Universidade Federal do Maranhão
Centro de Ciências Exatas e Tecnologia
Programa de Pós-Graduação em Engenharia de Eletricidade

Berto de Tácio Pereira Gomes

AGST (Autonomic Grid Simulation Tool): uma ferramenta para modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores

São Luís

2012

Berto de Tácio Pereira Gomes

AGST (Autonomic Grid Simulation Tool): uma ferramenta para modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão como requisito parcial para a obtenção do grau de MESTRE em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva

Doutor em Ciência da Computação – UFMA

São Luís

2012

Gomes, Berto de Tácio Pereira.

AGST (Autonomic Grid Simulation Tool): uma ferramenta para modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores / Berto de Tácio Pereira Gomes – São Luís, 2012.

134 f.

Orientador: Francisco José da Silva e Silva.

Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade, 2012.

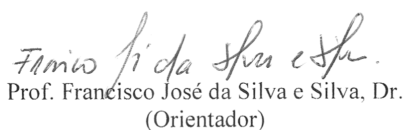
1. Computação em grade. 2. Computação autonômica. 3. Simulação. I. Título.

CDU 004

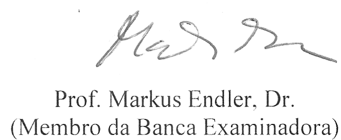
AGST(AUTONOMIC GRID SIMULATION TOOL):UMA FERRAMENTA PARA MODELAGEM, SIMULAÇÃO E AVALIAÇÃO DE ABORDAGENS AUTONÔMICAS PARA GRADES DE COMPUTADORES

Berto de Tácio Pereira Gomes

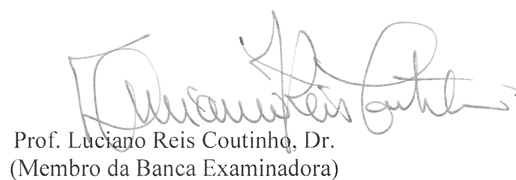
Dissertação aprovada em 09 de março de 2012.



Prof. Francisco José da Silva e Silva, Dr.
(Orientador)



Prof. Markus Endler, Dr.
(Membro da Banca Examinadora)



Prof. Luciano Reis Coutinho, Dr.
(Membro da Banca Examinadora)

Aos meus pais, minhas irmãs, meus amigos e meus professores.

Resumo

Grades de computadores são caracterizadas pelo alto dinamismo de seu ambiente de execução, alta heterogeneidade de recursos e tarefas, e por requererem grande escalabilidade. Essas características tornam tarefas como configuração, manutenção e recuperação em caso de falhas bastante desafiadoras e cada vez mais difíceis de serem realizadas exclusivamente por agentes humanos.

O termo Computação Autonômica denota sistemas computacionais capazes de mudar seu comportamento dinamicamente em resposta a variações do ambiente de execução. Para isso, o software é geralmente organizado seguindo-se a arquitetura MAPE-K (*Monitoring, Analysis, Planning, Execution and Knowledge*), na qual gerentes autonômicos realizam as atividades de monitoramento do ambiente de execução, análise de informações de contexto, planejamento e execução de ações de reconfiguração dinâmica, compartilhando algum conhecimento sobre o sistema controlado. Diversos esforços de pesquisa recentes buscam aplicar técnicas de computação autonômica à computação em grade, provendo-se maior autonomia e reduzindo-se a necessidade de intervenção humana na manutenção e gerenciamento destes ambientes computacionais, criando assim o conceito de grade autonômica.

Esta dissertação apresenta uma nova ferramenta de simulação que tem por objetivo auxiliar o desenvolvimento e avaliação de abordagens autonômicas para grades de computadores denominada AGST (*Autonomic Grid Simulation Tool*). A principal contribuição dessa ferramenta é a definição e implementação de um modelo de simulação baseado na arquitetura MAPE-K, que pode ser utilizado para simular todas as funções de monitoramento, analise e planejamento, controle e execução, permitindo assim a simulação de grades autonômicas. AGST provê ainda o suporte à execução de adaptações paramétricas e composicionais dos elementos gerenciados. Este trabalho também apresenta dois estudos de caso nos quais a ferramenta proposta foi utilizada com sucesso no processo de modelagem, simulação e avaliação de abordagens para grades computacionais.

Palavras-chave: Computação em Grade, Computação Autonômica, Simulação.

Abstract

Computer Grids are characterized by the high dynamism of its execution environment, resources and tasks heterogeneity, and high scalability. These features turn tasks such as configuration, maintenance and failure recovery quite challenging and is becoming increasingly difficult to perform them only by human agents.

The autonomic computing term denotes computer systems capable of changing their behavior dynamically in response to changes in the execution environment. For achieving this, the software is generally organized following the MAPE-K (Monitoring, Analysis, Planning, Execution and Knowledge) model, in which autonomic managers perform of the execution environment sensing activities, context analysis, planning and execution of dynamic reconfiguration actions, based on shared knowledge about the controlled system. Several recent research efforts seek to apply autonomic computing techniques to grid computing, providing more autonomy and reducing the need for human intervention in the maintenance and management of these computing environments, thus creating the concept an autonomic grid.

This thesis presents a new simulator tool for assisting the development and evaluation of autonomic grid approaches called AGST (Autonomic Grid Simulation Tool). The major contribution of this tool is the definition and implementation of a simulation model based on the MAPE-K autonomic management cycle, that can be used to simulate the monitoring, analysis and planning, control and execution functions, allowing the simulation of an autonomic computing grid. AGST also provides support for parametric and compositional dynamic adaptations of managed elements. This work also presents two case studies where the proposed tool was successfully used for the modeling, simulation and evaluation of approaches to grid computing.

Keywords: Grid Computing, Autonomic Computing, Simulation.

Agradecimentos

Ao bom Deus em primeiro lugar, pela minha vida e por todas as maravilhas.

Ao meu orientador, o Prof. Francisco José da Silva e Silva, pelo exemplo, apoio, compreensão, orientação, e por acompanhar sempre de perto o andamento das atividades. Nunca entendi como alguém tão ocupado consegue ser tão presente.

Aos professores Markus Endler (PUC-Rio) e Luciano Reis Coutinho (UFMA) por terem aceito fazer parte da banca examinadora.

Aos meus familiares pelo apoio que me deram trajetória. Eles me fazem querer ser alguém melhor todos os dias.

Ao membros do LSD-UFMA, com os quais compartilhei alegrias, tristezas, e conhecimento. Registro a importância de Jesseildo F. Gonçalves, Antônio Eduardo B. Viana, Márcio R. M. Martins, Rafael V. L. Araújo, José da S. Lucena, e Ariel S. Teles para a conclusão desta pesquisa.

Aos amigos do IFMA, Jefferson A. da Silva e Jackson A. da Silva, que desde a época da graduação sempre me ajudam em momentos difíceis.

Aos meus orientadores da graduação no IFMA, o Prof. Omar Andrés C. Cortes e o Prof. Rafael F. Lopes, que foram as primeiras pessoas que me incentivaram a fazer mestrado, ambos sempre me dão bons conselhos e os mesmos sempre foram grandes referenciais na minha trajetória acadêmica.

A Teresa Cristina de C. P. Mendes, pelo apoio e incentivo que concede constantemente às minhas pesquisas.

A UFMA e ao PPGEE pela estrutura dada a execução deste trabalho. Agradeço também ao chefe do PPGEE, ao coordenador, aos professores, e ao técnico administrativo Alcídes, pela dedicação desses profissionais.

A Camilla Ferreira Maciel, pelo apoio e compreensão durante esses anos de mestrado.

"Deus não me deu asas para voar, mas subindo alguns degraus por dia, mesmo cansado, chegarei aos céus"

Autor Desconhecido

Lista de Figuras

2.1 Ciclo de Gerenciamento de Sistemas [16] 22

2.2 Arquitetura MAPE-K [39]. 23

2.3 Arquitetura derivada do Sistema Ashby[Pleaseinsertintopreamble]s Ultra-stable [16]. 24

2.4 Combinação do código base do sistema com os aspectos 30

3.1 Taxonomia dos sistemas de grade 38

5.1 Arquitetura do GridSim [10] 52

5.2 Principais componentes do OGST 59

5.3 Arquitetura do SimGrid 62

6.1 Arquitetura do AGST 69

7.1 Classes que implementam o gerenciamento de ciclos autonômicos. 95

7.2 Classes utilizadas pela função de monitoramento. 97

7.3 Classes utilizadas pela função de análise e planejamento. 99

7.4 Classes utilizadas pela função de controle e execução de reconfigurações. 101

8.1 Ciclos Autonômicos da Abordagem de Viana. 111

8.2 Monitores da Abordagem de Viana. 112

8.3 Analisadores da Abordagem de Viana. 114

Lista de Tabelas

3.1 Cinco maiores classes de aplicações de grades 36

Lista de Siglas

AGST Autonomic Grid Simulator Tool.

ARM Application Replication Manager.

BoT bag-of-task.

BSP Bulk Synchronous Parallel.

CPU Central Processing Unit.

ECA Evento-Condição-Ação.

FTA Failure Trace Archive.

GFG Grid Feature Generator.

GS Grid Scheduler.

GWA Grid Workload Archive.

GWF Grid Workload Format.

IBM International Business Machines.

JVM Java Virtual Machine.

MAPE-K Monitoring, Analysis, Planning, Execution and Knowledge.

MPI Message Passing Interface.

MTBF Mean Time Between Failures.

OGST Opportunistic Grid Simulator Tool.

POA Programação Orientada a Aspectos.

RC Resource Controller.

RDS Resource Data Storage.

SDRM Simulation Data Record Manager.

TF Tolerância a Falhas.

UAST User Application Submission Tool.

UFMA Universidade Federal do Maranhão.

Sumário

Lista de Figuras vi

Lista de Tabelas vii

Lista de Siglas viii

1 Introdução 15

1.1 Objetivos 17

1.2 Estrutura da Dissertação 18

2 Introdução a Computação Autonômica 19

2.1 Conceitos Básicos de Computação Autonômica 19

2.1.1 Propriedades de Sistemas Autonômicos 20

2.2 Arquiteturas para Sistemas Autonômicos 21

2.2.1 Monitoramento 25

2.2.2 Análise e Planejamento 25

2.2.3 Controle e Execução 26

2.3 Abordagens para o Desenvolvimento de Sistemas Autonômicos 27

2.3.1 Reflexão Computacional 27

2.3.2 Programação Orientada a Aspectos (POA) 29

2.4 Conclusão 32

3 Grades de Computadores 34

3.1 Introdução a Grades de Computadores 34

3.1.1 Desafios para Construção de um Middleware de Grade 36

3.2 Taxonomia dos Sistemas de Grade 38

3.3 Grades de Computadores Pessoais (*Desktop Grids*) 39

3.4 Grades Autonômicas 41

3.5 Conclusões 42

4 Introdução a Simulação Computacional **44**

4.1 Conceitos Básicos de Simulação Computacional 44

4.2 Simulação Baseada em Eventos Discretos 45

4.3 Aplicações da Simulação Computacional 47

4.4 Conclusão 49

5 Trabalhos Relacionados **50**

5.1 GriSim 50

5.1.1 Arquitetura do GridSim 52

5.2 OGST 53

5.2.1 Extensões providas pelo OGST ao GridSim 53

5.2.2 Arquitetura do OGST 58

5.3 Outros Simuladores Baseados no GridSim 60

5.3.1 GSSIM 60

5.3.2 Alea 61

5.4 SimGrid 61

5.4.1 Arquitetura do SimGrid 62

5.5 DSiDE 63

5.6 Conclusão 64

6 AGST **66**

6.1 Introdução ao AGST 66

6.2 Requisitos para o desenvolvimento do AGST 68

6.3 Arquitetura do AGST 69
6.4 Funcionalidades do MAPE-K Simulation Framework 72
6.4.1 Gerenciamento de Ciclos Autonômicos 72
6.4.2 Monitoramento 75
6.4.3 Análise e Planejamento 77
6.4.4 Controle e Execução de Ações de Reconfiguração 79
6.4.5 Adaptações Paramétricas e Composicionais 79
6.4.6 Transferência de Estado 80
6.4.7 Sincronização 81
6.5 Limitações do AGST 84
6.6 Conclusões 85

7 Implementação do AGST 87

7.1 Bibliotecas utilizadas na implementação do AGST 87
7.1.1 *Java Reflection* 87
7.1.2 *AspectJ* 90
7.2 Implementação do MAPE-K Simulation Framework 94
7.2.1 Gerenciamento de Ciclos Autonômicos 94
7.2.2 Monitoramento 96
7.2.3 Análise e Planejamento 99
7.2.4 Controle e Execução de Reconfigurações 100
7.2.5 Adaptações Dinâmicas e Transferência de Estado 102
7.2.6 Sincronização 102
7.3 Considerações sobre o uso de POA e Reflexão na Sincronização 104
7.4 Conclusões 105

8 Estudos de Caso 107

8.1 Avaliação de uma Abordagem Autonômica para Tolerância a Falhas 107

8.1.1 1º Nível de Adaptação: Reconfiguração Paramétrica no *Checkpointing* . . . 108
8.1.2 1º Nível de Adaptação: Reconfiguração Paramétrica na Replicação 109
8.1.3 2º Nível de Adaptação: Reconfiguração Estrutural 110
8.1.4 Implementação da Abordagem de Viana 110
8.1.5 Resultados e Discussões 117
8.2 Avaliação de um Mecanismo de Gerenciamento de Execução de Aplicações 118
8.3 Conclusão 120

9 Conclusões e Trabalhos Futuros **122**

Referências Bibliográficas **125**

1 Introdução

Ao longo desta última década, a abordagem da Computação em Grade tem sido explorada para a execução de aplicações de alto desempenho. Grades de Computadores são sistemas que coordenam recursos distribuídos, como processadores, software, dados e periféricos conectados através de uma rede, com a finalidade de formar um sistema distribuído de larga escala que pode ser utilizado para realizar computações diversificadas. Estes recursos computacionais podem estar geograficamente dispersos e pertencerem a diversas organizações. Desta forma, uma questão fundamental em um sistema de computação em grade é que recursos de diferentes organizações podem ser reunidos para permitir a colaboração de um grupo de pessoas ou instituições. Tal colaboração é realizada sob a forma de uma Organização Virtual.

O componente central de uma arquitetura de grade é o seu *middleware*. Ele é utilizado para esconder a natureza heterogênea e a complexidade decorrente da distribuição dos recursos que compõem a grade. O *middleware* disponibiliza aos usuários e aplicações uma visão homogênea do ambiente, provendo interfaces padronizadas para diversos serviços. Um aspecto importante no desenvolvimento de um *middleware* de grade é o tratamento adequado da dinamicidade do ambiente. A Computação em Grade tornou possível a execução de aplicações cuja composição e interação é altamente dinâmica [63]. Além disso, a própria infraestrutura da grade é heterogênea e dinâmica. O dinamismo das aplicações e da infraestrutura da grade, a alta escalabilidade e alta heterogeneidade dos ambiente de grade tornam tarefas como a configuração, a manutenção, e a recuperação em caso de falhas, altamente complexas para serem realizadas exclusivamente por agentes humanos. Desta forma, deve-se prover mecanismos automatizados que facilitem o gerenciamento de grades de computadores.

O termo Computação Autonômica denota sistemas computacionais capazes de mudar seu comportamento dinamicamente em resposta a variações do ambiente de execução de acordo com políticas e objetivos estabelecidos, de forma análoga ao comportamento auto-regulatório de sistemas biológicos. Para isso, o software é

usualmente organizado seguindo-se o modelo *Monitoring, Analysis, Planning, Execution and Knowledge* (MAPE-K), no qual gerentes autonômicos realizam as atividades de sensoriamento do ambiente de execução, análise de contexto, planejamento e execução de ações de reconfiguração dinâmica, compartilhando algum conhecimento sobre o sistema controlado [60] [39].

Diversas pesquisas recentes buscam aplicar técnicas de Computação Autonômica à Computação em Grade, provendo-se maior autonomia e reduzindo-se a necessidade de intervenção humana na manutenção e gerenciamento destes ambientes computacionais, criando assim o conceito de Grade Autonômica. Alguns dos aspectos investigados incluem mecanismos autonômicos de autoproteção, tais como a detecção de sobrecargas que potencialmente poderiam levar à interrupção dos serviços; auto-otimização, através de ajustes de parâmetros e algoritmos ao se detectar degradações de desempenho; autocura, para contornar eventuais falhas parciais do sistema; e autoconfiguração, provendo-se máquinas virtuais configuradas sobre demanda e dinamicamente alocadas a recursos físicos [32] [41] [41] [15] [1] [63].

Durante o desenvolvimento de um *middleware* de grade, os pesquisadores frequentemente utilizam ferramentas de simulação para validar novos conceitos em suas implementações. Os simuladores de grade desempenham um papel fundamental na avaliação de novas abordagens para a construção de componentes de uma grade, uma vez que são inúmeras as dificuldades de acesso a um ambiente de grade com recursos em larga escala para testar e avaliar as mesmas, sejam elas autonômicas ou não. Devido ao alto custo econômico para adquirir ou pagar pelo uso de uma grande quantidade de recursos computacionais, geralmente os pesquisadores têm acesso apenas a uma restrita quantidade de recursos. Isso limita a capacidade de avaliar situações que demandam por recursos em grade quantidade. Além disso, devido à natureza dinâmica dos ambientes de grade e a dificuldade de coordenar seus usuários, é difícil explorar cenários que envolvam um grande número de aplicações e uma elevada quantidade de usuários de forma repetitiva e controlada.

Neste trabalho apresentamos um novo simulador de grades de computadores denominado *Autonomic Grid Simulator Tool* (AGST) [22]. Esse simulador possui como diferencial, com relação a outros simuladores de grades conhecidos, a disponibilização de um arcabouço para a modelagem e simulação de ciclos de gerenciamento autonômico em grades baseado na arquitetura MAPE-K e,

dessa forma, permite a construção e avaliação de abordagens autonômicas voltadas para ambientes de grade. Esse simulador permite o monitoramento de recursos do ambiente de grade, análise de informações de contexto, controle e execução de ações de reconfiguração dinâmica. O AGST implementa dois mecanismos de reconfiguração dinâmica: a Adaptação Paramétrica, que consiste na alteração de variáveis que determinam o comportamento da grade; e a Adaptação Composicional, que consiste na troca de algoritmos ou partes estruturais do *middleware* de grade, possibilitando que este adote novas estratégias para tratar novas situações e reagir às mudanças de contexto no ambiente de grade.

1.1 Objetivos

A pesquisa à qual se refere esta dissertação de mestrado tem como objetivo geral desenvolver uma ferramenta computacional capaz de auxiliar desenvolvedores de *middlawares* de grades e pesquisadores de forma geral no processo de modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores.

Os objetivos específicos desta pesquisa são:

- Levantar o estado da arte em computação autonômica e sua aplicação em ambientes de grades computacionais;
- Levantar o estado da arte em simuladores para ambientes de grades de computadores, investigando-se a disponibilidade de mecanismos de suporte à modelagem e simulação de grades autonômicas;
- Projetar e implementar funcionalidades para a modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores.
- Avaliar a adequação do modelo de simulação proposto ao processo de desenvolvimento de abordagens autonômicas para grades através de estudos de caso utilizando o AGST.

1.2 Estrutura da Dissertação

Esta dissertação está organizada como segue:

- O Capítulo 2 apresenta os principais conceitos sobre a Computação Autonômica e mostra as principais arquiteturas e abordagens para o desenvolvimento de sistemas autonômicos.

- O Capítulo 3 apresenta uma breve fundamentação teórica sobre Grades de Computadores. São apresentados conceitos, características, aplicações, desafios e uma classificação de abordagens para a computação em grade.

- O Capítulo 4 aborda os principais conceitos relacionados à Simulação Computacional, suas aplicações, bem como descreve o funcionamento de alguns tipos de simulação, entre elas a simulação baseada em eventos discretos que corresponde a técnica de simulação utilizada pelo AGST.

- O Capítulo 5 discute relevantes trabalhos relacionados com esta pesquisa. Nesse capítulo são apresentados os principais simuladores de grades de computadores, descrevendo-se suas arquiteturas e funcionalidades.

- O Capítulo 6 aborda o simulador AGST, apresentando as motivações e os requisitos para seu desenvolvimento. Este capítulo descreve ainda a arquitetura do AGST e suas funcionalidades, em especial aquelas responsáveis pela modelagem e simulação de abordagens autonômicas. Ainda nesse capítulo é feita ainda uma breve análise das principais limitações de uso do AGST.

- O Capítulo 7 descreve os principais aspectos da implementação do AGST, apresentando as principais bibliotecas utilizadas e descrevendo detalhes das principais classes e métodos que devem ser utilizadas para modelagem e simulação de abordagens autonômicas em grades.

- O Capítulo 8 apresenta dois estudos de caso de utilização do AGST na avaliação de abordagens voltadas para *Desktop Grids*.

- O Capítulo 9 apresenta as conclusões obtidas a partir desta pesquisa e apresenta trabalhos futuros que podem ser desenvolvidos a partir deste esforço inicial.

2 Introdução a Computação Autonômica

As grades de computadores são sistemas que possuem grande escalabilidade, alta heterogeneidade e ambiente de execução dinâmico. Essas características tornam o gerenciamento desses sistemas altamente complexo. A computação autonômica agrega vários campos da computação com o objetivo de dotar sistemas complexos com a capacidade de autogerenciamento.

Neste capítulo são apresentados alguns conceitos de computação autonômica, bem como algumas arquiteturas e abordagens para o desenvolvimento de sistemas autonômicos. Em relação as arquiteturas, será dada uma maior ênfase à arquitetura MAPE-K, por ser a mais utilizada. Todas as fases do ciclo de gerenciamento autonômico definidas por essa arquitetura são explicadas, na seguinte ordem: Monitoramento, Análise e Planejamento, Controle e Execução de Reconfigurações. Em relação as abordagens para o desenvolvimento de sistemas autonômicos, são abordadas: a Reflexão Computacional e a Programação Orientada a Aspectos (POA).

2.1 Conceitos Básicos de Computação Autonômica

O termo computação autonômica surgiu em 2001 com um manifesto publicado por Paul Horn, pesquisador da IBM, que lançou um desafio sobre o problema da crescente complexidade de gerenciamento do software [40]. O conceito de computação autonômica foi inspirado na corpo humano e está relacionado às reações fisiológicas involuntárias do sistema nervoso [60]. No corpo humano, o sistema nervoso autonômico cuida de reflexos inconscientes, isto é, de funções corporais que não requerem nossa atenção como a contração e expansão da pupila, funções digestivas do estômago e intestino, a frequência e profundidade da respiração, a dilatação e constrição de vasos sanguíneos, etc. Esse sistema reage às mudanças, ou perturbações, causadas pelo ambiente através de uma série de modificações, a fim de conter as perturbações causadas ao seu equilíbrio interno.

- **Autocura (*self-healing*)**: o sistema deve possuir a habilidade de identificar potenciais problemas e de se reconfigurar de forma a continuar operando normalmente;

- **Aberto (*open*)**: o sistema deve ser portável para diversas arquiteturas de hardware e software e, consequentemente, deve ser construído a partir de protocolos e interfaces abertos e padronizados;

- **Capacidade de Antecipação (*anticipatory*)**: o sistema deve ser capaz de antecipar, na medida do possível, suas necessidades e comportamentos considerando seu contexto.

As propriedades de autoconfiguração, auto-otimização, autocura e autoproteção são suficientes para realizar a visão original do termo [60]. Um requisito inerente aos sistemas autonômicos é que eles sejam adaptativos, ou seja, que sejam capazes de se reconfigurar dinamicamente de acordo com alterações percebidas em seu ambiente de execução. Desta forma, mecanismos de autoconsciência, consciência de contexto e autoconfiguração são usualmente considerados requisitos para a incorporação de mecanismos de auto-otimização e autocura.

2.2 Arquiteturas para Sistemas Autonômicos

Arquiteturas para sistemas autonômicos visam formalizar um quadro de referência que identifica as funções comuns e estabelece os alicerces necessários para automatizar o ciclo de gerenciamento de sistemas, conforme mostrado na Figura 2.1. De forma geral, as arquiteturas para sistemas autonômicos definem as seguintes funções: Monitoramento, Análise e Planejamento, Controle e Execução.

A seguir temos um detalhamento das atividades desenvolvidas em cada função:

- **Monitoramento ou Medição:** função que coleta, agrega, correlaciona e filtra dados sobre recursos gerenciados. Dados coletados incluem: informações de topologia, eventos, métricas, propriedades de configuração, etc. Recursos gerenciados incluem servidores, unidades de armazenamento, banco de dados, servidores de aplicação, serviços, aplicações, etc.

Ao contrário do corpo humano, em sistemas computacionais o autogerenciamento não é desempenhado involuntariamente, mas sim, através de tarefas que administradores delegam aos sistemas de acordo com políticas adaptativas. Estas últimas determinam o tipo de ação a ser executada em diferentes situações [40]. Dessa forma, os sistemas de computação autonômica são capazes de se autogerenciarem e se adaptarem dinamicamente às mudanças a fim de restabelecerem seus equilíbrios de acordo com as políticas e os objetivos do negócio do sistema. Para isso, devem dispor de mecanismos efetivos que os permitam monitorar, controlar e regular a si próprios, bem como recuperarem-se de problemas sem a necessidade de intervenções externas [61].

2.1.1 Propriedades de Sistemas Autonômicos

A essência da computação autonômica é o auto-gerenciamento. Para implementá-lo, o sistema deve ao mesmo tempo estar atento a si próprio e ao seu ambiente. Desta forma, o sistema deve conhecer com precisão a sua própria situação e ter consciência do ambiente operacional em que atua. Conforme Hariri [38], o termo computação autonômica tem sido utilizado para denotar sistemas que possuem algumas das seguintes propriedades:

- **Autoconsciência (*self-awareness*)**: o sistema conhece a si próprio: seus componentes e inter-relações, seu estado e comportamento;
- **Consciência do contexto (*context-aware*)**: o sistema deve ser ciente do contexto de seu ambiente de execução e ser capaz de reagir a mudanças em seu ambiente;
- **Autoconfiguração (*self-configuring*)**: o sistema deve ajustar dinamicamente seus recursos baseado em seu estado e no estado do ambiente de execução;
- **Auto-otimização (*self-optimizing*)**: o sistema é capaz de detectar degradações de desempenho e de realizar funções para auto-otimização;
- **Autoproteção (*self-protecting*)**: o sistema é capaz de detectar e proteger seus recursos de atacantes internos e externos, mantendo sua segurança e integridade geral;

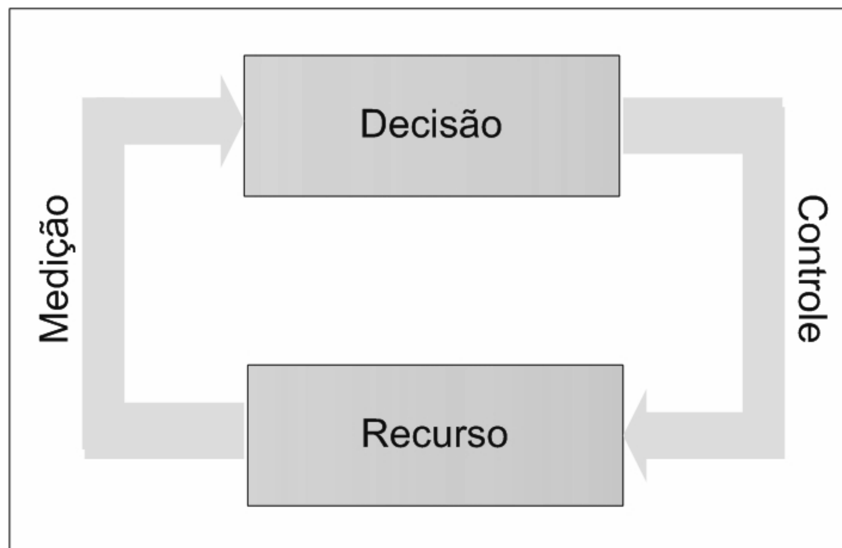


Figura 2.1: Ciclo de Gerenciamento de Sistemas [16]

- **Análise e Planejamento:** a função que examina os dados coletados e determina se devem ser feitas mudanças sobre as políticas ou estratégias correntes. Essa tomada de decisão assegura convergência em relação a valores limiares de parâmetros como desempenho, disponibilidade e segurança.

- **Controle e Execução:** a função que escalona e executa as mudanças identificadas como necessárias pela função de análise planejamento.

Em 2003, a IBM propôs uma versão automatizada do ciclo de gerenciamento de sistemas chamado de MAPE-K (\textit {Monitoring, Analysis, Planning, Execution and Knowledge}) [40], representado na Figura 2.2. Este modelo está sendo cada vez mais utilizado para interrelacionar os componentes arquiteturais dos sistemas autonômicos. De acordo com essa arquitetura, um sistema autonômico é formado por um conjunto de elementos autonômicos. Cada elemento autonômico (*Autonomic Element*) é constituído por um único Gerente Autonômico (*Autonomic Manager*). Esse gerente é responsável por monitorar e controlar um ou mais elementos gerenciados e por promover a produtividade dos recursos e a qualidade dos serviços providos pelo componente do sistema no qual está instalado. O elemento gerenciado (*Managed Element*) representa qualquer recurso de software e/ou hardware, ao qual é dado o comportamento autonômico através do acoplamento de um Gerente Autonômico. O Elemento Gerenciado pode ser, por exemplo, um servidor de aplicações *Web* ou banco de dados, um componente de software específico em um aplicativo (por exemplo, o

otimizador de consulta em um banco de dados), o sistema operacional, um conjunto de máquinas em um ambiente de rede, etc [39].

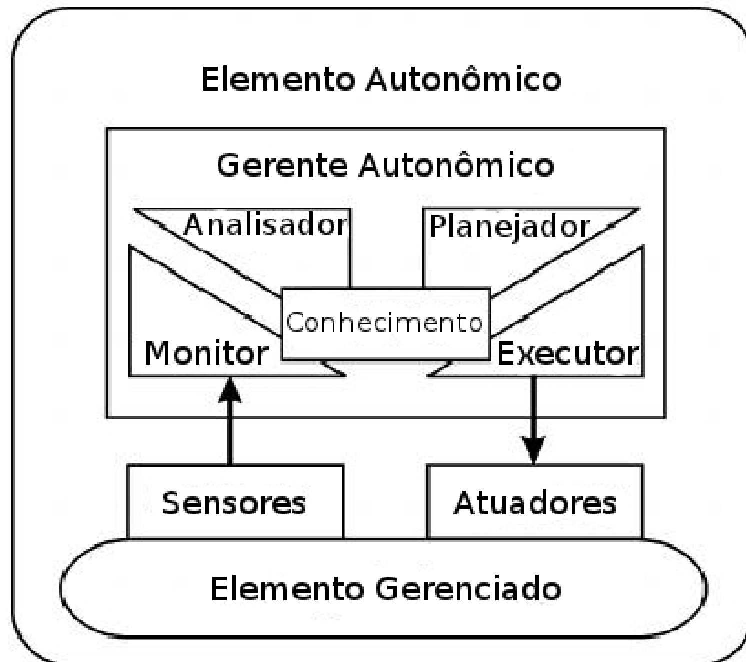


Figura 2.2: Arquitetura MAPE-K [39].

Os Sensores são componentes responsáveis por coletar informações do elemento gerenciado. Estes dados podem ser os mais diversos como, por exemplo, o tempo de resposta das requisições dos clientes, caso o elemento gerenciado seja um servidor de aplicações *Web*. As informações coletadas pelos sensores são enviadas ao Monitor, os quais realiza a interpretação e pré-processamento desses dados, colocando-os em um nível mais alto de abstração para, após isso, serem enviadas para a etapa seguinte do ciclo: a fase de análise e planejamento. Nesta fase temos como produto um plano de ações, que consiste de um conjunto de ações a serem executadas pelo Executor. Os componentes responsáveis por fazer as alterações no ambiente são chamados de Atuadores. Somente os Sensores e Atuadores possuem acesso direto ao elemento gerenciado. Como durante todo ciclo de gerenciamento autonômico existe a necessidade de tomadas de decisão, faz-se necessário também a presença de uma base de conhecimento, sendo esta mais explorada na fase de análise e planejamento.

Em muitos sistemas autonômicos faz-se necessário a utilização de mais de um ciclo de gerenciamento autonômico, usualmente organizados em um ciclo global e um ou mais ciclos de controle local. Os ciclos de controle local tratam apenas de estados conhecidos do ambiente local, sendo baseado no conhecimento encontrado

no próprio elemento gerenciado. Por esta razão, o ciclo local é incapaz de controlar o comportamento global do sistema. Já o ciclo global, a partir da análise de dados provenientes dos gerenciadores locais ou através de um monitoramento a nível global, pode tomar decisões e atuar em todo o sistema. No entanto, a implementação das interações entres os diversos níveis existentes vai depender das necessidades da aplicação.

Na Figura 2.3, é ilustrada uma arquitetura para sistemas autonômicos derivada do sitema *Ashby's Ultra-stable* [38]. Essa arquitetura, que apresenta semelhanças com a arquitetura MAPE-K, é voltada para a execução de aplicações de alto desempenho, onde o ambiente interno é caracterizado pelo estado interno das aplicações em execução e o ambiente externo caracteriza o estado do ambiente de execução. O controle do sistema envolve etapas semelhantes às definidas pelo ciclo MAPE-K: Monitoramento e Análise (M&A); Planejamento e Execução (PE). O laço de controle local (L) gerencia o comportamento de elementos individuais do sistema e o laço de controle global (G) gerencia o comportamento do sistema como um todo.

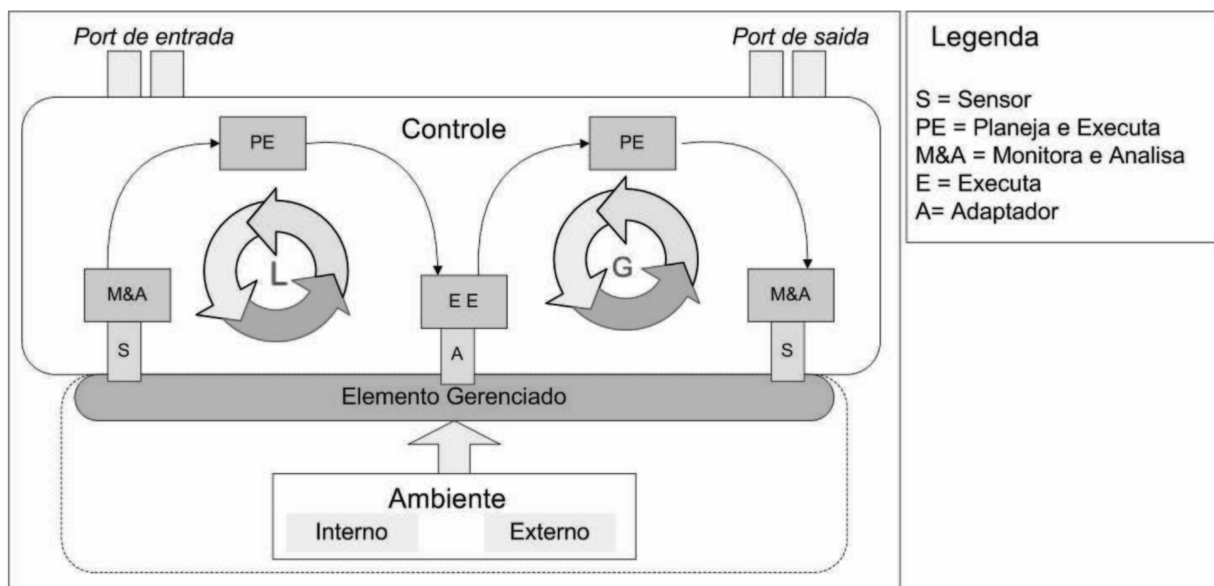


Figura 2.3: Arquitetura derivada do Sistema Ashby's Ultra-stable [16].

Uma vez apresentadas algumas arquiteturas para o desenvolvimento de sistemas autonômicos, serão detalhadas nas próximas subseções cada uma das fases que compõe o ciclo de gerenciamento autonômico de sistemas: Monitoramento, Análise e Planejamento, Controle e Execução de Ações de Reconfiguração.

2.2.1 Monitoramento

O monitoramento corresponde à primeira fase do ciclo autonômico MAPE-K. Nesta etapa, sensores são utilizados com a finalidade de se obter dados que reflitam mudanças de comportamento do elemento gerenciado ou informações do ambiente de execução que sejam relevantes ao processo de autogerenciamento. Os sensores são dispositivos de hardware ou software que estão diretamente ligados ao componente que se deseja monitorar.

O tipo de informação coletada, bem como o modelo dos monitores empregados, é específico para cada tipo de aplicação. Contudo, alguns requisitos são comuns entre eles, tais como a filtragem, a escalabilidade, a dinamicidade e a tolerância a falhas [57]. Um monitor deve prover filtragem, pois a quantidade de dados coletados pode crescer muito rapidamente e consumir recursos de transmissão, processamento e armazenamento. Grande parte desses dados pode ser de pouca relevância e, portanto, descartá-los não implicaria em prejuízo. Considerando-se que um sistema pode crescer indefinidamente, contendo inúmeros objetos sob monitoramento, a tarefa dos monitores não pode consumir recursos e reduzir o desempenho do sistema.

2.2.2 Análise e Planejamento

A análise é a fase que ocorre depois do monitoramento no ciclo de gerenciamento MAPE-K, tendo como fase seguinte o planejamento. Estas duas fases são geralmente implementadas em um único componente. O processo de análise e planejamento é essencial para o autogerenciamento, pois é nele que são geradas as decisões sobre quais modificações serão realizadas no sistema.

A fase de análise e planejamento recebe como entrada os eventos e os dados gerados pelo sistema de monitoramento, e gera como saída um conjunto de ações, também chamado de plano de ações. Estas ações correspondem às operações de reconfiguração dinâmicas que, de acordo com o mecanismo de tomada de decisão, devem ser executadas no sistema a fim de manter o equilíbrio do sistema, considerando seus objetivos. Muitas técnicas podem ser empregadas na fase de análise e planejamento, entre as quais se destacam o uso de regras Evento-Condição-Ação (ECA) [16], Funções de Utilidade [65], e Aprendizado por Reforço [74].

2.2.3 Controle e Execução

Na etapa de execução do ciclo MAPE-K são realizadas reconfigurações no sistema de forma a restabelecer seu equilíbrio. A finalidade da reconfiguração é permitir que um sistema evolua (ou simplesmente mude) incrementalmente de uma configuração para outra em tempo de execução, introduzindo pouco (ou, idealmente, nenhum) impacto sobre a execução do sistema. Desta forma, os sistemas (ou aplicações) não necessitam ser finalizados ou reiniciados para que as mudanças sejam concretizadas

A autoconfiguração (*self-configuring*) é a característica do sistema autonômico que o permite ajustar-se automaticamente às novas circunstâncias percebidas em virtude do seu próprio funcionamento, de forma a atender a objetivos especificados pelos processos de autocura, auto-otimização ou autoproteção [16].

O processo de reconfiguração é realizado pelos executores através dos atuadores. Executores recebem como entrada um plano de ações gerado na etapa de análise e planejamento e utiliza os atuadores pertinentes para implementar as ações de reconfiguração descritas no plano. As reconfigurações devem ser realizadas dinamicamente, sem impor a necessidade de parar e/ou reiniciar o sistema totalmente.

Muitas vezes a execução de ações de reconfiguração dinâmica requer mecanismos de transferência de estado. De maneira resumida, a transferência de estado refere-se ao processo que garante que o software recém configurado inicie suas operações com informações apropriadas sobre o seu estado, formado pelas configurações anteriores, garantindo assim a continuação coerente do fluxo da aplicação. Outro problema de quando se modifica um sistema em tempo de execução é a sincronização entre a execução de reconfigurações e a execução funcional do sistema. Dependendo da complexidade da reconfiguração, a parte do sistema a ser modificada não deve estar disponível para a execução funcional durante o tempo de reconfiguração. Ao preparar um componente a ser reconfigurado, é preciso assegurar que os canais de comunicação estejam vazios ou que as mensagens em trânsito não prejudiquem a consistência desse componente.

A execução da reconfigurações sobre o ambiente pode introduzir efeitos negativos sobre a qualidade dos serviços. O custo de reconfiguração é definido como uma medida desses efeitos. Entre os efeitos negativos pode-se citar a indisponibilidade

temporária do serviço, ou a perturbação induzida em outros serviços após o possível aumento do consumo de recursos de rede durante a reconfiguração. Os custos das reconfigurações dinâmicas podem ser derivados de várias partes, tais como: I) o custo do *download* dos componentes para os nós; II) o custo de instalação, ou seja, a colocação dos componentes à disposição do sistema; e III) o custo de instanciar componentes, incluindo eventuais transferências de estados. Os custos são medidas compostas, que envolvem o tempo de realização das ações, o custo computacional, o custo de comunicação e o nível de recursos consumidos durante a operação de reconfiguração [37].

2.3 Abordagens para o Desenvolvimento de Sistemas Autonômicos

No que se refere ao desenvolvimento de sistemas autonômicos, alguns princípios e tecnologias vêm sendo utilizados para facilitar esse processo do ponto de vista de programação. Esta seção apresenta duas das principais abordagens mais utilizadas no desenvolvimento de sistemas autonômicos, e que foram utilizadas no contexto deste trabalho de pesquisa: a reflexão computacional e a programação orientada a aspectos.

2.3.1 Reflexão Computacional

Reflexão computacional é a habilidade de um software observar ou até mesmo modificar a sua estrutura ou comportamento [56]. Essa abordagem expõe detalhes de implementação do software em um nível de abstração que permite mudanças em seu comportamento sem comprometer a sua portabilidade. Dessa maneira, um software reflexivo deve incorporar estruturas de dados que representam seus diversos aspectos em uma autorrepresentação. Esses aspectos são causalmente conectados com os aspectos de implementação do sistema. Portanto, modificações em qualquer um desses aspectos levam às mudanças em outro aspecto. Isto assegura que o software sempre tenha uma autorrepresentação precisa de si mesmo e que o estado e a computação do software estejam em conformidade com essa representação.

A reflexão computacional compreende basicamente duas atividades: introspecção e intercessão [71]. A introspecção denota a capacidade que um software tem de examinar sua própria estrutura, estado e representação. A esses elementos dá-se o nome de metainformação, que representa qualquer informação contida e manipulável por um software computacional que seja referente a si próprio. Por exemplo, um software pode observar quais os métodos que compõem uma interface, os parâmetros e seus tipos de dados.

Existem basicamente dois tipos de reflexão computacional: a estrutural e a comportamental. A primeira possibilita modificar a estrutura das classes ou objetos, através do acréscimo de atributos, métodos e modificação da hierarquia de herança. Já a reflexão comportamental torna possível modificar o comportamento dos programas em execução através da interceptação de ativações dos objetos, métodos, e atributos de interesse.

Em geral, software reflexivo é desenvolvido usando-se o padrão de desenvolvimento chamado de *Reflection* [56]. *Reflection* é um padrão arquitetural que oferece um mecanismo para modificação dinâmica da estrutura e do comportamento de sistemas de software. Esse padrão divide um software em um meta-nível e em um nível base. O meta-nível representa a estrutura e o comportamento do software em elementos chamados meta-objetos, enquanto que o nível base define a lógica da aplicação e a implementação das regras de negócio. Esses dois níveis são causalmente conectados através de um protocolo de meta-objetos [43], de forma que modificações em um serão refletidas no outro.

A principal vantagem da organização do software reflexivo em duas camadas é a nítida separação do código em dois níveis de funcionalidade, um nível base que provê funcionalidade do domínio e um meta-nível que permite que o comportamento, a forma ou a implementação do nível base, sejam manipulados, regulados ou influenciados. Em decorrência disto, software reflexivo apresenta outras vantagens como: maior reutilização do código, mais facilidade na manutenção e depuração do código, e maior transparência na incorporação de conteúdo adaptativo no software. Portanto, arquiteturas reflexivas mostram-se adequadas para desenvolvimento de sistemas autonômicos, uma vez que as mesmas favorecem uma separação clara entre o nível dos elementos autonômicos do sistema, o meta-nível,

que age sobre os elementos gerenciados, e o nível-base, constituído pelos elementos gerenciados, que reflete as adaptações desencadeadas pelo meta-nível.

Uma maneira de implementar o monitoramento do estado interno e do ambiente usando elementos autonômicos é empregar mecanismos reflexivos oferecidos pelas linguagens de programação. Além disso, os mecanismos de reflexão podem ser empregados para implementar as mudanças estruturais e comportamentais quando o ambiente de execução do sistema atingir um determinado estado. O comportamento de objetos individuais podem ser dinamicamente modificados ou mesmo substituído em tempo de execução, aumentando a flexibilidade do sistema. Portanto, arquiteturas baseadas em reflexão computacional são adequadas para introduzir propriedades de sistemas autonômicas, tais como: autoconsciência, ciência de contexto, e reconfiguração dinâmica.

2.3.2 Programação Orientada a Aspectos (POA)

Quado se fala em desenvolvimento de sistemas, um dos principais conceitos abordados é o conceito de **interesse**, em alguns casos usado como sinônimo de **responsabilidade**. Um interesse é alguma parte do domínio do sistema que se deseja tratar com uma unidade conceitual única. A Programação Orientada a Objetos (POO) permitiu uma melhor separação dos diversos interesses do sistema, com a estruturação de projetos e códigos mais próximos do que é idealizado naturalmente pelos desenvolvedores. No caso da POO, as abstrações básicas para os interesses são classes, objetos, métodos e atributos. Kiczales et al. [42] observaram que em alguns sistemas de software complexos, desenvolvidos segundo o paradigma da orientação a objeto, existem vários requisitos funcionais e não-funcionais entrelaçados ao longo do código computacional, tais como: *logging*, integridade de transações, autenticação, segurança, desempenho, distribuição, e persistência. Esses requisitos são chamados de interesses transversais (*crosscutting concerns*), já que, inerentemente, a sua implementação se dá através da adição de código em diversas classes ao longo do software. No entanto, esse código não está diretamente relacionado com a funcionalidade definida para essas classes, e isso dificulta o desenvolvimento e manutenção do software, pois a alteração em um desses requisitos implica na modificação de todas as classes que utilizam o mesmo [24].

A Programação Orientada a Aspectos (POA) surgiu como consequência do princípio de separação de interesses. A POA permite expressar interesses entrelaçados em termos de elementos chamados aspectos, desenvolvidos separadamente de outras partes do sistema e armazenados em uma unidade de código visível a todos os componentes do sistema [79]. Desta maneira, o software é organizado em classes ou componentes, que constituem o código base do sistema e representam as regras de negócio da aplicação, e aspectos que relacionam os requisitos funcionais e não-funcionais que afetam o comportamento do sistema e que estariam entrelaçados no código das classes ou componentes. A figura 2.4 ilustra o processo de combinação (*weaving*) do código base do sistema (*Base Code*) com os aspectos especificados (*Aspect Program*), resultando em um código executável dinamicamente modificado (*Modified Code*). O elemento responsável pelo processo de combinação é denominado *Aspect Weaver*.

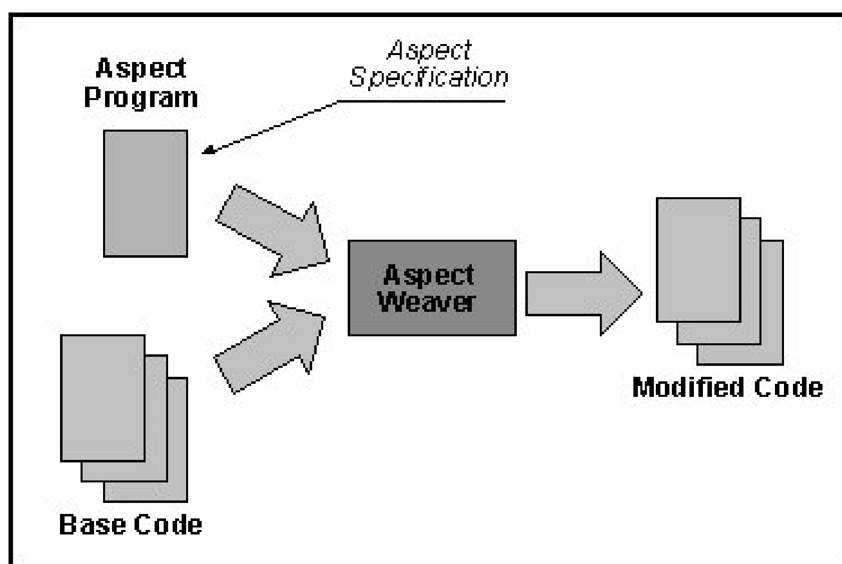


Figura 2.4: Combinação do código base do sistema com os aspectos

Existem diversos *frameworks* que dão suporte ao paradigma de desenvolvimento orientado a aspectos, tais como: *AspectJ* [1] e *JBoss AOP* [2], implementadas em Java [3]; AspectC [4] e XWeaver [5], implementadas em C; *Aspyct*

[1]http://eclipse.org/aspectj/
[2]http://www.jboss.org/jbossaop
[3]http://www.oracle.com/us/technologies/java/
[4]http://www.cs.ubc.ca/labs/spl/projects/aspectc
[5]http://www.xweaver.org/

AOP [6], implementada em Python [7]; e diversas outras. Além dos aspectos, para implementar a separação de interesses transversais, os *frameworks* POA podem incluir muitos dos seguintes conceitos e construções definidos por Laddad et al. [51]:

- **Identificação de pontos de execução do sistema**: O sistema expõe pontos durante a execução do sistema. Estes podem incluir a execução de métodos, criação de objetos, ou lançamento de exceções. Tais pontos identificados no sistema são chamados **pontos de junção** (*joinpoints*). Os ponto de junção estão presentes em todos os sistemas, mesmo aquelas que não usam POA. A POA apenas identifica e classifica esses pontos de junção.

- **Seleção de Pontos de junção**: A implementação de interesses transversais requer a seleção de um conjunto específico de pontos de junção. Por exemplo, um aspecto pode estar interessado em interceptar apenas os métodos públicos no sistema. A construção **pontos de corte** (*pointcuts*) seleciona qualquer ponto de junção que satisfaça os critérios. Isto é semelhante a uma consulta *Structured Query Language* (SQL) selecionando linhas no banco de dados. Um ponto de corte pode usar outro ponto de corte para formar uma seleção complexa. Os pontos de corte também coletam informações de contexto nos pontos selecionados. Por exemplo, um ponto de corte pode coletar argumentos passados a um método após a sua invocação.

- **Alteração da estrutura estática do sistema**: Algumas vezes, para implementar funcionalidades transversais de forma eficaz, é preciso alterar a estrutura estática do sistema. Em POA, isso ocorre por meio da construção denominada **declaração intertipo**. Declarações intertipos constituem um mecanismo que permite que um aspecto adicione outras declarações numa classe ou objeto existente. Por exemplo, pode ser adicionado um novo método ou atributo em uma classe. As declarações intertipos são descritas como formas de transversalidade estática (*static crosscutting*) e permitem introduzir alterações estáticas em classes, interfaces e aspectos do sistema. Alterações estáticas em módulos não têm efeito direto no comportamento do sistema.

[6] http://www.aspyct.org/
[7] http://python.org/

- **Alteração do comportamento do sistema**: Depois que um ponto de corte seleciona os pontos de junção, deve ser possível modificar esses pontos de junção com comportamentos adicionais ou alternativos. O construtor **adendo** (*advice*) em POA fornece um mecanismo para fazer isso. Um adendo pode acrescentar comportamentos antes (*before advice*), depois (*after advice*) ou durante (*around advice*) a execução dos pontos de junção selecionados. Os adendos constituem formas de transversalidade dinâmica (*dynamic crosscutting*) porque afetam diretamente a execução do sistema.

- **Módulos para expressar os interesses transversais**: Uma vez que o objetivo final da POA é ter um módulo que incorpora a lógica transversal, é preciso de uma unidade para expressar essa lógica. Essa unidade é o aspecto. Um aspecto contém pontos de junção, declarações inter-tipo, adendos. Um aspecto se relaciona a outros aspectos de uma forma semelhante a como uma classe se relaciona com outras classes. Aspectos se tornam uma parte do sistema e usam as classes do sistema para fazer seu trabalho.

A Programação Orientada a Aspectos pode auxiliar o desenvolvimento de software autoadaptativo, pois muitas mudanças no ambiente computacional estão relacionadas a requisitos que estão entrelaçados no código da aplicação. Dessa maneira, o encapsulamento desses requisitos em aspectos permite a modificação do software em um único ponto, ao invés de modificar todos os locais em que esse requisito está presente. Além disso, é possível substituir dinamicamente a implementação de um aspecto por outra, sem que haja modificação no código da aplicação. Por exemplo, a modificação do mecanismo de persistência de dados utilizado pela aplicação em face da diminuição da largura de banda disponível.

2.4 Conclusão

Neste capítulo foi visto que para um sistema ser autonômico ele deve ser capaz de se autogerenciar. Isso implica que ele deve possuir propriedades como autoconfiguração, auto-otimização, autoproteção, autocura, autoconsciência e ciência de contexto. Apresentamos a arquitetura mais utilizada no desenvolvimento de sistemas autonômicos: a MAPE-K. Vimos também que um sistema pode ter

mais de um ciclo autonômico, e que usualmente torna-se necessário a utilização de gerentes autonômicos locais e globais. Em sistemas distribuídos, os ciclos locais são usualmente utilizados para gerenciar o comportamento autonômico de componentes locais, enquanto que os ciclos globais controlam os ciclos locais e o comportamento do sistema como um todo.

Descreveu-se também as fases que envolvem o ciclo do gerenciamento autonômico. No monitoramento, os sensores enviam os dados puros do ambiente para os monitores, que são responsáveis por processá-los e disponibilizá-los em um nível mais alto de abstração. Em seguida a fase de análise e planejamento realiza a tomada de decisões e gera o plano contendo as de ações de reconfiguração que devem ser realizas para alcançar o equilíbrio do sistema. Na fase de execução, essas reconfigurações são executadas, sendo que as adaptações realizadas podem ser de dois tipos: paramétricas ou estruturais. Por fim, foram vistas algumas abordagens usualmente empregadas no desenvolvimento de sistemas autonômicos: a Reflexão Computacional e Programação Orientada a Aspectos.

3 Grades de Computadores

Este capítulo apresenta uma breve fundamentação teórica sobre as grades de computadores. São apresentados conceitos, características, aplicações, desafios e uma classificação de abordagens para a computação em grade. Neste capítulo, um tipo particular de grade ganha destaque, a grade de computadores pessoais (*Desktop Grid*). Dado que este trabalho está inserido no contexto de um projeto voltado para esse tipo de ambiente de grade, suas características principais são abordadas. Por fim, é feita uma breve introdução às Grades Autonômicas, mostrando alguns trabalhos relacionados a esse campo de pesquisa e aos conceitos apresentados no capítulo 2 .

3.1 Introdução a Grades de Computadores

Atualmente, diversos ramos de atividades científicas, comerciais e industriais como biologia, processamento de imagens para diagnóstico médico, previsão do tempo, física de alta energia, previsão de terremotos, simulações mercadológicas, prospecção de petróleo e computação gráfica, têm demandado grande capacidade de processamento, compartilhamento de dados e colaboração entre usuários e organizações. Porém, alternativas tradicionais para a realização de computação de alto desempenho, como o uso de supercomputadores ou de aglomerados de computadores como *clusters* Beowulf [64], requerem altos investimentos em hardware e software, podendo chegar a custar centenas de milhares de dólares.

Por outro lado, redes de computadores existentes em instituições tanto públicas quanto privadas formam hoje um enorme parque computacional interconectado principalmente por tecnologias ligadas à Internet. No entanto, apesar de permitir comunicação e troca de informações entre computadores, as tecnologias que compõem a Internet atual não disponibilizam abordagens integradas que permitam o uso coordenado de recursos pertencentes a várias instituições na realização de computações.

Uma abordagem, denominada Computação em Grade (*Grid Computing*) [29] [6], foi proposta com o objetivo de superar esta limitação. A origem do termo *Grid Computing* deriva de uma analogia com a rede elétrica (*Power Grid*), e reflete o objetivo de tornar o uso de recursos computacionais distribuídos tão simples quanto ligar um aparelho na rede elétrica.

A computação em grade permite a integração e o compartilhamento de computadores e recursos computacionais, como software, dados e periféricos, em redes corporativas e entre estas redes, estimulando a cooperação entre usuários e organizações, criando ambientes dinâmicos e multi-institucionais, fornecendo e utilizando os recursos de maneira a atingir objetivos comuns e individuais [4] [30].

Os primeiros projetos de pesquisa na área de computação em grade surgiram nos anos 90 com o objetivo de interligar centros de supercomputação. A abordagem na época era conhecida como metacomputação [5]. Projetos como FAFNER[1], SETI@home[2] e I-WAY [27] obtiveram grande repercussão e seus resultados serviram de base para o desenvolvimento de uma infraestrutura de grade mais ubíqua, capaz de interligar uma grande quantidade de instituições e recursos computacionais. Globus [28], Legion [35], OurGrid [2], GridBus [9] e InteGrade [19] são exemplos de projetos bem sucedidos desta segunda geração de grades de computadores.

O *middleware*[3] de grade é o componente de software central de um sistema de grade, sendo responsável pela integração dos recursos distribuídos, de modo a criar um ambiente unificado para o compartilhamento de dados, recursos computacionais e execução de aplicações.

O compartilhamento multi-institucional e dinâmico proposto pela computação em grade deve ser, necessariamente, altamente controlado, com provedores de recursos e consumidores definindo claramente e cuidadosamente o que é compartilhado, a quem é permitido compartilhar, e as condições sob as quais ocorre o compartilhamento. Ian Foster et al. [30], conceituam como **Organização**

[1] http://www.npac.syr.edy/factoring.html

[2] http://setiathome.ssl.berkeley.edu

[3] Middleware é uma camada de software que reside entre o sistema operacional e a aplicação a fim de facilitar o desenvolvimento de aplicações, escondendo do programador diferenças entre plataformas de hardware, sistemas operacionais, bibliotecas de comunicação, protocolos de comunicação, formatação de dados, linguagens de programação e modelos de programação.

Virtuais os grupos formados por indivíduos e/ou organizações que compõem esses ambientes dinâmicos e multi-institucionais.

Existem atualmente muitas aplicações para a computação em grade. Ian Foster e Carl Kesselman [29] identificaram as cinco principais classes de aplicações para grades de computadores: supercomputação distribuída, alto rendimento, computação sob demanda, computação intensiva de dados e computação colaborativa. Estas classes de aplicações são apresentadas na Tabela 3.1.

Categoria	**Características**	**Exemplos**
Supercomputação distribuída	Grandes problemas com intensiva necessidade de CPU, memória, etc.	Simulações interativas distribuídas, cosmologia, modelagem climática
Alto rendimento	Agregar recursos ociosos para aumentar a capacidade de processamento. A grade é utilizada para executar uma grande quantidade de tarefas independentes ou fracamente acopladas.	Projeto de chips, problemas de criptografia, simulações moleculares
Computação sob demanda	Recursos remotos integrados em computações locais, muitas vezes por um período de tempo limitado	Instrumentação médica, processamento de imagens microscópicas
Computação intensiva de dados	Síntese de novas informações de muitas ou grandes fontes de dados	Experimentos de alta energia, modernos sistemas meteorológicos
Computação colaborativa	Suporte à comunicação ou a trabalhos colaborativos entre vários participantes	Educação, projetos colaborativos

Tabela 3.1: Cinco maiores classes de aplicações de grades

3.1.1 Desafios para Construção de um Middleware de Grade

Para que a computação em grade se torne realidade, um *middleware* que atenda a diversos requisitos deve ser desenvolvido, entre os quais se destacam:

- **Heterogeneidade de plataformas de hardware:** um sistema de computação em grade pode, eventualmente, lidar com recursos heterogêneos, necessitando fornecer mecanismos que tratam essa diversidade de plataformas;

- **Interoperabilidade:** uma infraestrutura de grade de computadores deve seguir padrões na construção de suas interfaces, de forma a fomentar a interoperabilidade entre os *middlewares* de grade e o objetivo de integração em escala global;

- **Custo reduzido:** o uso de grade deve se constituir em um meio alternativo aos altos custos de hardware de alto desempenho e software;

- **Gerenciamento de recursos:** mecanismos que possibilitem o controle de recursos e monitoramento são essenciais para a alocação correta de acordo com requisitos pré-estabelecidos, balanceamento e escalonamento de tarefas para grade [33];

- **Qualidade de serviço:** com uso colaborativo da grade, deve ser garantido que proprietários de estações de trabalho não tenham suas aplicações afetadas devido às aplicações da grade que executam em seus recursos. O eventual uso comercial da grade deve ser negociado por meio de *Service Level Agreement* (SLA) entre usuários e provedores de serviço para garantir níveis adequados de custo e prestação de serviço [59];

- **Serviços de informação:** informações sobre a estrutura e o estado dos recursos (configuração, disponibilidade, carga de trabalho, e políticas de uso) são essenciais para o gerenciamento dos mesmos, dado que estas informações permitem aos gerenciadores de recursos alocá-los de maneira mais eficiente;

- **Políticas de segurança:** devido aos recursos computacionais encontrarem-se em diversos domínios administrativos, eventualmente, existem várias políticas de segurança. Assim, a infraestrutura de grade deve tratar questões de integração de políticas de segurança locais, mapeando identidades e autorizações, provendo acesso autenticado e confiável, bem como garantindo privacidade e a autonomia administrativa local;

- **Tolerância a falhas:** tratar falhas em grades é bastante complexo, visto que as grades agregam uma enorme quantidade de componentes de hardware e software, eventualmente heterogêneos, que fazem a probabilidade de falhas aumentar proporcionalmente ao tamanho do sistema, bem como dificulta a identificação da origem da falha.

3.2 Taxonomia dos Sistemas de Grade

Krauter et al. [49] propõem uma taxonomia para sistemas de grades de computadores ilustrada na Figura 3.1, descrita a seguir.

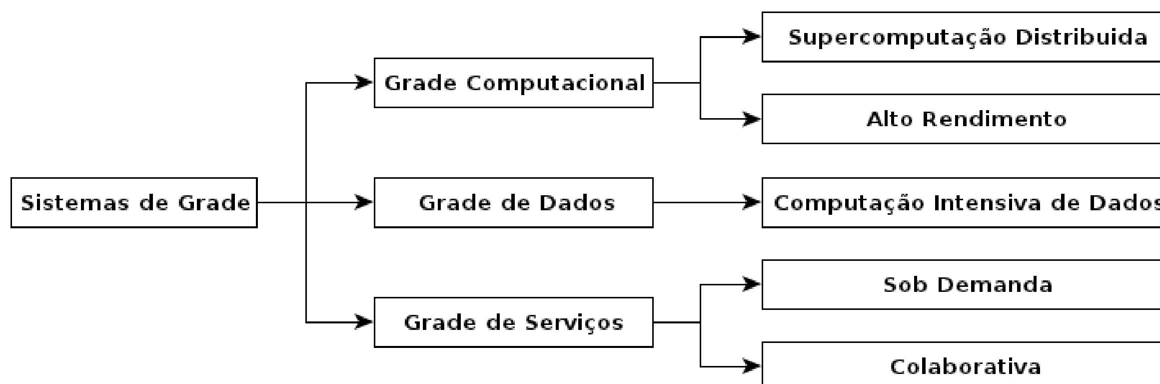


Figura 3.1: Taxonomia dos sistemas de grade

- **Grade Computacional** (*Computing Grid*): é um sistema de computação que objetiva prover maior capacidade computacional através da agregação de recursos computacionais distribuídos. Como exemplo de classes de aplicações que utilizam este tipo de grade temos:
  - **Supercomputação distribuída**: cujas aplicações usam grades para agregar recursos computacionais substanciais para resolver problemas que não poderiam ser resolvidos por um único sistema. Ex.: aplicações para planejamento e treinamento militar através de simulações interativas distribuídas, simulação de processos físicos complexos, modelagem climática;
  - **Alto rendimento**: onde a grade é usada para escalonar grande número de tarefas independentes ou fracamente acopladas, com o objetivo de utilizar ciclos de processadores ociosos. Ex.: aplicações para a resolução de problemas criptográficos;
- **Grade de Serviços** (*Service Grid*): provê serviços viabilizados pela integração de diversos recursos computacionais. Exemplos de classes de aplicações que utilizam esse tipo de grade são:
  - **Computação sob demanda:** aplicações usam as capacidades da grade por períodos curtos de acordo com solicitações por recursos como, por exemplo,

aplicações de instrumentação médica e requisições de utilização de software especializado por usuários remotos;

  - **Computação colaborativa:** aplicações que utilizam a grade para dar apoio a trabalhos cooperativos envolvendo diversos participantes, como, por exemplo, em projetos colaborativos e de educação;

- **Grade de Dados** (*Data Grid*): é um sistema de computação em grade que objetiva prover mecanismos especializados para publicação, descoberta, acesso e classificação de grandes volumes de dados distribuídos. O foco da computação intensiva de dados está na síntese de novas informações sobre dados que são mantidos em repositórios, bibliotecas digitais e bancos de dados geograficamente distribuídos.

3.3 Grades de Computadores Pessoais (*Desktop Grids*)

Grades de Computadores Pessoais, também chamadas de Grades Oportunistas, ou simplesmente *Desktop Grids*, são sistemas computacionais que proveem meios para usar uma base instalada de computadores pessoais não-dedicados para execução de aplicações computacionais de alto desempenho, aproveitando o poder de computação ocioso disponível [3].

Em uma *Desktop Grid*, um grande número de computadores pessoais regulares são integrados para a execução de aplicações distribuídas em larga escala. Os recursos computacionais são heterogêneos em relação às suas configurações de hardware e software. Várias tecnologias de rede podem ser usadas nas redes de interconexão, resultando em enlaces com diferentes capacidades no que diz respeito a propriedades como largura de banda, taxa de erros e latência de comunicação. Os recursos computacionais também podem ser distribuídos por vários domínios administrativos. No entanto, do ponto de vista do usuário, o sistema de computação deve ser visto como um recurso único e integrado, devendo ainda ser fácil de usar.

O foco de um *middleware* de grade oportunista não é a integração de *clusters* de computadores dedicados (ex.: Beowulf) ou recursos de supercomputação, mas o aproveitamento de ciclos de computação ociosos de computadores pessoais e estações regulares de trabalho que podem ser distribuídos por vários domínios administrativos.

Esses recursos não precisam ser dedicados para a execução de aplicações da grade, pois a carga de trabalho da grade irá coexistir com pedidos de execução de aplicações locais apresentados pelos usuários das máquinas regulares. O *middleware* da grade deve aproveitar ciclos ociosos de computação que surgem de períodos de tempo sem uso das estações de trabalho que compõem a grade.

Ao aproveitar o poder computacional ocioso de estações de trabalho existentes e conectá-los a uma infraestrutura de rede, o *middleware* da grade permite uma melhor utilização dos recursos computacionais existentes e a execução de aplicações paralelas computacionalmente intensivas que, de outro modo, exigiria um aglomerado ou máquinas paralelas de alto custo.

Durante a última década, desenvolvedores de *middleware* para *Desktop Grids* tem trabalhado na construção de várias abordagens para permitir a execução de diferentes classes de aplicações, tais como: (a) aplicações sequenciais, onde a tarefa a ser executada é atribuída a um único nó da grade; (b) aplicações paramétricas ou *bag-of-tasks*, onde várias cópias de uma tarefa são atribuídas aos diferentes nós da grade, sendo que cada tarefa possui um subconjunto dos dados de entrada que são processados de forma independente e sem a troca de dados; (c) aplicações paralelas fortemente acopladas, cujos processos trocam dados entre si através da troca de mensagens ou abstrações de memória compartilhada.

Dentro desta linha, o projeto InteGrade [19] é uma iniciativa multi-institucional da qual a Universidade Federal do Maranhão (UFMA) participa para o desenvolvimento de um *middleware* de grade capaz de usufruir do poder computacional de estações de trabalho que estejam ociosas, aproveitando seus ciclos para a execução de aplicações paralelas de alto desempenho. O objetivo é permitir que organizações utilizem sua infraestrutura computacional para realizar este tipo de computação, sem a necessidade de adquirir novo e específico hardware.

Devido à heterogeneidade, alta escalabilidade e dinamismo do ambiente de execução, apoio eficiente para a execução de aplicações em grades oportunistas compreende um grande desafio para os desenvolvedores de *middleware*, que devem prover soluções inovadoras para resolver os problemas encontrados em áreas, tais como:

1. **Gestão de recursos**: que engloba desafios como a forma de monitorar eficientemente um grande número de recursos computacionais distribuídos em múltiplos domínios administrativos. Em *Desktop Grids*, essa questão é ainda mais difícil devido à natureza dinâmica do ambiente de execução, onde os nós podem entrar e sair da grade a qualquer momento. Uma vez que as máquinas utilizadas pela grade não são dedicadas, a disponibilidade das mesmas para executar aplicações da grade pode oscilar frequentemente por vários motivos, tais como: (1) a realização de computações dos usuários locais que exijam maior desempenho, forçando a interrupção da execução de aplicações submetidas por usuários da grade, para que o usuário local não tenha sua computação local prejudicada); e (2) o simples desligamento/reinício do sistema operacional da máquina utilizada por parte dos usuários locais.

2. **Programação de aplicações e gerenciamento de execução**: que deve fornecer mecanismos amigáveis para executar aplicações no ambiente de grade, para controlar a execução das tarefas, fornecer ferramentas para coletar os resultados das computações e gerar relatórios sobre as situações atuais e passadas. O gerenciamento de execução da aplicação deve abranger todos os modelos de execução suportados pelo *middleware*.

3. **Tolerância a falhas** : ambientes de grade são altamente propensos a falhas. Esta característica é amplificada em grades oportunistas devido ao seu dinamismo e à utilização de máquinas não-dedicadas, levando a um ambiente de computação não-controlado. Um mecanismo de detecção eficiente e escalável de falhas deve ser fornecido pelo *middleware* de grade, juntamente com um mecanismo de recuperação automática de execução de aplicações que não requeira a necessidade de intervenção humana.

3.4 Grades Autonômicas

O dinamismo das aplicações e da infraestrutura da grade, sua alta escalabilidade e heterogeneidade tornam tarefas como configuração, manutenção e recuperação em caso de falhas altamente complexas para serem realizadas exclusivamente por agentes humanos. Desta forma, deve-se prover mecanismos automatizados que facilitem o gerenciamento de grades de computadores [40].

Um iniciativa da UFMA no contexto de grades autonômicas é denominada AutoGrid [67], um sistema de grade autogerenciável que utiliza o *middleware* InteGrade como sua fundamentação. O projeto do AutoGrid prevê a capacidade do *middleware* para (re) configurar-se de acordo com dados de contexto regularmente observados a partir do ambiente de execução. São também previstos recursos de autocura, a capacidade de se recuperar de falhas de aplicações sem intervenção humana, e auto-otimização, capacidade de minimizar o tempo de resposta das aplicações e otimizar o desempenho global da grade. O AutoGrid enfatiza a abertura e flexibilidade, portanto, sua infraestrutura de monitoramento foi projetada para permitir introduzir, remover e substituir os monitores de acordo com o estado do ambiente de execução. Além disso, o mecanismo de adaptação previsto é extensível, pois permite que os usuários projetem mecanismos adaptativos para tratar características autonômicas diferentes e mais robustas.

No trabalho de Liu [53] é proposta uma arquitetura autonômica para que se possa tratar a heterogeneidade e dinamismo do ambiente de grades. Esta arquitetura permite que o comportamento de serviços e aplicações e suas interações sejam dinamicamente especificados e adaptados de acordo com regras de alto nível, baseadas nos requisitos, estados e contexto de execução das aplicações.

Diversos outros esforços de pesquisa recentes buscam aplicar técnicas de computação autonômica à computação em grade, provendo-se maior autonomia e reduzindo-se assim a necessidade de intervenção humana na manutenção e gerenciamento destes ambientes computacionais. Alguns dos aspectos investigados incluem mecanismos autonômicos de: autoproteção, tais como a detecção de sobrecargas que potencialmente poderiam levar à interrupção dos serviços; auto-otimização, através de ajustes de parâmetros e algoritmos ao se detectar degradações de desempenho; autocura, para contornar eventuais falhas parciais do sistema; e autoconfiguração, provendo-se máquinas virtuais configuradas sobre demanda e dinamicamente alocadas a recursos físicos [32] [41] [15] [1] [63].

3.5 Conclusões

Este capítulo abordou aspectos gerais do estado da arte na tecnologia de Grades de Computadores. Foram descritas as classes de aplicações da computação em

grade, os desafios para construção de *middleware* de grade e também uma taxonomia que classifica as grades em três tipos: computacionais, de dados, e de serviços.

Abordou-se também as *Desktop Grids*, um tipo de grade cujo objetivo é executar tarefas aproveitando ciclos de computação ociosos de máquinas pessoais não-dedicadas, e que apresentam grande variação na carga de trabalho local (*hostload*) devido a computações realizadas pelos usuários locais dessas máquinas. O entendimento do conceito de *Desktop Grid* é fundamental para os estudos de caso envolvendo o AGST, pois as abordagens autonômicas que foram avaliadas com esse simulador são voltadas especificamente para esses ambientes computacionais.

Por fim, foram apresentadas pesquisas recentes que aplicam técnicas de computação autonômica em grades de computadores, o que reforça a importância de uma ferramenta capaz de simular as funcionalidades desejáveis em um ambiente de grade autogerenciável como forma de auxiliar seu desenvolvimento e avaliação. A construção de uma ferramenta de simulação deste tipo é o objetivo central deste trabalho.

4 Introdução a Simulação Computacional

Neste capítulo são abordados os principais conceitos relacionados à simulação computacional e suas aplicações. Descreve-se também o funcionamento de alguns tipos de simulação, entre elas a simulação baseada em eventos discretos, que corresponde à técnica de simulação utilizada pelo AGST.

4.1 Conceitos Básicos de Simulação Computacional

Dentre as várias definições existentes na literatura, escolhemos uma que acreditamos ser bastante aproximada ao contexto de nossa pesquisa. Segundo Shannon [69], a simulação é definida como o processo de projetar modelos de um sistema (ou processo) e conduzir experimentos com esses modelos com o propósito de entender o comportamento do sistema ou avaliar várias estratégias para a operação do sistema.

A simulação computacional de sistemas consiste na utilização de certas técnicas matemáticas, empregadas em computadores, as quais permitem imitar o funcionamento de operações ou processos do mundo real. Ou seja, é o estudo do comportamento de sistemas reais através do exercício de modelos. Um modelo de simulação é aquele que captura as características de um sistema real, tais como sua complexidade, dinamicidade e aleatoriedade, repetindo em um computador o mesmo comportamento que o sistema apresentaria quando submetido às mesmas condições de contorno. [14].

Para Forrester [26], um sistema é definido como um agrupamento de partes que operam juntas, visando um objetivo comum. Sendo assim, um sistema pressupõe uma interação causa-efeito entre as partes que o compõem. Para que tais partes sejam identificadas, o objetivo do sistema precisa ser conhecido com clareza. O estado do sistema pode ter natureza contínua ou discreta. É de natureza contínua se o estado pode assumir qualquer valor dentro de um intervalo de valores como, por exemplo, a temperatura do sistema. O estado é de natureza discreta se apenas valores discretos são possíveis como, por exemplo, o estado de um interruptor (aberto ou fechado) [68].

A simulação computacional pode ser classificada em três categorias básicas: simulação de "Monte Carlo", simulação contínua e simulação discreta. Em alguns casos pode ser necessário construir um modelo de simulação que compreenda simultaneamente aspectos de simulações contínuas e discretas. Nestes casos, a simulação é denominada simulação combinada ou híbrida.

A simulação de "Monte Carlo" utiliza geradores aleatórios para simular sistemas físicos ou matemáticos nos quais não se tem o tempo explicitamente como uma variável. Já a simulação contínua e a simulação de eventos discretos levam em consideração a mudança de estado do sistema ao longo do tempo.

A simulação contínua é utilizada para modelar sistemas cujo estado varia ao longo do tempo. Por exemplo, uma xícara de chá quente colocada à temperatura ambiente. O fenômeno de resfriamento do chá é contínuo e o seu estado pode ser mais bem conduzido por uma simulação contínua. A simulação contínua utiliza-se de equações diferenciais para o cálculo das mudanças das variáveis de estado ao longo do tempo.

Já a simulação de eventos discretos é utilizada para modelar sistemas que mudam seu estado em momentos discretos de tempo, a partir da ocorrência de eventos. A preparação da xícara de chá do exemplo anterior poderia ser dividida em três eventos: (1) colocação da água quente na xícara; (2) colocação do chá na aguá quente; e (3) disponibilização do chá. Cada evento ocorre em instantes determinados no tempo. Uma vez que o AGST é um simulador de eventos discretos, descreve-se na seção seguinte o funcionamento básico de uma simulação baseada em eventos discretos.

4.2 Simulação Baseada em Eventos Discretos

Simuladores baseados em eventos discretos utilizam o conceito de "tempo simulado" ou "tempo da simulação", em vez de fazer a simulação em tempo real. O que caracteriza a simulação a eventos discretos é o fato do tempo da simulação ser descontínuo, ou seja, o tempo avança em saltos, e sem qualquer relação com o tempo real da simulação [14].

Neste tipo de simulação o tempo é uma variável global que deve ser sempre igual ao tempo associado ao evento que está acontecendo no momento ou ao último evento que ocorreu. O intervalo de tempo entre cada evento não é necessariamente igual, uma vez que a simulação é orientada a eventos e não a tempo. A ação ou acontecimento que provoca uma mudança de estado em um determinado instante de tempo é um evento. Por exemplo, se uma máquina está no estado "em funcionamento", e acontece uma "falha", a máquina passa para o estado "parado". Portanto, "falha" é um evento [68].

Durante a simulação, os eventos devem ocorrer obedecendo a uma ordem cronológica, ou seja, os eventos de menor tempo devem ser tratados primeiros. Por exemplo, suponha que um evento e_a ocorreu no instante t_a do tempo simulado e foi sucedido pelo evento e_b, o qual ocorreu no instante t_b do tempo simulado. Se não aconteceu nenhum evento de interesse entre e_a e e_b, então o tempo simulado pula de t_a diretamente para t_b. Um evento deve simplesmente anunciar que está ocorrendo e todo o sistema deve reagir a essa ocorrência

Cada evento da simulação está associado a um tempo simulado. Os eventos que ainda não foram tratados pelo simulador (ou seja, que ainda não aconteceram no tempo simulado) são mantidos em uma fila com prioridade, denominada fila de eventos. A prioridade de um evento é o tempo simulado associado a ele. Os dados de entrada da simulação determinam o conjunto de eventos iniciais (uma lista de tempos de chegada e durações de tarefas) ou, opcionalmente, permitem a geração aleatória desse conjunto (tempo médio entre chegadas de tarefas e duração média de uma tarefa). A simulação acaba quando a fila de eventos esvaziar.

Considerando-se um modelo de simulação de uma grade de computadores na qual tarefas são submetidas para execução, antes da simulação começar, todos os eventos iniciais devem ser colocados na fila de eventos. Um exemplo de evento inicial poderia ser: "chegada de tarefa com duração 500, no instante $t = 35$ do tempo simulado". O evento é a chegada de tarefa com duração 500. Ele está associado ao tempo simulado $t = 35$. Ao final da simulação, o simulador poderia imprimir os seguintes dados: duração em tempo simulado; número de tarefas tratadas; duração média de uma tarefa; tempo de espera médio de uma tarefa; tempo total médio de execução de uma tarefa, etc.

4.3 Aplicações da Simulação Computacional

O campo de aplicação da simulação é bastante vasto. Técnicas de simulação têm sido aplicadas em diversas áreas conhecidas, tais como a acadêmico-científica, industrial, militar, manufatura e serviço. A simulação computacional é um instrumento muito útil na modelagem e estudo de vários tipos de sistemas, desde sistemas naturais até sistemas econômicos e sociais.

Dentre as diversas vantagens do uso da simulação podemos destacar:

- Uma vez criado, um modelo de simulação pode ser utilizado inúmeras vezes para avaliar projetos e abordagens propostas;
- Uma vez que os modelos de simulação podem ser bastante detalhados, aproximando-se do sistema real, novas políticas e procedimentos operacionais, regras de decisão, fluxos de informação, entre outros, podem ser avaliados sem que o sistema real seja perturbado;
- Hipóteses sobre como ou porque certos fenômenos acontecem podem ser testadas para confirmação;
- O tempo real de execução da simulação pode ser controlado, podendo ser comprimido ou expandido. Isso permite a reprodução dos fenômenos de forma mais lenta ou acelerada, para um melhor estudo dos mesmos;
- Novas situações sobre as quais se têm pouco conhecimento e experiência podem ser tratadas, de tal forma que se tenha, teoricamente, alguma preparação diante de futuros eventos.

Naturalmente, existem outras metodologias para a análise de sistemas, tais como o modelo matemático construído pela teoria das filas, por exemplo, o qual é composto de fórmulas matemáticas que usualmente fornecem soluções mais rápidas computacionalmente do que a abordagem da simulação. Por outro lado, a teoria das filas impõe hipóteses simplificadoras para sistemas complexos, ou mesmo é incapaz de tratá-los de maneira conveniente. Quanto mais complexo, dinâmico e aleatório for um problema, então maior será aplicabilidade das ferramentas de simulação [14].

Ainda quanto à aplicabilidade da simulação, em [14] afirma-se que se o problema for estático, isto é, se os estados do sistema não se alteram com o tempo, então a simulação de eventos discretos não tem nenhuma utilidade prática. Porém, se o problema for determinístico, isto é, não apresenta nenhum comportamento aleatório, a simulação pode ser utilizada, mas obviamente será subutilizada. Essa é uma das técnicas de verificação/validação de modelos de simulação. Se o problema em questão for complexo, dinâmico e apresenta aleatoriedade, então a melhor técnica para analisar o sistema é a simulação.

Apesar das inúmeras vantagens da utilização de técnicas de simulação, há também algumas desvantagens, tais como:

- A construção de modelos requer treinamento especial. O aprendizado se dá ao longo do tempo com a aquisição de experiência. Dois modelos de um mesmo sistema construídos por indivíduos diferentes podem apresentar muitas divergências;
- Os resultados da simulação são às vezes de difícil interpretação. Uma vez que os modelos tentam capturar a aleatoriedade do sistema, muitas vezes existe dificuldade em determinar quando uma observação realizada durante uma execução se deve a alguma significante relação no sistema ou a aleatoriedade construída no modelo.
- A modelagem e a experimentação associadas a modelos de simulação podem consumir muitos recursos, principalmente tempo. A tentativa de simplificação na modelagem ou nos experimentos objetivando economia de recursos costuma levar a resultados insatisfatórios. Em muitos casos, a aplicação de métodos analíticos (como a teoria das filas, por exemplo) pode trazer resultados menos ricos, porém mais econômicos;
- Embora o modelo de simulação consiga capturar com elevado grau de fidelidade diversas características dos sistemas reais, em sistemas altamente complexos é praticamente impossível garantir que durante a simulação os sistemas simulados apresentarão os mesmos comportamentos dos sistemas reais.

4.4 Conclusão

A simulação faz com que sistemas possam ser estudados sem a utilização de um sistema real, permitindo que mudanças em vários aspectos do sistema possam ser experimentadas sem correr o risco de sofrer consequências indesejadas. Isso permite antever resultados, melhorar o desempenho do sistema, evitar acidentes, reduzir custos de projetos, estudar sistemas, entre outros.

Foram vistos neste capítulo os principais conceitos ligados a simulação computacional. Os três tipos de simulação abordados foram: Monte Carlo, contínua e discreta. A simulação baseada em eventos discretos teve seu funcionamento descrito a fim de possibilitar a compreensão do tipo de técnica de simulação utilizada pelo AGST. Por fim, foi visto neste capítulo que técnicas de simulação podem ser aplicadas nas mais diversas áreas, mas que sua aplicabilidade depende da natureza do sistema que se deseja modelar e que quanto mais aleatório, dinâmico e complexo for o sistema, maiores são as chances da simulação ser a opção mais acertada para analisá-lo.

Devido a alta escalabilidade, a complexidade e a heterogeneidade inerente de uma grades computacionais reais, é difícil produzir de forma repetitiva e controlada uma avaliação do desempenho de abordagens voltadas para esses ambientes, tais como algoritmos para o escalonamento de tarefas ou estratégias para tolerância a falhas na execução de aplicações. Além disso, devido ao alto custo econômico, pesquisadores dificilmente tem acesso a um ambiente de grade com grande quantidade de recursos disponíveis para a execução dos experimentos.

Fatores como esses motivam a utilização de simuladores computacionais, já que os modelos de simulação podem ser bastante detalhados, aproximando-se do sistema real e possibilitando que novas abordagens para esses ambientes possam ser avaliados sem que o sistema de grade real seja perturbado. Os resultados dessas avaliações poderão proporcionar, teoricamente, maior preparo no projeto e implementação futura dessas abordagens em ambientes de grades reais, pois já haverá algum conhecimento prévio sobre o comportamento das mesmas.

5 Trabalhos Relacionados

Técnicas de simulação são usualmente empregadas em pesquisas na área de computação em grade para a realização de avaliações de desempenho de abordagens desenvolvidas para esses ambientes [8, 12, 23, 44, 82]. Ao longo dos últimos anos, vários simuladores de grade foram desenvolvidos, alguns deles baseados em eventos discretos, tais como: GridSim [7], OGST [18], Alea2 [46], GSSIM [50], SimGrid [11] e DSiDE [13].

Este capítulo apresenta as principais características dos mais relevantes simuladores de grade computacionais, enfatizando-se o GridSim e o OGST por terem sido utilizados como base para o desenvolvimento do AGST.

5.1 GriSim

O GridSim[1] é um simulador de grades baseado em eventos discretos desenvolvido em Java[2] que permite a modelagem e a simulação de diferentes classes de recursos, usuários, aplicações, escalonadores e a conectividade entre esses recursos através de uma ou várias redes que interligam diversos domínios administrativos. A versão mais recente do GridSim, até o presente momento, é a 5.2 beta, disponibilizada em 25 de novembro de 2010.

GridSim fornece primitivas para criação, gerenciamento e mapeamento de tarefas para recursos, de forma que pode ser usado para simular escalonadores (*brokers*) para ambientes distribuídos como *clusters* e grades, permitindo assim a avaliação de heurísticas de escalonamento. Diversos trabalhos, tais como [8, 44, 82], utilizaram o GridSim para essa finalidade. Além do suporte à simulação de grades computacionais, essa ferramenta também oferece suporte à simulação de grades de dados. Estatísticas da totalidade ou de operações selecionadas podem ser armazenadas em arquivos e então serem analisadas usando métodos de análise estatística disponibilizadas pelo GridSim.

[1]Página Inicial: `http://www.gridbus.org/gridsim`
[2]http://www.oracle.com/us/technologies/java/index.html

GridSim permite a modelagem de vários tipos de recursos, tais como monoprocessadores, multiprocessadores de memória compartilhada, máquinas de memória distribuída, tais como SMPs e *clusters* com diferentes capacidades e configurações. Os recursos podem ser modelados utilizando diferentes políticas de alocação, tais como a de espaço compartilhado (*space-shared*) ou de tempo compartilhado (*time- shared*). A capacidade dos recursos deve ser definida em termos de MIPS (Milhões de Instruções Por Segundo). GridSim permite ainda que o usuário implemente suas próprias políticas de alocação de recursos [66].

Os recursos simulados pelo GridSim podem ser localizados em qualquer zona de fuso-horário, ou seja, o simulador permite o ajuste do relógio para fusos horários diferentes para simular a distribuição geográfica dos recursos. Fins de semana e feriados podem ser mapeados em função da hora local do recurso para modelar carga de trabalho não referente à grade (carga de trabalho local). Esses recursos podem ser reservados antecipadamente (*reservation-based allocation*) para sua utilização durante um determinado período de tempo [66]. Falhas de recursos também podem ser simuladas.

As arquiteturas de rede simuladas pelo GridSim permitem que suas entidades possam ser conectadas usando links e roteadores, com políticas de escalonamento de pacotes tais como FIFO (*First-In*, *First-Out*), WFQ (*Weighted Fair Queuing*) e SCQF (*Self Clocked Fair Queuing*) implementadas nativamente pelo simulador. O roteamento pode ser feito através de tabelas estáticas ou métodos dinâmicos, como o *Routing Information Protocol* (RIP) e *Open Shortest Path First* (OSPF) [66].

Simulações determinísticas podem ser realizadas através do uso de arquivos de *traces* coletados de ambientes reais. O GridSim oferece suporte à simulações a partir de *traces* tanto de *workload*, quanto de falhas de recursos.

Durante a simulação, o GridSim cria uma série de entidades *multi-threaded*, cada uma executando paralelamente em seu próprio segmento. O comportamento de uma entidade deve ser descrito no método `body()`. Dentre as principais entidades primitivas definidas pelo simulador para a construção de componentes mais elaborados estão: Usuário `(GridUser)`, Serviço da Informações da Grade (`AbstractGIS`), Recurso (`GridResource`), Máquina (`Machine`,) Processador `PE`.

Qualquer outra entidade que necessite se comunicar com outras entidades a partir do envio e recebimento eventos pode ser construída estendendo a classe `GridSim`.

5.1.1 Arquitetura do GridSim

A arquitetura do GridSim, segundo [10], é dividida em multi-camadas de forma modular, como ilustrado na figura 5.1. Esta abordagem facilita a integração de novos componentes ou camadas.

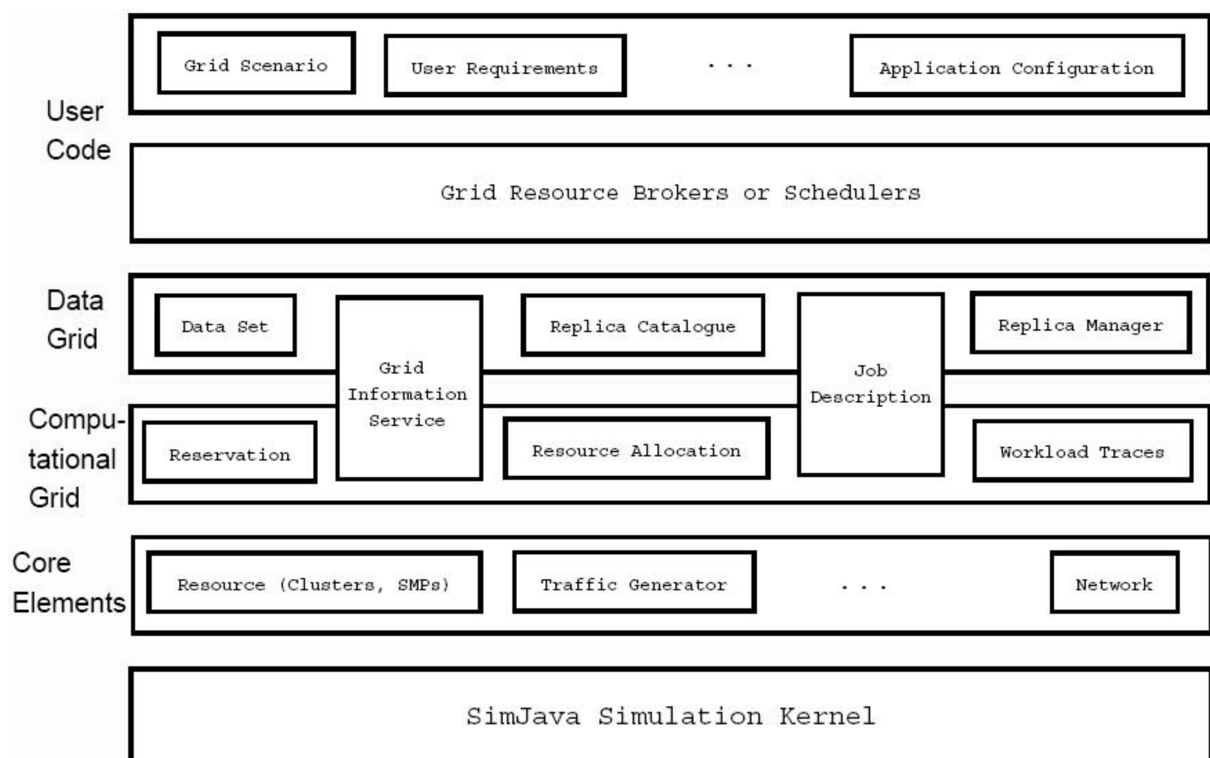


Figura 5.1: Arquitetura do GridSim [10]

A versão 5.2 beta do GridSim é baseada no SimJava2, um pacote de simulação baseado em eventos discretos e de propósito geral implementado em Java. Portanto, a primeira camada na parte inferior da Figura 5.1 é constituída pelo SimJava2, responsável pela interação entre os componentes ou eventos do GridSim. Todos os componentes do GridSim se comunicam uns com os outros através de operações de troca de mensagens, conforme definido pelo SimJava2. O GridSim acrescenta características de redes de comunicação e de entregas de eventos ao SimJava2 que permitem comunicação síncrona ou assíncrona para o acesso e liberação de serviços.

A segunda camada contém os elementos do GridSim utilizados para criação das simulações. Esses componentes servem para modelar a infra-estrutura distribuída, ou seja, os recursos da grade, tais como *clusters*, repositórios de armazenamento e enlaces de rede. Já as camadas de Grade de Computação e Grade de Dados são responsáveis pela modelagem e simulação de grades de computação e grades de dados, respectivamente. Entretanto, os serviços de informação e gerenciamento de tarefas são comuns a ambas as camadas.

A quinta camada contém componentes que ajudam os usuários na implementação dos seus próprios escalonadores de recursos e tarefas, para que eles possam testar os seus próprios algoritmos e estratégias de escalonamento. A camada acima desta ajuda os usuários a definir os seus próprios cenários e configurações para validar seus algoritmos.

5.2 OGST

O OGST[3] (*Opportunistic Grid Simulation Tool*) [17, 18] é uma ferramenta de simulação cujo principal objetivo é ajudar desenvolvedores de *middleware* de grades oportunistas na avaliação de novos conceitos e em suas implementações, considerando diferentes condições do ambiente de execução e cenários. A motivação inicial para o desenvolvimento do OGST foi prover uma ferramenta para avaliar o comportamento de algoritmos de escalonamento comumente usados em ambientes de grades quando sujeitos à diferentes condições do ambiente de execução, de forma a permitir a investigação de abordagens de escalonamento adaptativo e reescalonamento dinâmico de aplicações. O OGST permite a simulação de aplicações e recursos em grande escala, além de permitir a criação de cenários envolvendo diversos usuários de uma maneira controlada e repetitiva.

5.2.1 Extensões providas pelo OGST ao GridSim

O OGST foi construído tendo por base o GridSim, ao qual realizou diversas extensões a fim de obter mecanismos mais flexíveis e extensíveis que permitissem a geração automatizada de aplicações, recursos computacionais e enlaces de rede, bem como a implementação e avaliação de novas abordagens de escalonamento e de

[3]http://www.lsd.ufma.br/ ogst/

tolerância a falhas em grades. Além disso, o OGST disponibiliza um conjunto nativo de heurísticas de escalonamento e de estratégias para tolerância a falhas. As extensões mais importantes são descritas a seguir.

Geração Automatizada de Aplicações e Tarefas

O GridSim não explicita um modelo de aplicações, fornecendo apenas a entidade básica para representar uma tarefa denominada *Gridlet*. Portanto, a geração de aplicações no GridSim não é automática. Extensões providas pelo OGST ao GridSim permitem gerar automaticamente dois tipos de aplicações: regulares (constituídas de uma única tarefa) e *bag-of-tasks* (constituídas de várias tarefas que executam paralelamente de forma independente umas das outras, ou seja, sem trocar dados durante o processo de execução). O tamanho das tarefas é definido em Milhões de Instruções (MIs).

No OGST, as aplicações podem ser geradas de três formas: (1) sinteticamente, através de distribuições de probabilidade (uniforme, *poisson*, exponencial e hiper-exponencial) utilizadas na geração do tamanho das tarefas e de seus respectivos tempos de chegada no processo de simulação; (2) a partir de *traces* de *workloads* nos formatos GWF[4] (*Grid Workload Format*) e SWF[5] (*Standart Workload Format*); e (3) através do modelo probabilístico de Lublin [55].

Geração Automatizada de Recursos Computacionais e Enlaces de Rede

O GridSim fornece as entidades necessárias para a modelagem e simulação de recursos da grade, tais como máquinas, e dos enlaces de rede que conectam esses recursos. No entanto, esse simulador não implementa nenhum mecanismo de geração automática do ambiente de grade. Por esse motivo, o OGST implementa um mecanismo de geração automática de recursos computacionais baseado em distribuições de probabilidade. Dessa forma o ambiente de grade pode ser gerado sinteticamente. O poder de processamento dos nós é definido em termos de MIPS. Nesse mesmo mecanismo, o usuário do simulador deve fornecer a quantidade de recursos a serem gerados, bem como um valor representativo do poder de processamento médio da grade, a fim de que as distribuições, por exemplo, a

[4] http://gwa.ewi.tudelft.nl/pmwiki/
[5] http://www.cs.huji.ac.il/labs/parallel/workload/swf.html

distribuição uniforme, gerem máquinas cuja heterogeneidade varia em função desse valor. Esse mecanismo requer ainda o fornecimento de informações sobre a quantidade de roteadores que interconectam os aglomerados de recursos e a capacidade de transmissão de dados através da rede (definida em termo de bits por segundo)

A fim de simular *desktop grids* oportunistas, foi implementado no OGST um mecanismo que permite gerar recursos computacionais, que além da carga de trabalho demandada por aplicações da grade (*workload*), possuem também uma carga de trabalho proveniente da execução de aplicações por parte dos usuários locais dessas máquinas (*hostload*). Esse mecanismo é baseado em uma análise estatística de arquivos de *traces* que mostram a variação da carga de trabalho local de diversas máquinas de laboratórios da Universidade de São Paulo ao longo de vários meses. A partir dessa análise foi implementado um mecanismo que gera um ambiente de grade no qual a variação da carga de trabalho local segue o mesmo padrão das máquinas contidas no trace.

Falhas de Recursos e Técnicas de Tolerância a Falhas

Falhas de recursos podem ser geradas sinteticamente ou a partir de bancos de dados de *traces* de falhas coletados de ambientes reais no padrão FTA (*Failure Trace Archive*[6]) [48]. A geração sintética de falhas é baseada em distribuições de probabilidade, tais como exponencial, hiper-exponencial e weibull [14]. Deve-se fornecer os parâmetros de acordo com a distribuição utilizada. Por exemplo, no caso de uma distribuição exponencial, devem ser fornecidos o MTBF (tempo médio entre falhas) e o MTTR (tempo para recuperação) dos recursos.

No OGST, a recuperação da execução de aplicações em decorrência de falhas de nós pode ser baseada em três técnicas de tolerância a falhas que atuam no nível da tarefa apresentadas em [81], e que são usualmente empregadas em grades computacionais. São elas:

- **Reinício**: é a técnica mais simples usada para prover tolerância a falhas em grades computacionais e que consiste em tentar reiniciar a execução da tarefa que falhou. O reinício pode ser feito na mesma máquina em que a tarefa estava

[6] http://fta.inria.fr/

executando quando ocorreu a falha (*retry*), ou pode ser em um novo recurso para o qual a tarefa deva ser reescalonada (*alternate resource*).

- **Replicação**: cria e executa diferentes réplicas da mesma tarefa em diferentes recursos da grade simultaneamente na expectativa de que ao menos uma delas terminará sua execução com sucesso. Quando uma réplica termina as demais são eliminadas. Dado que essas réplicas são escalonadas para diferentes recursos e a probabilidade de uma delas executar em um recurso mais veloz torna-se maior, consequentemente essa técnica contribui no ganho de desempenho em relação ao tempo de conclusão da aplicação.

- **Pontos de Controle (*Checkpointing*)**: é a técnica que periodicamente pausa a execução de uma tarefa e salva o seu estado, permitindo que, na ocorrência de falhas, ela seja reiniciada a partir do último estado salvo (*checkpoint*). Em um ambiente real, o *checkpoint* pode ser feito de forma transparente pelo sistema operacional, contudo a heterogeneidade dos recursos da grade torna essa abordagem menos usual pelos *middlewares* de grade. Uma outra opção é a instrumentação da aplicação que pode ser feita pelo usuário, alterando diretamente o código fonte da aplicação ou utilizando bibliotecas específicas que devem ser ligadas ao código fonte ou os objetos da aplicação gerados pelo compilador [81]. No OGST, o *checkpoint* das aplicações é simulado através do armazenamento periódico (em relação ao tempo simulado) da quantidade de instruções (expressa em MIs) que foram executadas até o instante do *checkpoint*. Em caso de falhas, a aplicação só precisa executar a quantidade de instruções que restavam no instante do último *checkpoint* realizado.

Biblioteca de Algoritmos de Escalonamento

O modelo de simulação do GridSim não provê algoritmos de escalonamento em grades, devendo estes serem implementadas pelos próprios utilizadores desse simulador. Por meio de extensões ao GridSim, foi implementada uma biblioteca de heurísticas de escalonamento largamente utilizadas em grades, compreendendo os algoritmos InteGrade, Min-Min, WQR e MCT [23]. Novas estratégias podem ser facilmente adicionadas ao OGST por meio da adoção do padrão de projeto *Strategy*, através do qual são definidas famílias de algoritmos de escalonamento. Para criar uma

nova heurística de escalonamento no OGST baseada nesse padrão basta estender o componente `Schedule Strategy` (ver seção 5.2.2), responsável pela implementação desse padrão.

Persistência de Dados das Simulações

O GridSim permite a coleta de dados estatísticos relativos à execução de uma simulação que são armazenados em um arquivo texto. A partir deste arquivo, pode-se gerar um relatório de execução. No entanto, consultar e procurar relacionamentos em um grande volume de dados através da manipulação de um arquivo texto pode se tornar uma tarefa árdua. Para simplificar a consulta e análise dos dados gerados no processo de simulação, o OGST utilizada um banco de dados relacional. Por padrão, o OGST utiliza o Gerenciador de Banco de Dados MySQL[7] (mas pode usar diversos outros SGBDs relacionais) para armazenar os dados referentes à configuração dos cenários simulados (quantidade e configuração de recursos computacionais, quantidade e tamanho das tarefas das aplicações, tipo de algoritmo de escalonamento, tipo de estratégia de tolerância a falhas, etc.) e os dados gerados após o término das simulações (*status* de execução das tarefas, tempo de início e término de cada tarefa, etc.). O esquema do banco de dados é automaticamente gerado pelo OGST.

Mecanismo de Erro de Predição do Tempo de Conclusão das Aplicações

Uma vez que no OGST a capacidade de processamento dos nós é definido em MIPS e o tamanho das tarefas é definido em MIs, calcular o tempo simulado de conclusão de uma tarefa em um determinado nó é algo bastante simples, já que basta dividir o tamanho da tarefa pela capacidade de processamento do nó. Essas informações estão disponíveis durante a simulação e podem ser facilmente obtidas.

Em grades reais, calcular o tempo de conclusão das tarefas não é algo tão simples assim, já que em máquinas reais o tempo que uma aplicação leva para concluir sua execução depende da combinação de vários fatores, tais como o tipo de máquina na qual a aplicação da grade irá executar (processador, memória, dispositivos de E/S) o tipo de aplicação a ser submetida (tamanho e tipo das instruções da aplicação, tamanho

[7] http://dev.mysql.com

dos arquivos de entrada e saída, grau de paralelismo e acoplamento entre as tarefas, etc.), dentre outros fatores pertencentes ao ambiente de execução da grade.

O tempo de conclusão de uma tarefa em ambientes de grades reais pode ser estimado através da utilização de algoritmos de predição [54]. Geralmente os algoritmos de predição seguem duas abordagens. Uma delas consiste em calcular a estimativa do tempo de execução da aplicação baseado no registro de execuções anteriores da mesma ou de aplicações semelhantes. A outra abordagem é baseada no conhecimento do modelo de execução da aplicação. O código da aplicação é analisado, estimando-se o tempo de execução de cada tarefa de acordo com a capacidade dos recursos da grade. Porém, as informações necessárias para realizar a predição do tempo de execução das aplicações nem sempre podem ser obtidas em um ambiente real.

Para tentar tornar o ambiente de grade simulado mais semelhante ao ambiente de grade real, o OGST provê um mecanismo que adiciona um percentual de erros de predição no cálculo de estimativa do tempo de conclusão das aplicações. Esse mecanismo é baseado nos resultados experimentais descritos nos trabalhos de Sun e Wu [73], que desenvolveram um modelo matemático para prever o desempenho de um ambiente distribuído não-dedicado com base no trabalho de Gong et al. [34]. O modelo de predição leva em consideração que as estações de trabalho que compõem o ambiente distribuído são heterogêneas e não-dedicadas, o que é exatamente o caso das *Desktop Grids*. Deste modo, as tarefas paralelas submetidas à grade competem para a execução com tarefas sequenciais locais submetidas pelos usuários das máquinas. Esse modelo pressupõe que a tarefa pode ser particionada livremente em pequenos pedaços e não consideram os efeitos de sincronização, comunicação, processo de migração, ou grau de paralelismo das tarefas.

5.2.2 Arquitetura do OGST

A Figura 5.2 ilustra os principais componentes do OGST. O `Grid Feature Generator` (GFG) é um componente usado para definir um ambiente de grade simulado (nós da grade) e as aplicações a serem executadas com suas respectivas taxas de chegada. O `User Application Submission Tool` (UAST) representa

o usuário da grade e é o componente responsável pela submissão da aplicação, recebendo uma notificação sobre sua conclusão.

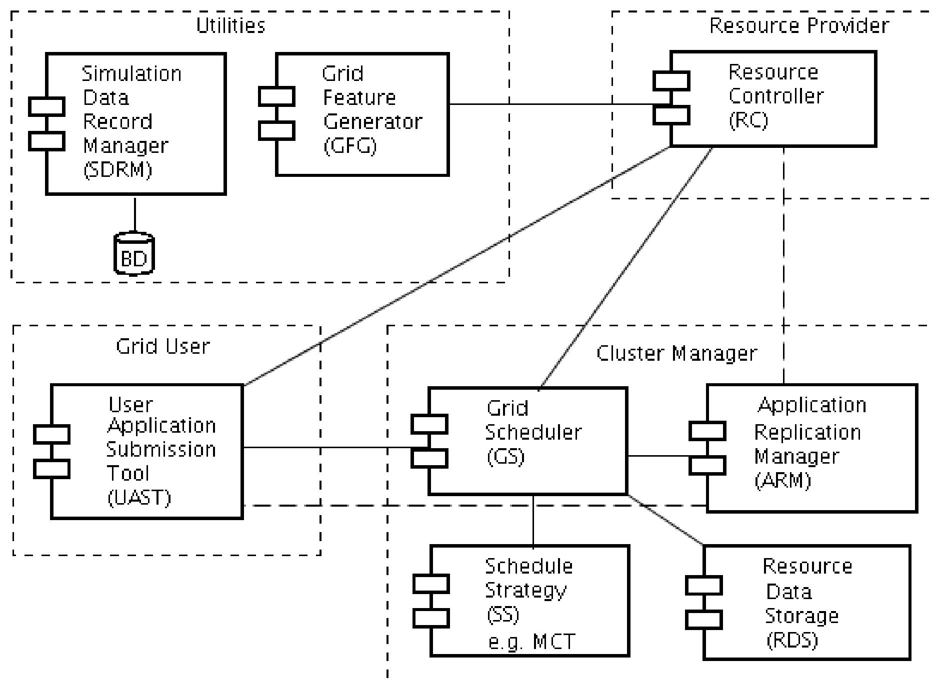


Figura 5.2: Principais componentes do OGST

O `GridScheduler` (GS) recebe submissões para execução de aplicações do UAST e executa o algoritmo de escalonamento, encapsulado no componente `SchedulingStrategy` (SS). A estratégia de escalonamento usa dados sobre a disponibilidade dos recursos da grade providos pelo componente `Resource Data Storage` (RDS). Cada tarefa da aplicação é então mapeada para a execução em um nó específico da grade. Cada nó da grade executa um `ResourceController` (RC), responsável pela instanciação e execução das tarefas das aplicações escalonadas para o nó, mantendo uma lista de tarefas esperando para serem executadas. Através desse componente, o OGST permite também a simulação de variação na carga local de trabalho para o recurso. O `SimulationDataRecordManager` (SDRM) é o componente responsável pelo armazenamento dos conjuntos de dados utilizados para gerar o ambiente de grade simulado (*data-sets*), bem como os resultados coletados após a simulação, tais como os tempos de conclusão e execução de tarefas. O `ApplicationReplicationManager` (ARM) é o componente responsável pelo gerenciamento da execução de réplicas de tarefas.

5.3 Outros Simuladores Baseados no GridSim

Assim como o OGST, diversos outros simuladores de grades utilizaram o GridSim como base para suas implementações e através de extensões conseguiram automatizar diversas tarefas, tais como a geração de recursos computacionais, redes, usuários, aplicações, dentre outros componentes. As próximas subseções descrevem resumidamente os simuladores GSSIM e Alea.

5.3.1 GSSIM

GSSIM (*Grid Scheduling Simulator*) [50] é um *framework* de simulação discreta que permite realizar estudos experimentais sobre algoritmos de escalonamento. A proposta desse simulador é ser uma ferramenta fácil de usar (abordagem *easy-to-use*), permitindo que algoritmos de escalonamento implementados e avaliados nesse simulador sejam facilmente portados para sistemas de grades reais.

Esse *framework* ainda permite a geração automatizada de recursos, enlaces de rede e outros elementos de um ambiente de grade a partir de aquivos que contêm a descrição do ambiente a ser gerado, tais como: quantidade de recursos, capacidade de processamento, capacidade de transmissão de dados, dentre outras informações que permitem detalhar o ambiente de grade simulado. As falhas de recursos também podem ser geradas. Porém, não há nenhuma técnica de tolerância a falhas nativamente implementada, dificultando assim a investigação de abordagens para a tolerância a falhas de recursos em grades computacionais usando esse simulador.

GSSIM permite gerar automaticamente as cargas de trabalho provenientes das aplicações da grade (*workload*, tanto de forma sintética, a partir de distribuições de probabilidade ou de modelos probabilísticos, quanto a partir de *traces* nos formatos GWF e SWF e MWF (*Metacentrum Workload Format*[8]) e permite também a descrição dos *workloads* utilizando o formato XML.

De acordo com o trabalho de [46], o GSSIM apresenta alguns problemas, tais como a execução lenta, fraca escalabilidade, e visualização de saída de dados deficiente. Além disso, o GSSIM não é compatível com as versões padrão do GridSim

[8]http://www.metacentrum.cz/en/

(incluindo a última versão, a GridSim 5.2 beta), uma vez que utiliza como base de sua implementação uma versão modificada do GridSim 4.

5.3.2 Alea

Alea é um outro simulador que também permite a implementação e avaliação de algoritmos de escalonamento em grades, disponibilizando inclusive uma biblioteca contendo algumas heurísticas baseadas em políticas de alocação de recursos do tipo FCFS (*First-Come*, *First-Served*), tais como, EDF [52], EASY Backfilling [70], a EDF-Backfilling [80].

O termo Alea2 [46] designa uma versão estendida e melhorada do simulador Alea [45]. As extensões abrangem melhorias no *design* da ferramenta, escalabilidade e velocidade das simulações. Além disso, o Alea2 trouxe consigo uma nova interface de visualização dos dados de saída da simulação, estendeu as políticas de alocação de recursos do GridSim e implementou a geração de falhas de recursos a partir de arquivos de *traces*. Alea2 tem suporte à geração de recursos, enlaces de rede e outros elementos de um ambiente de grade. Além do suporte à geração de cargas de trabalho a partir de *traces* dos formatos GWF, SWF e MWF, o Alea2 também dá suporte ao formato PWF (Pisa Workload Format) [9]. Assim como no GSSIM, falhas de recursos também podem ser geradas, mas não há nenhuma técnica de tolerância a falhas nativamente implementada.

5.4 SimGrid

SimGrid[10] [11] é um *framework* de simulação baseado em eventos discretos escrito em C. Esse simulador provê um conjunto de abstrações e funcionalidades básicas que permitem construir simulações para aplicações de domínio específico, visando o estudo de algoritmos de escalonamento para aplicações paralelas em plataformas computacionais distribuídas. O SimGrid utiliza arquivos XML como entrada para definir a topologia de rede e as características e responsabilidades dos recursos. Através da API SimGrid, tarefas podem ser atribuídas aos recursos de acordo com a política de escalonamento a ser simulada. Diferentemente do GridSim, o

[9] http://www.fi.muni.cz/ xklusac/alea/

[10] Página Inicial: `http://simgrid.gforge.inria.fr/`

SimGrid não é capaz de modelar uma grande quantidade de fenômenos, tais como os eventos de erros na comunicação em rede e também não consegue modelar topologias de rede complexas [66].

5.4.1 Arquitetura do SimGrid

A Figura 5.3 ilustra a arquitetura do SimGrid. Esta arquitetura possui diversas camadas, descritas a seguir, que são organizadas de acordo com as funcionalidades específicas disponibilizadas por cada conjunto de APIs (*Application Programing Interface*).

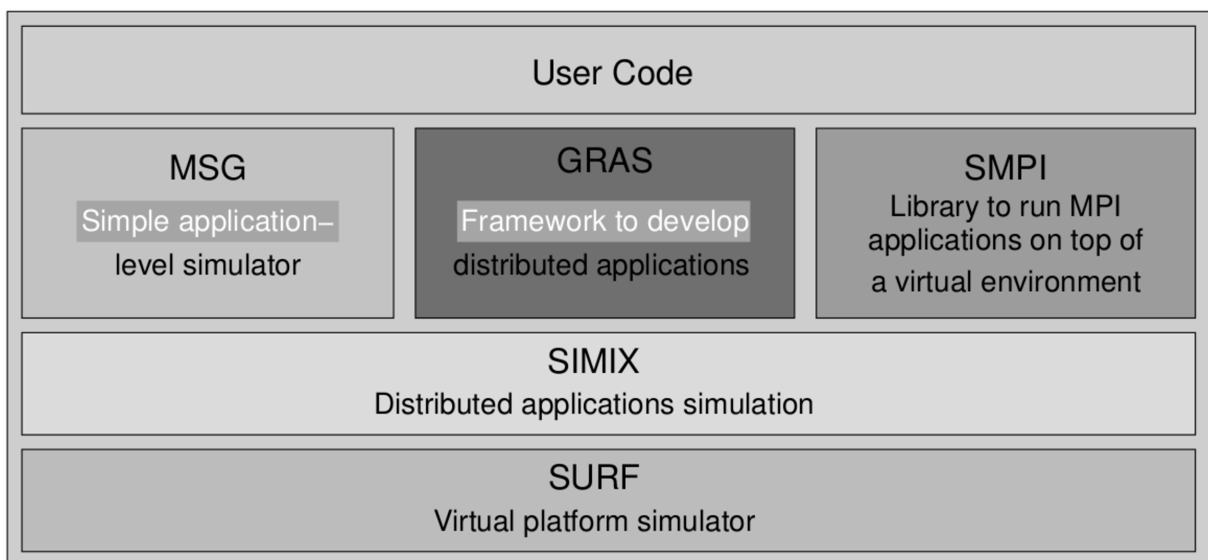


Figura 5.3: Arquitetura do SimGrid

- **XBT (*eXtended Bundle of Tools*)**: implementa uma biblioteca de estruturas de dados utilizadas pela camada SURF, e possui ainda mecanismos de exceção, um serviço de *logging*, e suporte à edição de configurações da ferramenta.

- **SURF**: disponibiliza uma API de baixo nível que, entre suas funcionalidades, é responsável por preparar o início da simulação, criando as estruturas utilizadas durante a simulação (ex: recursos da grade), gerenciar o relógio da simulação, calculando o tempo de conclusão das ações (ex: execução de uma tarefa em um recurso) e notificar ações que são finalizadas à interface do usuário.

- **MSG (*Meta-SimGrid*)**: o MSG foi o primeiro ambiente de programação provido no SimGrid e é a API mais utilizada para realizar a simulação e avaliação de

heurísticas de escalonamento. Ela permite simulações no modelo de aplicações paramétricas (com mestre e escravos). O mestre envia tarefas aos escravos, que lhe devolvem resultados.

- **GRAS (*Grid Reality And Simulation*)**: provê uma API para implementar aplicações distribuídas (ex: serviço de informação de recursos da grade) no topo de plataformas heterogêneas. Esta interface provê um suporte para o ciclo de desenvolvimento de aplicações através do simulador, permitindo que o código para plataformas reais seja automaticamente criado. Este ambiente faz uso da API fornecida pelo MSG para implementar a aplicação da simulação e de *sockets* para implementar a aplicação real.

- **SMPI**: visa permitir o desenvolvimento do mesmo código fonte para as aplicações do tipo MPI (*Message Passing Interface*) que serão executadas tanto no simulador, quanto em um ambiente real. Esta API ainda está em fase de desenvolvimento.

- **SimDag** : permite criar protótipos e simular heurísticas de escalonamento para aplicações paralelas estruturadas como grafos de tarefas (DAG - *Direct Acyclic Graphs*) [47]. Com esta API pode-se criar tarefas, adicionar dependências entre as tarefas, escalonar tarefas para execução nos recursos computacionais e computar o tempo de execução da aplicação DAG.

5.5 DSiDE

DSiDE é um simulador de grades escrito em C++. Inicialmente, esse simulador foi desenvolvido a partir de um modelo de grade simples, consistindo de um único escalonador, um servidor de informações da grade com múltiplos recursos computacionais e de armazenamento. Posteriormente, esse modelo inicial foi refinado e estendido com mecanismos de tolerância a falhas, *checkpointing* e replicação, além de mecanismos para permitir simular a reserva antecipada de recursos. O DSiDE permite gerar recursos, cargas de trabalho, e falhas de recursos tanto de forma sintética, quanto a partir de arquivos de *traces*.

No trabalho de Chtepen et al. [12], o DSiDE foi utilizado na investigação de uma abordagem autonômica para tolerância a falhas em grade computacionais que

combina as vantagens de duas abordagens estáticas, *checkpointing* e replicação. De acordo com os autores desse trabalho, simuladores como GridSim, SimGrid, e NSGrid [76] não são adequados para este estudo por causa de suas limitadas possibilidades para a modelagem dinâmica de sistemas distribuídos.

Os resultados da pesquisa mostraram que a abordagem autonômica, por ser adaptativa, obteve melhor desempenho em diversos cenários quando comparada as abordagens estáticas. No entanto, não é mencionado nesse trabalho se os mecanismos que simulam o comportamento autonômico da abordagem avaliada são providos nativamente pelo DSiDE, ou se esse simulador precisou ser estendido para que tal avaliação fosse possível. O referido trabalho também não menciona os tipo de adaptação suportados pelo DSiDE. Uma vez que não foi encontrada nenhuma referência para *download* do DSiDE, não foi possível constatar a presença de mecanismos autonômicos nativos nesse simulador.

5.6 Conclusão

No presente capítulo foram exploradas as características dos principais simuladores de grades apresentados na literatura. Foi possível observar que boa parte desses simuladores são baseados no GridSim, que fornece as primitivas básicas para a implementação de novos componentes e funcionalidades.

Dentre as funcionalidades básicas essenciais para a avaliação de abordagens autonômicas voltadas para grades estão: o suporte ao monitoramento de recursos, análise de informações de contexto e o controle e execução de ações de reconfiguração dinâmica. No entanto, como visto neste capítulo, apesar das variedades de funcionalidades apresentadas pelos simuladores analisados, nenhum deles foi concebido tendo por objetivo a disponibilização de funcionalidades que permitam a simulação de grades autonômicas.

A falta de um suporte nativo para a modelagem e simulação de abordagens autonômicas para grades nos atuais simuladores força os desenvolvedores desse tipo de abordagem a se preocupar também com a implementação desse suporte. A implementação de mecanismos autonômicos em simuladores requer o conhecimento de abordagens para o desenvolvimento de software adaptativo, tais como a reflexão

computacional e/ou programação orientada a aspectos. Essas abordagens permitem o desenvolvimento de mecanismos autonômicos mais transparentes.

O AGST, diferentemente de outros simuladores de grades citados anteriormente, já fornece um suporte nativo para a modelagem, simulação, e avaliação de abordagens autonômicas para grades. Os mecanismos autonômicos providos por esse simulador retira dos desenvolvedores de abordagens autonômicas a responsabilidade de implementar mecanismos autonômicos complexos, tais como ciência de contexto, adaptação dinâmica, transferência de estado e sincronização entre as ações de reconfigurações e a execução funcional dos elementos gerenciados.

6 AGST

O presente capítulo aborda o simulador de grades autonômicas desenvolvido nesta pesquisa, o AGST. São apresentadas as motivações e os requisitos para o desenvolvimento desse simulador, bem como são explicadas a sua arquitetura e as funcionalidades, em especial aquelas responsáveis pela modelagem, simulação e avaliação de abordagens autonômicas para grades de computadores. Por fim, é feita uma breve análise das principais limitações de uso do AGST.

6.1 Introdução ao AGST

O AGST[1] [22] é uma ferramenta que dá suporte à modelagem e simulação de ciclos de gerenciamento autonômico em grades de computadores, permitindo a construção e avaliação de abordagens autonômicas voltadas para esses ambientes computacionais. O principal objetivo com o desenvolvimento do AGST é auxiliar desenvolvedores de *middlewares* autonômicos para grades na avaliação de novos conceitos em suas implementações.

As motivações para o desenvolvimento do AGST surgiram no contexto do projeto "Desenvolvimento de Mecanismos Autonômicos para Grades Computacionais", financiado pelo CNPq e desenvolvido pelo Laboratório de Sistemas Distribuídos da Universidade Federal do Maranhão (LSD-UFMA). O objetivo principal desse projeto era o desenvolvimento e incorporação de mecanismos de computação autonômica ao *middleware* InteGrade (ver seção 3.3). Durante esse projeto, pesquisadores do LSD-UFMA começaram a desenvolver abordagens autonômicas voltadas para ambientes de grades de computadores, especialmente *Desktop Grids*. As abordagens autonômicas foram desenvolvidas seguindo a arquitetura mais conhecida para o desenvolvimento de sistemas autonômicos: a MAPE-K.

Para a avaliação das mecanismos autonômicos desenvolvidos, buscou-se ferramentas de simulação que fornecessem algum tipo de suporte à modelagem e simulação de ciclos de gerenciamento autonômico em grades de computadores

[1]http://www.lsd.ufma.br/ agst/

baseados no modelo MAPE-K. Dos diversos simuladores analisados no capítulo 5, nenhum possuía os requisitos necessários para a modelagem e simulação das abordagens autonômicas desenvolvidas. Esse fato, aliado à crescente importância de aplicações da computação autonômica em grades de computadores, como evidenciado na Seção 3.4, foram as principais motivações para o desenvolvimento de um simulador que facilitasse a implementação e avaliação de mecanismos autonômicos para grades computacionais. Esse simulador foi denominado *Autonomic Grid Simulation Tool*, ou simplesmente AGST.

O AGST é um simulador de grades autonômicas baseado na arquitetura MAPE-K, e, portanto, oferece um arcabouço que implementa essa arquitetura, provendo suporte à todas as fases de um ciclo de gerenciamento autonômico MAPE-K: monitoramento de recursos do ambiente de grade por meio de sensores; análise das informações de contexto; planejamento de ações de reconfiguração dinâmica; controle e execução de ações de adaptação dinâmica através de atuadores. O AGST oferece suporte à execução de dois tipos de adaptação dinâmica: paramétrica e composicional. A adaptação paramétrica consiste na alteração de variáveis que determinam o comportamento de algoritmos utilizados pelo *middleware* de grade. Já a adaptação composicional consiste na troca de algoritmos ou partes estruturais do *middleware*, permitindo que este adote novas estratégias para tratar novas situações e reagir às mudanças de contexto no ambiente da grade. O AGST implementa um mecanismo de transferência de estado durante a substituição de componentes em adaptações estruturais, e oferece um mecanismo que trata a sincronização entre a execução das ações de reconfiguração e a execução funcional dos elementos gerenciados.

A principal vantagem do uso do AGST está na diminuição da complexidade no processo de modelagem e simulação de abordagens autonômicas para grades. As funcionalidades do simulador proposto foram implementadas seguindo um conceito de transparência. Esse conceito significa que não é necessário ao usuário do AGST realizar instrumentações[2] estáticas dos elementos gerenciados para que os mesmos se tornem passíveis de serem monitorados e/ou reconfigurados dinamicamente. Por meio de extensões aos componentes do AGST, os usuários podem facilmente especificar seus próprios critérios de classificação e filtragem dos dados de contexto,

[2]A instrumentação é o processo de adicionar/alterar código em um programa ou classe, podendo ser estática (diretamente na classe base) ou dinâmica (em tempo de execução).

além de implementar as decisões e ações que devem ser tomadas pelas abordagens autonômicas que pretendem avaliar.

O AGST é um simulador orientado a objetos escrito em Java, desenvolvido tendo como base de sua implementação os simuladores OGST, GridSim, e SimJava2, aos quais proveu diversas extensões. Considerando a taxonomia de simuladores proposta por Salutio et al. [72], o AGST é um simulador de eventos discretos orientado a objetos pertencente às categorias: *Sistemas Distribuídos* e *Sistemas de Escalonamento em Grades*.

6.2 Requisitos para o desenvolvimento do AGST

Em sistemas autonômicos, a funcionalidade de auto-gerenciamento é obtida através da composição de diversas propriedades, como a autoconfiguração, autorrecuperação, auto-otimização e autoproteção. A manutenção dessas propriedades exige a provisão de mecanismos de implementação que as apoiem, como, por exemplo: a ciência de contexto interna (*self-aware*) e externa (*environment-aware*), o monitoramento desses contextos (*self-monitoring*) e adaptação dinâmica (*self-adjusting*) [16]. Por esse motivo, os princípios que nortearam o desenvolvimento do AGST foram definidos de forma a atender os requisitos considerados necessários para tornar possível a simulação de grades autonômicas. Os requisitos considerados são:

- **Ciência de contexto**: o simulador deve prover aos usuários mecanismos nativos para o monitoramento de dados que reflitam mudanças de comportamento do elemento gerenciado ou informações do ambiente de execução que sejam relevantes;

- **Adaptação dinâmica**: o simulador deve prover aos usuários o suporte nativo para execução de ações reconfiguração dinâmica, de forma que os componentes da grade possam ser adaptados de forma paramétrica e estrutural. Em adaptações que envolvem a substituição de componentes da grade, deve-se garantir a transferência de estado entre os componentes quando necessário;

- **Transparência**: os mecanismos de acesso a dados de contexto e adaptação dinâmica dos componentes devem ser transparentes em relação aos elementos

gerenciados, ou seja, nenhum componente deve precisar ter seu código-fonte alterado para que ele se torne passível de ser monitorado ou adaptado dinamicamente.

- **Arquitetura Autonômica**: o simulador deve implementar a arquitetura mais adotada para sistema autonômicos, a MAPE-K, de modo a facilitar a portabilidade das abordagens autonômicas implementadas no simulador para um *middlware* de grade real.

6.3 Arquitetura do AGST

A arquitetura do AGST é dividida em camadas, conforme ilustrado na figura 6.1. Esta abordagem facilita a integração de novos componentes ou camadas.

Figura 6.1: Arquitetura do AGST

A primeira camada corresponde a infraestrutura de suporte à execução do AGST. Essa é composta pela JVM (*Java Virtual Machine*) e pelo Sistema Gerenciador de Banco de Dados (SGBD) MySQL. A JVM é necessário dado que o código das camadas superiores é escrito na linguagem Java. Já o SGBD MySQL é utilizado por padrão para armazenar os dados obtidos durante a execução das simulações.

A segunda camada corresponde ao pacote de simulação de propósito geral baseado em eventos discretos responsável pela interação entre as entidades

da simulação, o SimJava2. Essa camada permite que os componentes das camadas superiores se comuniquem uns com os outros através de eventos (`Sim_Event`).

A terceira camada corresponde ao GridSim, que fornece para as camadas superiores os componentes primitivos necessárias para criação e gerenciamento de componentes do ambiente de grade, tais como: tarefas (`Gridlet`), recursos computacionais (`GridResource`), enlaces de rede (`Link`), usuários (`GridUser`), e o serviço de informações da grade (`AbstractGIS`). Qualquer outro componente que precise se comunicar com outros componentes através do envio e recebimento de eventos, tal como um escalonador, pode ser implementado pelas camadas superiores através de extensões da classe `GridSim`.

A quarta camada corresponde ao simulador de grades oportunistas OGST, cujas principais funcionalidades foram apresentadas na seção 5.2. Mais informações sobre o OGST podem ser encontradas nos trabalhos de Cunha Filho [17, 18]. Essa camada faz uso dos componentes primitivos fornecidos pela camada inferior (GridSim), e a partir deles implementa componentes específicos do modelo de grade simulado pelo OGST, tais como: Escalonador (`GridSchduler`), o Serviço de informações da Grade `ResourceDataStorage`, e a Ferramenta de Submissão de Aplicações (`UserApplicationSubmitionTool`). Essa camada provê mecanismos flexíveis para a geração automatizada de aplicações, recursos computacionais e enlaces de rede.

Ao longo do desenvolvimento do trabalho de pesquisa proposto nesta dissertação, foram implementadas novas funcionalidades no OGST e adicionadas melhorias em funcionalidades já existentes, motivados pela necessidade de prover novos serviços considerados essenciais para os tipos de experimentos e cenários aos quais houve necessidade simular com o AGST. Entre estas alterações, destaca-se a implementação da geração automatizada de recursos computacionais com carga de trabalho local (*hostload*). Também foram realizadas alterações no mecanismo que gerencia a ocorrência de falhas de recursos, que antes só podiam ser geradas sinteticamente a partir de distribuições de probabilidade (normal, poisson, uniforme, exponencial, hiper-exponencial e *weibull*), e que agora também podem ser geradas a partir de bancos de dados de traces no padrão FTA (*Failure Trace Archive*[3]) [48] coletados de *Desktop Grids* reais. A geração automatizada da aplicações, que

[3]http://fta.inria.fr/

antes podiam ser geradas sinteticamente a partir de distribuições de probabilidade, passaram a ser geradas também sinteticamente a partir do modelo probabilístico de Lublin [55], e ainda a partir de arquivos de *traces* de *workloads* coletados também de ambientes reais. A implementação dessas funcionalidades permite a realização de experimentos cujas características dos cenários simulados tornam-se mais próximas de cenários reais.

Além de recursos para a geração do ambientes de grade e das aplicações, a camada do OGST também fornece uma biblioteca de heurísticas de escalonamento de tarefas comumente utilizadas por *middlewares* de grades reais: MCT, Min-Min, Work-Queue, e InteGrade. Outras heurísticas podem ser facilmente implementadas através de extensões ao componente `ScheduleStrategy`, provido pelo OGST. Também é fornecido por essa camada um conjunto de estratégias para a tolerância a falhas na execução de aplicações: *checkpointing*, reinício e replicação. Implementamos algumas melhorias no primeiro mecanismo, que agora consegue simular o custo computacional (medido em segundos) gerado na execução do *checkpointing*.

Esta camada é ainda responsável pela geração automática do esquema utilizado pelo banco de dados (primeira camada), provida pelo componente `SimulationDataRecordManager`. O modelo de dados relacional definido pelo OGST teve que ser estendido, de forma a permitir o armazenamento de dados referentes a execução de adaptações paramétricas e estruturais durante a execução de uma simulação. São armazenados o instante em que ocorreu a adaptação, o tipo de parâmetro ajustado e seu respectivo valor (em caso de adaptação paramétrica), bem como o tipo de componente ajustado ou substituído (em caso de adaptação composicional). Isso permite ao usuário analisar o comportamento adaptativo de suas abordagens ao longo do tempo da simulação.

A quinta, e última camada, corresponde ao MAPE-K Simulation Framework. Essa camada fornece um conjunto básico de componentes extensíveis e reutilizáveis para modelagem e simulação dos ciclos de gerenciamento autonômico em grades, e provê ao AGST os seguintes mecanismos: I) gerenciamento das operações de acesso e modificação das variáveis de estados dos elementos gerenciados; II) execução de ações de reconfiguração dinâmicas no ambiente de grade e; III) gerenciamento da comunicação e da interação entre componentes dos ciclo autonômicos. As próximas seções tratam exclusivamente do detalhamento de cada funcionalidade provida pelo

MAPE-K Simulation Framework, dado que este é o produto principal deste trabalho de mestrado.

6.4 Funcionalidades do MAPE-K Simulation Framework

Para possibilitar a modelagem e a simulação de grades autonômicas, o MAPE-K Simulation Framework oferece as seguintes funcionalidades: Gerenciamento de Ciclos Autonômicos, Monitoramento, Análise e Planejamento, Controle e Execução de Ações de Reconfiguração, Adaptações Paramétricas e Composicionais, Transferência de Estado e Sincronização. As funcionalidades citadas estão descritas nas próximas subseções. Essas funcionalidades estão em conformidade com a versão 2.0 do AGST, que até a data de publicação deste trabalho era a mais recente.

6.4.1 Gerenciamento de Ciclos Autonômicos

A fim de prover as funcionalidades referentes a todas as etapas definidas pela arquitetura MAPE-K, o AGST provê componentes para a modelagem e simulação de ciclos de gerenciamento autonômico. Cada ciclo autonômico possui um único gerente autonômico. Cada gerente autonômico possui seus respectivos componentes autonômicos: monitores, analisadores, e executores. Os gerentes autonômicos utilizam pontos de contato (*touchpoints*) para capturar informações dos elementos gerenciados e/ou prover reconfigurações sobre os mesmos. Esses pontos de contato são os sensores e os atuadores.

O AGST trabalha com os conceitos de ciclo local e ciclo global de gerenciamento (ver seção 2.2), e por esse motivo oferece dois tipos de gerentes autonômicos. Ambos estendem da classe `AbstractAutonomicManager`. São eles:

1. `LocalAutonomicManager`: gerencia um ciclo local. É capaz de monitorar e tomar decisões que não afetam a estrutura global do sistema, uma vez que suas ações estarão baseadas em informações obtidas localmente;

2. `GlobalAutonomicManager`: gerencia o ciclo global. É capaz de receber informações a partir dos diversos ciclos de controle local ou a partir de um monitoramento global, e tomar decisões que afetam o ambiente da grade

globalmente com base nessas informações. Além disso, o gerente global tem a capacidade de exercer controle sobre a execução dos ciclos locais, podendo interromper e continuar a execução dos mesmos quando necessário.

A possibilidade de simular diversos ciclos autonômicos é importante para o desenvolvimento de abordagens que possuem dois níveis de adaptação. Um exemplo prático de uma abordagem dessa natureza pode ser encontrado nos trabalhos de Viana [77, 78], que propôs uma abordagem autonômica para tolerância a falhas de aplicações em grades oportunistas baseada tanto no ajuste de parâmetros (primeiro nível de adaptação), como na alternância de duas estratégias distintas ao longo do tempo, replicação e *checkpointing*, de acordo com o contexto que for mais favorável para cada estratégia (segundo nível de adaptação). Se no ambiente da grade há bastante recursos disponíveis para a grade, então utiliza-se a replicação, porém, se há poucos recursos, então utiliza-se a técnica de *checkpointing*.

Essa abordagem utiliza, além do gerente global, dois gerentes locais: um para gerenciar a execução da técnica replicação e outro para gerenciar a execução da técnica de *checkpointing*. O gerente local da replicação monitora a quantidade de recursos disponíveis, e com base nessa informação decide se aumenta ou diminui a quantidade de réplicas a serem geradas para cada aplicação. Já o gerente local de *checkpointing* monitora a frequência com a qual os recursos falham (MTBF - *Mean Time Between Failures*) e, com base nessa informação, decide se aumenta ou diminui o intervalo entre *checkpoints*. Essas decisões são tomadas em um nível local, já que um simples ajuste de parâmetros em ambos os ciclos locais não representam uma mudança em relação ao tipo de estratégia de tolerância a falhas utilizada. Enquanto isso, o gerente autonômico global, por ter uma visão geral da grade, pode decidir o momento em que a estratégia de tolerância a falhas vigente deve ser trocada, caso perceba que os esforços locais para melhorar o desempenho da estratégia não sejam mais suficientes.

Além dos conceitos de ciclo autonômico local e global, AGST trabalha com os conceitos de: ciclo de gerenciamento simples e cilo de gerenciamento composto [62]. Um ciclo de gerenciamento simples é formado pela interação entre monitores, analisadores e executores pertencentes a um mesmo ciclo autonômico, seja ele local ou global. Já o ciclo de gerenciamento composto é formado pela interação de componentes autonômicos (sensores, monitores, analisadores, executores, e atuadores) pertencentes

à ciclos de gerenciamento distintos. Por esse motivo, quando um gerente autonômico global decide interromper a execução de um determinado ciclo autonômico local, essa decisão irá afetar, na prática, apenas a interação entre os componentes autonômicos pertences ao a esse ciclo. Caso algum dos componentes autonômicos pertencentes ao ciclo interrompido esteja interagindo com algum componente estrangeiro, ainda assim esses componentes poderão se comunicar entre si e trocar dados. Considerando o exemplo anterior, caso o analisador do ciclo de *checkpointing* esteja interessado em receber informações de contexto providas pelo monitor do ciclo de replicação, eles poderão enviar e receber dados ainda que seus respectivos ciclos locais tenham tido suas execuções paradas pelo gerente autonômico global.

A interação entre os diversos componentes autonômicos dentro de um mesmo ciclo e/ou em ciclos diferentes ocorre através do mecanismo de envio e recebimento de eventos provido pela quarta camada (SimJava2). Para que um componente autonômico possa enviar um evento para outro componente autonômico é preciso conhecer o identificador do destinatário. Esse identificador (String) é fornecido obrigatoriamente no ato da instanciação de todos os componentes autonômicos.

Todas as informações acerca dos componentes autonômicos instanciados durante a simulação são armazenadas em um registrador global denominado `ConfigurationManagmentDataBase`. Esse registrador global pode ser utilizado para consultar informações sobre os componentes autonômicos. Por exemplo, durante a simulação um determinado analisador pode consultar quais monitores estão aptos a fornecer um determinado tipo de dado de contexto. A partir desse desse registrador global é possível descobrir também se determinados componentes estão em um mesmo ciclo autonômico ou em ciclos distintos. Por exemplo, o mecanismo de interrupção dos ciclos autonômicos utiliza essa informação para definir se deve ou não interromper a interação entre os componentes, já que essa interrupção deve acontecer apenas quando os mesmos estão dentro do mesmo ciclo autonômico. O `ConfigurationManagmentDataBase` é capaz de fornecer ainda informações acerca de quais componentes possuem um ponto de contato (sensor ou atuador) acoplado.

Qualquer componente do OGST (quarta camada), ou até mesmo do MAPE-K Simulation Framework (quinta camada) pode ser um elemento gerenciado. No entanto, esses componentes não precisam de alterações na implementações de suas

classes, tal como implementar algum tipo de interface ou estender alguma outra classe específica do MAPE-K Simulation Framework, devido ao uso de técnicas de reflexão computacional e programação orientada a aspectos, que serão vistas no capítulo 7, que descreve a implementação do AGST.

6.4.2 Monitoramento

Dentro de cada ciclo autonômico, a função de monitoramento do AGST é responsável por coletar informações que reflitam mudanças significativas no comportamento dos elementos gerenciados e/ou obter outros dados de contexto do ambiente de grade simulado que sejam significativos para iniciar algum tipo de adaptação que leve o sistema ao seu estado de equilíbrio. Considerando o exemplo anterior, uma queda na quantidade de recursos disponíveis para a execução de aplicações, que corresponde a uma das propriedades poder ser considerada relevante, dependendo de sua intensidade. Essa quantidade de recursos (propriedade de contexto) é obtida a partir de um monitoramento do Serviços de Informações da Grade (elemento gerenciado). Neste caso, um ajuste da quantidade de réplicas de aplicações geradas pelo *middleware* pode levar o sistema a uma melhora no seu desempenho.

Para exercer a função de monitoramento no AGST, implementamos dois componentes que trabalham de forma integrada: sensores e monitores. O sensores são implementados pelo classe *Sensor*, já os monitores são implementados através de extensões da classe *AbstractMonitor*. A função de coleta dos dados é exercida por sensores acoplados diretamente aos elementos gerenciados. Cada sensor só coleta dados quando requisitado por um ou mais monitores. Já os monitores são os responsáveis pela filtragem e classificação dos dados coletados pelos sensores. No AGST, cada monitor está restrito a observação de uma única propriedade do sistema, podendo receber dados de contexto de apenas um sensor. Cada monitor possui um intervalo de monitoramento (periodicidade) definido no ato de sua instanciação. Por esse motivo, a requisição dos dados de contexto é realizada pelos monitores para os sensores. Além do monitoramento periódico, no qual os dados de contexto são coletados em intervalos de tempo regulares, foi implementado também uma forma de monitoramento chamada de monitoramento sob-demanda. Nessa última, os dados de contexto podem ser monitorados a qualquer instante, a partir de uma requisição

realizada por um analisador. Cada monitor pode trabalhar com ambas as formas de monitoramento, até mesmo simultaneamente.

Os dados de contexto coletados pelos sensores podem chegar aos monitores em um formato inadequado para os analisadores (que receberão esses dados na fase de análise e planejamento), exigindo, portanto, algum tipo de modificação, classificação, ou refinamento desses dados. Com base no último exemplo, se o sensor acoplado ao serviço de informações de uma grade obtém a lista dos recursos computacionais disponíveis para a execução de aplicações em um dado momento, essa lista pode então ser pré-processada de forma que chegue aos analisadores ordenada pelo poder de processamento dessas máquinas, e/ou ser pré-processada para converter o *hostname* de cada recurso para o seu respectivo endereço IP. Devido a essa necessidade, cada monitor no AGST pode executar, quando necessário, um pré-processamento sobre os dados obtidos pelos sensores. Para os casos em que não é necessário realizar nenhum tipo de pré-processamento, o AGST já disponibiliza um monitor padrão (`DefaultMonitor`).

Após o pré-processamento, cabe aos monitores fazer uma análise prévia dos dados de contexto com o objetivo de identificar a ocorrência de alguma mudança relevante no contexto da grade que possa subsidiar alguma tomada de decisão por parte de algum analisador. Como vimos no exemplo anterior, uma queda repentina na quantidade recursos disponíveis para a execução de aplicações pode ser considerada uma mudança drástica no contexto, e, portanto, bastante significativa.

No AGST, a detecção de mudanças significativas em relação aos valores apresentados por uma determinada propriedade do ambiente monitorado é feita através de regras. Para implementar uma regra no AGST é necessário estender a classe `AbstractRule`. Considerando o exemplo anterior, uma mudança no percentual de ocupação dos recursos de 20.1% para 20.2%, pode não representar uma mudança significativa para a grade. Por isso, as regras servem para definir intervalos. São consideradas mudanças significativas se os valores obtidos no monitoramento mudem de um intervalo para o outro. Por exemplo, podemos definir dois intervalos: um BAIXO entre 0% e 50% e outro ALTO entre 50% e 100%. Se o último valor monitorado for igual a 58, e no monitoramento anterior esse valor era 30, significa que o estado do ambiente mudou de BAIXO para ALTO. Cada regra recebe o valor de uma propriedade do sistema e retorna um valor de verdadeiro ou falso. Se a regra retornar um valor

verdadeiro, isso indica que uma notificação de mudança de contexto deve ser enviada aos analisadores interessados.

Em virtude disso, cada monitor do AGST mantém uma lista dos analisadores interessados em mudanças de contexto da propriedade monitorada. Nem toda notificação, ou seja, nem toda mudança de intervalo na propriedade de contexto monitorada, pode ser de interesse de todos os analisadores registrados. Por exemplo, um analisador pode estar interessado em receber notificações somente quando o valor do monitoramento da disponibilidade de recursos na grade mudar de ALTO para BAIXO, embora o monitor também gere notificações quando ocorrer mudanças de BAIXO para ALTO. Prevendo essas situações, o AGST fornece filtros que permitem selecionar as notificações de contexto de interesse de um dado analisador. O filtro deve ser informado no ato do registro do analisador junto ao monitor.

No AGST os dados monitorados são enviados para os analisadores encapsulados em forma de eventos (`MonitoredDataEvent`). Além do valor corrente da propriedade monitorada, cada evento carrega informações acerca de qual monitor enviou o dado, bem como o instante no qual a medição da propriedade foi realizada pelo sensor.

Uma característica fundamental do mecanismo de monitoramento é a sua transparência com relação ao código fonte do elemento gerenciado. Desta forma, para se ter acesso às propriedades monitoradas, não é necessário ao usuário realizar qualquer instrumentação estática no código fonte do componente (elemento) gerenciado. Para tanto, utiliza-se uma abordagem bastante comum para desenvolvimento de sistemas autonômicos: a reflexão computacional. Essa técnica permite acessar informações acerca da implementação dos componentes que compõem uma aplicação em tempo de execução.

6.4.3 Análise e Planejamento

Ao receber uma notificação da ocorrência de um evento relativo à alguma mudança de contexto em uma propriedade monitorada, o analisador inicia a fase de análise e planejamento do ciclo autonômico. No AGST, para implementar um analisador, é preciso estender a classe `AbstractAnalizer`. Os analisadores decidem,

com base nos dados de contexto coletados e no conhecimento que possuem sobre o ambiente de grade, quais ações deverão ser tomadas pela abordagem autonômica.

A implementação do mecanismo de tomada de decisão a ser utilizado por cada analisador durante as simulações é de responsabilidade do usuário do AGST. Para simplificar esta tarefa, o AGST provê um mecanismo de decisão baseado em regras do tipo evento-condição-ação (ECA *rules*). Para implementar essas regras deve-se estender a classe `AbstractECARule`. Cada regra recebe um evento contendo um dado monitorado (`MonitoredDataEvent`) e retorna a ação que deverá ser executada na próxima fase o cilo MAPE-K caso as condições impostas pela regra sejam satisfeitas. Considerando o exemplo do mecanismo de tolerância a falhas autonômico descrito anteriormente, o analisador responsável pela política de replicação pode implementar uma regra que defina a ação de diminuir a quantidade de réplicas a serem geradas a partir de novas submissões de aplicações à grade caso receba a notificação de um evento que indique baixa disponibilidade de recursos na mesma.

É importante frisar que a implementação dessas regras pode também ser feita diretamente no analisador. Porém, isso impede que um outro analisador a reutilize, gerando duas especificações da mesma regra. A utilização do componente `AbstractECARule` como base para a implementação de regras evita isso, já que dois analisadores distintos podem utilizar a mesma instancia de uma regra. Para os caso em que os usuário do AGST queiram implementar suas regras estendendo a classe `AbstractECARule`, eles devem utilizar um tipo padrão de analisador denominado `DefaultAnalyzer`. Esse analisador pode ser obtido através de simples instanciações, dispensando assim a implementação de um novo tipo de analisador. É possível adicionar diversas regras ECA ao `DefaultAnalyzer`, cada uma retornando uma ação específica.

Ao final da função de análise e planejamento, cada analisador deve construir um plano de ações (`ActionPlan`). Cada plano de ações contém uma lista de todas as ações a serem executadas na próxima fase do ciclo MAPE-K. É importante frisar que se esse plano for composto por mais de uma ação, elas devem ser inseridas na lista na mesma ordem em que devem ser executadas. Uma vez pronto, esse plano de ações é enviado ao executor, que pode estar no mesmo ciclo autonômico do analisador, ou em outro. Assim como os eventos de monitoramento, o plano de ações também

é enviado como um evento, denominado `ActionPlanEvent`. Esse evento contém, além das ações a serem realizadas, a informação de qual analisador construiu o plano.

6.4.4 Controle e Execução de Ações de Reconfiguração

Dentro do ciclo autonômico, a função de controle e execução das ações de reconfiguração que foram planejadas pelos analisadores é exercida por executores através dos atuadores. Cada ação de reconfiguração é implementada pelo usuário do AGST estendendo-se a classe `AbstractAction`. Os executores do AGST são implementados através de extensões da classe `AbstractExecutor`. O AGST já provê um executor padrão, que pode ser obtido através de uma simples instanciação da classe *DefaultExecutor*.

Uma vez que os executores não possuem acesso direto aos elementos gerenciados, eles se utilizam de atuadores conectados diretamente a esses elementos. No AGST, os atuadores são implementados pela classe `Effector`. A relação entre os atuadores e os elementos gerenciados é de um para um. O atuador de cada elemento gerenciado só pode ser gerado a partir de uma fábrica específica para esse tipo de componente, denominada `EffectorFactory`. Essa fábrica garante que para cada elemento gerenciado só haverá um atuador acoplado. Essa singularidade impede que duas ações de reconfiguração sejam efetuadas simultaneamente sobre o mesmo elemento gerenciado. O componente `EffectorFactory` faz uma consulta ao `ConfigurationManagmentDataBase` para descobrir se já existe algum atuador acoplado um determinado elemento.

6.4.5 Adaptações Paramétricas e Composicionais

Os atuadores providos pelo AGST são capazes de efetuar dois tipos de adaptação dinâmica: a paramétrica e a composicional. O mecanismo de adaptação paramétrica permite a alteração no estado dos elementos gerenciados, ou seja, permite modificar os valores atribuídos às variáveis declaradas em suas classes. Considerando o mecanismo de tolerância a falhas proposto por Viana, a alteração da quantidade de réplicas e a modificação do intervalo entre *checkpoints* constituem um tipo de adaptação paramétrica. Já a modificação do intervalo entre *checkpoints* é feita nas tarefas das aplicações, sendo que o `FaultToleranceManager` utiliza esse intervalo

para conhecer o momento em que deve proceder aos *chekcpoints*. Já o mecanismo de adaptação composicional permite modificações de nível estrutural no ambiente de grade, tais como a substituição de componentes e algoritmos. Esse tipo de adaptação foi implementado para ser usado, por exemplo, nos casos em que a simples variabilidade de parâmetros não for suficiente para atingir as metas de desempenho das abordagens autonômicas implementadas. A substituição da estratégia de tolerância a falhas por outra constitui um tipo de adaptação composicional. O elemento gerenciado nos exemplos de adaptação paramétrica e composicional citados é o componente `SimulationSettings`, que corresponde ao componente utilizado pelo AGST para armazenar as configurações do *middlware de grade*, dentre elas a estratégia de tolerância a falhas utilizada (*FaultoStrategy*).

Semelhante ao mecanismo de monitoramento, os mecanismos de adaptação paramétrica e composicional também dispensam instrumentações estáticas nos componentes que serão adaptados durante simulação. Mais detalhes de como empregamos a reflexão computacional para a implementação dos mecanismos de adaptação paramétrica e composicional serão vistos no Capítulo 7.

Uma vez que as ações de reconfiguração são executadas em tempo de execução, é preciso tratar de duas questões principais: I) a transferência de estado entre os componentes que participam de um processo de substituição; e II) a sincronização entre a execução de reconfigurações e a execução funcional do sistema. Comentamos nas próximas subseções a maneira como o AGST trata ambas as questões.

6.4.6 Transferência de Estado

Durante o processo de substituição dinâmica de componentes pode ser necessário transferir informações de um componente para outro. Essa transferência de estado entre o componente original e o componente substituto garante que este último inicie sua atividade com as informações apropriadas, mantendo assim a consistência do sistema após a reconfiguração. Se tomarmos como exemplo uma abordagem autonômica para o escalonamento de aplicações que troca uma heurística de escalonamento em uso por outra de acordo com mudanças no contexto de execução, percebemos que é preciso transferir a lista de tarefas que ainda não foram mapeadas da primeira heurística para a segunda. Do contrário, essas aplicações seriam perdidas

após o processo de reconfiguração. Isso caracterizaria uma inconsistência, já que as aplicações perdidas apesar de submetidas jamais seriam executadas.

No AGST, a transferência de estado também é provida pelos atuadores, dado que para se conseguir transferir as variáveis que caracterizam o estado do objeto para um outro, é preciso ter acesso garantido a implementação dos mesmos. A transferência de estado provida pelo AGST funciona da seguinte forma: o atuador responsável pela reconfiguração identifica inicialmente se o componente substituto é compatível com o componente original, ou seja, esse processo descobre se ambos os componentes estão dentro uma hierarquia de classe comum. Caso os componentes sejam de uma mesma família, procede-se então a busca pelas variáveis comuns aos componentes, que apesar de serem da mesma família, podem estar em níveis diferentes da hierarquia. Após a identificação das variáveis comuns inicia-se o processo de espelhamento. Esse processo atribui para variáveis do componente substituto o mesmo valor de suas equivalentes no componente original.

A transferência de estado se utiliza dos mesmos princípios dos mecanismos de monitoramento e adaptação dinâmica, uma vez que para transferir os estados das variáveis de um componente para o outro, esse mecanismo necessita tanto acessar os dados do componente original (como no monitoramento) quanto modificar os dados do componente substituto (como na adaptação dinâmica). Consequentemente, esse mecanismo se vale do mesmo conceito de transparência e também utiliza a reflexão computacional.

6.4.7 Sincronização

Dependendo da complexidade da reconfiguração, a parte do sistema a ser modificada não deve estar disponível para a execução funcional durante o tempo de reconfiguração, do contrário isso poderia levar o sistema a um estado inconsistente. Considerando o exemplo do escalonamento autonômico, antes de substituir uma heurística de escalonamento por outra, é necessário parar a execução funcional do escalonador, ou seja, deve-se suspender temporariamente o escalonamento de tarefas até que o processo de substituição da heurística e eventuais transferências de estado sejam completadas.

Para compreender a forma como o AGST trata a sincronização entre a execução de reconfigurações e a execução funcional do sistema é preciso relembrar alguns aspectos sobre o funcionamento do OGST (quarta camada da Figura 6.1). Os componentes que compõem a simulação se comunicam através de eventos do GridSim (`Sim_event`), utilizando os métodos `send()` e `sim_get_next()` para enviar e receber eventos discretos. Os eventos da simulação podem transportar qualquer tipo de dado (`Object`) e são identificados por um número inteiro chamado de *tag*. Cada evento está associado a um tempo simulado. Durante a simulação, os eventos que ainda não foram tratados pelo componente, ou seja, que ainda não aconteceram no tempo simulado são mantidos em uma fila e são ordenados de forma crescente de acordo com sua prioridade. A prioridade de um evento é o tempo simulado associado a ele. Por exemplo, se o `GridScheduler` receber dois eventos de submissão de tarefas, e_1 e e_2, cujos tempos simulados associado a esses evento são respectivamente 5 e 6, isso implica que a tarefa encapsulada no evento e_1 será escalonada primeiro.

O mecanismo de sincronização implementado pelo AGST é provido por componentes especiais capazes de interceptar os elementos gerenciados em pontos específicos de sua execução. Esses componentes especiais devem ser implementados pelo usuário a partir de extensões da classe `AbstractInterceptor`. O mecanismo de sincronização entre a execução de reconfigurações e a execução funcional dos elementos gerenciados funciona seguindo a seguinte sequência:

1. O analisador envia para o executor um plano de ações específico para esse tipo de reconfiguração. Esse plano deverá conter uma ação primária, que envia para o elemento gerenciado um evento discreto contendo uma ação secundária, que corresponde a ação adaptativa propriamente dita. Por exemplo, se o analisador (`AbstractAnalyzer`) decide que a heurística de escalonamento (`ScheduleStrategy`) deve ser trocada, então o plano de ações deve conter uma ação (`AbstractAction`) que determina o envio de um evento (`Sim_event`) para o `GridScheduler` identificado pela *tag* `OgstTags.DYNAMIC_RECONFIGURATION` (indicando que se trata de uma ação de reconfiguração dinâmica). Esse evento deverá carregar uma outra ação de forma encapsulada, esta última corresponde a ação propriamente dita, ou seja, a ação que substitui uma heurística por outra transferindo o estado entre ambas;

2. O executor recebe o plano de ações e executa a ação primária. Isso faz com que a ação de adaptação propriamente dita seja colocada na fila de eventos discretos do elemento gerenciado, que no exemplo anterior é o `GridScheduler`;

3. O elemento gerenciado recebe o evento de reconfiguração dinâmica. No entanto, a implementação do elemento gerenciado não especifica nenhum tratamento para esse tipo de evento, já que a execução de ações adaptativas não faz parte da execução funcional desse elemento. Por exemplo, se `GridScheduler` recebe um evento com a *tag* `GridSimTags.EXPERIMENT`, esse componente sabe que trata-se de um evento de submissão de um aplicação (`Application`), que está encapsulada no evento. O código para tratar esse evento já faz parte da implementação funcional desse componente. Porém, quando o `GridScheduler` recebe um evento com a `OgstTags.DYNAMIC_RECONFIGURATION`, ele não sabe que trata-se de uma ação de reconfiguração dinâmica;

4. A execução do elemento gerenciado é interceptada no ponto da execução em que se extrai o evento da fila utilizando-se programação orientada a aspectos, transferindo o fluxo da execução para o componente de interceptação. Esse ponto de interceptação é definido pelos usuários do AGST nas subclasses de `AbstractInterceptor`. Uma vez interceptado, o elemento gerenciado só poderá processar outro evento da fila assim que o fluxo da execução for devolvido para esse elemento.

5. O componente de interceptação verifica que o referido evento possui a *tag* `OgstTags.DYNAMIC_RECONFIGURATION`, e conclui que trata-se de um evento de reconfiguração dinâmica;

6. O componente de interceptação extrai a ação adaptativa encapsulada no evento e a executa através dos atuadores definidos pela ação adaptativa usando reflexão computacional. Considerando o exemplo anterior, é nesse ponto que a substituição de uma heurística de escalonamento por outra é efetuada. O `GridScheduler` só poderá voltar a escalonar aplicações quando o fluxo da execução for devolvido para ele.

7. Após a execução da ação adaptativa, o componente de interceptação devolve o fluxo da execução para o elemento gerenciado, que agora pode processar o próximo evento da fila.

Uma característica fundamental do mecanismo de sincronização do AGST é a sua transparência. A abordagem proposta não requer qualquer instrumentação estática na implementação do elemento gerenciado para que o mesmo passe a tratar os eventos de reconfiguração. Ao invés disso, esse evento é interceptado e a reconfiguração é tratada por outro componente, claramente separando-se esta responsabilidade. Essa característica foi obtida a partir da utilização da programação orientada a aspectos (POA), conforme detalhado na Seção 2.3.

6.5 Limitações do AGST

Em se tratando de grades autonômicas, simular um ambiente capaz de se auto-monitorar, de tomar decisões com base no contexto do ambiente, e de se auto-reconfigurar dinamicamente, é uma tarefa desafiadora. Além da dificuldade de simular características típicas de grades de computadores, outras características específicas de sistemas autonômicos são difíceis de serem simuladas, tais como problemas de escalabilidade e o custo imposto pela execução de ações de reconfiguração. Isso impõe algumas limitações ao processo avaliação de abordagens autonômicas utilizando o AGST.

Uma limitação do AGST reside no fato de não simular problemas relacionados à escalabilidade do ambiente de grade. Em um ambiente de grade real é bastante provável que existam limitações quanto a quantidade de componentes autonômicos (sensores, monitores, analisadores, executores e atuadores) que podem ser adicionados ao sistema, já que a adição desses componentes e a frequência das operações realizadas por eles podem ocasionar uma perda de desempenho (*overhead*), ou até mesmo tornar indisponíveis os serviços oferecidos pelo *middleware* de grade. Já no ambiente de grade simulado pelo AGST, a quantidade de componentes autonômicos que podem ser executados não está limitada pela escalabilidade do sistema, mas apenas pela quantidade de memória e ao poder de processamento disponibilizados pela máquina na qual a simulação é executada. Sendo assim, não

é possível avaliar o impacto que a quantidade de componentes autonômicos tem sobre a qualidade dos serviços.

A execução de reconfigurações também apresenta limitações. Não foram desenvolvidos mecanismos que permitam mensurar o custo de comunicação entre os componentes autonômicos ou medir a quantidade de recursos consumidos durante a execução das reconfigurações. É sabido que mudanças de contexto podem ocorrer enquanto a reconfiguração está sendo executada. Durante essa fase do processo de adaptação, o serviço já estará totalmente ou parcialmente suspenso. A principal decisão a tomar nesta situação é: completar a reconfiguração em curso, ou atrasar, ou cancelar a reconfiguração e considerar a nova mudança [37]. No AGST, a execução de uma ação de reconfiguração não pode ser interrompida, portanto, ainda que em um ambiente de grade autonômica real essa suspensão seja possível e/ou desejável, no ambiente de grade simulado pelo AGST não existe tal possibilidade.

Uma vez que o ambiente de grade está sujeito a falhas, é provável que no ambiente real ocorram interrupções no funcionamento de monitores, analisadores e executores, em face da ocorrência de falhas de hardware e/ou software. Durante as simulações com o AGST assume-se que os componentes autonômicos jamais falham. Essa limitação dificulta a avaliação, por exemplo, de abordagens autonômicas tolerantes a falhas na execução dos componentes autonômicos do *middleware* de grade.

O mecanismo de tomada de decisões baseado em regras do tipo *evento-condição-ação* AGST não oferece suporte à detecção da existência de conflitos entre as regras. Sendo assim, é possível que regras conflitantes sejam executadas no ambiente de grade. Isso também constitui uma limitação do AGST.

6.6 Conclusões

Neste capítulo foram abordados diversos aspectos do simulador AGST. Foram mostradas as motivações e os requisitos para o seu desenvolvimento e descreveu-se também a sua arquitetura e suas principais funcionalidades.

Foi visto que o desenvolvimento do AGST foi motivado pela necessidade de se avaliar abordagens autonômicas para grades de computadores baseadas na arquitetura MAPE-K e a escassez de simuladores com suporte necessário para a

simulação de ciclos autonômicos em grades. O AGST é um simulador de eventos discretos orientado a objetos baseado nos simuladores OGST, GridSim, e SimJava2.

Foram apresentadas as camadas que compõem a arquitetura do AGST, descrevendo as funcionalidades providas por cada uma delas. Vimos que ao longo do desenvolvimento do AGST novas funcionalidades foram implementadas na camada provida pelo OGST, tais como o suporte à geração de falhas a partir de arquivos de traces. Essas implementações foram necessárias a fim de possibilitar a modelagem dos cenários que gostaríamos de simular usando o AGST. No entanto, a principal contribuição do AGST para a avaliação de abordagens autonômicas para grades de computadores está no conjunto de funcionalidades providas pela última camada: o MAPE-K Simulation Framework.

Foi mostrado que o MAPE-K Simulation Framework provê diversas funcionalidades para a modelagem e simulação de gerenciamento de ciclos, monitoramento de recursos, análise e planejamento, controle e execução de ações de reconfiguração dinâmica. Esta última funcionalidade é apoiada pelo suporte à execução de adaptações paramétricas e estruturais, transferência de estado e sincronização entre a execução das ações de reconfiguração e a execução funcional dos elementos gerenciados. Essas funcionalidades foram descritas em detalhes, de forma que foi possível perceber a interação entre os componentes que as implementam. Por fim, foi feita uma reflexão a respeito das limitações no uso do simulador AGST.

7 Implementação do AGST

Este capítulo mostra detalhes da implementação do AGST. Nesse capítulo são apresentadas as principais bibliotecas utilizadas para implementar as funcionalidades do MAPE-K Simulation Framework. São apresentadas ainda as principais classes e métodos que devem ser utilizadas no processo de modelagem e simulação de abordagens autonômicas para grades de computadores.

7.1 Bibliotecas utilizadas na implementação do AGST

Como visto na seção 6.4, a implementação das funcionalidades do MAPE-K Simulation Framework é baseada em duas abordagens usualmente empregadas para o desenvolvimento de sistemas autonômicos: a reflexão computacional e a programação orientada a aspectos. Os conceitos fundamentais sobre essas abordagens foram apresentados na seção 2.3.

Para o desenvolvimento do MAPE-K Simulation Framework utilizamos duas implementações dessas abordagens em Java: a API (*Application Programming Interface*) *JavaReflection* [25], que implementa conceitos da reflexão computacional, e o *framework AspectJ* [51], que implementa conceitos da POA. Portanto, antes de detalharmos as principais classes e métodos do MAPE-K Simulation Framework, descreveremos sucintamente os recursos providos por essas bibliotecas de software utilizados no desenvolvimento do AGST.

7.1.1 *Java Reflection*

Conforme visto na seção 2.3, a reflexão computacional é a habilidade de um software observar ou até mesmo modificar a sua estrutura ou comportamento (intercessão) dinamicamente. Em uma visão mais simples, a reflexão computacional é um mecanismo para descobrir e modificar dados a respeito de um programa em tempo de execução. Em java, a reflexão computacional é implementada pela API *Java Reflection*, que consiste de classes nos pacotes `java.lang` e `java.lang.reflect`.

A API *Java Reflection* implementa diversos conceitos da reflexão computacional, entre os quais destacamos:

- **Metadados**: representação que um objeto contém de si mesmo;
- **Meta-objeto**: objeto que contém a representação da estrutura interna de um outro objeto;
- **Introspecção**: capacidade de examinar a estrutura interna dos objetos;
- **Interseção**: capacidade de alterar a estrutura e comportamento dos objetos.

A API *Java Reflection* provê a introspecção através de meta-objetos para muitas construções importantes de linguagem, incluindo, mas não limitado a: classes, métodos, campos, interfaces e modificadores de acesso. Por exemplo, as meta-classes `Class` e `Method` são utilizadas para acessar respectivamente informações sobre classes e métodos de programas em tempo de execução. Esses meta-objetos também fornecem uma interface para mudar ou adaptar a estrutura e comportamento dos objetos. Por exemplo, um meta-objeto da classe `Method` pode ser usado para invocar um determinado método a partir de sua assinatura. Da mesma forma, um meta-objeto da classe `Field` expõe os atributos de um campo, tais como nomes e modificadores de acesso, permitindo consultar e modificar os valores desses atributos.

Entre as variadas operações reflexivas que a API *Java Reflection* pode executar, estão:

- Determinar a classe de um objeto;
- Carregar dinamicamente uma classe;
- Obter informações sobre os modificadores, campos, métodos, construtores e superclasses de uma classe;
- Alterar os modificadores de acesso à variáveis, métodos e construtores;
- Descobrir quais constantes e declarações de métodos pertencem a uma interface;
- Criar uma instância de uma classe cujo nome não se sabe até o tempo de execução;

- Obter e definir o valor do campo de um objeto;
- Invocar um método em um objeto dinamicamente;
- Interceptar chamadas de métodos.

A API *Java Reflection* é geralmente usada para criar ferramentas de desenvolvimento tais como *debuggers*, *class browsers* e construtores de GUI (*Graphical User Interface*). Geralmente, nestas ferramentas precisamos interagir com classes, objetos, métodos e campos não conhecidos em tempo de compilação. Assim, a aplicação deve, dinamicamente, encontrar e acessar estes itens.

Devido às suas potencialidades, essa API também tem sido empregada no desenvolvimento de sistemas autonômicos. No trabalho de Dawson et al. [20] é proposto um mecanismo que explora as capacidades reflexivas das linguagens de programação para monitorar sistemas com capacidades adaptativas. Nesse artigo é comentado como esse monitoramento pode ser implementado utilizando API *Java Reflection*. Em outro trabalho [21], além do monitoramento, explora-se outras potencialidades dessa API para efetuar mudanças comportamentais e estruturais em sistemas autonômicos.

Basicamente três funcionalidades do MAPE-K Simulation Framework são baseadas na API *Java Reflection*: I) monitoramento de propriedades de contexto; II) adaptação paramétrica e composicional; e III) transferência de estado. De maneira geral, todos esses mecanismos usam essa API para acessar e/ou modificar os atributos dos objetos que representam os elementos gerenciados. O mecanismo de transferência de estado, em particular, utiliza essa API também para descobrir se dois componentes (original e substituto) são compatíveis a partir de uma análise comparativa de suas hierarquias de classes. Essa análise envolve também descobrir os campos que ambos os componentes possuem em comum. Neste trabalho, essa API é utilizada nos seguintes casos:

- Para descobrir os campos declarados por um elemento gerenciado, utiliza-se o método `getDeclaredFields()` da meta-classe `Class`;
- Para descobrir o nome de um determinado campo, utiliza-se o método `getField(String fieldName)` da meta-classe `Field`;

- Para acessar o valor de instância de um campo específico do elemento gerenciado, utiliza-se o método `get(Object target)` da meta-classe `Field`, passando como parâmetro o próprio elemento gerenciado;

- Para alterar o valor de instância de um campo específico do elemento gerenciado, utiliza-se o método `set(Object target, Object value)` da meta-classe `Field`, passando como parâmetros o próprio elemento gerenciado e o valor a ser atribuído;

- Muitas vezes os modificadores de acesso aos campos são privados. Nesses casos, antes de acessar esses campos via reflexão, é necessário utilizar o método `setAccessible(Boolean flag)` da meta-classe `Field`, passando como parâmetro a *flag* `true`, que modifica o acesso ao campo de privado (`private`) para público (`public`);

- Para descobrir a classe de um dado objeto, utiliza-se o método `getClass()` da meta-classe `Object`. Já a descoberta da super classe do elemento gerenciado é feita a partir de uma chamada ao método `getSuperClass()` da meta-classe `Class`.

7.1.2 *AspectJ*

Conforme visto na seção 2.3, a programação orientada a aspectos é um paradigma de programação que permite encapsular interesses entrecortantes (que afetam diversas partes da aplicação) em módulos fisicamente separados do restante do código. Esses módulos são denominados aspectos.

A linguagem *AspectJ* é uma extensão orientada a aspectos da linguagem Java, desenvolvida inicialmente pela Xerox PARC em 1997 e posteriormente agregada ao projeto Eclipse da IBM em 2002. Como qualquer implementação POA, *AspectJ* consiste em duas partes: I) a especificação da linguagem, que define a gramática e semântica da linguagem; e II) a implementação da linguagem, que inclui combinadores denominados de *weavers*. Os *weavers* são utilizado para entrelaçar os aspectos com classes do sistema. Em *AspectJ*, a combinação entre aspectos e classes Java é baseada na manipulação de *byte-code*. O *byte-code* gerado pelo compilador AspectJ está em conformidade com a especificação do *byte-code* Java. Dessa forma, qualquer JVM

compatível com essa especificação pode executar as aplicações que utilizam esse *framework*. Alguns ambientes de desenvolvimento integrado (IDEs), tais como o Eclipse [1], oferecem suporte para simplificar a criação e depuração de aplicações com *AspectJ*.

Além dos elementos oferecidos pela programação orientada a objetos (classes, métodos, atributos, etc), diversas construções da programação orientada a aspectos são implementadas pelo *framework* AspectJ, tais como: aspectos, pontos de corte, pontos de junção, adendos e declarações inter-tipos. A seguir descreveremos sucintamente essas construções, utilizadas na implementação do AGST.

O aspecto (`Aspect`) e á a principal construção do *AspectJ*. Essa construção permite alterar a estrutura estática ou dinâmica das aplicações. A estrutura estática é alterada adicionando, por meio das declarações inter-tipos, membros (atributos, métodos ou construtores) a uma classe, modificando assim a hierarquia do sistema. Já a alteração em uma estrutura dinâmica de um programa ocorre em tempo de execução por meio dos pontos de junção (os quais são selecionados por pontos de corte) e através da adição de comportamentos (adendos) antes ou depois dos pontos de junção.

Utiliza-se a seguinte sintaxe para implementar um aspecto no *AspectJ*:

Código 7.1: Declaração de um aspecto no AspectJ

```
1 public aspect Aspect {
2
3 //pontcuts
4
5 //advices
6
7 //inter-type declarations
8
9 }
```

Os pontos de junção (*joinpoints*) são pontos na execução de um programa onde os aspectos serão aplicados. O *AspectJ* pode detectar e operar sobre diversos tipos de pontos de junção, tais como: chamada e execução de métodos e construtores; inicialização e pré-inicialização de objetos; referência a campos; e execução de tratamento de exceções.

[1] http://www.eclipse.org/aspectj/

Um aspecto no *AspectJ* pode definir pontos de corte (*pointcut*), que são aninhados através de operadores lógicos *e* (&&), *ou* (||), *não* (!). Os pontos de corte são responsáveis por selecionar os pontos de junção, isto é, eles detectam quais os ponto do programa que deverão ser interceptados pelos aspectos. É possível declarar um ponto de corte semelhante a uma classe em Java, podendo, da mesma maneira que atributos e métodos dessas classes, se especificar um quantificador de acesso aos pontos de corte. Estes podem ser públicos, privados ou final, mas não podem ser sobrecarregados. Eles também podem ser declarados abstratos, mas somente dentro de aspectos abstratos, e ainda podem ser nomeados ou anônimos. Para declarar um ponto de corte no *AspectJ* utiliza-se a seguinte sintaxe:

Código 7.2: Estabelecendo um ponto de corte no AspectJ

```
1 pointcut <name> (args): <body>
```

No AGST, os pontos de corte são empregados na interceptação de eventos de reconfiguração enviados aos elementos gerenciados. O ponto de corte seleciona o ponto de junção no qual o evento é recebido pelo elemento gerenciado, transferindo para o aspecto o fluxo da execução. Para definição do ponto de corte utiliza-se construções do *AspectJ*. Para ilustrar, segue um exemplo de como interceptar chamadas ao método `sim_get_next(Sim_event event)` no componente `GridScheduler` utilizando pontos de corte. Para isso, utiliza-se o designador `call`. Esse designador permite a interceptação do método a partir de sua assinatura. Já o designador `within` é usado para definir em qual classe essa interceptação deverá acontecer. O operador lógico `&&` é utilizado para restringir a interceptação somente a objetos da classe `GridScheduler`, conforme demonstra o código a seguir:

Código 7.3: Interceptando um método com AspectJ

```
1 pointcut joinpoint (): call(* sim_get_next(Sim_event)) && within (GridScheduler);
```

Em *AspectJ*, elementos *wildcards* são utilizados para possibilitar que em especificações de assinatura (*signature*) sejam definidas o número de caracteres (*) e o número dos argumentos (..).

Os adendos (*advices*) são construções que permitem a inserção de código a ser executado em um ponto de junção que está sendo referenciado pelo ponto de corte.

Existem três tipos de adendos: *before* (antes do ponto de junção), *around* (antes e depois do ponto de junção) e *after* (depois do ponto de junção). O adendo pode modificar a execução do código no ponto de junção, pode substituir, ou passar por ele. Utilizando adendos, pode-se, por exemplo, obter o *logging* das mensagens antes de executar o código de determinados pontos de junção que estão espalhados em diferentes módulos. O corpo de um adendo é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um ponto de junção é alcançado.

A seguir temos o exemplo de um aspecto `Interceptor` que utiliza adendo *after* (linhas 7 a 9) para inserir código após o ponto de junção definido no exemplo anterior (linhas 4 e 5).

Código 7.4: Utilizando adendos do AspectJ

```
1   public aspect Interceptor {
2
3     //selecionando o ponto de juncao
4     pointcut joinpoint (): call(* sim_get_next(Sim_event))
5                            && within (GridScheduler);
6
7      after (): joinpoint () {
8      //inserir codigo a ser executado depois do ponto de juncao
9      };
10 }
```

Enquanto que as construções pontos de junção, ponto de corte, e adendos são utilizados para modificar a estrutura dinâmica de um programa através da adição de comportamentos, as declarações inter-tipos são utilizadas para alterar a estrutura estática de um programa. Essas declarações permitem, por exemplo, adicionar novos membros a uma classe ou alterar a sua hierarquia. A modificação na hierarquia das classes é feita por meio da declaração inter-tipos `declare parents`. Essa declaração permite alterar a hierarquia e indicar que alguma classe herda de outra classe ou implementa certas interfaces. A seguir é mostrada a sintaxe da declaração `declare parents`, para os dois casos (herança e implementação de interface).

Código 7.5: Sintaxe da declaração declare parents

```
1 declare parents : [PadraoDeTipo] implements [ListaDeInterfaces];
2 declare parents : [PadraoDeTipo] extends [Classe/ListaDeInterfaces];
```

Além da sintaxe tradicional, apresentada nesta seção, o *AspectJ* oferece também uma sintaxe alternativa com base na facilidade das anotações Java (`@Annotations`), essa sintaxe é denominada `@AspectJ`. As anotações Java usadas pela sintaxe `@AspectJ` são parte do arquivo jar *aspectjrt.jar*. É preciso colocar este jar no `classpath` da aplicação quando aspectos forem compilados utilizando essa sintaxe para mapear aspectos para classes Java [51].

7.2 Implementação do MAPE-K Simulation Framework

Nesta seção descrevemos as principais classes do MAPE-K Simulation Framewok, comentando aspectos centrais de suas implementações e seus principais métodos. A estrutura desta seção é semelhante à utilizada na seção 6.4, na qual foram explicadas as respectivas funcionalidades implementadas pelas classes aqui descritas.

7.2.1 Gerenciamento de Ciclos Autonômicos

O diagrama da figura 7.1 contém as classes definidas no AGST para a modelagem de ciclos autonômicos em grades computacionais.

Todos os componentes do AGST estendem direta (tais como o gerente autonômico) ou indiretamente (tais como: monitores, analisadores e executores) da classe `AbstractComponent`. Essa classe estende da classe `GridSim` a fim de possibilitar que todas as suas subclasses possam se comunicar através de eventos discretos (`Sim_event`). Todos os componentes que estendem de `AbstractComponent` devem receber obrigatoriamente um nome (`String`) no ato de sua instanciação, e são automaticamente registrados junto ao `ConfigurationManagmentDataBase` utilizando esse identificador.

Um gerente autonômico é composto por alguns componentes: monitores (`AbstractMonitor`), analisadores (`AbstractAnalyzer`) e executores (`AbstractExecutor`). Cada gerente autonômico mantém um registro dos seus próprios componentes, afim de que seja possível descobrir quais componentes encontram-se em um mesmo ciclo autonômico, e quais pertencem a ciclos autonômicos distintos. Cada componente do gerente autonômico implementa respectivamente uma fase do ciclo MAPE-K. Todos eles estendem da classe `AbstractMAPEKEntity`, e

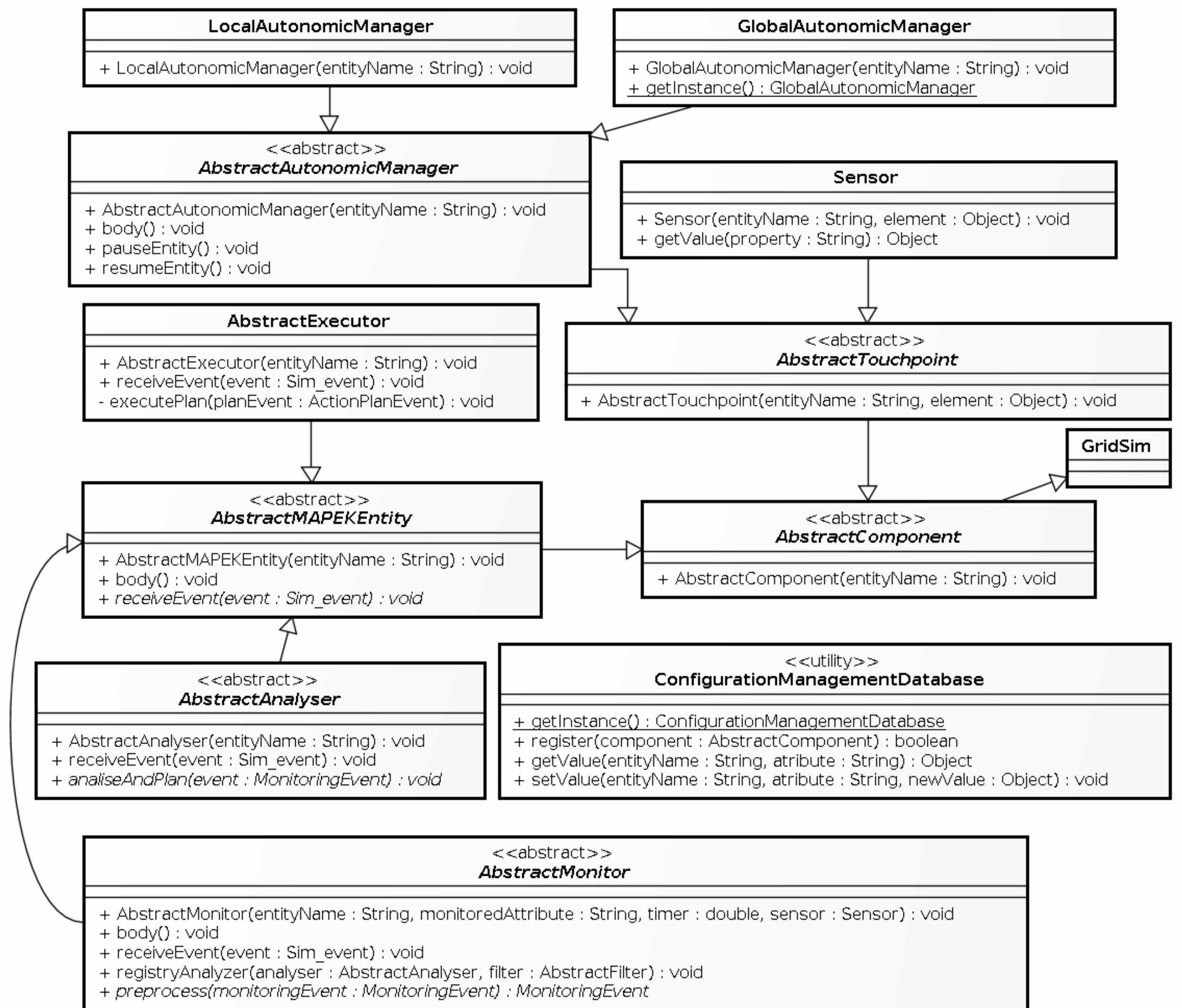


Figura 7.1: Classes que implementam o gerenciamento de ciclos autonômicos.

implementam obrigatoriamente o método `receiveEvent( )`. Esse método permite definir o comportamento de cada componente do gerente autonômico diante da recepção de um evento do GridSim. Por exemplo, um analisador só deve implementar a lógica que permite tratar eventos do tipo `OgstTags.MONITORING_EVENT`.

Os pontos de contato utilizados pelos gerentes autonômicos, sensores (`Sensor`) e atuadores (`Effector`) estendem da classe `AbstractTouchpoint`, que proveem funcionalidades reflexivas para acessar propriedades dos elementos gerenciados, e para efetuar adaptações paramétricas e estruturais dos elementos gerenciados.

Os gerentes autonômicos locais são implementados pela classe `LocalAutonomicManager`, enquanto que o gerente autonômicos global é

implementados pela classe `GlobalAutonomicManager`. Uma vez que só pode haver uma instância do gerente global, a classe `GlobalAutonomicManager` implementa o padrão de projeto *singleton* [31]. Esse padrão garante que haverá uma única instância dessa classe ao longo da simulação. `LocalAutonomicManager` e `GlobalAutonomicManager` estendem da classe `AbstractAutonomicManager`.

Os gerentes autonômicos comunicam-se entre si através do envio e recebimento de eventos do GridSim, e portanto, esses componentes implementam seu ciclo de vida no método `body()`. Todos os componentes precisam receber obrigatoriamente um nome (`String`) no ato de sua instanciação. Todos os componentes do AGST utilizam o método `send (String dest, double delay, int tag, Object data)` para enviar eventos discretos. Esse método recebe quatro parâmetros: o identificador do destinatário; o retardo (*delay*) de propagação do evento, que corresponde ao tempo simulado de entrega do evento; a *tag* de identificação do tipo de evento; e um dado opcional, que pode ser qualquer objeto. Do lado do destinatário utiliza-se o método `sim_get_next(Sim_event)` para receber o evento. Nas simulações realizadas, o retardo de propagação utilizado por padrão para simular o atraso no envio das mensagens é sempre zero, pois no desenvolvimento do AGST não focamos na modelagem do tempo necessário para este tipo propagação, conforme analisado na seção 6.5.

A decisão de parar ou continuar um ciclo autonômico local é tomada pelo analisador do `GlobalAutonomicManager`. Utilizando o método `send (String dest, double delay, int tag, Object data)`, esse analisador envia para os ciclos locais os eventos de interromper, cuja *tag* é `OgstTags.PAUSE`, e de continuar. cuja *tag* é `OgstTags.CONTINUE`.

7.2.2 Monitoramento

O diagrama de classes da figura 7.2 contém as classes modeladas no AGST para implementar a função de monitoramento.

O sensores são implementados pela classe `Sensor`. Essa classe estende de `AbstractTouchpoint` e recebe dois argumentos em seu construtor: o nome do sensor (`String`), e uma instância do elemento gerenciado (`Object`

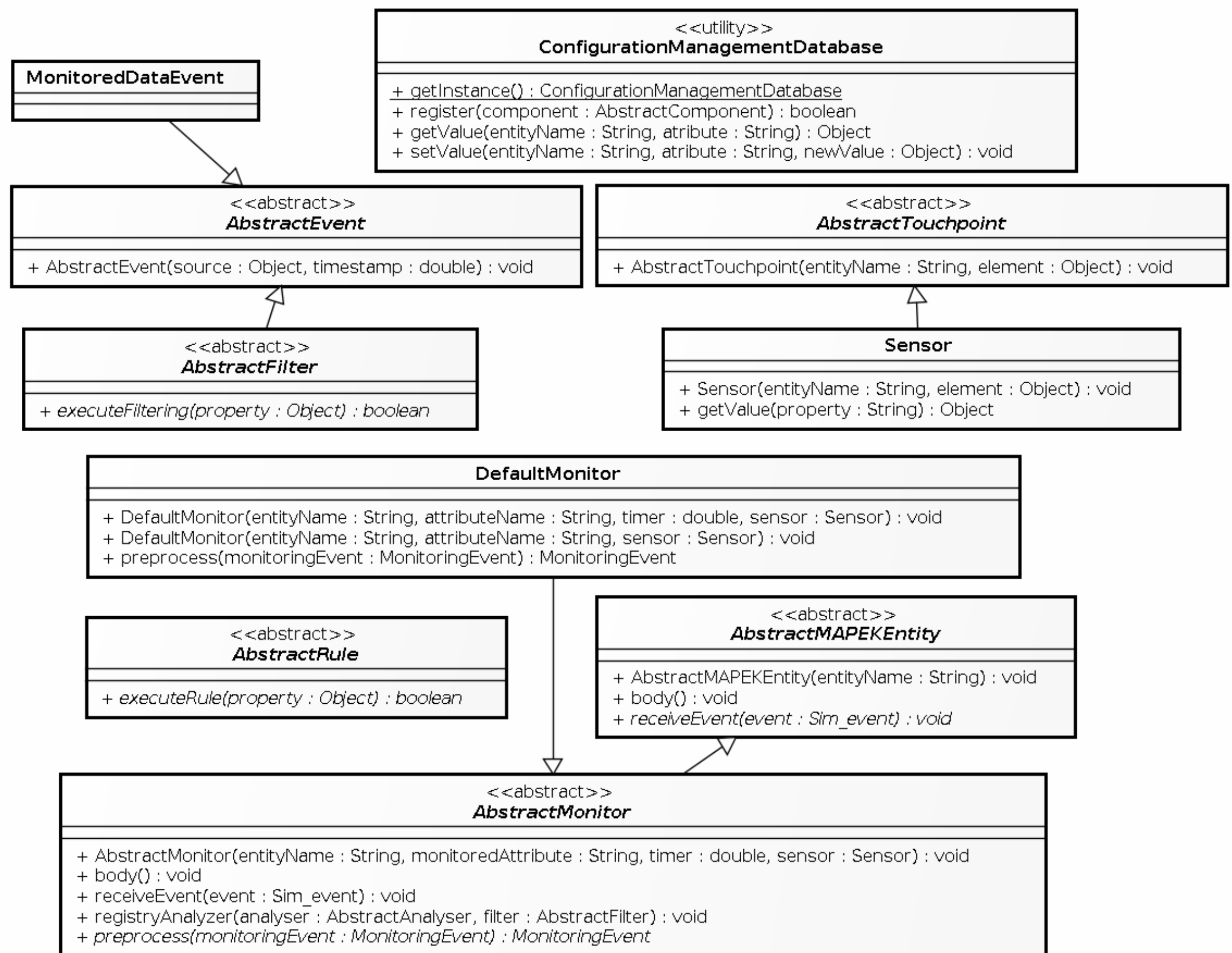


Figura 7.2: Classes utilizadas pela função de monitoramento.

`managedElement`). Essa classe utiliza as meta-classes `Class` e `Field` da API *JavaReflection* para acessar as propriedades de contexto do elemento gerenciado (conforme ilustrado na seção 7.1). Quando um monitor precisa que o sensor colete uma propriedade de contexto, esse monitor invoca o método `getValue(String property)` da classe `Sensor`. Esse método recebe como parâmetro o nome da propriedade a ser coletada, e retorna o valor (`Object`) dessa propriedade.

Os monitores do AGST são implementados partir de extensões da classe `AbstractMonitor`, tal como o monitor padrão que não realiza pré-processamento e é implementado pela classe `DefaultMonitor`. A classe `AbstractMonitor` recebe três parâmetros em seu construtor: o identificador do monitor (`String`), o tempo entre os eventos de monitoramento (`Double`), e o sensor (`Sensor`) ao qual o monitor solicitará o valor de uma propriedade correspondente a algum tipo de informação de contexto. Caso o tempo entre os eventos de mapeamento fornecido

seja negativo, o monitor enviará informações de contexto da propriedade monitorada apenas quando for requisitado por algum analisador (monitoramento sob demanda). Os analisadores se registram junto aos monitores através de uma chamada ao método `registryAnalyzer(AbstractAnalyser analyser, AbstractFilter filter)` da classe `AbstractMonitor`, que recebe como parâmetros o analisador e o filtro associado a ele.

Nos monitores, o pré-processamento das propriedades de contexto é implementado no método `preprocess(Object property)`. Esse método recebe como parâmetro uma propriedade de contexto. Os monitores utilizam regras implementadas a partir de extensões da classe `AbstractRule` para determinar a ocorrência de variações significativas nas propriedades de contexto. Cada regra deve implementar o método `executeRule(Object property)`, que recebe como parâmetro a propriedade que será analisada, e deve retornar `true` em caso de variação significativa na mesma, ou `false` em caso de variação irrelevante. Esse método é invocado pelo monitor logo após ele obter o valor da propriedade a partir do sensor.

Os filtros utilizados pelos monitores para selecionar os analisador que irão receber uma dada notificação de mudança de contexto são implementados a partir de extensões da classe `AbstractFilter`. Os critérios de filtragem de cada analisador são implementados através do método `executeFiltering (Object property)`, que recebe como parâmetro o valor da propriedade e retorna `true`, caso o analisador associado ao filtro esteja interessado no valor que essa propriedade apresenta. Do contrário, deve-se retornar `false`.

As informações enviadas ao analisador são encapsuladas em um objeto da classe `MonitoredDataEvent`, que estende da classe `AbstractEvent`. Para enviar a notificação ao analisador, o monitor invoca o método `send (String dest, double delay, int tag, Object data)`, passando como parâmetros o identificador do analisador, o atraso de propagação padrão (zero), a *tag* `OgstTags.MONITORING_EVENT` (indicando que se trata de um evento de monitoramento), e o objeto que encapsula as informações de contexto (`MonitoredDataEvent`).

7.2.3 Análise e Planejamento

O diagrama da figura 7.3 contém as classes modeladas no AGST para implementar a função de análise e planejamento.

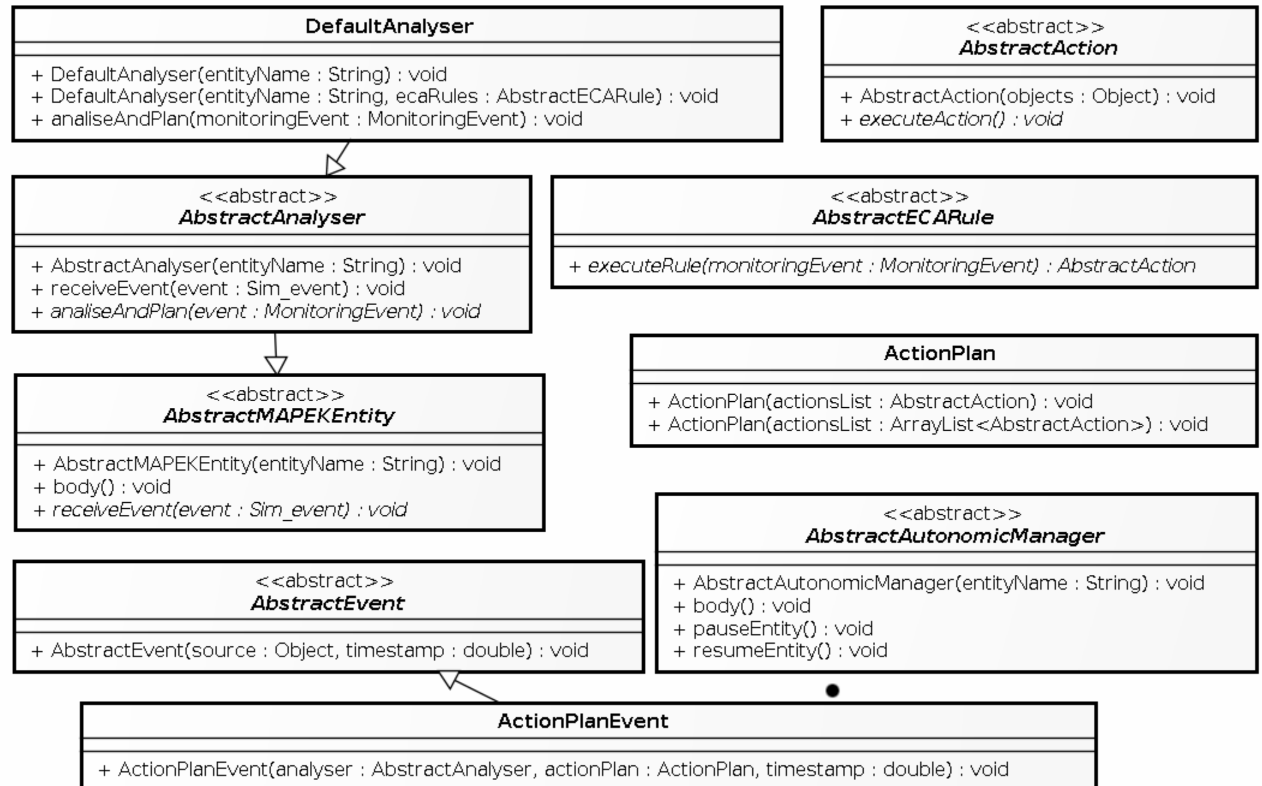


Figura 7.3: Classes utilizadas pela função de análise e planejamento.

O analisadores do AGST são implementados através de extensões da classe `AbstractAnalyzer`. Os analisadores recebem os eventos enviados pelos monitores através de uma chamada ao método `receiveEvent()`. O código responsável pela tomada de decisões deve ser implementado no método `analiseAndPlan()`.

A classe `DefaultAnalyzer`, que estende de `AbstractAnalyzer`, permite ao usuário implementar a lógica de tomada de decisão utilizando regras do tipo *evento-condição-ação* (*ECA rules*). Dessa forma, não há necessidade do usuário implementar diretamente essa lógica no método `analiseAndPlan()`. Para implementar as regras, estende-se a classe `AbstractECARule`. O único método a ser implementado pelas subclasses de `AbstractECARule` é o `executeRule(MonitoredDataEvent event)`, que recebe como parâmetro um evento contendo uma propriedade monitorada e o instante da medição, e retorna a ação (`AbstractAction` a ser realizada. As ações retornadas pelas regras irão compor

o plano de ações (`ActionPlan`) elaborado pelo analisador. As regras utilizadas pelo `DefaultAnalyzer` podem ser associadas a ele de duas formas: através do construtor dessa classe, passando uma lista das regras a serem executadas; ou através do método `addEcaRule (AbstractECARule rule)`, passando a regra como parâmetro para esse método.

Para gerar um plano de ações, os analisadores utilizam o método `generateActionPlanEvent(AbstractAction...actions)`, que recebe como parâmetros um vetor de ações cujo tamanho é indefinito, e retorna um objeto do tipo `ActionPlan` contendo todas ações definidas. De forma semelhante ao adotado por outras entidades, os analisadores encapsulam o plano de ações em um objeto do tipo `ActionPlanEvent`, e o envia para os executores através de um evento do GridSim. O método utilizado para enviar o evento é novamente o `send (String dest, double delay, int tag, Object data)`. Dessa vez, os parâmetros passados a esse método são o identificador do executor (`String`), uma *tag* do tipo `OgstTags.PLAN_EVENT` (indicando que se trata de um evento contendo um plano de ações), e o plano de ações encapsulado (`ActionPlanEvent`).

7.2.4 Controle e Execução de Reconfigurações

O diagrama da figura 7.4 contém as classes utilizadas pelo AGST a implementar a função de controle e execução de reconfigurações dinâmicas.

Executores são implementados a partir de extensões da classe `AbstractExecutor`. O plano de ações (`ActionPlan`) é recebido pelos executores a partir de um chamada ao método `receiveEvent()`. O *DefaultExecutor* é um executor padrão que estende de `AbstractExecutor`. Esse tipo de executor extrai as ações contidas no plano (`ActionPlan`), e as executa na mesma ordem em que foram adicionadas na lista de ações (`ArrayList<AbstractAction>`) do plano.

A ações de reconfiguração a serem executadas são implementadas a partir de extensões da classe `AbstractAction`. Todas as subclasses de `AbstractAction` devem obrigatoriamente implementar um único método: o `executeAction()`. É no escopo desse método que são instanciados os atuadores apropriados para efetuar as ações adaptativas sobre os elementos gerenciados.

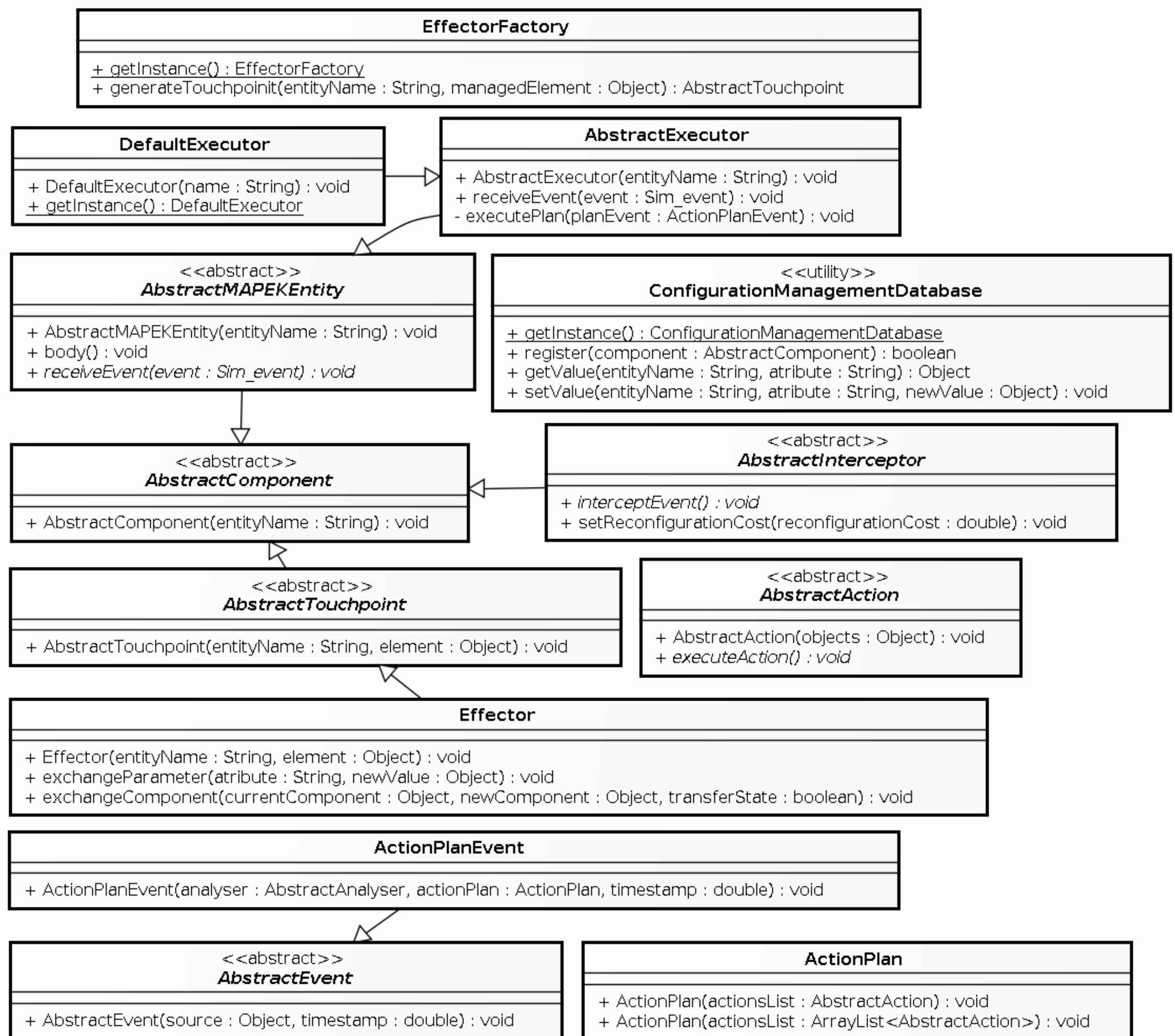


Figura 7.4: Classes utilizadas pela função de controle e execução de reconfigurações.

Os atuadores são implementados pela classe `Effector`, que estende da classe `AbstractTouchpoint`. A classe `EffectorFactory` é a responsável pela instanciação dos atuadores. Essa classe garante que nenhum elemento gerenciado terá mais do que um atuador acoplado. A instanciação dos atuadores é feita através do método `generate(String effectorName, Object managedElement)`. Esse método recebe como parâmetros o nome do atuador e o elemento gerenciado ao qual será acoplado. Caso o usuário tente instanciar mais de um atuador para cada elemento gerenciado é lançada uma exceção do tipo `DuplicatedEffectorException`.

7.2.5 Adaptações Dinâmicas e Transferência de Estado

A classe `Effector` utiliza as meta-classes `Class` e `Field` da API *JavaReflection* não apenas para acessar valores dos campos dos elementos gerenciados (como no caso dos sensores), mas também para modifica-los, bem como para percorrer a hierarquia de classes e descobrir quais campos podem ser transferidos de um componente para o outro (conforme ilustrado na seção 7.1). Essas capacidades são empregadas na implementação dos mecanismos de adaptação dinâmica e também na implementação do mecanismo de transferência de estado.

O método implementado pela classe `Effector` para efetuar adaptações paramétricas é o `exchangeParameter(String property, Object value)`. Esse método recebe como argumentos o nome da propriedade a ser modificada e novo valor a ser atribuído a essa propriedade. Já o método `exchangeComponent(Object substituted, Object substitute, boolean transferState)`, implementado por essa classe, é responsável por efetuar adaptações composicionais. Esse método recebe como parâmetros: o componente original, o componente substituto, e uma *flag*. Sinalizando essa *flag* como `true`, o usuário informa que os estados do componente original devem ser transferidos para o componente substituto. A sinalização `false` indica que o processo de substituição não requer transferência de estado.

7.2.6 Sincronização

Conforme visto na seção 6.4.7, o mecanismo de sincronização do AGST consiste em interceptar as chamadas ao método `sim_get_next(Sim_event event)` no elemento gerenciado. Essa interceptação interrompe temporariamente a execução do elemento gerenciado e desvia o fluxo da execução para o componente interceptador. Esse interceptador captura o evento de reconfiguração enviado ao elemento gerenciado, dado que o mesmo não possui o código para tratá-lo. O evento de reconfiguração (`OgstTags.DYNAMIC_RECONFIGURATION`) encapsula a ação de reconfiguração (`AbstractAction`) a ser executada. Essa ação é extraída pelo interceptador, que procede então a execução da mesma. Ao final da reconfiguração, o interceptador

devolve o fluxo da execução para o elemento gerenciado, que a partir de então pode continuar a executar seu código funcional.

A sincronização entre a execução de reconfigurações e a execução funcional dos elementos gerenciados é provida por aspectos, que são implementados a partir de extensões do aspecto `AbstractInterceptor`. Na implementação do `AbstractInterceptor`, ilustrada pelo código 7.6, ao invés de utilizarmos a sintaxe normal do *AspectJ* apresentada na seção 7.1.1, optamos pela utilização da sintaxe alternativa *@AspectJ*, baseada em anotações (*@Annotations*) Java. Adotamos essa sintaxe porque acreditamos que para o usuário do AGST seria mais cômodo trabalhar com classes Java, do que força-los a aprender a sintaxe tradicional do *AspectJ*. Portanto, o aspecto `AbstractInterceptor` foi implementado como uma classe Java abstrata. No entanto, uma característica da utilização de aspectos é que eles são combinados dinamicamente com as classes da aplicação, não requerendo sua instanciação de forma explícita pelo programador.

Código 7.6: Classe AbstractInterceptor

```
1  public abstract class AbstractInterceptor extends AbstractComponent{
2
3      public AbstractInterceptor() throws Exception {
4          super(AbstractInterceptor.class.getName());
5      }
6      @Pointcut ()//ponto de corte abstrato
7      public abstract void interceptEvent();
8      //adendo para executar o metodo depois ponto de juncao
9      @After("interceptEvent()")
10     public void afterReceiveEvent( JoinPoint joinPoint ) {
11         //obtendo o evento de enviao ao elemento gerenciado
12         Sim_event event = (Sim_event) joinPoint.getArgs()[0];
13         int op = event.get_tag();
14         switch ( op ) {
15         case OgstTags.DYNAMIC_RECONFIGURATION:
16             //obtendo a acao de reconfiguracao dinamica
17             AbstractAction action = (AbstractAction) event.get_data();
18             //executando a acao
19             executeAction(action);
20             break;
21         }
22     }
23
24     private void executeAction(AbstractAction action) {
25         action.executeAction();
26     }
27 }
```

A classe `AbstractInterceptor` define um ponto de corte abstrato. Isso significa que a seleção dos pontos de junção a serem interceptados durante a sincronização tem que ser implementada nas subclasses de `AbstractInterceptor`. Por tratar-se de um aspecto implementado com a sintaxe *@AspectJ*, a definição do ponto de corte é feita declarando um método abstrato precedido da anotação `@Pointcut`. No caso da classe `AbstractInterceptor`, o método abstrato é o `interceptEvent()`. O que determina o ponto de corte como abstrato é o modificador `abstract` desse método

A classe `AbstractInterceptor` implementa o método `afterReceiveEvent(Joinpoint joinpoint)`. Esse método utiliza a anotação `@After(''interceptEvent'')` pelo fato de que esse método precisa ser invocado automaticamente pelo `AspectJ` sempre após o ponto de junção. Esse método recebe como parâmetro um meta-objeto do tipo `Joinpoint`, que fornece meta-informações sobre ponto de junção. Através do método `getArgs()` da classe `Joinpoint` é possível acessar via reflexão computacional os argumentos recebidos por um método. A partir do método `getArgs()`, a classe `AbstractInterceptor` consegue obter o evento discreto (`Sim_event`) enviado ao elemento gerenciado. A partir do método `get_tag()` da classe `Sim_event` obtém-se a *tag* identificadora do evento. Caso essa *tag* seja equivalente à constante `OgstTags.DYNAMIC_RECONFIGURATION` (indicando que se trata de um evento de reconfiguração), invoca-se o método `executeAction(AbstractAction action)`, que recebe como parâmetro a ação de reconfiguração a ser executada.

7.3 Considerações sobre o uso de POA e Reflexão na Sincronização

Assim como a programação orientada a aspectos, a reflexão computacional também provê mecanismos para a interceptação e instrumentação dinâmica de código. Porém, dependendo da API de reflexão utilizada, existe a necessidade de declarar uma cláusula especial para instanciação de objetos da classe base, introduzindo sobrecarga e custos de manutenção.

Geralmente, as APIs de reflexão computacional, tais como *Java Reflection*, utilizam *Proxies* para prover interceptação. *Proxy* é uma classe especial que permite

interceptar um conjunto de métodos baseado em uma interface. Nesse caso, é preciso implementar fábricas (*factories*) de objetos que retornem esses *proxies* como se fossem instâncias da classe base. Isso faz com que os pontos do sistema que deveriam instanciar os objetos da classe base precisem ser modificados para que passem a invocar as fábricas de objetos para obter uma instância do *proxy*, ao invés de uma instância da classe base.

Apesar de termos utilizado a reflexão computacional para acessar o contexto dos pontos de junção, a interceptação desse ponto é feita via POA. Se a interceptação fosse feita via reflexão, isso implicaria adição as cláusulas especiais de instanciação dos elementos gerenciados ao longo do código do simulador, gerando custos de alteração no código-fonte da aplicação. Já o uso dos aspetos dispensa a adição dessas clausulas. Além disso, a reflexão computacional em Java permite interceptar apenas métodos, enquanto que o pontos de corte da programação orientada a aspectos permitem interceptar uma variedade significativamente maior de pontos de junção.

7.4 Conclusões

Neste capítulo foram abordados os principais aspectos da implementação do AGST. Foram abordadas as principais bibliotecas, classes e métodos utilizados em seu desenvolvimento. Foram descritos ainda a implementação dos principais componentes do MAPE-K Simulation Framework, detalhando suas principais classes e métodos, explicando a interação entre elas.

Foi mostrado que a API *Java Reflection* implementa diversos conceitos da reflexão computacional, tais como metadados, meta-objetos, introspecção, e interseção. Já o Framework *AspectJ* estende a sintaxe Java e implementa diversas construções da programação orientada a aspectos, tais como aspectos, pontos de corte, pontos de junção, adendos e declarações inter-tipos. Foram mostrados alguns exemplos de aplicações de cada uma dessas bibliotecas e falou-se das principais funcionalidades que elas proveem e como algumas delas foram empregadas na implementação do MAPE-K Simulation Framework.

Além disso, foi descrito como a reflexão computacional foi empregada na implementação dos mecanismos transparentes de monitoramento, adaptação paramétrica e composicional, e de transferência de estado. O emprego da reflexão

computacional nesses mecanismos evita a instrumentação estática das classes dos elementos gerenciados.

Foram descritos também como a interceptação e a instrumentação dinâmica de código via POA foi utilizada na implementação do mecanismo de sincronização entre a execução de reconfigurações e a execução funcional dos elementos gerenciados. O mecanismo de instrumentação dinâmica de código permite que módulos de programas sejam reescritos em tempo de execução, possibilitando a introdução de novas funcionalidades após a disponibilização da aplicação.

Por fim, foram descritas algumas considerações sobre o uso da reflexão computacional e da programação orientada a aspectos na implementação do mecanismo de sincronização do AGST, destacando que a facilidade para adicionar e remover comportamentos adaptativos através de aspectos compreende a principal vantagem do uso da programação orientada a aspectos sobre reflexão computacional para a implementação da interceptação e de instrumentação do código dos elementos gerenciados.

8 Estudos de Caso

Este capítulo descreve dois estudos de caso de utilização do AGST no processo de avaliação de abordagens voltadas para *Desktop Grids*. O primeiro trata de uma abordagem autonômica para tolerância a falhas na execução de aplicações. O segundo trata de mecanismo de gerenciamento de aplicações que possuam restrições de tempo de execução definidas pelos usuários no ato de suas submissões.

O objetivo deste capítulo é mostrar como as funcionalidades providas pelos componentes do AGST foram utilizadas para modelar e simular essas abordagens. A descrição completa das referidas abordagens, suas implementações detalhadas, e a totalidade dos resultados de suas avaliações por meio de simulações utilizando o AGST podem ser encontradas nas publicações de Viana [77, 78] e Martins [58], respectivamente.

8.1 Avaliação de uma Abordagem Autonômica para Tolerância a Falhas

Um ponto chave em grades computacionais é o suporte à tolerância a falhas na execução de aplicações na grade, pois os usuários esperam que suas aplicações submetidas à grade sejam concluídas com sucesso. Os usuários também esperam receber os resultados das computações submetidas o mais breve possível. Nessa linha, foi proposta por Viana uma abordagem autonômica para tolerância a falhas de aplicações em *Desktop Grids* que busca alcançar esses dois objetivos: maximizar a taxa de sucesso das aplicações e minimizar o seu tempo de conclusão.

As técnicas de tolerância a falhas mais utilizadas em grades computacionais são: reinicio, replicação e *checkpointing*. Segundo Viana, as abordagens de replicação e *checkpointing* apresentam vantagens em diferentes condições do ambiente de execução da grade. Entretanto o ambiente de grades é altamente dinâmico, sofrendo frequentes mudanças no decorrer de sua execução. Se essas técnicas de tolerância a falhas forem utilizadas de modo estático (fixo), ou seja, não se adequarem dinamicamente

às mudanças sofridas pelo ambiente, elas podem causar o desequilíbrio do sistema, degradando seu desempenho e eficiência.

Levando-se em consideração o exposto, que evidencia as desvantagens de se adotar uma abordagem estática para prover tolerância a falhas na execução de aplicações em *Desktop Grids*, Viana propôs uma abordagem autonômica que utiliza as vantagens do uso das técnicas de replicação e *checkpointing* e que tem como base dois níveis de adaptação. No primeiro nível, são levadas em consideração as variações do ambiente de execução para as quais os ajustes em parâmetros usados pela técnica de tolerância a falhas em uso sejam suficientes para manter o sistema em equilíbrio com seus objetivos. O segundo nível de adaptação lida com variações mais significativas do ambiente de execução que provocam uma degradação de desempenho da abordagem de tolerância a falhas em uso que não podem ser contornadas apenas através de adaptações paramétricas, exigindo uma reconfiguração estrutural da técnica de tolerância a falhas, substituindo-se a abordagem em uso por outra.

8.1.1 1º Nível de Adaptação: Reconfiguração Paramétrica no *Checkpointing*

A técnica de *checkpointing* naturalmente produz uma sobrecarga sobre o tempo de execução das tarefas, uma vez que é necessário parar o processo a cada vez que é salvo o estado do progresso da aplicação. A periodicidade estática nas tomadas dos *checkpoints* das tarefas torna esta abordagem não muito vantajosa quando levamos em consideração a volatilidade dos recursos em *Desktop Grids*. Um recurso é dito volátil quando, em seu histórico de funcionamento, ele apresenta um grande número de falhas ou, em se tratando de *Desktop Grids*, o proprietário do recurso não o disponibiliza com muita frequência para executar as tarefas da grade. Já os recursos estáveis são aqueles que são menos suscetíveis a falhas e, portanto, estão na maior parte do tempo disponíveis ou executando as computações submetidas à grade.

Nas abordagens estáticas, o *checkpointing* pode ser configurado com intervalos curtos para evitar que as tarefas, ao serem recuperadas das falhas, executem o mínimo possível para retornar ao mesmo estado de quando falharam. Contudo, essa configuração irá executar muitas vezes o procedimento que pausa e salva o estado da tarefa, prolongando o tempo de sua conclusão. Quando o ambiente se torna estável em

relação à disponibilidade dos recursos, essa sobrecarga é desnecessária. Por outro lado, em ambientes voláteis, a definição de longos intervalos para a tomada do *checkpointing* pode levar a muita reexecução de código da aplicação e, dependendo do grau de volatilidade dos recursos do ambiente, o *checkpointing* das tarefas pode nem chegar a ser feito. Se conhecermos o grau de volatilidade dos recursos da grade, podemos estimar intervalos entre *checkpoints* mais adequados para cada situação.

O gerente autonômico ajusta intervalos entre *checkpoints* através de regras que têm como base duas estimativas: (1) uma previsão sobre quando irá ocorrer a próxima falha (`failPrediction`); e (2) uma previsão sobre o tempo que resta para a conclusão da tarefa (`concRemExecTime`). Se o tempo previsto para a ocorrência da próxima falha for menor ou igual ao tempo que resta para a conclusão da tarefa, o intervalo entre *checkpoints* deve ser reduzido. Por outro lado, se o tempo previsto para a ocorrência da próxima falha for maior que tempo restante para a conclusão da tarefa, o intervalo entre *checkpoints* deve ser incrementado.

8.1.2 1º Nível de Adaptação: Reconfiguração Paramétrica na Replicação

Determinar um número fixo para a quantidade de réplicas na técnica de replicação em grades computacionais é uma tarefa difícil quando se utiliza uma abordagem estática. A geração de uma grande quantidade réplicas a fim de aumentar a probabilidade de sucesso das tarefas pode saturar a grade a medida que novas tarefas vão sendo submetidas pelos usuários. Em contrapartida, a geração de uma pequena quantidade de réplicas diminui as chances de conclusão das aplicações a despeito da ocorrência de falhas.

Visando superar essa dificuldade, o modelo de Viana utiliza uma abordagem de adaptação paramétrica para ajustar a quantidade de réplicas usada na estratégia de replicação à medida que os nós da grade vão sendo alocados para execução das computações dos usuários. A abordagem funciona com base no percentual recursos que estão ocupados executando alguma aplicação da grade. Uma vez que a ocupação cresce proporcionalmente ao número de réplicas empregado, a quantidade de réplicas é alterada quando são atingidas diferentes faixas desse percentual.

8.1.3 2º Nível de Adaptação: Reconfiguração Estrutural

O gerente autonômico busca ajustar parâmetros da técnica de tolerância a falhas em uso pelo *middleware* de grade de forma a manter o equilíbrio do sistema. No entanto, mudanças no ambiente de execução podem tornar estas ações insuficientes para a manutenção desse equilíbrio. Nesse caso, é necessário que o gerente autonômico global realize uma reconfiguração estrutural, procedendo então a substituição da abordagem de tolerância a falhas em uso.

Por esse motivo, o modelo proposto por Viana possui um segundo nível de adaptação que prevê a substituição da abordagem de tolerância a falhas em uso para uma das seguintes possibilidades: o uso de replicação ou a adoção de *checkpointing*. O uso da técnica de *checkpointing* impõe um custo adicional ao tempo de execução da aplicação. A técnica de replicação não impõe este custo, mas requer maior uso de recursos da grade, podendo atrasar a execução de novas tarefas submetidas à mesma dependendo da disponibilidade de seus recursos.

Assim, a reconfiguração estrutural é efetuada a partir de uma regra que utiliza o percentual de ocupação de recursos como métrica para tomada de decisão. À medida que o consumo de recursos cresce e os ajustes na técnica de replicação não mais surtem os efeitos desejados, a substituição da técnica de replicação por *checkpointing* se torna necessária. De outra forma, se o contexto da grade indica redução desse percentual a valores que permitam utilização da técnica anterior, volta-se a utilizar replicação.

8.1.4 Implementação da Abordagem de Viana

Esta seção descreverá resumidamente os principais componentes do AGST utilizados na implementação da abordagem autonômica de Viana, a fim de se destacar as vantagens na utilização desta ferramenta para a avaliação de mecanismos autonômicos desenvolvidos para grades de computadores. Destaca-se ainda importância deste trabalho para a validação das funcionalidades do AGST, para a percepção da necessidade de implementação de novas funcionalidades na ferramenta, bem como para a detecção e correção de *bugs*.

Para a implementação da abordagem autonômica de Viana foram definidos três ciclos de gerenciamento autonômico, conforme podemos observar na Figura 8.1.

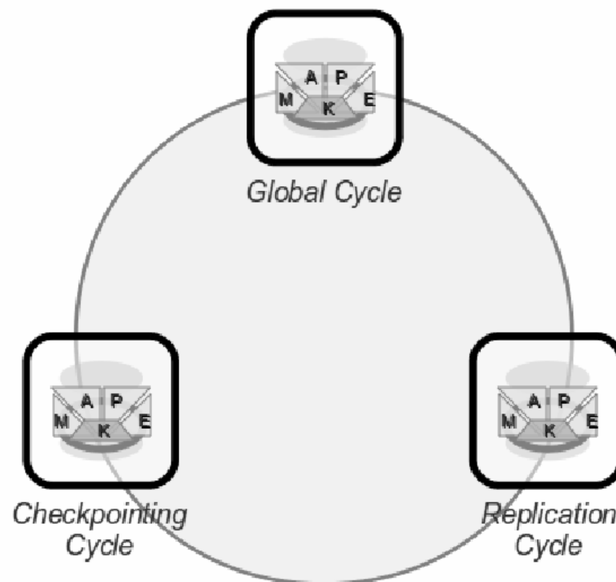


Figura 8.1: Ciclos Autonômicos da Abordagem de Viana.

Cada um desses ciclos possui um gerente autonômico que atua sobre um respectivo elemento gerenciado, os quais são diversos componentes do *middleware* da grade. Dois desses ciclos são responsáveis por realizar adaptações paramétricas: um deles controla o ciclo de gerenciamento autonômico local do *checkpointing* e o outro controla o ciclo de gerenciamento local da replicação. Para implementar os gerentes responsáveis por esses dois ciclos autonômicos foram instanciados respectivamente dois objetos da classe `LocalAutonomicManager`: o `checkpointingManager` e `replicationManager`. O terceiro ciclo é relativo ao gerenciamento global da abordagem autonômica, sendo responsável por realizar as adaptações estruturais, exercendo a função de ativar e desativar os outros dois ciclos. O gerente para esse ciclo é obtido a partir de uma instância da classe `GlobalAutonomicManager`, cuja nomenclatura desse objeto na implementação de Viana é `globalFaultToleranceManager`.

Em cada ciclo autonômico são processadas as funções de monitoramento, análise e planejamento, controle e execução de reconfigurações. Para facilitar a explicação, a descrição da implementação foi dividida de acordo com essas fases.

Monitoramento

Conforme foi descrito na seção 8.1, a abordagem autonômica de Viana necessita obter diversas informações sobre o contexto do ambiente de execução da grade. Para isso, foram desenvolvidos diversos sensores e monitores, utilizando como base as classes fornecidas pelo AGST. Os sensores foram desenvolvidos a partir de instâncias da classe `Sensor`, enquanto que os monitores foram implementados a partir de extensões da classe `AbstractMonitor`. Para obter as informações necessárias ao funcionamento da estratégia autonômica, foram desenvolvidos três monitores, sendo que cada um deles está associado a um sensor específico. O diagrama de classes que representa essa estrutura pode ser visto na figura 8.2.

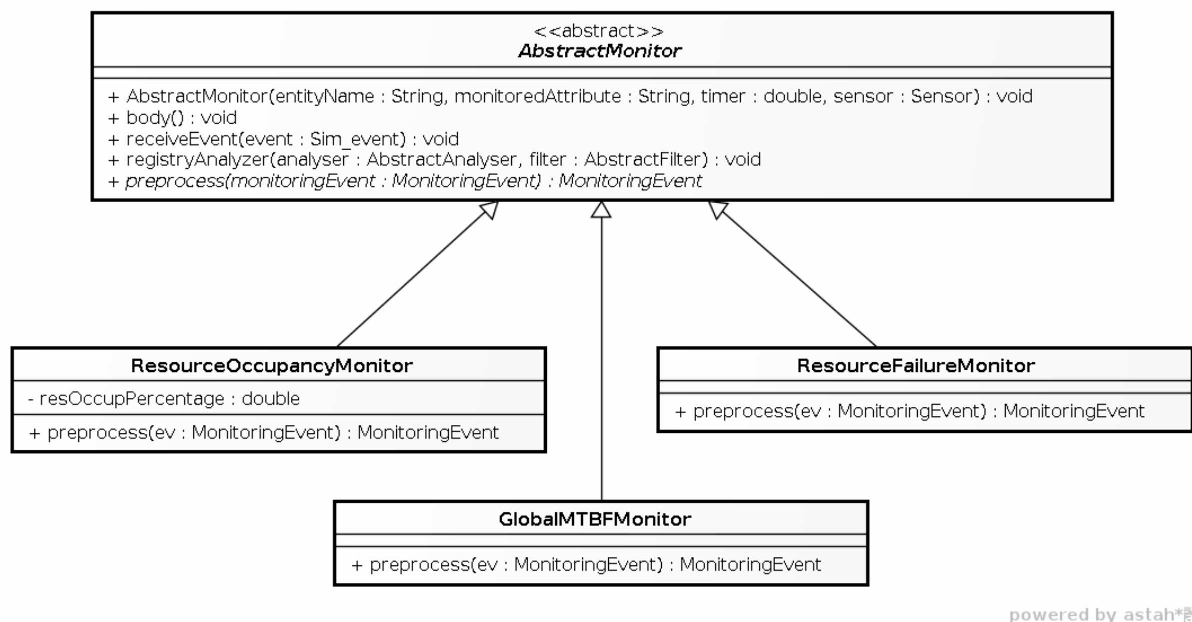


Figura 8.2: Monitores da Abordagem de Viana.

Para coletar do ambiente de grade as informações necessárias para a tomada de decisão, foram implementadas três classes de monitores: `ResourceOccupancyMonitor`, `ResourceFailuresMonitor`, e `GlobalMTBFMonitor`. Como foi visto na seção 7.2, para implementar um monitor é necessário estender a classe `AbstractMonitor`. Cada monitor pode realizar um pré-processamento sobre os dados obtidos a partir dos sensores, e avaliar através de regras se o valor das propriedade monitoradas são relevantes para o processo de adaptação da grade, bem como utilizar filtros para definir quais analisadores receberão notificações de mudanças de contexto.

`ResourceDynamicInfoSensor` é um sensor acoplado junto ao componente *ResourceDataStorage* (Serviços de Informações da Grade) que coleta desse componente uma lista dos recursos contendo seus respectivos estados: ocupado ou livre. O `ResourceOccupancyMonitor` obtém os dados desse sensor e calcula a porcentagem dos recursos disponíveis através da função de pré-processamento. Esse monitor é utilizado nos três ciclos da abordagem autonômica.

Como visto na seção 8.1.1, a abordagem de Viana necessita realizar ajustes no intervalo entre os *checkpoints* de cada tarefa. Para isso, é necessário coletar informações sobre o tempo médio entre falhas dos recursos. Essa função é exercida pelo `ResourceFailuresSensor`, um sensor acoplado ao *ResourceDataStorage* que coleta desse componente uma lista contendo as ocorrências de falhas dos recursos. O `ResourceFailuresMonitor` obtém esses dados desse sensor e calcula o tempo médio entre falhas de cada recurso da grade através da função de pre-processamento. Em seguida, essas informações são enviadas ao analisador do ciclo local de *checkpointing*.

De acordo com o que foi apresentado na seção 8.1.2, a abordagem de Viana utiliza adaptação paramétrica para ajustar o número de réplicas de acordo com o percentual de recursos ocupados e a taxa de falhas da grade. Entretanto, essa adaptação só é viável quando a taxa de falhas da grade como um todo for baixa. Essas informações são obtidas por um monitor global denominado `GlobalMTBFMonitor`, cuja função é realizar o monitoramento desses dados sempre que o analisador do ciclo local de replicação requisitar (sob demanda). Esse monitor recebe dados do sensor `ResourceFailuresSensor`, o mesmo utilizado pelo `ResourceFailureMonitor`. Uma vez que o monitoramento é realizado sob demanda, o `GlobalMTBFMonitor` não utiliza regras, e nem filtros.

Análise e Planejamento

A tomada de decisão na estratégia autonômica proposta por Viana é implementada por meio de regras que são executadas pelos analisadores. Foram implementadas três classes de analisadores: `CheckpointingAnalyser`, `ReplicationAnalyser`, e `GlobalAnalyserFaultTolerance`, conforme pode ser observado no diagrama de classes da Figura 8.3. Como visto na seção 7.2, todos os

analisadores devem estender da classe `AbstractAnalyser`, sendo que toda a lógica da análise e planejamento deve ser implementada no método `analiseAndPlan()`.

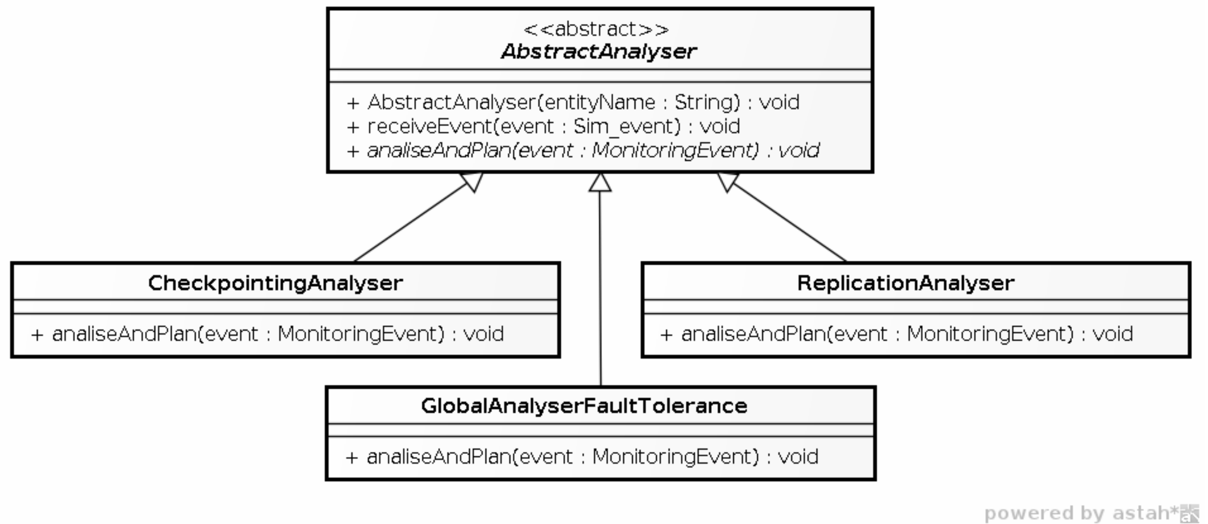


Figura 8.3: Analisadores da Abordagem de Viana.

O `ReplicationAnalyser` recebe dados de contexto do monitor `ResourceOccupancyMonitor` relativos ao percentual de ocupação dos recursos da grade. Essa informação é utilizada para que sejam tomadas decisões sobre adaptações paramétricas durante a vigência da técnica de replicação. De acordo com percentual informado, o analisador deve decidir qual o número de réplicas que deve ser utilizado, bem como criar a ação de reconfiguração apropriada e enviá-la para o executor. Nesse caso, é instanciada uma ação do tipo `ActionChangeReplicasNum`, passando como parâmetro no construtor da classe a quantidade de réplicas.

Conforme visto na seção 7.2, para implementar uma ação é necessário estender a classe `AbstractAction` e implementar o método `executeAction()`. O plano contendo todas as ações definidas pelo analisador é gerado através da invocação do método `createActionPlan(AbstractAction...actions)`. Esse método está implementado na própria classe `AbstractAnaliser`. O plano de ações gerado é enviado para o executor através da invocação do método `sendPlanToExecutor(ActionPlanEvent planEvent)`. Esse método recebe como parâmetro um evento contendo o plano de ações criado pelo analisador.

O `CheckpointingAnalyser` recebe dados de contexto do monitor `ResourceFailuresMonitor` relativos ao percentual de falhas de recurso da grade. A partir dessas informações ele pode decidir se deve modificar o intervalo de

checkpointing das tarefas em execução na grade. Para isso, esse analisador instancia ações do tipo `ActionChangeChekpointingInterval`, passando como parâmetro no construtor da classe o valor do novo intervalo. Além disso, esse analisador recebe dados de contexto sob-demanda do monitor `ResourceOccupancyMonitor`. Esses dados são relativos ao percentual de ocupação dos recursos da grade, e são utilizados para que sejam tomadas decisões acerca do cancelamento de tarefas replicadas.

O `GlobalAnalyserFaultTolerance` é o responsável pela tomada de decisões do ciclo autonômico global (gerenciado pelo `GlobalAutonomicManager`). O `GlobalAnalyserFaultTolerance` recebe dados de contexto do monitor `ResourceOccupancyMonitor` relativos ao percentual de ocupação dos recursos da grade, bem como informações sobre a taxa de falhas dos recursos da grade.

Essas informações são utilizadas pela abordagem autonômica para a tomada de decisão sobre adaptações no nível estrutural, que se necessário promoverá a substituição de uma técnica de tolerância a falhas por outra. Para isso, o `GlobalAnalyserFaultTolerance` instancia ações do tipo `ActionChangeToCheckpointing`, caso a substituição seja feita no sentido da replicação para o *checkpointing*, ou então, ele instancia uma ação do tipo `ActionChangeToReplication`, caso a substituição seja no sentido inverso. Nesse último caso, é necessário passar o número de réplicas como argumento no construtor ação.

Os analisadores locais que controlam os ciclos de gerenciamento autonômico das técnicas de replicação e de *checkpointing* devem ser reiniciados ou parados de acordo com a reconfiguração que será realizada. Por exemplo, enquanto a estratégia vigente for a de replicação, o ciclo de *checkpointing* deverá estar inativo, e vice-versa. Uma vez que o ciclo global é controlado pelo `GlobalAnalyserFaultTolerance`, esse analisador fica responsável por enviar os eventos de parada ou reinício para os ciclos autonômicos locais.

Controle e Execução

Após a elaboração do plano de ações por parte do analisadores, é necessário que o executor receba esse plano e proceda a execução das ações nele contidas através dos atuadores. Conforme visto na seção 7.2, a execução do plano de ações é

atingida instanciando-se objetos executores e atuadores a partir das classes `Executor` e `Effector`, respectivamente.

Na implementação da abordagem de Viana todos os gerentes autonômicos (`CheckpointingManager`, `ReplicationManager`, e `GlobalFaultToleranceManager`) compartilham um único executor (abordagem centralizada), de forma que todas as ações de reconfiguração dinâmica são colocadas em uma fila, e a execução das mesmas obedece a ordem de chegada dos planos de ação (FIFO). Essa abordagem garante que apenas uma ação de reconfiguração será executada por vez no ambiente de grade simulado.

A abordagem de Viana utilizou diversos atuadores em sua implementação para realizar adaptações paramétricas e estruturais. Entre os principais atuadores, estão os utilizados para alterar parâmetros como o número de réplicas da técnica de replicação, denominado `ChangeReplicasNumberEffector`, e o intervalo que determina a periodicidade do *checkpointing* das tarefas, denominado `ChangeIntervalCheckpointingEffector`. Merecem destaque também os atuadores responsáveis por realizar adaptações estruturais, como o atuador que efetua a substituição da técnica de tolerância a falhas, denominado `ChangeFaultToleranceStrategyEffector`. Esse último atuador age sobre o `GridScheduler`, que corresponde ao componente do `middleware` de grade responsável pelo escalonamento de tarefas.

Uma vez que não é desejável que o `GridScheduler` escalone aplicações durante uma ação de adaptação, foi desenvolvido um sincronizador de reconfigurações específico para esse componente, denominado `GridSchedulerInterceptor`. Esse componente foi implementado estendendo o aspecto `AbstractIntercept`, provido pelo AGST. Como visto na seção 7.2, esse é implementado como uma classe utilizando anotações (*@Annotations*) Java, e pode ser utilizada para interceptar os eventos de reconfiguração enviados aos elementos gerenciados. Essa interceptação impede que qualquer outro evento seja processado pelo escalonador durante a execução a reconfiguração. Dessa forma, o `GridScheduler` fica impossibilitado de escalonar aplicações durante a substituição de uma técnica de tolerância a falhas por outra.

8.1.5 Resultados e Discussões

O suporte à modelagem e simulação de grades autonômicas provido pelo AGST foi de vital importância para tornar possível a avaliação do desempenho da abordagem de Viana. Nesse aspecto, é importante frisar que o conjunto de componentes autonômicos responsáveis pelas funções do ciclo MAPE-K facilitou a implementação da referida abordagem através do reuso. O uso desses componentes permitiu que o maior esforço ficasse concentrado na implementação da lógica da tomada de decisão, na definição dos critérios de filtragem, e na criação de ações de reconfiguração. Além disso, a utilização de técnicas como a reflexão computacional e programação orientada a aspectos dispensaram a instrumentação direta no código fonte dos elementos gerenciados, permitindo uma separação clara entre o código funcional desses componentes e o código responsável pela execução das ações de adaptação.

Graças a flexibilidade oferecida pelo AGST na simulação de *Desktop Grids*, a abordagem de Viana foi analisada em diversos cenários, considerando diferentes condições no ambiente de execução: muitos recursos e poucas falhas, muitos recursos e muitas falhas, poucos recursos e poucas falhas, etc. Nesse sentido, foi de fundamental importância tanto o suporte à geração de recursos computacionais com carga de trabalho local (*hostload*), bem como o suporte à geração de falhas de recursos providos pelo AGST. Falhas e recursos podem ser gerados automaticamente, tanto de forma sintética (a partir de distribuições probabilísticas), quanto a partir de arquivos de `traces` coletados de ambientes de *Desktop Grids* reais.

A avaliação da abordagem de Viana foi feita tomando como base comparativa o desempenho das principais abordagens estáticas para tolerância a falhas na execução de aplicações em grades computacionais: reinício, replicação, *checkpointing*. Nesse ponto é importante destacar que essas três abordagens já são providas pelo AGST, de modo que não foi necessário implementá-las para tornar a comparação de desempenho possível. Isso possibilitou ainda uma considerável economia no tempo de implementação da abordagem de Viana, uma vez que a mesma depende da alternância de duas das três abordagens já disponibilizadas pelo AGST.

As métricas utilizadas para avaliação da eficiência das abordagens para tolerância a falhas nas simulações foram: (1) o percentual de aplicações submetidas

para execução concluídas com sucesso; e (2) o tempo médio de conclusão dessas aplicações. Quanto a isso, é importante destacar que o fato do AGST utilizar uma base de dados relacional para o armazenamento dos resultados das simulações facilitou bastante o processo de consulta a esses dados através da linguagem SQL. Isso tornou o processo de avaliação da abordagem de Viana mais eficiente, se comparado ao esforço para a mineração dos dados a partir de arquivos de texto, prática adotada por outros simuladores.

Os resultados gerados pelas simulações realizadas utilizando o AGST permitiram que Viana observasse que as técnicas de reinício, replicação e *checkpointing* apresentam desempenhos diferentes de acordo com o estado do ambiente de execução. Viana pode observar que a estratégia autonômica produziu bons resultados, aproximando-se sempre da abordagem estática que obteve o melhor desempenho em cada um dos cenários avaliados. Portanto, Viana concluiu que em ambientes de grade que apresentem variações na disponibilidade de recursos e no quantitativo de falhas desses recursos, a abordagem autonômica proposta por ele contribui significativamente para reduzir o tempo de conclusão das aplicações, mantendo elevado percentual de sucesso em suas execuções [77,78].

8.2 Avaliação de um Mecanismo de Gerenciamento de Execução de Aplicações

Outro exemplo de aplicação do AGST pode ser encontrado nos trabalhos de Martins [58], que desenvolveu um mecanismo de gerenciamento de aplicações em *Desktop Grids* que possuam restrições de tempo de execução definidas pelos usuários no ato de sua submissão. Esse mecanismo foi desenvolvido para ser incorporado ao *middlware* InteGrade, no entanto, optou-se por avaliá-lo previamente através de simulações. O referido mecanismo é composto basicamente por cinco componentes:

1. Um mecanismo de predição do tempo de execução das aplicações baseado na capacidade de processamento dos recursos da grade. Esse mecanismo é provido pela quarta camada do AGST (ver seção 5.2).
2. Uma heurística de escalonamento que mapeia as tarefas que compõem a aplicação para nós que potencialmente possam cumprir o prazo de execução

da aplicação especificado pelo usuário. Essa abordagem considera que existem duas classes de aplicações: a classe *soft-deadline*, que designa as aplicações que possuem restrições quanto ao tempo máximo em que devem ser executadas; e a classe *nice*, que designa aplicações que não possuem restrições de tempo de execução. Essa heurística foi implementada através de uma extensão da classe `SchedulingStrategy` do AGST, que permite definir famílias de algoritmos de escalonamento utilizando o padrão *Strategy* (ver seção 5.2).

3. Um mecanismo que monitora o progresso da execução das tarefas com restrição de tempo em cada nó da grade, verificando a necessidade ou não de reescaloná-las para outros nós de forma a atender o prazo estipulado. Este mecanismo foi implementado utilizando os sensores providos pelo AGST;

4. Um mecanismo que monitora o tempo médio entre falhas de cada recurso. Este mecanismo também foi implementado utilizando os sensores providos pelo AGST;

5. Um mecanismo de tolerância a falhas na execução de aplicações baseado em *checkpointing*. Esse mecanismo é inspirado na abordagem de Viana (ver seção 8.1).

O algoritmo de escalonamento de aplicações utilizado pela abordagem der Martins é do tipo *on-line*, isto é, cada aplicação é mapeada imediatamente após sua submissão (caso haja recurso disponível). Essa abordagem pode mapear mais de uma tarefa para execução em um dado nó da grade. O mapeamento de tarefas da classe *soft-deadline* é realizado levando-se em consideração o intervalo médio entre falhas dos nós e a sua capacidade de processamento. Escalona-se uma tarefa da classe *soft-deadline* para nós identificados como estáveis (que apresentem maior MTBF) e cuja capacidade permita concluir a execução da tarefa dentro do prazo estipulado pelo usuário ao submeter a aplicação para execução na grade. Para que mudanças no contexto da execução das aplicações *soft-deadline* sejam percebidas, é necessário monitorar o progresso de execução de cada tarefa. Além disso, as informações sobre o MTBF de cada nó precisam ser monitoradas periodicamente para que a abordagem de escalonamento possa mapear as aplicações para aqueles considerados mais estáveis.

A combinação entre a abordagem de escalonamento e a abordagem de tolerância tem o objetivo de garantir a execução com sucesso das aplicações, mesmo na ocorrência de falhas de recursos. Porém, devido ao custo de tempo imposto pelo uso da técnica de *checkpointing* e a eventual variação na carga de trabalho local dos nós, considera-se como aptos para executar a tarefa os nós cujas capacidades de processamento estimadas pelo mecanismo de predição adotado permitam concluir a execução da tarefa com certa margem de antecedência em relação ao *deadline* definido pelo usuário. Essa margem serve para garantir que a aplicação consiga executar no prazo determinado, ainda que eventuais atrasos ocorram. Caso a carga local de trabalho exceda a estimativa inicial, o mecanismo de acompanhamento do progresso da execução das tarefas é responsável por identificar a incapacidade do nó em cumprir o prazo estipulado e ressubmeter a tarefa para um novo escalonamento.

A métrica utilizada como critério de avaliação foi a porcentagem de aplicações executadas com sucesso dentro do prazo estipulado pelo usuário. Os resultados das simulações permitiram concluir que o mecanismo de gerenciamento de execução de aplicações proposto por Martins e a abordagem para tolerância a falhas na execução de aplicações baseada nos trabalhos de Viana, quando adequadamente combinados, compõem um modelo de gerenciamento que permite executar aplicações com prazo de conclusão de forma eficiente, mesmo em um ambiente tão dinâmico quanto o das *Desktop Grids*. O mecanismo proposto por Martins foi avaliado em diversos cenários, levando-se em consideração características típicas de *Desktop Grids*, apresentando resultados significativamente melhores do que os apresentados pelo mecanismo tradicional de escalonamento de aplicações utilizado pelo *middlware* InteGrade, o qual serviu como base para a comparação dos resultados.

8.3 Conclusão

Este capítulo mostrou dois estudos de caso envolvendo a aplicação das funcionalidades providas pelo AGST para facilitar o processo de modelagem, simulação e avaliação de duas abordagens especificamente voltadas para *Desktop Grids*: a abordagem autonômica para tolerância a falhas na execução de aplicações proposta por Viana e o mecanismo de gerenciamento de aplicações com restrições de tempo de execução definidas pelos usuários no ato de sua submissão.

A partir dos resultados obtidos através das simulações foi possível avaliar previamente essas duas abordagens sem a utilização de um ambiente de grade real. Os diversos cenários simulados permitiram que mudanças em vários aspectos das abordagens pudessem ser experimentadas. Portanto, a utilização do AGST foi importante para analisar o comportamento das abordagens avaliadas, antever seus resultados, e ainda melhorar projeto dessas abordagens, antes de proceder a implementação e testes das mesmas em ambientes de *Desktop Grids* reais.

Os diversos experimentos realizados com o AGST nos estudos de caso descritos foram extremamente importantes para testar a própria capacidade desse simulador em atender os requisitos considerados necessários para tornar possível a simulação de grades autonômicas: ciência de contexto, adaptação dinâmica, transparência e arquitetura autonômica (ver seção 6.2). Uma vez que ambas as abordagens puderam ser simuladas a partir de simples extensões ao componentes do AGST, pode-se assumir que o simulador atende de fato a esses requisitos. Além disso, os estudos de caso descritos permitiram ao longo dos experimentos detectar e corrigir *bugs*, bem como perceber a necessidade de prover melhorias e implementar novas funcionalidades.

9 Conclusões e Trabalhos Futuros

Sistemas de computação autonômica são aqueles capazes de se auto-gerenciar e se adaptar dinamicamente às mudanças a fim restabelecer seu equilíbrio de acordo com as políticas e os objetivos definidas pelos administradores dos sistemas. Para isso, devem dispor de mecanismos efetivos que os permita monitorar, controlar e regular a si próprios, bem como recuperarem-se de problemas sem a necessidade de intervenções externas. Uma arquitetura para o desenvolvimento de sistemas autonômicos bastante utilizada é a MAPE-K, que define um ciclo de gerenciamento com fases de monitoramento, análise e planejamento, controle e execução de reconfigurações.

Em computação em grade, a criação de mecanismos que proporcionem um maior grau de autonomia ao *middleware* da grade, de forma que a intervenção humana na manutenção e gerenciamento desses ambientes computacionais seja reduzida, constitui um dos temas de pesquisa mais explorados ultimamente e mostra-se também um grande desafio. Apesar disso, os simuladores de grades até então referenciados na literatura não apresentam o suporte adequado para auxiliar os desenvolvedores de *middlewares* de grade no processo de avaliação de abordagens autonômicas voltadas para esses ambientes computacionais.

Este trabalho apresentou um levantamento do estado de arte em computação autonômica, computação em grade, simulação computacional, e analisou as características encontradas nas principais ferramentas de simulação conhecidas. Porém, o ponto central deste trabalho foi a apresentação do simulador AGST. Esse simulador, diferente dos simuladores de grade analisados, implementa uma arquitetura voltada para desenvolvimento de sistemas autonômicos, a MAPE-K. O AGST provê um conjunto de funcionalidades adequadas ao processo de modelagem, simulação e avaliação de abordagens autonômicas voltadas para grade computacionais, podendo auxiliar os desenvolvedores de *middlware* de grades e pesquisadores em geral na avaliação de novos conceitos em suas implementações. Este trabalho também descreveu dois estudos de caso de utilização do AGST para

demonstrar como as funcionalidades do MAPE-K Simulation Framework podem ser utilizadas.

As principais contribuições deste trabalho são:

- A investigação do estado da arte em ferramentas de simulação para o apoio ao desenvolvimento e avaliação de *middlware* para ambientes de computação em grade. Através desta investigação, ficou evidenciado a escassez de mecanismos específicos para a avaliação de abordagens autonômicas para ambientes de grades computacionais, o que motivou o desenvolvimento do AGST.

- O desenvolvimento de uma ferramenta que permite modelar e simular um ambiente de grade computacional capaz de monitorar, controlar e regular a si próprio, bem como se recuperar de problemas sem a necessidade de intervenções externas. O AGST provê, através do MAPE-K Simulation Framework, um conjunto de funcionalidades que permite modelar e simular ciclos autonômicos em grades: Gerenciamento de Ciclos Autonômicos, Monitoramento, Análise e Planejamento, Controle e Execução de Reconfigurações. Esse simulador permite que abordagens autonômicas sejam avaliadas através da técnicas simulação sem forçar os desenvolvedores dessas abordagens a implementar o suporte do qual necessitam em suas investigações.

- A disponibilização de funcionalidades que simulam mecanismos de ciência de contexto, execução de adaptações paramétricas e composicionais, transferência de estado, e sincronização. Essas funcionalidades são baseadas em reflexão computacional e programação orientada a aspectos, e evitam que os usuários do AGST precisem realizar instrumentações estáticas nas classes dos componentes gerenciados. Isso facilita a modelagem das abordagens, economizando esforço de implementação e manutenção do código adaptativo.

- A avaliação dos mecanismos de suporte a modelagem, simulação e avaliação providos pelo AGST através de estudos de caso. Isso permitiu constatar que o modelo de simulação proposto se mostrou adequado às necessidades exigidas para o desenvolvimento e avaliação dessas abordagens através de técnicas simulação. A partir do desenvolvimento do AGST, outros pesquisadores

podem realizar novo estudos de caso e propor a implementação de novas funcionalidades.

A partir deste trabalho inicial, identificamos algumas possibilidades de trabalhos futuros que poderiam ser desenvolvidos, tais como:

- O AGST deixa o usuário livre para implementar o mecanismo de tomada de decisão de acordo com a abordagem autonômica a ser avaliada. Para simplificar este processo de implementação, o AGST disponibiliza o suporte para o desenvolvimento de mecanismos de tomada de decisão expressos através de regras do tipo evento-condição-ação (*ECA rules*). No entanto, o suporte a outras técnicas, tais como funções de utilidades e aprendizagem por reforço, técnicas Bayesianas [36] e técnicas híbridas [75] podem ser futuramente desenvolvidas.

- Atualmente o AGST permite a avaliação de abordagens autonômicas com base na arquitetura MAPE-K e em abordagens de desenvolvimento de sistemas autonômicas como POA e reflexão computacional. Pode-se futuramente estudar a possibilidade de implementar novas arquiteturas e/ou utilizar novas abordagens ou variantes delas. Por exemplo, a utilização do *AspectJ* limita a implementação do mecanismo de sincronização do AGST ao uso de POA estática. Isso significa que aspectos não podem ser adicionados ou removidos em tempo de execução. O uso POA dinâmica resolveria esse problema e tornaria o mecanismo de sincronização mais flexível e transparente.

- A implementação de mecanismos para a detecção de conflitos entre regras do tipo evento-condição-ação durante o processo de tomada de decisão da abordagem autonômica.

Referências Bibliográficas

[1] L. Abraham, M. A. Murphy, M. Fenn, and S. Goasguen. Self-provisioned hybrid clouds. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 161–168, New York, NY, USA, 2010. ACM.

[2] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, USA, jun 2003.

[3] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski. Fault-aware scheduling for bag-of-tasks applications on desktop grids. In *Grid Computing, 7th IEEE/ACM International Conference on*, pages 56 –63, September 2006.

[4] M. Baker, R. Buyya, and D. Laforenza. The grid: International efforts in global computing. *In Proc. of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2000.

[5] F. Berman, G. Fox, and A. Hey. *Grid computing: making the global infrastructure a reality*. Wiley series on parallel and distributed computing. Wiley, 2003.

[6] R. Buyya. Understanding the Grid. *Grid Computing Planet Conference*, 2002.

[7] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13):1175–1220, 2002.

[8] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Softw. Pract. Exper.*, 35:491–512, April 2005.

[9] R. Buyya, S. Venugopal, R. Ranjan, and C. S. Yeo. The Gridbus Middleware for Market-Oriented Computing. In R. Buyya and K. Bubendorfer, editors, *Market-Oriented Grid and Utility Computing*, chapter 26, pages 589–622. John Wiley and Sons, Hoboken, NJ, USA, 2009.

[10] A. Caminero, A. Sulistio, B. Caminero, C. Carrion, and R. Buyya. Extending gridsim with an architecture for failure detection. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 01*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[11] H. Casanova, A. Legrand, and M. Quinson. Simgrid: A generic framework for large-scale distributed experiments. *Tenth International Conference on Computer Modeling and Simulation*, pages 126–131, April 2008.

[12] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. de Turck, P. Demeester, and P. A. Vanrolleghem. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):180–190, Fevereiro 2009.

[13] M. Chtepen, B. Dhoedt, F. H. A. Claeys, F. de Turck, P. Demeester, and P. A. Vanrolleghem. Dynamic Scheduling of Computationally Intensive Applications on Unreliable Infrastructures. In *Proceedings of International Mediterranean Modeling Multiconference (I3M'06)*, pages 539–544, Barcelona, Espanha, Outubro 2006. IEEE Computer Society.

[14] L. Chwif and A. C. Medina. *Modelagem e Simulação de Eventos Discretos: Teoria & Aplicações*. Ed. dos Autores, São Paulo, SP, Brasil, 2 edition, 2007. ISBN 8590597822.

[15] P. Collet, F. Křikava, J. Montagnat, M. Blay-Fornarino, and D. Manset. Issues and scenarios for self-managing grid middleware. In *Proceeding of the 2nd workshop on Grids meets autonomic computing*, GMAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[16] S. Corrêa and R. Cerqueira. Computação autônoma: Conceitos, infra-estruturas e soluções em sistemas distribuídos. In *Anais do 27o. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'09)*, pages 151–198, Recife, PE, Brasil, Maio 2009.

[17] G. Cunha Filho. OGST (Opportunistic Grid Simulation Tool): Uma Ferramenta de Simulação para Avaliação de Estratégias de Escalonamento de Aplicações em Grades Oportunistas. Master's thesis, Universidade Federal do Maranhão, São Luís, MA, Brasil, 2009.

[18] G. Cunha Filho and F. J. da Silva e Silva. Ogst: An opportunistic grid simulation tool. In *LAGrid 2008: 2nd International Workshop Latin American Grid*, Campo Grande, Mato Grosso do Sul, August 2008.

[19] F. J. da Silva e Silva, F. Kon, A. Goldman, M. Finger, R. Y. de Camargo, F. C. Filho, and F. M. Costa. Application execution management on the integrade opportunistic grid middleware. *Journal of Parallel and Distributed Computing*, 70:573–583, 2010.

[20] D. Dawson, R. Desmarais, H. M. Kienle, and H. A. Müller. Monitoring in adaptive systems using reflection. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 81–88, New York, NY, USA, 2008. ACM.

[21] Dawson, Dylan. Facilitating autonomic computing using reflection. Master's thesis, Department of Computer Science, University of Victoria, Victoria, BC, Canada, 2003.

[22] B. de Tácio P. Gomes and F. J. da S. e Silva. Agst - autonomic grid simulation tool. a simulator of autonomic functions based on the mape-k model. In *Proceedings of First International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2011)*, pages 354–359, Noordwijkerhout, Netherlands, 2011. SciTePress.

[23] B. de Tácio Pereira Gomes, G. C. Filho, I. R. Campos, J. F. Gonçalves, and F. J. da Silva e Silva. Scheduling strategies evaluation for opportunistic grids. *Computing Systems, Symposium on*, 0:88–95, 2010.

[24] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. *Commun. ACM*, 44:33–38, October 2001.

[25] I. R. Forman and N. Forman. *Java Reflection in Action*. Manning Publications Co., 2006.

[26] J. W. Forrester. *Principles of systems*. Pegasus Communications, 1968.

[27] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the i-way high-performance distributed computing experiment. In *Proceedings*

of the 5th IEEE International Symposium on High Performance Distributed Computing, HPDC '96, pages 562–, Washington, DC, USA, 1996. IEEE Computer Society.

[28] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[29] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*, chapter Globus: A Toolkit-Based Grid Architecture, pages 259–278. Morgan Kaufmann Publisher, 1999.

[30] I. Foster, C. Kesselman, and S. Tueke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 2001.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[32] C. Germain-Renaud and O. F. Rana. The convergence of clouds , grids , and autonomics. *IEEE Internet Computing*, 13(December):9–9, 2009.

[33] A. Goldchleger, F. Kon, A. Goldman, F. M., and S. W. Song. Integrade: Rumo a um sistema de computação em grade para aproveitamento de recursos ociosos em máquinas compartilhadas. Technical report, Departamento de Ciência da Computação. Instituto de Matemática e Estatística - Universidade de São Paulo, Set 2002.

[34] L. Gong, X.-H. Sun, and E. F. Watson. Performance modeling and prediction of nondedicated network computing. *IEEE Transactions on Computers*, 51(9):1041–1055, September 2002.

[35] A. S. Grimshaw, W. A. Wulf, and C. The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40:39–45, January 1997.

[36] H. Guo. A Bayesian Approach for Automatic Algorithm Selection. In *Proceedings of the IJCAI Workshop on AI and Autonomic Computing: Developing a Research Agenda for Self-Managing Computer Systems*, Acapulco, Mexico, 2003.

[37] S. Hallsteinsen and J. Floch. Theory of adaptation - specification of the madam core architecture and middleware services. Technical Report D 2.2, SINTEF

Information ad Communication Technology, Trondheim, Norway, December 2006.

[38] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The Autonomic Computing Paradigm. *Cluster Computing*, 9(1):5–17, Janeiro 2006.

[39] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing - Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3):7:1–7:28, Agosto 2008.

[40] IBM. An architectural blueprint for autonomic computing. Technical report, 2003.

[41] S. Jha, M. Parashar, and O. Rana. Investigating autonomic behaviours in grid-basedcomputational science applications. In *Proceedings of the 6th international conference industry session on Grids meets autonomic computing*, GMAC '09, pages 29–38, New York, NY, USA, 2009. ACM.

[42] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, Jyväskylä, Finlândia, Junho 1997. Springer-Verlag.

[43] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, EUA, Abril 1991.

[44] K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 541–548, Washington, DC, USA, 2007. IEEE Computer Society.

[45] D. Klusáček, L. Matyska, and H. Rudová. Alea: Grid Scheduling Simulation Environment. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics (PPAM'07)*, pages 1029–1038, Gdańsk, Polônia, Setembro 2008. Springer-Verlag.

[46] D. Klusáček and H. Rudová. Alea 2: job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 61:1–61:10, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[47] D. R. Koes and S. C. Goldstein. Near-optimal instruction selection on dags. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 45–54, New York, NY, USA, 2008. ACM.

[48] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'10)*, pages 398–407, Melbourne, Australia, Maio 2010. IEEE Computer Society.

[49] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computin. In *Software - Practice and Experience*, volume 32, pages 135–164. February 2002.

[50] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. Grid Scheduling Simulations with GSSIM. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS'07)*, pages 1–8, Hsinchu, Taiwan, Dezembro 2007. IEEE Computer Society.

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

[52] C. L. Liu and J. W. Layland. Tutorial: hard real-time systems. chapter Scheduling algorithms for multiprogramming in a hard real-time environment, pages 174–189. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[53] H. Liu, V. Bhat, M. Parashar, and S. Klasky. An autonomic service architecture for self-managing grid applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 132–139, Washington, DC, USA, 2005. IEEE Computer Society.

[54] Y. Liu. Grid Scheduling. Technical report, Department of Computer Science, University of Iowa, Iowa City, IA, EUA, 2004.

[55] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63:2003, 2001.

[56] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL, EUA, Outubro 1987. ACM.

[57] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, London, England, 1995.

[58] M. R. M. Martins, B. de Tácio P. Gomes, J. F. Gonçalves, and F. J. da Silva e Silva. Execution management of applications with runtime restrictions on opportunistic grids environments. In *The Second International Conference on Performance, Safety and Robustness in Complex Systems and Applications - PESARO 2012*.

[59] D. A. Menascé and E. Casalicchio. Qos in grid computing. *IEEE Internet Computing*, pages 85–87, July - August 2004.

[60] R. Murch. *Autonomic Computing*. IBM Press/Prentice-Hall, 2004.

[61] M. Parashar and S. Hariri. Autonomic Computing: An Overview. *Unconventional Programming Paradigms*, pages 247–259, 2005.

[62] M. Parashar and S. Hariri. *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press, 2006.

[63] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, 9(2):161–174, Abril 2006.

[64] P. R. Prins. Teaching parallel computing using beowulf clusters: a laboratory approach. *J. Comput. Sci. Coll.*, 20(2):55–61, Dec. 2004.

[65] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Ed. Prentice-Hall, Englewood Cliffs, NJ, EUA, 2nd edition, 2003.

[66] R. J. Sabatine. Implementação de um gerenciador de redes overlay para o gridsim. Master's thesis, Instituto de Física de São Carlos. Universidade de São Paulo, São Paulo, SP, Brasil, 2010.

[67] M. A. S. Sallem, S. A. de Sousa, and F. J. da Silva e Silva. Autogrid: Towards an autonomic grid middleware. In *Proceedings of the 16th IEEE International Workshops*

on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE '07, pages 223–228, Washington, DC, USA, 2007. IEEE Computer Society.

[68] N. K. Sasaki. *Simulação de Sistemas de Comunicação Óptica Baseada em Simulação a Eventos Discretos*. PhD thesis.

[69] R. E. Shannon. Simulation modeling and methodology. *SIGSIM Simul. Dig.*, 8:33–38, April 1977.

[70] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The easy - loadleveler api project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, London, UK, 1996. Springer-Verlag.

[71] C. Souza and C. Mazieiro. Intercessão em Tempo de Implantação - uma Abordagem Reflexiva para a Plataforma J2EE. In *Anais do XV Simpósio Brasileiro de Engenharia de Software (SBES'01)*, pages 286–301, Rio de Janeiro, RJ, Brasil, Outubro 2001.

[72] A. Sulistio, C. S. Yeo, and R. Buyya. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *Softw. Pract. Exper.*, 34:653–673, June 2004.

[73] X.-H. Sun and M. Wu. Grid harvest service: A system for long-term, application-level task scheduling. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[74] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Ed. MIT Press, Cambridge, MA, EUA, 1998.

[75] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 3rd International Conference on Autonomic Computing (ICAC'06)*, pages 65–73, Dublin, Ireland, Junho 2006. IEEE Computer Society.

[76] P. Thysebaert, B. Volckaert, F. de Turck, B. Dhoedt, and P. Demeester. Evaluation of grid scheduling strategies through nsgrid: a network-aware grid simulator. *Neural, Parallel Sci. Comput.*, 12:353–378, September 2004.

[77] A. E. B. Viana. Uma abordagem autonômica para tolerância a falhas na execução de aplicações em desktop grids. Master's thesis, Universidade Federal do Maranhão, São Luís, MA, Brasil, 2011.

[78] A. E. B. Viana, B. de Tácio P. Gomes, J. F. Gonçalves, L. R. Coutinho, and F. J. da Silva e Silva. Design and evaluation of autonomic fault tolerance strategies using the agst autonomic grid simulator. In *Proceedings Of CLCAR 2011 - 4th Latin American Conference on High Performance Computing*, Colima, Mexico, 2011. Instituto Tecnológico de Colima.

[79] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 120–130, Los Angeles, CA, EUA, Maio 1999. ACM.

[80] C. Yeo and R. Buyya. Integrated risk analysis for a commercial computing service in utility computing. *Journal of Grid Computing*, 7:1–24, 2009. 10.1007/s10723-008-9103-2.

[81] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4):171–200, Setembro 2005.

[82] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Sci. Program.*, 14:217–230, December 2006.