distribuído foi modelado com uma página de alto nível descrevendo os processos, canais e as suas ligações (*links*), uma página para descrever um processo, outra página para descrever um canal e uma página com os detalhes da captura de estados globais. A figura A.2 mostra a página de mais alto nível. Na mesma, cada processo é uma transição de substituição que é composta por uma página chamada `Process`. No nosso cenário, existem três processos denominados P, Q, R, cada um tendo uma página `Process` correspondente. Além disso, cada processo tem um lugar para as mensagens a serem enviadas, um lugar para configuração (para descrever seus canais de entrada e saída) e dois lugares representam *buffers* de entrada e de saída para permitir conexão, respectivamente, com canais de entrada e saída.

Cada um dos seis canais é uma transição de substituição para uma página chamada `Channel`. Para cada transição há uma página individual `Channel`. Cada canal está ligada a um buffer de entrada e outro de saída e também um local para descrever a sua configuração (para descrever o seu processo de entrada).

A hierarquia de um modelo CPN pode ter mais que dois níveis. No nosso cenário, cada página `Process` tem uma transição de substituição que está ligado a página de *snapshot*, tornando o nosso modelo contendo três níveis hierárquicos.

A seguir vamos explicar o conjunto de declarações, funções e cada uma das páginas de

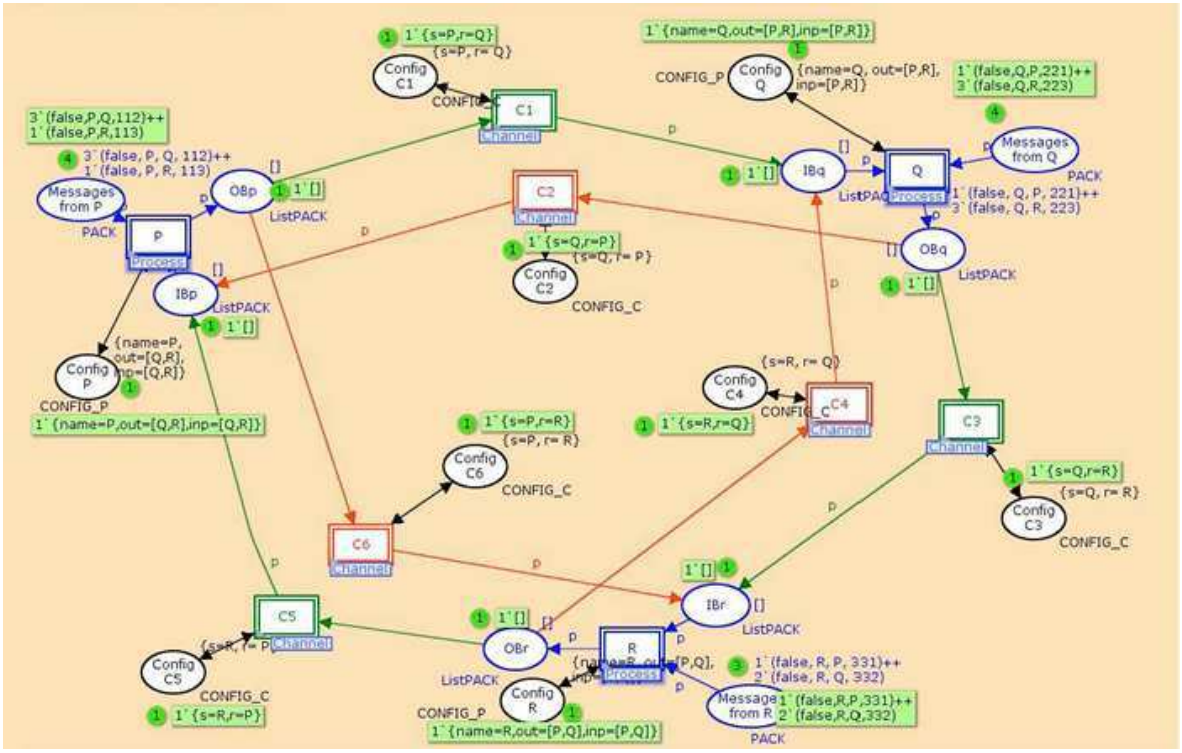

Figura A.2: Modelo CPN para três processos - Página `Top`

mais baixo níveis.

A.3.2 Declarações e Funções

Redes de Petri Coloridas utilizam uma linguagem funcional de alto nível para descrever os seus lugares, transições e arcos. Esta linguagem é chamada CPN-ML, que foi adaptada a partir do linguagem funcional ML, a fim de lidar com múltiplos conjuntos. *CPN Tools* tem suporte para declarações e funções baseadas em CPN-ML, que têm alcance em toda a rede.

Em nosso modelo, que foi necessário declarar tipos (`color sets`) para os processos, dados, os pacotes e outras combinações destes tipos e os tipos primitivos (por exemplo: `integers`, `booleans` e `unit types`). Um pacote é uma tuplas contendo quatro componentes: pacote tipo (ordinárias ou marcador), remetente, o recebedor e dados. As combinações de tipos feita através de tuplas, listas ou registros. Sempre que foi necessário simular uma fila, preservando uma ordenação FIFO, uma lista foi utilizada. Listas também foram utilizadas quando era necessária a contagem do número de elementos em um local. Para combinar o *snapshot* a ser encontrado com os dados que devem ser armazenados, usamos

tuplas (`product color sets`). Os registros foram usados para armazenar processos de canais e de configuração.

Funções foram usadas quando necessário para gerar mais do que um símbolo de um componente de um `color set` (por exemplo, gera genM marcadores para cada canal de saída) e também quando foi necessário filtrar dados de uma lista. Filtragem de dados a partir de uma lista era necessário para a rotear pacotes para canais da direita e para armazenar os estados de cada canal, de acordo com o algoritmo de *snapshots* distribuídos. Funções também foram utilizados para produzir tuplas ou para recuperar uma componente de um tupla.

A.3.3 Representação de um Canal

Um canal foi modelado considerando-se um canal livre de erros, com *buffers* de tamanho infinito e entrega de mensagens de acordo com a ordem de chegada. Usamos listas para representar uma fila, como podem ser observados para a página `Channel` na figura A.3.

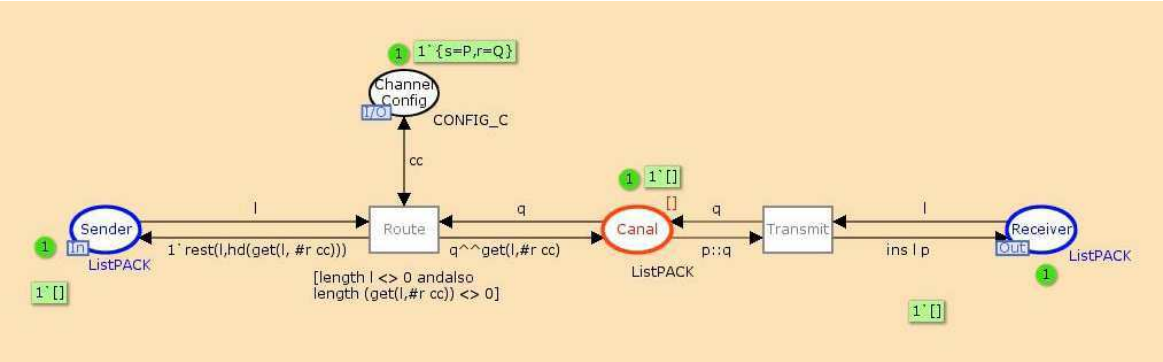

Figura A.3: Modelo CPN de um canal - Página `Channel`

A fim de juntar processos com canais então incluímos na página `Channel` a capacidade de roteamento, ou seja, entregando pacotes apenas para os seus respectivos recebedores, conforme informado no pacote cabeçalho. Três buffers foram utilizados: o primeiro como o canal de entrada, o segundo como o próprio canal e, o terceiro como o canal de saída. Duas transições são usadas entre estes buffers de roteamento e transmissão de dados.

A.3.4 Representação de um Processo

Um processo tem a capacidade de enviar pacotes por todos seus canais de saída e receber pacotes de todos os seus canais de entrada. Estas são as suas mais importantes funções, que são modelados na página `Process` na figura A.4, em conjunto com a capacidade de solicitar o registro de um estado global.

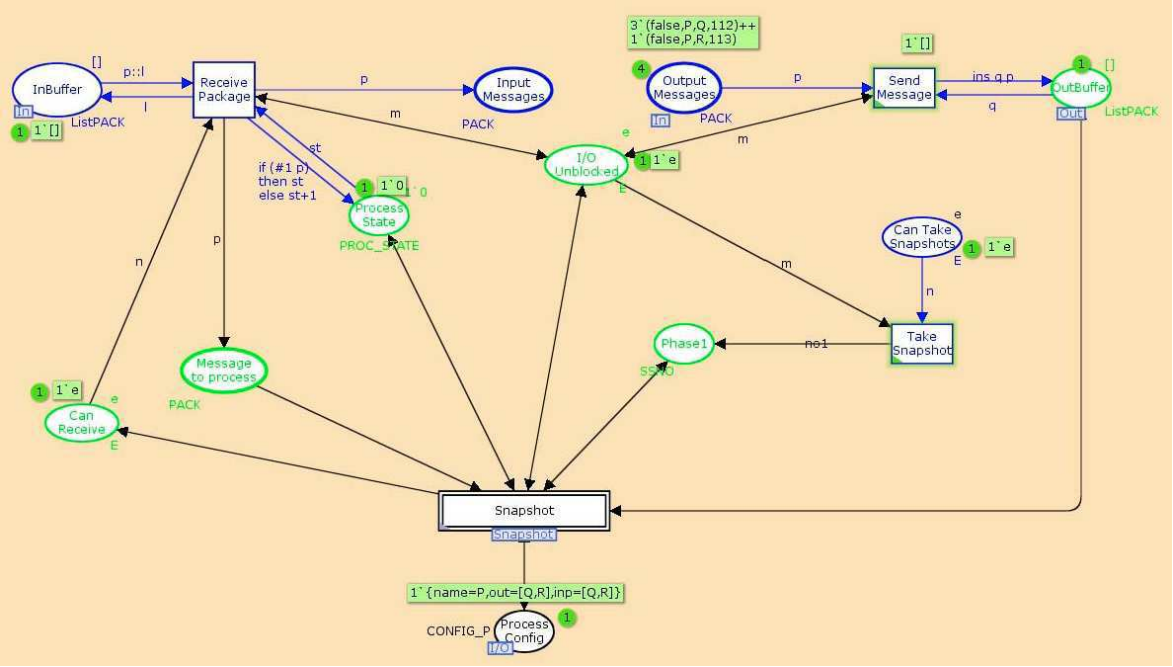

Figura A.4: Modelo CPN de um processo - Página `Process`

Mensagens a serem enviadas estão em um lugar chamado Mensagens de Saída (*Output Messages*), vinculada à página de nível superior para permitir que qualquer conjunto de mensagens em uma simulação seja escolhido pelo usuário no nível superior. As mensagens são transferidas para o *buffer* de saída quando a transição Enviar mensagem (*Send Message*) é executada.

As mensagens atingem o *buffer* de entrada e são recebidos quando a transição Recebedor de Mensagem (*Receive Message*) é executada. Esta ação armazena os dados em um local chamado Mensagens de Entrada (*Input Messages*).

Todo o resto da página está relacionada com o processo de captura de um estado global e será explicada na página `Snapshot`. Mas é importante para descrever aqui o recurso de tirar *snapshots*. A transição "Tirar" *Snapshot* (*Take Snapshot*) começa a gravação de um

estado global e o lugar *Can Take Snapshots* grava a quantidade de estados globais que um processo efetuou pedido de gravação.

A.3.5 Representação de um *Snapshot*

O processo de registro de um *snapshot* para compôr um estado global através de *snapshots* distribuídos foi modelado na página `Snapshot`, conforme ilustrado na figura A.5. A mesma página é utilizada tanto para a solicitação de um *snapshot* por um processo como por outros processos que recebem pedidos de um *snapshot*. Todo o processo é dividido em três fases, que são marcados por quatro eventos: iniciar *snapshot*, armazenar estado do processo, o enviar marcadores e finalizar o *snapshot*.

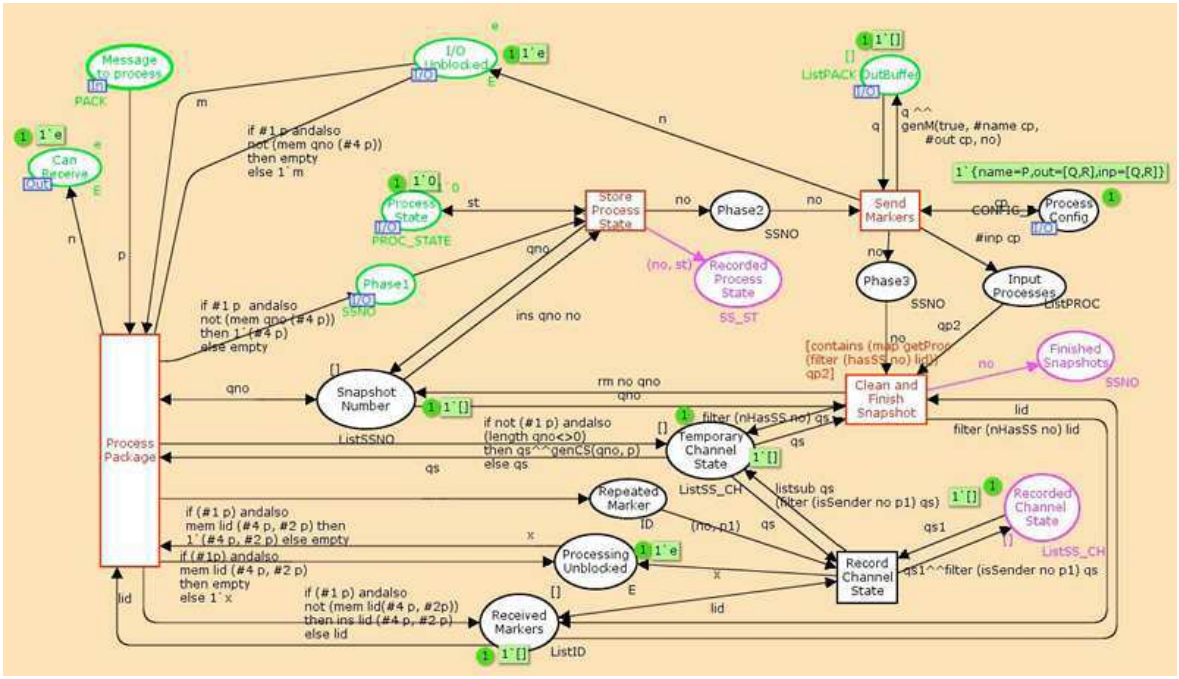

Figura A.5: Modelo CPN de uma registro de um *Snapshot* - Página `Snapshot`

Em primeiro lugar, ou o processo inicia a requisição de um *snapshot* (transição *Take Snapshot* na página `Process`) ou o processo recebe um marcador indicando que outros processos solicitaram um *snapshot*. Se este último ocorreu, a transição Processa Mensagem (*Process Message*) percebe que um snapshot deve ser sido solicitado e começa a receber o processo. Em ambos os casos, o *snapshot* entra na fase 1, onde entrada e saída são bloqueadas. Durante esta fase, o estado do processo está sendo gravado e o estado do canal

é registrado como vazio, e o final desta fase é assinalada pela transição Estado do Processo Gravado (*Store Process State*). Em nosso modelo, o estado do processo é simplesmente modelado como um `integer`, que muda conforme processamento de cada mensagem recebida. O estado de canal vazio não exige qualquer ação no modelo. Com os estados registrados, o *snapshot* entra na fase dois e termina quando a transição Enviar Marcadores (*Send Markers*) é executada. Neste momento, são enviados marcadores através de todos os canais de saída, o processo de entrada e saída são desbloqueados e o *snapshot* entra na fase três. Esta última fase é geralmente a mais longa, porque irá terminar somente quando um marcador para o *snapshot* corrente é recebido por todos os canais de entrada de cada processo. Uma função especial é utilizada na guarda de transição Limpo e termina *snapshot* (*Clean and Finish Snapshot*) que permite a transição quando esta última condição é estabelecida. Antes desta transição ser disparada pode acontecer de um processo (digamos q) recebe um marcador a partir de uma canal c mais de uma vez. Para cada um destes marcadores repetidos a transição Record Canal Estado (*Record Channel State*) é executada e o estado do canal é registrado com a seqüência de mensagens recebidas através c depois que o estado de q foi gravado e antes de q receber o marcador através de c.

Esta página é a mais complexa do modelo, porque requer o uso de listas de a gravação dos estados dos canais (lugares Estado do Canal Temporário (*Temporary Channel State*) e Estado do Canal Gravado (*Recorded Channel State*)) e também começou a gravar *snapshots* (lugar Número de *snapshot* (*Snapshot Number*)) e marcadores recebidos (local Marcadores Recebidos (*Received Markers*)), então os eventos concorrentes não afeta o algoritmo. Alguns outros lugares auxiliares, transições e arcos são usados para modelar o algoritmo, mas não são fundamentais para serem detalhados aqui.

A.4 Análise e Resultados da Modelagem

Um modelo é útil não só para a descrição de um sistema, mas também para extrair informações úteis sem a necessidade de se construir o mesmo. Análise é um meio de extrair informações a partir do modelo com o auxílio de ferramentas matemáticas. O tipo de análise formal que o *CPN Tools* é a partir da análise do espaço de estados através da utilização de ocorrência grafos. Uma ocorrência de um grafo CPN é um grafo dirigido onde existe um nó

para cada marcação alcançável e um arco para cada elemento possível.

Ocorrência de grafos são derivados de modelos finitos ou a partir simulações de modelos infinitos. O nosso modelo de um sistema distribuído com *n* processos, infelizmente, é um modelo infinito, devido ao uso de contadores, listas e fichas aleatoriamente geradas. Portanto, outros tipos de análise devem ser utilizados. Análise de desempenho poderia ser utilizado se o nosso modelo fosse uma CPN temporizada, mas não é o caso. Assim, decidimos utilizar simulações como uma análise técnica informal. Como modelos CPN são executáveis, é possível executar simulações onde cada passo é uma transição disparando fichas (*tokens*) entre lugares. Este tipo de simulação pode ser controlada com o *CPN Tools*, permitindo uma grande quantidade de cenários diferentes.

Monitores, em *CPN Tools*, são mecanismos utilizados para observar, inspecionar, controlar e modificar simulações. Quatro tipos de monitores são oferecidos, mas vamos centrar-se nos monitores de ponto crucial (*breakpoint*) e de coleta de dados. Monitores *breakpoint* são utilizados para interromper uma simulação. Monitores Coletores de dados são utilizados para extrair dados numéricos a partir de uma rede, como o número de vezes que uma transição foi disparada ou o número de fichas em um lugar para um determinado passo.

A.4.1 Cenários de Simulação

Alguns dados importantes para serem obtidos a partir de simulações podem ser: memória necessária para capturar um estado global, o tempo decorrido para registrar um estado global e a quantidade de mensagem gerais sobre o sistema distribuído devido a realização do algoritmo.

Para medir a memória necessária, deveríamos ser capazes de contar o número de pacotes armazenados por um *snapshot* em um determinado momento. Isto não foi viável para o nosso modelo, uma vez que o uso de fichas de listas não permitem calcular o número de pacotes em cada simulação passo. O tempo gasto para registrar um estado global depende de uma temporização na rede, o que não é o caso do nosso modelo. No entanto, foi possível registrar o número de transições disparadas que aconteceu durante a captura de um estado global, que está relacionado com o tempo. Finalmente, a mensagem era um *overhead* fácil de medir duranta a monitoração, necessitando apenas de uma simples contagem de mensagens regulares e marcadores.

Os seguintes cenários são baseados nas situações de pior caso: decidimos considerar uma conectividade completa entre processos, ou seja, cada processo é conectado diretamente com todos os outros processos.

Dois diferentes cenários simulação foram escolhidos:

1. Um estado global: fixando um único estado global, variamos o número de processos e medimos:

   (a) O número de transições disparadas (directamente relacionada com o tempo ellapsed), a partir do momento um processo solicita um estado global até o momento cada processo tem um *snapshot* e o estado global é completado;

   (b) A mensagem overhead (a taxa de envio marcadores durante o enviado ordinário mensagens) causada pelo pedido de um estado global a partir do momento de um processo solicita que, até ao momento em que for concluída;

2. Vários estados globais: fixando o número de processos, variamos o número de estados globais registrados e medimos:

   (a) O número de transições disparadas (diretamente relacionadas com o tempo decorrido), quando um processo realiza a solicitação do primeiro estado global até o momento que todos os estados globais foram concluídos;

   (b) A mensagem *overhead* (a taxa de envio marcadores durante o enviado ordinário de mensagens) causada pelo pedido de um estado global a partir do momento do primeiro estado global até o momento que todos os estados globais foram concluídos.

O segundo cenário foi repetido para 10, 15, 20 e 25 processos.

Cada simulação foi feita apenas uma vez, mas os resultados não foram notados significativamente diferente quando se repete uma simulação.

A.4.2 Resultados de Simulação

A primeira simulação mostra alguns resultados evidentes como podemos ver na Figura A.6.

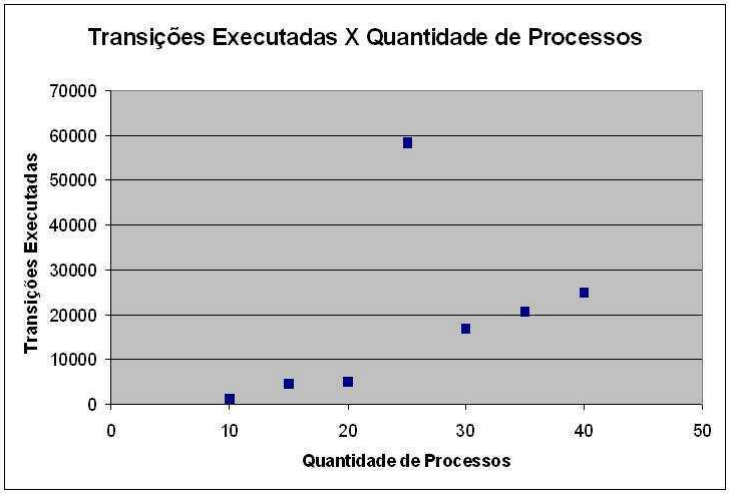

Figura A.6: Transições executada *versus* Quantidade de Processos (para um estado global)

Como é esperado, à medida que aumenta o número de processos que fazem parte de um sistema distribuído, é preciso mais tempo para registrar um estado global usando *snapshots* distribuídos. O número de transições disparadas, com exceção de uma anomalia para 15 processos, parece crescer linearmente com o número de processos.

A mensagem *overhead* é a taxa de mensagens de monitoração (marcadores) sobre as mensagens ordinárias do sistema. Ela mostra o quão intrusivo um sistema de monitoração se comporta quando monitora um sistema distribuído. A Figura A.7 mostra os resultados de uma série de 10 a 40 processos.

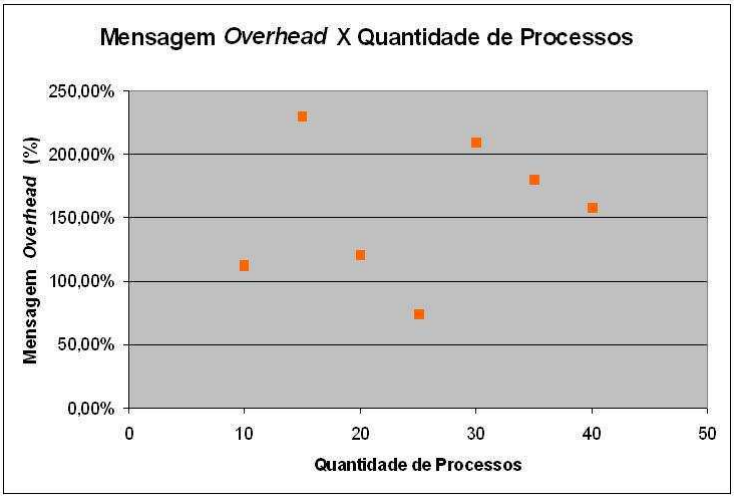

Figura A.7: *Overhead* de Mensagens *versus* Quantidade de Processos (para um estado global)

Isto pode ser inferido como um algoritmo muito intrusivo (*overhead* entre 70 e 250%), embora não temos dados suficientes para inferir uma taxa crescente com o número de processos. Os resultados parecem um pouco aleatório, o que indica que mais simulações são necessárias para entender melhor esse comportamento. Talvez, nós precisássemos repetir as simulações para cada número de processos e calcular a média. Aumentando o número de processos também pode ser interessante para descobrir esta tendência, embora os custos de simulação iriam aumentar fortemente.

Pode ser interessante medir os efeitos da gravação de mais de um estado global, uma vez que a meta de monitoração requer muitos estados global para inferir a ordenação de eventos, por exemplo. Isso foi feito na segunda simulação. A Figura A.8 mostra que o número de transições disparadas (e, consequentemente, o tempo decorrido) necessários para o registro de estados globais cresce com o número de estados registrados.

Podemos inferir que as transições disparadas crescem linearmente com o número de estados, mas não podemos extrapolar os resultados de linearidade para a variável tempo, uma vez que não temos uma temporização na rede. Podemos notar também das várias curvas que cresce o número de processos cresce também o número de transições disparadas, como ficou demonstrado na primeira simulação.

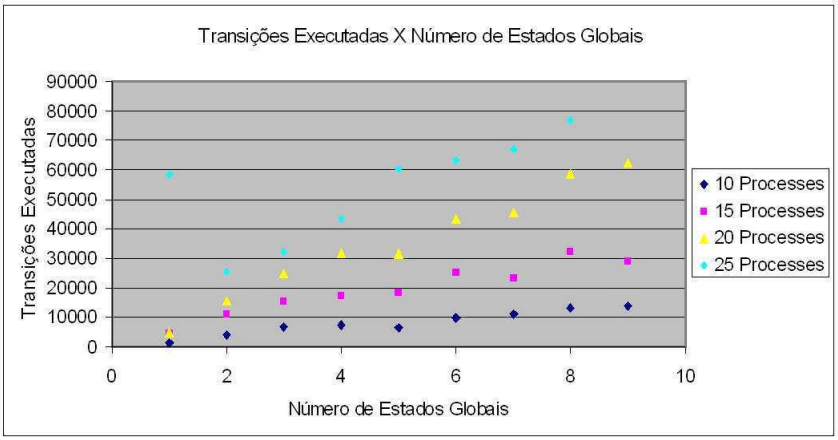

Figura A.8: Transições disparadas *versus* Número de Estados Globais (para 10, 15, 20 e 25 processos)

O *overhead* com o número de *snapshots* também tem indícios de ser elevado quando aumenta-se o número de estados globais registrados. Tal como na primeira simulação, não se pode inferir uma crescente tendência, uma vez que os dados são escassos na simulação. Mas

os resultados mostram que, pelo menos, independentemente do número de estados globais gravados, o *overhead* é elevado, como podemos ver na Figura A.9. Repetindo simulações para remover a aleatoriedade dos resultados poderá ser a primeira solução. O aumento do número de estados globais registados também poderá mostrar a tendência de aumento do *overhead*.

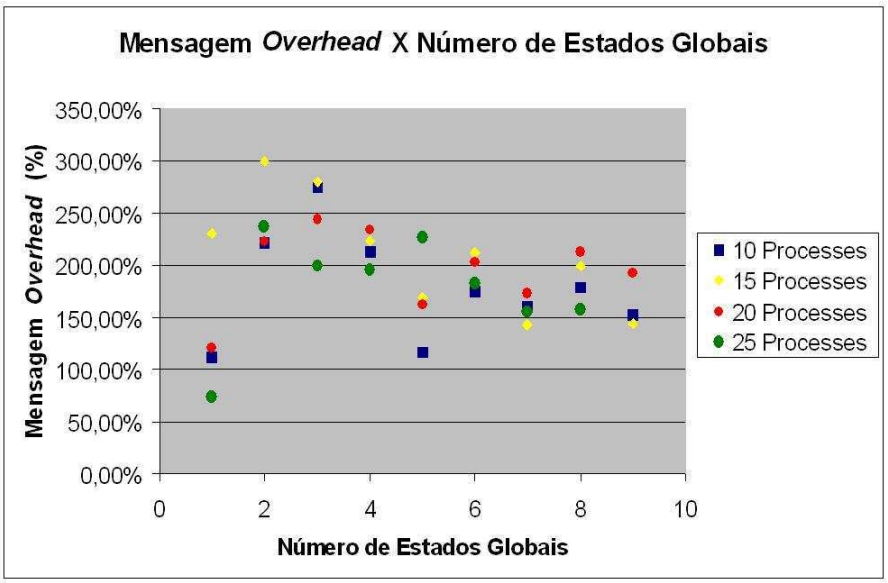

Figura A.9: *Overhead* de Mensagens *versus* Quantidade de Estados Globais (para 10, 15, 20 e 25 processos)

Em suma, os resultados mostram que o tempo de registro de estados globais cresce com o número de processos e com o número de estados gravados. Além disso, o algoritmo impõe um considerável *overhead* da comunicação, embora mais simulações poderia torná-la mais clara, em termos da tendência de crescimento.