UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

GUSTAVO CIPRIANO MOTA SOUSA

Desenvolvimento de Máquinas de Execução para Linguagens de Modelagem Específicas de Domínio: Uma Estratégia Baseada em Engenharia Dirigida por Modelos

Goiânia
2012

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Desenvolvimento de Máquinas de Execução para Linguagens de Modelagem Específicas de Domínio: Uma Estratégia Baseada em Engenharia Dirigida por Modelos

Autor(a): Gustavo Cipriano Mota Sousa

Goiânia, 09 de Outubro de 2012.

Gustavo Cipriano Mota Sousa – Autor

Fábio Moreira Costa – Orientador

GUSTAVO CIPRIANO MOTA SOUSA

Desenvolvimento de Máquinas de Execução para Linguagens de Modelagem Específicas de Domínio: Uma Estratégia Baseada em Engenharia Dirigida por Modelos

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Sistemas Distribuídos.

Orientador: Prof. Fábio Moreira Costa

Goiânia
2012

GUSTAVO CIPRIANO MOTA SOUSA

Desenvolvimento de Máquinas de Execução para Linguagens de Modelagem Específicas de Domínio: Uma Estratégia Baseada em Engenharia Dirigida por Modelos

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 09 de Outubro de 2012, pela Banca Examinadora constituída pelos professores:

Prof. Fábio Moreira Costa
Instituto de Informática – UFG
Presidente da Banca

Prof. Ricardo Couto Antunes da Rocha
Instituto de Informática – UFG

Prof. Nelson Souto Rosa
Centro de Informática – UFPE

Gustavo Cipriano Mota Sousa

Graduou-se em Ciências da Computação na UFG - Universidade Federal de Goiás.

Agradecimentos

Antes de tudo, gostaria de agradecer de forma especial ao meu orientador e professor Fábio Moreira Costa por toda a atenção e paciência dedicada à minha orientação. Sem o seu contínuo apoio e motivação, este trabalho certamente não seria possível.

Também quero agradecer ao Prof. Peter Clarke e o grupo de pesquisa da CVM na Florida International University não só pelas discussões e várias contribuições à este trabalho, mas também pela forma como me acolheram durante minha visita. Em especial, quero agradecer ao Dr. Andrew Allen pela sua disposição em ajudar no esclarecimento de minhas dúvidas.

Agradeço também ao colega Roberto Vito Rodrigues Filho pela sua ajuda no início deste trabalho e a todos os demais colegas que me acompanharam e me incentivaram durante esses anos.

Também devo agradecer à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior e à Fundação de Amparo à Pesquisa do Estado de Goiás pelo apoio financeiro para o desenvolvimento deste trabalho.

Resumo

Sousa, Gustavo Cipriano Mota. **Desenvolvimento de Máquinas de Execução para Linguagens de Modelagem Específicas de Domínio: Uma Estratégia Baseada em Engenharia Dirigida por Modelos**. Goiânia, 2012. 104p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Abordagens de engenharia de software dirigida por modelos propõem o uso de modelos como uma forma de lidar com a crescente complexidade das aplicações atuais. Por meio de linguagens de modelagem específicas de domínio, essas abordagens visam elevar o nível de abstração utilizado na engenharia de software, possibilitando que usuários que conheçam o domínio de negócio sejam capazes de construir aplicações. As aplicações são definidas como modelos que são então processados de forma automatizada por mecanismos capazes de executá-los. Essa abordagem tem sido aplicada em domínios como comunicação e redes elétricas inteligentes para possibilitar a construção de aplicações por meio de modelos que podem ser criados e modificados em tempo de execução. Nessa abordagem, modelos são processados por máquinas de execução específicas de domínio, que encapsulam o conhecimento necessário para executá-los. No entanto, a aplicação dessa mesma abordagem em outros domínios exige que novas máquinas de execução sejam implementadas por completo, o que exige um grande esforço de implementação. Neste trabalho, apresentamos uma abordagem dirigida por modelos para a construção dessas máquinas de execução de modelos. Essa abordagem propõe um metamodelo que captura os aspectos independentes de domínio de uma classe particular de máquinas de execução de modelos, os quais descrevem aplicações baseadas no provimento de serviços a partir de um conjunto heterogêneo de recursos. A partir do metamodelo proposto, podem ser construídos modelos que definem máquinas de execução para domínios específicos, as quais são capazes de executar modelos descritos na linguagem de modelagem específica do domínio em questão.

Palavras–chave

Engenharia Dirigida por Modelos, Linguagens de Modelagem Específicas de Domínio, Metamodelagem

Abstract

Sousa, Gustavo Cipriano Mota. **Model-driven development of Domain-Specific Execution Engines**. Goiânia, 2012. 104p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

The combination of domain-specific modeling languages and model-driven engineering techniques hold the promise of a breakthrough in the way applications are developed. By raising the level of abstraction and specializing in building blocks that are familiar in a particular domain, it has the potential to turn domain experts into application developers. Applications are developed as models, which in turn are interpreted at runtime by a specialized execution engine in order to produce the intended behavior. In this approach models are processed by domain-specific execution engines that embed knowledge about how to execute the models. This approach has been successfully applied in different domains, such as communication and smart grid management to execute applications described by models that can be created and changed at runtime. However, each time the approach has to be realized in a different domain, substantial re-implementation has to take place in order to put together an execution engine for the respective DSML. In this work, we present a generalization of the approach in the form of a metamodel that captures the domain-independent aspects of runtime model interpretation and allow the definition of a particular class of domain-specific execution engines which provide a high-level service upon an underlying set of heterogenous set of resources.

Keywords

Model-Driven Engineering, Domain-Specific Modeling Languages, Metamodeling

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Em abordagens tradicionais de desenvolvimento, software é comumente implementado por meio de codificação em linguagens de programação de propósito geral. Apesar de amplamente empregadas, estas linguagens se baseiam em conceitos relacionados à plataforma de implementação e não ao problema a ser resolvido. Esta distância semântica entre o problema em questão e as abstrações disponíveis nas linguagens é uma das causas da complexidade envolvida na especificação de uma solução para um determinado problema [26], o que por sua vez exige um grande esforço durante o desenvolvimento de software.

Com o intuito de amenizar esta complexidade, abordagens dirigidas por modelos propõem o uso de modelos como um meio de preencher a lacuna entre o domínio do problema e a plataforma de implementação. Para isto, essas abordagens empregam modelos que são construídos a partir de abstrações próximas ao domínio do problema e processados de forma automatizada em abstrações da plataforma de implementação.

O termo *Model-Driven Engineering* (MDE) é utilizado para descrever esse conjunto de abordagens que utilizam modelos como os principais artefatos no processo de engenharia de software [50]. Nessas abordagens, o uso de modelos não se limita à documentação ou ao projeto do software, mas também se presta ao seu desenvolvimento. Assim sendo, a principal atividade de desenvolvimento passa a ser a modelagem utilizando construções próximas ao domínio do problema, em substituição à atividade de codificação baseada em construções da plataforma de implementação.

Não obstante, abordagens de MDE não restringem o uso de modelos à etapa de desenvolvimento dentro do ciclo de vida de software. De forma geral, o principal objetivo dessas abordagens é abstrair dos desenvolvedores as complexidades essencialmente tecnológicas, que guardam mais relação com as capacidades da plataforma de implementação do que com o problema a ser resolvido. Nessa perspectiva, modelos também podem ser empregados para representar aspectos da execução do software, permitindo o emprego de abstrações mais apropriadas para monitorar ou adaptar um sistema em tempo de execução [9].

Devido a essas capacidades, essas abordagens tem sido propostas como uma

forma de lidar com a crescente demanda por aplicações complexas, notadamente aquelas que envolvem elementos de computação distribuída em ambientes heterogêneos e em constante mudança, apresentando requisitos de disponibilidade e segurança, adaptabilidade e evolução, dentre outros [26, 14]. A construção desse tipo aplicação empregando uma abordagem tradicional exige um grande esforço, sendo comumente desempenhada por especialistas em desenvolvimento de software. O emprego de abstrações próximas ao domínio do problema, possibilitado pelas tecnologias de MDE, abre espaço para um novo cenário, onde especialistas de domínio, ou até mesmo usuários finais podem construir aplicações complexas através da manipulação de modelos de alto nível.

Para a construção destes modelos são empregadas linguagens especificamente projetadas para capturar os conceitos de um determinado domínio. Os modelos, por sua vez, são transformados de forma automatizada em construções da plataforma de implementação. No entanto, para que o processamento desses modelos seja realizado com um alto grau de automatização é necessário que essas transformações incorporem conhecimento específico de domínio [26]. Assim sendo, a construção de linguagens específicas de domínio e mecanismos projetados para processá-las é necessária para se obter os benefícios previstos pelas abordagens de MDE. Essa infraestrutura possibilita que modelos abstratos, descritos diretamente pelos usuários, sejam executados de forma automatizada.

Apesar dessa abordagem trazer um grande benefício para a construção de aplicações em um determinado domínio de negócio, um grande esforço ainda é exigido na construção dos mecanismos de transformação ou plataformas voltadas para a execução de modelos específicos desse domínio. Esses mecanismos são comumente construídos em linguagens de programação de propósito geral, o que exige um grande esforço em sua codificação. Além disso, por incorporarem conhecimento específico de domínio, novos mecanismos de processamento precisam ser codificados para cada nova linguagem.

A despeito disso, diferentes classes de aplicações em domínios diversos podem apresentar necessidades similares de processamento. O processamento de modelos de alto nível usualmente envolve a comparação e transformação de modelos, avaliação de políticas, adaptação em tempo de execução, entre outras operações. Logo, a aplicação de uma abordagem dirigida por modelos na construção desses mecanismos específicos de domínio que compartilham características comuns pode trazer benefícios interessantes.

Ao se aplicar tal abordagem, é possível ampliar o uso de modelos para além da descrição de aplicações, possibilitando também a descrição da plataforma para execução dessas aplicações. Além de simplificar a construção dessas plataformas, esta abordagem também facilita sua evolução com o intuito de atender ao surgimento de novos requisitos, a mudanças na plataforma de implementação, ou à evolução da linguagem específica de domínio correspondente.

Uma categoria interessante de aplicações que podemos destacar está relacionada ao provimento de serviços a partir de um conjunto heterogêneo de recursos. Uma aplicação desta categoria conta com um conjunto de recursos que apresentam diferentes características, e tem como objetivo fornecer um serviço de alto nível ao usuário, capaz de abstrair a complexidade envolvida em sua realização. Assim sendo, tais aplicações precisam ser capazes de se auto-gerenciar, abstraindo do usuário detalhes da escolha, configuração, monitoramento, e utilização de recursos.

A Máquina Virtual de Comunicação (*Communication Virtual Machine* - CVM) é um exemplo de plataforma para construção e execução de aplicações que se enquadram nessa categoria [22], destinada à realização de serviços de comunicação.

Como uma plataforma dirigida por modelos, a CVM atua a partir do processamento de modelos descritos em uma linguagem específica de domínio chamada Linguagem de Modelagem de Comunicação (*Communication Modeling Language* - CML). Um modelo em CML descreve um cenário de comunicação, incluindo participantes, tipos de dados e tipos de mídia de um serviço de comunicação a ser realizado. Detalhes sobre os dispositivos ou tecnologias de comunicação que serão efetivamente utilizados para estabelecer a comunicação não precisam ser descritos em um modelo em CML.

A CVM é considerada uma plataforma de comunicação centrada no usuário devido ao alto nível de abstração de seus modelos, que podem ser facilmente construídos por usuários finais [18]. A partir de um determinado modelo descrito em CML, a CVM é capaz de fornecer o serviço desejado de forma automatizada. Além disso, um modelo em CML pode ser modificado durante o curso de uma sessão de comunicação, desencadeando adaptações da CVM para atender ao cenário atualizado descrito pelo modelo modificado. Isso qualifica os modelos em CML como modelos de desenvolvimento e de tempo de execução [9].

Para realizar um serviço de comunicação de forma automatizada e transparente ao usuário, a CVM precisa realizar diversas operações, desde transformações e negociação de modelos até a seleção e gerenciamento de provedores de comunicação. A CVM conta com uma arquitetura organizada em camadas que possuem responsabilidades bem definidas e encapsulam os principais aspectos envolvidos na realização da comunicação.Essa mesma estratégia também é empregada para possibilitar a construção de aplicações de gerenciamento dos elementos de uma rede elétrica inteligente local por usuários finais. Com esse intuito, foi projetada a Linguagem de Modelagem de Redes Elétricas Locais (*Microgrid Modeling Language*, MGridML) que possibilita ao usuário criar modelos que descrevem fontes e cargas de energia em uma *micro-grid* e preferências para utilização desses recursos. A partir desses modelos a Máquina Virtual de Redes Elétricas Locais (*Microgrid Virtual Machine*, MGridVM) é capaz de monitorar e gerenciar os elementos de uma *micro-grid* para atender aos requisitos definidos pelo modelo em execução [6]. O

emprego dessa estratégia nesses domínios demonstra o seu potencial para a construção de plataformas destinadas ao provimento de serviços descritos por modelos a partir de um conjunto heterogêneo de recursos.

Neste trabalho, nos apoiamos nas soluções empregadas pela CVM e MGridVM para propor o uso de uma abordagem dirigida por modelos para o desenvolvimento de máquinas de execução de modelos específicas de domínio, destinadas à construção e execução de aplicações que se enquadram na categoria descrita anteriormente. Ao fazê-lo, buscamos uma forma sistematizada de capturar as soluções empregadas por essas máquinas de execução, de forma que possam ser reutilizadas em outros domínios de aplicação.

Assim sendo, o presente trabalho propõe o uso de modelos como um meio de construir máquinas virtuais projetadas para a execução de uma determinada linguagem de modelagem específica de domínio. Para isso, propomos a construção de um metamodelo que permita descrever modelos que representem máquinas de execução que se enquadram na categoria descrita. Além disso, ao adotarmos CVM e MGridVM como referências, propomos o emprego de uma arquitetura baseada em camadas com responsabilidades bem definidas, de forma independente do domínio de negócio. Dentro dessa proposta, o metamodelo para descrição dessas máquinas de execução deve apresentar construções relativas às responsabilidades identificadas para as camadas que as compõem. A partir do metamodelo proposto, poderão ser criados modelos que descrevem máquinas de execução para suporte a linguagens de modelagem específicas de domínio.

Com o intuito de demonstrar a viabilidade desta abordagem, construímos um metamodelo independente de domínio que permite a construção de uma das camadas da arquitetura empregada. Essa camada, a mais inferior da arquitetura, tem como responsabilidade o gerenciamento dos recursos subjacentes disponíveis. Denominada *intermediador de serviços*, essa camada fornece à camada superior uma interface capaz de abstrair a heterogeneidade e os detalhes envolvidos na utilização dos recursos.

Baseado no metamodelo proposto, e com o intuito de validá-lo, construímos um modelo que equivale à implementação existente dessa camada na CVM. Além disso, desenvolvemos um ambiente de execução para validar e avaliar o modelo construído. Os resultados obtidos nessa execução foram comparados com a implementação original dessa camada na CVM.

1.1 Objetivos

O objetivo geral deste trabalho é propor o uso de abordagens dirigidas por modelos na construção de máquinas de execução de modelos capazes de prover serviços a partir de um conjunto heterogêneo de recursos. As máquinas construídas a partir desta

abordagem são projetadas para processarem modelos de alto nível descritos em uma linguagem de modelagem específica de domínio, e que podem ser criados e modificados em tempo de execução.

Além disso, com o intuito de demonstrar a viabilidade da proposta, este trabalho tem os seguintes objetivos específicos:

- Propor uma arquitetura independente de domínio de negócio para máquinas de execução de modelos projetadas para execução de aplicações que fornecem um serviço descrito por um modelo de alto nível, a partir de um conjunto heterogêneo de recursos.
- Projetar um metamodelo para definição da camada de intermediação de serviços que integra a arquitetura da máquina de execução.
- Desenvolver um ambiente de execução que possibilite a execução de modelos construídos em conformidade com o metamodelo projetado.
- Fazer uma validação inicial da proposta por meio da construção de um modelo, em conformidade com o metamodelo proposto, que descreve uma camada de intermediação de serviços equivalente à camada de intermediação de comunicação em rede que integra a CVM.
- Estabelecer uma comparação entre a camada de intermediação de comunicação em rede construída por meio do modelo e a camada equivalente que integra a CVM, implementada em uma linguagem de programação de propósito geral.

1.2 Metodologia

Com o intuito de atingir os objetivos estabelecidos, a seguinte metodologia de pesquisa foi adotada:

- Revisão do estado de arte de abordagens dirigidas por modelo, incluindo sua relação com outras abordagens, conceitos, técnicas e ferramentas associadas.
- Estudo da plataforma CVM e sua interpretação à luz das abordagens dirigidas por modelo.
- Refatoração da camada de intermediação de serviços presente na CVM com o intuito de isolar os elementos independentes de domínio.
- Elaboração de um metamodelo independente de domínio que forneça abstrações apropriadas para a descrição dos elementos específicos de domínio identificados na fase anterior.
- Construção de um modelo equivalente à camada de intermediação de serviços da CVM.

- Avaliação do modelo construído em relação à implementação existente da camada de intermediação de comunicação em rede.

1.3 Contribuições

As contribuições desse trabalho incluem:

- A aplicação de uma abordagem dirigida por modelos na construção de mecanismos específicos de domínio destinados à execução de modelos de alto nível que podem ser criados e modificados em tempo de execução. Ao propor o uso de uma abordagem dirigida por modelos para esse fim, o presente trabalho visa trazer os mesmos benefícios obtidos na construção de aplicações para a construção de suas plataformas de execução, reconhecendo que mesmo sendo específicas de domínio, ainda existem várias similaridades entre essas plataformas.
- A construção de um metamodelo e um ambiente de execução que, associados, permitem construir a camada de intermediação de serviços de máquinas de execução de modelos definidas em conformidade com a arquitetura geral utilizada. Essa contribuição, por sua vez, representa uma etapa em direção à realização da visão completa da abordagem proposta, a qual envolveria a extensão do metamodelo para abranger as demais camadas que constituem a arquitetura.

Este trabalho foi apresentado no *7th International Workshop on Models@run.time* [51] e publicado na forma de artigo nos anais do evento.

1.4 Organização da dissertação

Os capítulos seguintes descrevem, além da abordagem proposta e sua implementação, as abordagens e conceitos associados. O Capítulo 2 introduz os conceitos associados à abordagem proposta e essenciais para a sua compreensão. O Capítulo 3 descreve a proposta de execução de modelos na forma de uma arquitetura genérica para construção de máquinas de execução e como essa arquitetura pode ser especializada para domínios específicos. O Capítulo 4 detalha o metamodelo proposto para construção da camada de intermediação de serviços da arquitetura. O Capítulo 5 detalha a implementação de um ambiente para a execução de camadas de intermediação de serviços descritas por meio do metamodelo proposto, e o Capítulo 6 demonstra como o metamodelo, juntamente com o ambiente de execução podem ser utilizados para construir uma camada de intermediação de serviços para a CVM. O Capítulo 7 discute outros trabalhos existentes que estão relacionados à construção de máquinas para execução de DSMLs. Por fim, o Capítulo 8 resume este trabalho, discutindo suas contribuições, limitações e possíveis trabalhos futuros.

CAPÍTULO 2

Referencial teórico

Neste capítulo revemos os conceitos sobre os quais o presente trabalho foi desenvolvido. Na Seção 2.1 tratamos dos princípios de abordagens de engenharia de software dirigida por modelos e seus fundamentos associados. A Seção 2.2 fornece uma visão geral da máquina virtual de comunicação (CVM), descrevendo seu comportamento e sua arquitetura. Por fim, a Seção 2.3 descreve a arquitetura de computação autônoma e políticas, como tecnologias utilizadas para a construção de sistemas auto-gerenciáveis.

2.1 Engenharia dirigida por Modelos

O termo *Model-Driven Engineering* (MDE) é geralmente empregado para identificar o conjunto de abordagens que promovem o uso de modelos como os principais artefatos no processo de engenharia de software [48, 26]. Nessas abordagens o uso de modelos não se limita à documentação ou compreensão de um sistema de software, mas pode abranger todas as áreas da engenharia de software, incluindo desde o seu desenvolvimento, até sua operação e manutenção.

O emprego de abordagens dirigidas por modelo evoluiu a partir da necessidade de lidar com a crescente complexidade encontrada no desenvolvimento de sistemas de software. O avanço nas tecnologias de rede e processamento observado nas últimas décadas possibilitou a construção de aplicações cada vez mais elaboradas, que comumente envolvem elementos de computação distribuída em ambientes heterogêneos que estão em constante mudança.

O desenvolvimento dessas aplicações utilizando abordagens tradicionais de desenvolvimento, baseadas na codificação em linguagens de programação, exige um grande esforço. Apesar da notável evolução nas linguagens de programação, suas construções continuam sendo em grande parte baseadas nas capacidades fornecidas pelo ambiente computacional, e assim não guardam relação direta com o problema a ser resolvido. A distância semântica entre o problema a ser resolvido e a plataforma utilizada na sua implementação é considerada um dos principais fatores que dificultam o desenvolvimento das aplicações atuais [48, 26].

Abordagens de MDE visam reduzir essa distância promovendo o uso de abstrações mais próximas ao domínio do problema, que são sistematicamente transformadas em construções do ambiente computacional. Nessas abordagens, modelos são utilizados como um meio de descrever um sistema de software sob diferentes aspectos e níveis de abstração, sendo então processados de forma automatizada em elementos da plataforma de implementação.

A abordagem de arquitetura dirigida por modelos (*Model-Driven Architecture*, MDA) [41] surgiu com o objetivo de definir funcionalidades do sistema de forma independente da plataforma de implementação, buscando a interoperabilidade e portabilidade do sistema. A MDA pode ser considerada como a primeira iniciativa a propor um conjunto de princípios e padrões para o uso sistematizado de modelos. O termo *Model-Driven Engineering*, usado mais recentemente, propõe o uso de modelos de forma mais abrangente, e engloba além de MDA outras iniciativas como *Model-Driven Development* (MDD) [50, 7], e *Model-Integrated Computing* (MIC) [53], entre outras.

A maior parte das abordagens de MDD propõe a construção de linguagens de modelagem e transformações capazes de transformar modelos descritos por meio dessas linguagens em artefatos da plataforma de implementação. A abordagem MIC pode ser considerada como uma aplicação de MDD, e propõe a construção de ambientes de desenvolvimento específicos de domínio que compreendem linguagens de modelagem específicas de domínio, transformadores e ferramentas que auxiliam na construção de aplicações naquele domínio.

Apesar de empregarem modelos com diferentes objetivos, essas abordagens apresentam várias similaridades, sendo por vezes sobrepostas. A engenharia dirigida por modelos busca sistematizar o uso de modelos, de forma que possam ser empregados com diversos objetivos em todas as atividades da engenharia de software. Dessa forma, o princípio básico da engenharia dirigida por modelos propõe o uso de modelos como objetos de primeira classe, utilizados na construção de todos os artefatos de software [8, 13].

Apesar de ser considerada uma abordagem recente de engenharia de software, a engenharia dirigida por modelos se baseia em idéias de várias outras abordagens como linguagens específicas de domínio, programação orientada por linguagens, programação generativa e programação dirigida por domínio, entre outras [13]. Nesse contexto, a engenharia dirigida por modelos surge como uma tentativa de sistematizar o uso de modelos, apoiando-se em técnicas e princípios já existentes.

2.1.1 Modelos

Um modelo é geralmente utilizado como um meio de representar simplificadamente algum aspecto de um sistema. Um modelo descreve um sistema de acordo com uma perspectiva, ocultando informações que não estão relacionadas à natureza dessa perspectiva. Modelos podem ser empregados para capturar informações sobre um sistema existente, ou para descrever um sistema a ser construído. Um sistema complexo, por sua vez, é representado por meio de um conjunto de modelos, que capturam diferentes aspectos em diversos níveis de abstração.

Em abordagens de MDE, um modelo não é apenas um diagrama ou uma outra representação visual, mas sim uma definição formal dos conceitos a serem representados [49]. Nessas abordagens, o uso de modelos vai além da análise e projeto de um sistema, podendo ser utilizado em todas as atividades relacionadas à engenharia de software. O emprego de modelos formalmente definidos possibilita que modelos representando visões abstratas de um sistema sejam processados de forma automatizada, resultando em artefatos de implementação.

Modelos são formalizados por meio de linguagens de modelagem, cujas construções são definidas por um metamodelo. Um metamodelo, que também é um modelo, formaliza os conceitos fornecidos pela linguagem de modelagem, e define como estes estão relacionados.

2.1.2 Metamodelagem

Conforme mencionado na seção anterior, modelos são construídos em conformidade com uma linguagem de modelagem. Assim como outras linguagens formais, uma linguagem de modelagem é definida por sua sintaxe e semântica. A sintaxe de uma linguagem de modelagem pode ainda ser divida em sintaxe concreta, que representa sua notação textual ou gráfica, e sintaxe abstrata que representa os conceitos disponíveis na linguagem e como eles se relacionam. A semântica, por sua vez, também é tratada separadamente como semântica estática, que define critérios para que modelos sejam considerados válidos, e a semântica dinâmica que define o significado dos modelos ao serem executados.

A sintaxe abstrata e a semântica estática de uma linguagem de modelagem são comumente formalizadas por meio de um metamodelo. Um metamodelo é um modelo que descreve um conjunto de abstrações e como estas se relacionam. As abstrações descritas por um metamodelo representam as construções que irão compor a linguagem por ele formalizada. Assim sendo, os elementos do metamodelo devem estar relacionados ao nível de abstração da linguagem por ele descrita e também ao aspecto do sistema a ser descrito por meio dessa linguagem.

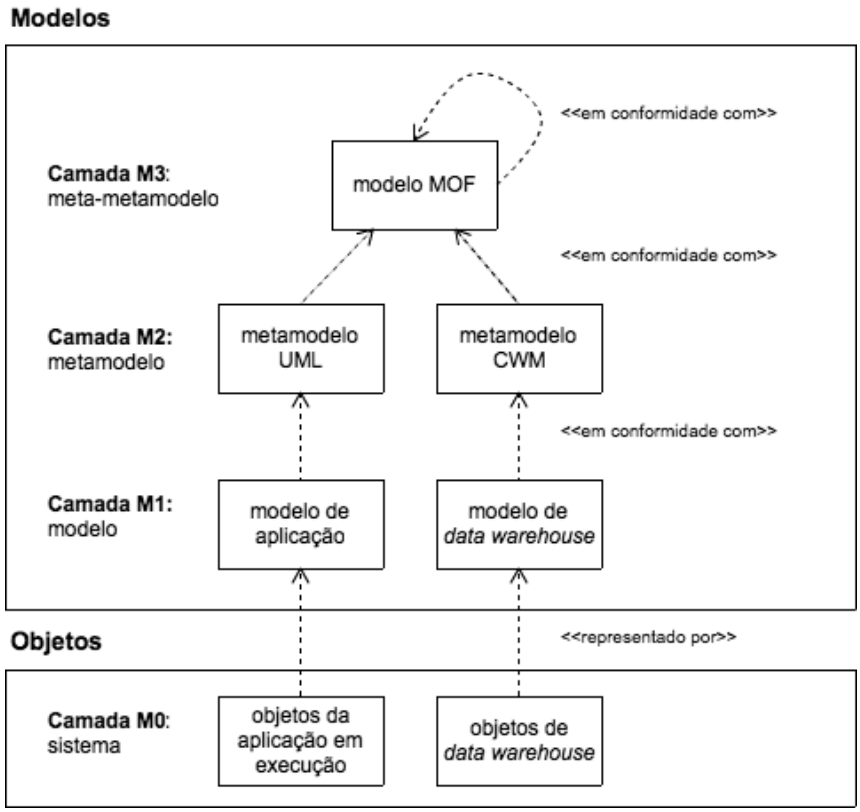


Figura 2.1: *Arquitetura de metamodelagem da MOF.*

Por se tratar de um modelo, um metamodelo também é construído por meio de uma linguagem de modelagem. Esta meta-linguagem, por sua vez, também é definida através de um metamodelo, que neste caso é chamado de meta-metamodelo.

A construção de modelos e metamodelos foi padronizada pelo OMG (*Object Management Group*) por meio de uma arquitetura de metamodelagem denominada *Meta-Object Facility* (MOF) [46]. A especificação da MOF descreve uma arquitetura de quatro camadas, ilustrada na Figura 2.1, onde os elementos em uma determinada camada são definidos como instâncias de elementos da camada imediatamente superior. A camada mais superior, denominada M3, também integra a especificação da MOF, e representa o meta-metamodelo, que pode ser utilizado para a construção de metamodelos. O modelo da camada M3, também denominado de modelo MOF, é formalizado por meio de suas próprias abstrações, o que elimina a necessidade de um outro nível de metamodelagem.

A partir do modelo da camada M3, é possível construir metamodelos que descrevem a sintaxe abstrata e semântica estática de linguagens de modelagem para diversos fins. Exemplos de linguagens de modelagem definidas a partir do modelo MOF incluem a *Unified Modeling Language* (UML) e *Common Warehouse Metamodel* (CWM), também padronizadas pelo OMG. Essas linguagens integram a camada M2 da arquitetura da MOF, e são utilizadas para construir os modelos que integram o nível M1. Os modelos da camada M1, por sua vez, são representações de objetos existentes no nível M0, que

podem ser considerados como objetos do mundo real. Além disso, a especificação da MOF ainda define um padrão para representação de modelos no formato XMI (*XML Metadata Interchange*).

O *Eclipse Modeling Framework* (EMF) [24] é um *framework* de modelagem que fornece um conjunto de ferramentas para construção e processamento de modelos. O EMF inclui um metamodelo denominado Ecore, que representa um subconjunto do modelo MOF correspondente às abstrações existentes em linguagens de programação orientadas a objetos. O Ecore pode ser utilizado para construir modelos que representam o metamodelo de uma linguagem. Utilizado dessa forma, o Ecore assume a mesma função do nível M3 da arquitetura da MOF. O desenvolvimento do EMF e do Ecore influenciou a padronização da MOF 2.0, que passou a contar com um metamodelo equivalente ao Ecore, denominado *Essential MOF* (EMOF) [46].

2.1.3 Linguagens específicas de domínio

Uma linguagem específica de domínio (*Domain-Specific Language*, DSL) é uma linguagem de programação ou especificação, projetada com o objetivo de prover abstrações para a resolução de problemas em um determinado domínio.

Diferente de linguagens de propósito geral, que podem ser empregadas para um grande conjunto de tarefas em variados domínios, linguagens específicas de domínio normalmente estão restritas a um conjunto limitado de tarefas em um determinado domínio. Não obstante, DSLs apresentam um maior poder de expressividade na solução de problemas no seu domínio específico [42, 56].

Os ganhos em expressividade tem como principal benefício o aumento da produtividade na construção de soluções. Além disso, o uso de conceitos e notações apropriadas próximas ao domínio pode permitir que especialistas de domínio, ou até mesmo usuários finais sejam capazes de construir aplicações [39].

Um domínio pode ser considerado como uma área de interesse delimitada. Podemos categorizar domínios como técnicos e de negócio [55, 33]. Domínios técnicos estão diretamente relacionados aos aspectos tecnológicos de um software, como persistência, interfaces gráficas, bancos de dados etc. Domínios de negócio, por sua vez, estão relacionados com atividades econômicas ou profissionais, como telecomunicações, sistema bancário, seguros, vendas, entre outros. Apesar disso, essa separação é nebulosa, sendo que a delimitação de um domínio deve ser definida de acordo com o problema a ser resolvido.

A abordagem de engenharia dirigida por modelos fornece um conjunto de princípios e ferramentas para construção de linguagens de modelagem, que se adequam perfeitamente à construção de DSLs [38, 14]. Através do uso de técnicas de metamodelagem é

possível especificar as construções dessas linguagens, que nesse caso são geralmente denominadas linguagens de modelagem específicas de domínio (*Domain-Specific Modeling Languages*, DSMLs).

Ao mesmo tempo, grande parte do sucesso no emprego de abordagens dirigidas por modelos está associada ao emprego de DSMLs, sendo consideradas tecnologias que apresentam grande sinergia [12, 33, 38]. O uso de abordagens dirigidas por modelos em um domínio específico permite atingir um alto grau de automatização no processamento de modelos.

Apesar de metamodelos poderem ser empregados na descrição de DSMLs, sua capacidade limita-se à descrição da sintaxe abstrata e da semântica estática. A definição da sintaxe concreta é simples, e existem várias ferramentas que permitem descrever sua notação, seja ela textual ou gráfica, e o seu mapeamento para a sintaxe abstrata. A formalização da semântica dinâmica de uma DSML, no entanto, é bem mais complexa, e atualmente não há um método estabelecido como padrão [38].

A semântica dinâmica de uma DSML é, na maior parte das situações, descrita informalmente, por meio de linguagem natural, e codificada na implementação do mecanismo responsável pelo processamento da DSML. A formalização da semântica dinâmica de linguagens de modelagem tem sido objeto de estudos, e tem o potencial de trazer vários benefícios, principalmente associados à geração automática de ferramentas [12].

Existem várias abordagens para descrição da semântica dinâmica de linguagens de modelagem as quais podem ser agrupadas nas seguintes categorias [17]:

- Por tradução, onde a semântica da linguagem é definida por meio da tradução de seus conceitos em conceitos de uma outra linguagem cuja semântica já é estabelecida.
- Operacional, que descreve como os modelos descritos por meio da linguagem são processados. A semântica operacional é incorporada ao interpretador ou mecanismo responsável pelo processamento e execução dos modelos.
- Por extensão, que possibilita a descrição da semântica como uma extensão de uma outra linguagem.
- Denotacional, que é definida por meio de um mapeamento entre as construções da linguagem e um domínio semântico que engloba elementos matemáticos representando elementos primitivos de semântica.

Uma das abordagens proeminentes para definição de semântica de DSMLs por meio de tradução é denominada ancoragem semântica [16]. Nessa abordagem, a definição da semântica e se dá pelo mapeamento dos elementos da DSML em modelos computacionais, cuja semântica é bem definida, como máquinas de estado abstratas, sistemas de eventos discretos, autômatos etc. Outra abordagem baseada em tradução

emprega regras de reescrita [1], que descrevem como os elementos de um modelo podem ser sucessivamente reescritos, até que representem elementos da linguagem alvo.

Outra técnica também empregada na formalização da semântica dinâmica de DSMLs consiste em incorporar comportamento ao metamodelo (*behavior weaving*) [44]. Esta abordagem propõe uma forma de estender a linguagem de meta-metamodelagem para possibilitar a construção de metamodelos que incorporem a semântica operacional.

2.1.4 Modelos em tempo de execução

Modelos também têm sido empregados com sucesso como um meio para lidar com a complexidade envolvida no gerenciamento de sistemas em tempo de execução. O gerenciamento de sistemas que precisam ser monitorados, adaptados e evoluídos durante sua execução é realizado por meio da inspeção e manipulação de estruturas que representam o sistema em tempo de execução.

Nesse cenário, o uso de modelos em tempo de execução tem como intuito prover representações mais apropriadas dos elementos de um sistema em execução, ocultando a complexidade de sua plataforma de execução. Assim, os modelos representam interfaces que podem ser utilizadas por agentes para monitorar e manipular sistemas em execução. De acordo com essa visão, os modelos são o meio pelo qual desenvolvedores ou outros agentes podem compreender, configurar e modificar o comportamento e estrutura de um sistema em execução [26].

Modelos de tempo de execução estão diretamente relacionados à reflexão computacional, visto que ambos buscam definir representações que refletem um sistema em execução e que possuem uma relação de causalidade com o mesmo. No entanto, diferente da reflexão cujas representações estão associadas à plataforma de execução do sistema, modelos em tempo de execução empregam representações mais próximas ao domínio do problema [9]. Assim como na reflexão computacional, modelos em tempo de execução podem ser empregados para capturar a estrutura ou comportamento de um sistema em execução, ou ainda, podem capturar outras facetas do sistema relacionadas ao domínio do problema.

2.2 A Máquina Virtual de Comunicação

A Máquina Virtual de Comunicação (*Communication Virtual Machine*, CVM) é uma plataforma para especificação e realização de serviços de comunicação [22]. Sendo uma plataforma dirigida por modelos, a CVM funciona por meio do processamento de modelos descritos em uma linguagem de modelagem específica de domínio, denominada Linguagem de Modelagem de Comunicação (*Communication Modeling Language*, CML)

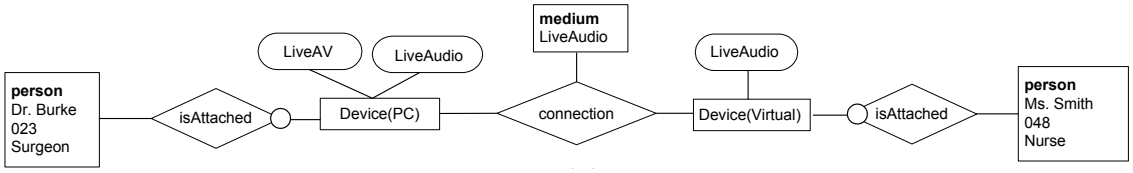


Figura 2.2: *Exemplo de modelo CML [22].*

[18]. Diferente de muitas abordagens dirigidas por modelo que se apoiam na geração de código, os modelos CML são diretamente interpretados pela CVM.

A CVM é considerada uma plataforma centrada no usuário devido ao alto nível dos modelos descritos por meio da CML, que podem ser facilmente construídos por usuários especialistas de domínio, ou até mesmo por usuários finais. A CML é uma linguagem declarativa, por meio da qual são descritos os participantes, dados, e tipos de mídia envolvidos em uma comunicação a ser realizada. Informações associadas às tecnologias e dispositivos efetivamente utilizados na realização da comunicação, não são descritos em um modelo CML. A Figura 2.2 apresenta um exemplo de modelo descrito utilizando a representação gráfica da CML. O modelo em questão define uma sessão de comunicação entre dois participantes por meio de áudio em tempo real.

A partir de um modelo CML, a CVM é capaz de realizar o serviço de comunicação descrito de forma automática, sem a necessidade de intervenção do usuário. Um modelo CML também pode ser modificado ao longo do curso de uma comunicação, sendo a CVM capaz de identificar essas mudanças e adaptar a comunicação em andamento para atender aos novos requisitos definidos. Devido a esta natureza, os modelos em CML podem ser considerados como modelos de desenvolvimento e de tempo de execução [57].

Para tornar isso possível, a CML e a CVM incorporam conhecimento relacionado à realização de serviços de comunicação, permitindo que modelos CML se limitem a descrever os aspectos específicos do serviço a ser realizado. A CVM se baseia em uma arquitetura de camadas que encapsulam as tarefas necessárias à realização de um serviço de comunicação. Um modelo CML fornecido pelo usuário (ou aplicação) é sucessivamente processado e transformado pelas camadas da CVM, até que sejam geradas chamadas aos componentes que efetivamente realizam os serviços solicitados. A Figura 2.3 ilustra as camadas que integram a CVM, descritas logo adiante.

- Interface de Comunicação com o Usuário (*User Communication Interface*, UCI). A UCI representa a interface para interação com a CVM, provendo meios para a definição e gerenciamento de modelos em CML. É por meio dessa interface que usuários ou aplicações podem solicitar a realização de uma sessão de comunicação descrita por um modelo, podendo também ser notificados sobre eventos ocorridos durante o seu andamento. Além disso, a UCI também conta com um ambiente

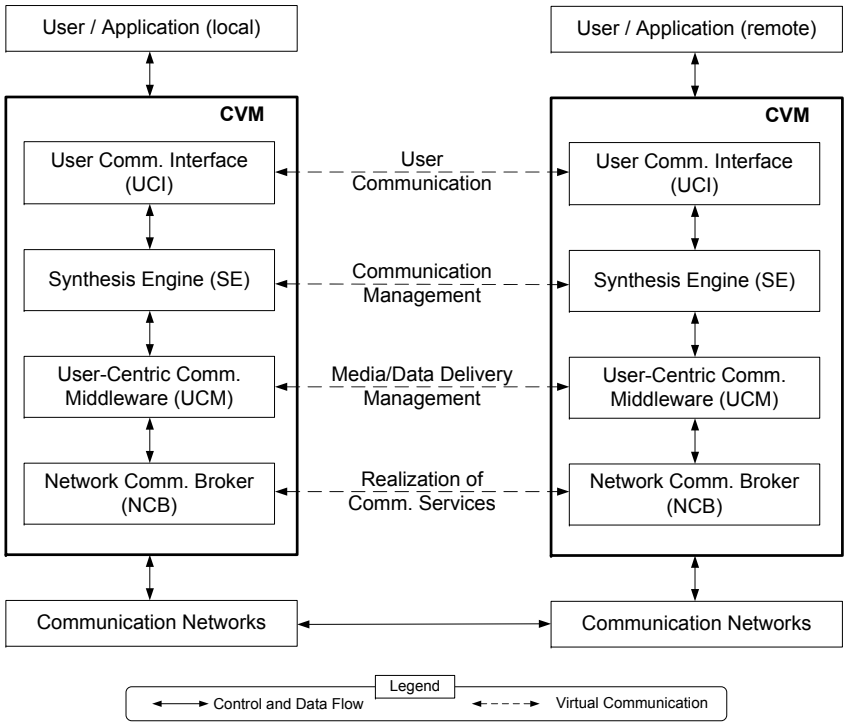


Figura 2.3: *Arquitetura em camadas da CVM [22].*

de modelagem que permite que usuários construam modelos CML por meio de manipulação direta da representação gráfica.

- Mecanismo de Síntese (*Synthesis Engine*, SE). O mecanismo de síntese é responsável por duas principais tarefas: a negociação de um modelo em CML recebido da UCI com os demais participantes de uma comunicação; e a transformação desse modelo em um *script* de controle da comunicação, o qual contém a lógica para o estabelecimento da comunicação. O mecanismo de síntese implementa a semântica operacional da CML [58], sendo o responsável por transformar os modelos declarativos descritos através da CML em *scripts* executáveis pela camada seguinte.
- *Middleware* de Comunicação Centrado no Usuário (*User-centric Communication Middleware*, UCM). A camada de *middleware* da CVM, tem como função executar o *script* de comunicação recebido do mecanismo de síntese e coordenar as sessões de comunicação. Além disso, cabe a esta camada garantir que sejam aplicadas as políticas de segurança, qualidade de serviço, entre outras relativas à comunicação.
- Intermediador de Comunicação em Rede (*Network Communication Broker*, NCB). A camada NCB provê uma interface de comunicação independente de tecnologia para ser utilizada pela camada de *middleware*. O intermediador tem como função blindar a camada superior da heterogeneidade e complexidade envolvidas na interação com os componentes que efetivamente realizam a comunicação. Assim, a NCB recebe requisições da camada UCM e intermedia o acesso aos *frameworks* de comunicação para que realizem a solicitação recebida.

A Máquina Virtual de Redes Elétricas Locais Inteligentes (*Microgrid Virtual Machine*, MGridVM) [6] também utiliza essa arquitetura para a execução de modelos construídos por usuários para descrever requisitos para a utilização de energia elétrica em um ambiente local. A partir desses modelos a MGridVM gerencia a utilização de fontes de energia, cargas e baterias.

2.2.1 O Intermediador de Comunicação em Rede

Como introduzido no Capítulo 1, neste trabalho nos concentramos na camada de intermediação de serviços da arquitetura de máquinas de execução de modelos. Para melhor compreender a função da camada de intermediação de serviços, nesta seção descrevemos em mais detalhes o Intermediador de Comunicação em Rede (NCB) que integra a CVM.

A camada NCB foi projetada com o intuito de fornecer uma interface de serviços de comunicação independente da tecnologia de comunicação empregada, capaz de abstrair para a camada superior a heterogeneidade e dinâmica envolvidas na utilização de diversos provedores alternativos de comunicação [22]. Para isso, o intermediador de comunicação deve ser capaz de se auto-gerenciar, adaptando-se a mudanças no ambiente de rede, nos provedores de comunicação, ou nas solicitações recebidas da camada de *middleware*.

Apesar de prever o uso de diversos provedores de comunicação, a versão inicialmente desenvolvida da NCB comportava apenas um provedor diretamente incorporado à sua implementação, limitando a qualidade e tipos de serviços de comunicação disponíveis. Em trabalhos seguintes [3], a NCB foi ampliada para possibilitar a integração, e auto-configuração de vários *frameworks* provedores de comunicação como Skype, Smack, Asterisk etc. Para o auto-gerenciamento da camada foi adotada a arquitetura de computação autônoma, e proposto o uso de políticas como meio de guiar decisões sobre a seleção de *frameworks* de comunicação [5] .

2.3 Sistemas auto-gerenciáveis

Um sistema auto-gerenciável é aquele capaz de se adaptar de forma automática às mudanças nos seus requisitos e ambiente operacional. O objetivo do auto-gerenciamento é reduzir a necessidade do uso de controles manuais para manter um sistema de acordo com os seus objetivos quando da ocorrência de mudanças em seu ambiente operacional ou requisitos [36].

A construção de sistemas auto-gerenciáveis apresenta vários desafios cujas soluções envolvem diversas áreas de pesquisa, incluindo computação sensível ao contexto,

inteligência artificial, adaptação dinâmica, entre outras. Nesse contexto, arquiteturas de auto-gerenciamento surgem como uma abordagem para integração de soluções para os desafios envolvidos na realização da visão de sistemas auto-gerenciáveis [60, 36].

Esta seção apresenta um resumo da arquitetura de computação autônoma proposta pela IBM para construção de sistemas auto-gerenciáveis e do uso de políticas para guiar o gerenciamento de sistemas. Ambos conceitos são empregados pela CVM e neste trabalho para prover capacidades de auto-gerenciamento.

2.3.1 Computação autônoma

A crescente complexidade dos sistemas baseados em software modernos tem dificultado não só a sua construção de aplicações, mas também a sua implantação, configuração e operação. A provisão de serviços que atendam às demandas atuais dos usuários muitas vezes exige a construção de aplicações capazes de integrar vários vários recursos.

Esses recursos, geralmente heterogêneos, podem incluir, além de componentes de software, bancos de dados, servidores de aplicações, aplicações legadas e, até mesmo, subsistemas completos. Além disso, com a expansão das tecnologias de rede e Internet, esses recursos podem estar distribuídos em inúmeros dispositivos. Nessa situação, até mesmo a implantação e gerenciamento de aplicações pode tornar-se uma atividade intratável, mesmo para especialistas em tecnologia da informação.

As dificuldades relacionadas à integração e gerenciamento desses sistemas complexos levaram ao surgimento de iniciativas [34, 61, 43] que propõem uma abordagem sistematizada para a construção de sistemas capazes de se auto-gerenciar. O emprego de mecanismos de computação autônoma, proposto pela IBM [34], visa dotar os sistemas de software de capacidades similares àquelas presentes no sistema nervoso autônomo humano, que controla as funções vitais do organismo de forma inconsciente ao ser humano. De forma análoga, um sistema de computação autônoma é capaz de ajustar sua operação de acordo com mudanças no seu ambiente e nas demandas de usuários.

Conforme essa visão, um sistema autônomo deve possuir as seguintes capacidades de auto-gerenciamento:

- Auto-configuração: Capacidade de um sistema de automaticamente adaptar para atender mudanças em seu ambiente.
- Auto-otimização: Capacidade de monitorar os recursos e realizar ajustes para aumentar a sua eficiência.
- Auto-proteção: Capacidade de identificar e antecipar ataques, tomando as ações necessárias para se proteger.

- Auto-recuperação: Capacidade de identificar problemas como falhas ou mal-funcionamento em recursos e tomar as ações necessárias para garantir a manutenção dos serviços fornecidos.

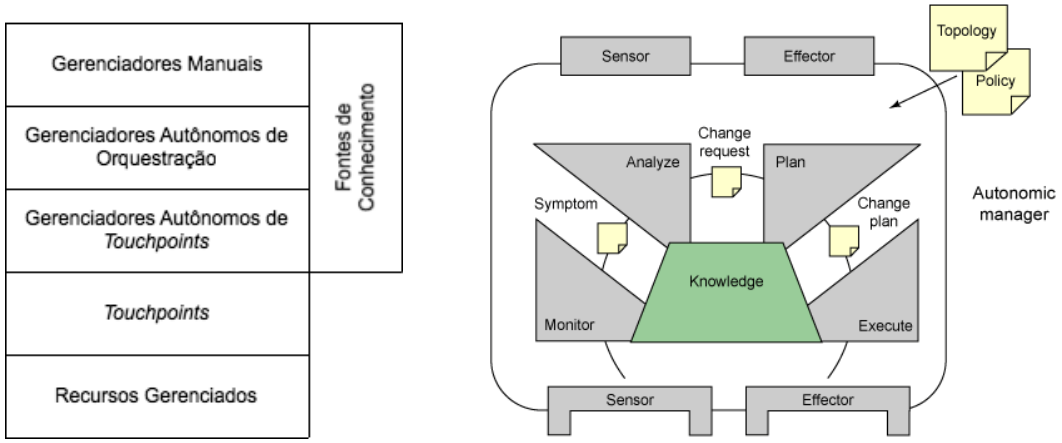
Para realizar essa visão, foi proposta uma arquitetura baseada em camadas para a construção de sistemas autônomos [28] . A Figura 2.4(a) ilustra essa arquitetura, cujo elemento principal é o gerenciador autônomo. A camada mais inferior contém os recursos a serem gerenciados, que podem ser componentes de hardware ou software, e podem incorporar capacidades de auto-gerenciamento. A camada imediatamente acima representa os componentes de acesso aos recursos, denominados *touchpoints*. Um *touchpoint* representa uma interface padrão para interação com os recursos apresentando o comportamento de sensor e atuador para um ou mais recursos. Enquanto sensores tem como função observar o estado dos recursos gerenciados, atuadores são utilizados para efetuar mudanças nos mesmos.

O gerenciamento dos recursos é realizado por meio de gerenciadores autônomos e manuais. Gerenciadores autônomos de *touchpoint* atuam diretamente sobre os recursos, interagindo com eles por meio de seus *touchpoints*. Gerenciadores autônomos de orquestração gerenciam outros gerenciadores autônomos, coordenando diferentes grupos de recursos ou diferentes tarefas de gerenciamento. Por fim, um gerenciador manual fornece uma interface para que um profissional de tecnologia da informação possa intervir manualmente na gerência dos recursos. Ao realizarem sua função, os gerenciadores autônomos e manuais podem obter e compartilhar informações, que neste caso são chamadas de conhecimento e mantidas em fontes de conhecimento. Em um sistema autônomo são considerados como conhecimento tipos de dados particulares como sintomas, políticas, solicitações e planos de mudança.

O elemento principal da arquitetura de computação autônoma é o gerenciador autônomo, que implementa um ciclo de controle que realiza as funções de monitoramento, análise, planejamento e execução (MAPE). Essas funções, conjuntamente, são responsáveis por monitorar os recursos e eventualmente tomar a ação apropriada em resposta a alguma mudança. Durante a sua execução essas funções são influenciadas pelos dados mantidos em uma fonte de conhecimento e interagem por meio da troca de dados como sintomas, solicitações de mudança e planos de mudanças, também tratados como conhecimento.

A Figura 2.4(b) mostra uma visão conceitual do gerenciador autônomo, onde podem ser identificados os elementos que implementam as funções do ciclo de controle:

- Monitor, que é responsável por coletar, agregar e filtrar informações coletadas de recursos gerenciados por meio de sensores, identificando sintomas relevantes para a análise do estado do recurso.



(a) (b)

Figura 2.4: *(a) Arquitetura de computação autônoma (b) Gerenciador autônomo [28].*

- Analisador, que analisa os sintomas gerados pelo monitor e determina se alguma mudança é necessária, gerando uma solicitação de mudança.
- Planejador, que fornece meios para construção de planos de mudança que determinam quais ações devem ser tomadas em resposta a uma solicitação de mudança recebida.
- Executor, que coloca um plano de mudança em prática, interagindo com os atuadores para executar as ações definidas pelo plano.

Como ilustrado na parte superior da Figura 2.4(b), o gerenciador autônomo também expõe as mesmas interfaces de sensores e atuadores, permitindo que um gerenciador autônomo seja utilizado por um gerenciador autônomo orquestrador. Além disso, políticas podem ser utilizadas para direcionar o comportamento de um gerenciador autônomo, as quais também são consideradas como conhecimento.

A aplicação de abordagens dirigidas por modelos em conjunto com essa arquitetura de computação autônoma pode ser muito benéfica. O uso de técnicas de metamodelagem possibilita a formalização da estrutura do conhecimento utilizado para direcionar o comportamento autônomo. Modelos podem ser empregados na representação de políticas, sintomas, solicitações e planos de mudança, além de outros conhecimentos relacionados aos recursos gerenciados.

2.3.2 Políticas de gerenciamento

O uso de políticas no gerenciamento de sistemas tem como intuito isolar as regras que controlam o gerenciamento de um sistema de sua funcionalidade [11]. Essa separação visa reduzir o esforço exigido na manutenção de sistemas e aumentar a sua flexibilidade e adaptabilidade. Nesse contexto, políticas são utilizadas como um meio

de definir objetivos de alto nível, que direcionam o comportamento do sistema. Devido a isto, o uso de políticas está incorporado a várias arquiteturas e abordagens de auto-gerenciamento de sistemas, incluindo a de computação autônoma.

Políticas podem ser utilizadas para definir regras para se atingir diversos objetivos de gerenciamento, como escalabilidade, desempenho, otimização do uso de recursos, flexibilidade etc. Políticas são definidas por linguagens de definição de políticas, que oferecem meios para especificar regras que podem ser mapeadas para ações de gerenciamento de um sistema. O uso de políticas para auto-gerenciamento de sistemas guarda semelhanças com técnicas de MDE, pois se baseia na construção de representações formais de políticas por meio de linguagens, e na sua aplicação de forma automatizada.

Há uma grande diversidade entre as linguagens para definição de políticas, que variam de acordo com os aspectos a serem gerenciados ou o domínio aplicado. Apesar disso, algumas linguagens de políticas buscam oferecer construções mais genéricas que podem ser utilizadas para uma gama maior de sistemas.

A PDL (*Policy Definition Language*) [40] considera uma política como um conjunto de regras que determinam o comportamento esperado de um sistema e propõe a definição de políticas por meio de regras do tipo Evento-Condição-Ação (ECA). Ponder [21] é uma linguagem orientada a objetos declarativa para especificação de políticas de segurança baseada em papéis e políticas de gerenciamento de propósito geral. A linguagem Ponder estende a abordagem baseada em regras do tipo ECA, que passam a ser utilizadas na composição de outros tipos mais elaborados de políticas, que incluem obrigações, restrições, delegações, papeis, relacionamentos etc. *Policy Management for Autonomic Computing* (PMAC) [2] é uma plataforma de políticas para o gerenciamento de vários aspectos de sistemas distribuídos de larga escala. Na PMAC, uma política é uma regra que contempla condições, ações, prioridade e papel. A plataforma proposta pela PMAC se integra com a arquitetura de computação autônoma proposta pela IBM, sendo o gerenciador autônomo considerado um gerenciador baseado em políticas e, assim, suas funções MAPE operam de acordo com um conjunto de políticas.

O uso de linguagens de políticas mais específicas ou genéricas é uma questão sujeita a debates, assim como na área de linguagens de programação. Entretanto, sob uma ótica mais abrangente de abordagens de MDE, linguagens de política são necessariamente de propósito específico e podem ser consideradas como uma linguagem específica de domínio. Na implementação da camada NCB da CVM, políticas são utilizadas para direcionar a seleção de *frameworks* de comunicação [10, 5].

2.3.3 Considerações finais

Neste capítulo descrevemos a Máquina Virtual de Comunicação (CVM), destacando sua arquitetura e o funcionamento de sua camada de intermediação de serviços. Para isso, revemos os fundamentos de abordagens de engenharia de software dirigida por modelos, que são utilizados na CVM para possibilitar a construção de serviços de comunicação por meio de modelos de alto nível. Além disso, também tratamos dos princípios relacionados à construção de sistemas auto-gerenciáveis, que são empregados na camada de intermediação de serviços da arquitetura da CVM. Os conceitos discutidos neste capítulo tem como intuito fornecer um referencial para a compreensão do presente trabalho.

CAPÍTULO 3

Construção dirigida por modelos de máquinas de execução específicas de domínio

Como introduzido no capítulo anterior, para viabilizar o emprego de modelos de alto nível para a construção e adaptação de aplicações é necessário construir linguagens específicas de domínio e mecanismos capazes de interpretá-las em tempo de execução. Enquanto o emprego de metamodelos permite descrever a sintaxe abstrata e a semântica estática de uma linguagem, sua semântica dinâmica é geralmente incorporada aos mecanismos responsáveis pelo seu processamento. Assim, metamodelos associados a máquinas de execução podem ser utilizados como um meio para definição de DSMLs e para o processamento de modelos descritos por meio delas.

A Seção 3.1 descreve de forma geral uma abordagem para a construção dirigida por modelos de máquinas de execução capazes de realizar serviços descritos na forma de modelos de alto nível. A Seção 3.2 descreve uma arquitetura genérica para máquinas de execução de modelos que se enquadram na categoria descrita na seção anterior. A Seção 3.3, por sua vez, detalha a camada de intermediação de serviços da arquitetura descrita.

3.1 Visão geral

A construção de um metamodelo e uma máquina de execução projetados para uma DSML permite definir completamente a sintaxe e semântica da linguagem. Enquanto o metamodelo descreve a sintaxe e semântica estática da linguagem, sua semântica dinâmica é capturada de forma operacional pela máquina de execução.

A criação de DSMLs de alto nível, baseadas em construções próximas do domínio do problema, possibilita que usuários sejam capazes de construir aplicações complexas. Para tornar isso possível, é preciso que o metamodelo da DSML, assim como a máquina de execução capaz de processá-la, incorporem grande parte do conhecimento específico de seu domínio. Isto, por sua vez, permite que os modelos descritos usando uma DSML possam se limitar aos requisitos específicos das aplicações por eles descritas,

pois as informações relacionadas ao domínio já estão integradas à DSML. Dessa forma, para se obter proveito das técnicas de MDE e DSMLs é preciso que sejam projetados metamodelos e máquinas de execução específicos para os domínios de aplicação a serem utilizados.

Enquanto existem várias técnicas de modelagem padronizadas para a descrição de metamodelos, a construção de máquinas de execução de DSMLs é usualmente realizada por meio da codificação em linguagens de programação de propósito geral. Nesse contexto, o uso de linguagens de programação apresenta as mesmas limitações encontradas na construção de aplicações complexas, pois oferecem construções muito primitivas que não estão relacionadas aos problemas encontrados na construção de máquinas de execução para processamento de modelos.

A construção de máquinas de execução para modelos que podem ser criados e modificados em tempo de execução exibe vários problemas comuns, mesmo que essas máquinas sejam projetadas para diferentes DSMLs. Comparação e transformação de modelos, avaliação de políticas, adaptação em tempo de execução, entre outros são alguns dos problemas que esse tipo de máquina de execução necessita lidar. A solução desses problemas por meio da codificação em linguagens de programação não só exige bastante esforço, mas também reduz a flexibilidade das máquinas de execução construídas.

Esse contexto sugere a aplicação de técnicas de MDE como forma de amenizar a complexidade da construção de máquinas de execução de modelos. Assim sendo, a mesma abordagem utilizada para construção de aplicações é utilizada na construção das plataformas para execução dessas aplicações. O uso de modelos nesse contexto visa não apenas a simplificar a construção dessas máquinas de execução, mas também facilitar a sua evolução com o intuito de atender a necessidades de mudança na sintaxe ou semântica das DSMLs por elas processadas.

Neste trabalho, propomos o emprego de uma abordagem dirigida por modelos para a construção de uma categoria específica de máquinas capazes de executar modelos que podem ser criados e modificados em tempo de execução. Essa categoria, que inclui a CVM [22] e MGridVM [6], contempla máquinas virtuais para execução de DSMLs que descrevem serviços de alto nível realizados a partir de um conjunto heterogêneo de recursos. A realização de serviços descritos por essas DSMLs comumente envolve diversas operações, como transformação e negociação de modelos, avaliação de políticas, recuperação de falhas, seleção e configuração de recursos etc.

Para que máquinas de execução que exibem essas características possam ser definidas por meio de modelos é preciso uma linguagem de modelagem especialmente projetada para isso. No entanto, para que seja vantajoso o emprego de uma linguagem de modelagem para a descrição de uma máquina de execução, esta linguagem deve possuir uma abrangência suficiente para que possa ser utilizada em diferentes domínios de

aplicação. Ao mesmo tempo, tal linguagem não deve ser muito genérica, de forma que não torne a manipulação de modelos tão complexa quanto a codificação em uma linguagem de propósito geral.

Assim sendo, essa linguagem deve conter construções que implementam as operações necessárias para fornecer a partir de um conjunto heterogêneo de recursos, serviços descritos por modelos que podem ser criados e modificados em tempo de execução. Essas operações, que englobam transformação e negociação de modelos, avaliação de políticas, recuperação de falhas, seleção e configuração de recursos, entre outras, podem ser tratados como um domínio técnico. Deste modo, essa linguagem pode ser considerada uma linguagem de modelagem específica do domínio que abrange os aspectos técnicos envolvidos na realização de serviços transparentes de alto nível a partir de um conjunto heterogêneo de recursos. Essa linguagem pode então ser utilizada para construir modelos que descrevem máquinas de execução específicas de domínio, voltadas para o processamento de modelos descritos em conformidade com uma DSML.

3.2 Uma arquitetura para máquinas de execução de modelos

Neste trabalho propomos o uso das soluções empregadas no desenvolvimento da CVM e MGridVM para elaboração dessa linguagem para definição de máquinas de execução que se enquadram na categoria identificada. Assim, essa linguagem incorpora uma arquitetura genérica [19] que pode ser especializada para produzir máquinas de execução específicas de domínio. A arquitetura utilizada é uma generalização da arquitetura em camadas empregada pela CVM e MGridVM e inclui as seguintes camadas:

- Interface com o Usuário (*User Interface* - UI), que provê uma interface externa para utilização da plataforma. Além disso, esta camada possibilita a definição e gerenciamento de modelos;
- Mecanismo de Síntese (*Synthesis Engine* - SE), que possui como principal responsabilidade a transformação de um modelo declarativo fornecido pela UI em uma representação algorítmica a ser executada pela camada inferior;
- *Middleware*, que além de executar as requisições geradas pelo Mecanismo de Síntese, também gerencia os serviços providos pela máquina de execução e as tarefas em execução. Essa também é a camada responsável pela aplicação de restrições de segurança, qualidade de serviço, entre outras específicas do domínio de negócio.
- Intermediador de Serviços (*Service Broker* - SB), que é a camada responsável pelo gerenciamento dos recursos. Assim sendo, essa camada tem como objetivo

prover uma interface de acesso aos recursos de forma independente da tecnologia empregada por estes, provendo um serviço transparente à camada de *middleware*.

Ao nos basearmos na arquitetura da CVM e MGridVM, aproveitamos o conhecimento adquirido em relação à separação de responsabilidades necessárias para a realização de serviços no domínio técnico ao qual ambas pertencem, conforme descrito acima.

A construção de uma linguagem para modelagem de máquinas de execução de modelos, por sua vez, também requer a construção de um metamodelo que forneça abstrações que possam ser utilizadas para descrever máquinas de execução de acordo com a arquitetura descrita. Logo, um modelo elaborado a partir desse metamodelo representa uma especialização da arquitetura genérica proposta visando seu uso em um determinado domínio. A Figura 3.1 ilustra a arquitetura genérica descrita e apresenta a CVM e MGridVM como especializações dessa arquitetura genérica.

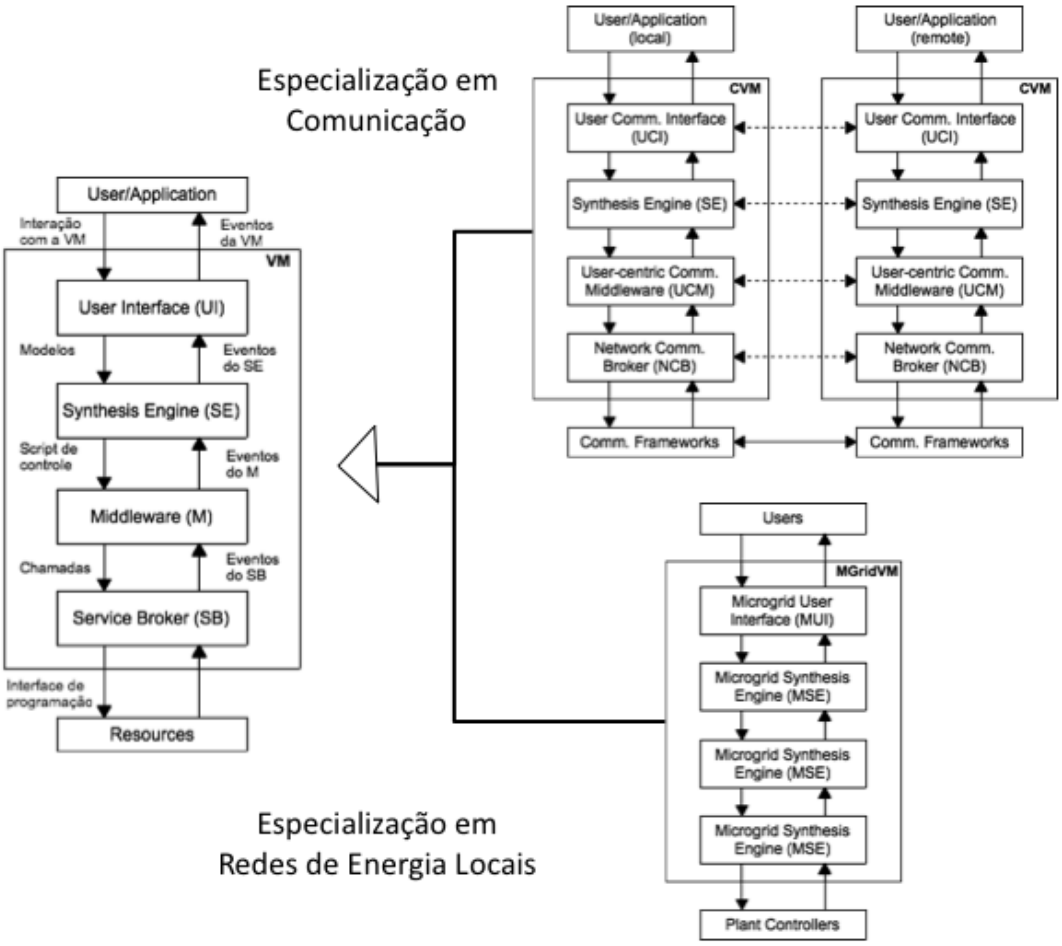


Figura 3.1: *Arquitetura em camadas proposta para construção de máquinas de execução de modelos.*

Devido ao extenso trabalho envolvido na construção desse metamodelo, neste trabalho construímos um metamodelo que engloba apenas os elementos associados à camada de intermediação de serviços da arquitetura genérica de máquinas de execução

de modelos. A construção do metamodelo representa uma etapa em direção à realização da visão proposta neste trabalho, e tem como objetivo investigar sua viabilidade. Para isso, além do metamodelo também construímos um ambiente de execução que define a semântica operacional da linguagem. A seção seguinte descreve em mais detalhes a camada de intermediação de serviços, incluindo suas responsabilidades e como estas são exercidas dentro da arquitetura empregada.

3.3 A camada de intermediação de serviços

O intermediador de serviços é a camada da arquitetura que é responsável pelo gerenciamento dos recursos que serão efetivamente utilizados para realizar os serviços especificados por meio de modelos. Cabe a esta camada disponibilizar uma interface de serviços que abstraia da camada superior a heterogeneidade dos recursos subjacentes. Além disso, o intermediador deve possuir capacidade de auto-gerenciamento, ocultando os detalhes envolvidos na seleção, monitoramento e preparação dos recursos sob sua gerência.

Ao longo deste trabalho consideramos um recurso como um componente de software ou hardware que fornece algum serviço. De acordo com essa definição, aplicações, bancos de dados, controladores de hardware, componentes de software, servidores de aplicação, ou até mesmo complexos sistemas podem ser tratados como recursos. Para ser utilizado por uma camada de intermediação de serviços, um recurso deve expor uma interface que permita a utilização de seus serviços e o seu gerenciamento. Um serviço, por sua vez, é uma funcionalidade bem definida fornecida por meio de um ou mais recursos. Assim sendo, serviços podem ser diretamente fornecidos por um recurso, ou combinados em serviços capazes de fornecer funcionalidades que envolvem o uso de vários recursos.

Como uma interface de serviços, o intermediador se comunica com a camada de *middleware* através de chamadas que podem ser feitas por esta e eventos que podem ser gerados sinalizando para ela alguma situação. Assim, ao definirmos o comportamento dessa camada, precisamos descrever como serão tratadas as chamadas realizadas pela camada superior, e em que situações serão gerados determinados eventos. Mas, além disso, também precisamos descrever como outras tarefas serão realizadas, incluindo o monitoramento e seleção de recursos, manutenção de informações, adaptação da camada, entre outros.

Com isto em mente, neste trabalho construímos um metamodelo que possibilita a modelagem do comportamento necessário para atender as responsabilidades definidas para a camada de intermediação de serviços. O metamodelo em questão contempla a descrição das seguintes tarefas envolvidas no cumprimento destas responsabilidades:

- Gerenciamento de recursos. A camada deve atuar como intermediário na utilização dos recursos, interceptando os eventos por eles gerados e controlando o acesso aos recursos.
- Gerenciamento de estado. A camada deve ser capaz de manter, em tempo de execução, dados associados aos serviços providos, e que podem influenciar o processamento de chamadas e eventos.
- Auto-gerenciamento. As tarefas de auto-gerenciamento da camada incluem o monitoramento dos recursos e dos dados mantidos pela camada, identificando situações que podem exigir uma mudança na organização interna da camada. Ao identificar a necessidade de uma mudança a camada deve determinar a ação que deve ser tomada.

Também construímos um ambiente de execução para auxiliar na execução de modelos construídos a partir do metamodelo. Assim sendo, esse ambiente implementa a semântica operacional da linguagem para descrição da camada de intermediação de serviços. O ambiente de execução possibilita que um modelo definido a partir do metamodelo proposto seja executado.

3.4 Considerações finais

O metamodelo e o ambiente de execução possibilitam que intermediadores de serviços sejam descritos por meio de modelos e utilizados em tempo de execução pela camada de *middleware*. Um intermediador de serviços, por sua vez, se integra às demais camadas de acordo com a arquitetura apresentada para a construção de máquinas de execução de modelos. Os próximos capítulos apresentam, na seguinte ordem: o metamodelo elaborado para a definição de intermediadores de serviços; o ambiente de execução capaz de processar modelos descritos por meio do metamodelo; e um modelo criado a partir desse metamodelo para descrever um intermediador de serviços com comportamento equivalente à camada NCB que integra a plataforma CVM.

CAPÍTULO 4

Metamodelo do Intermediador de Serviços

A camada de intermediação de serviços é responsável pelo gerenciamento dos recursos que serão efetivamente utilizados para provisão de serviços. Um intermediador de serviços gerencia um conjunto heterogêneo de recursos, e provê à camada superior uma interface uniforme, capaz de abstrair as diferenças existentes entre os recursos gerenciados.

Além de disponibilizar uma interface uniforme para uso dos recursos, a camada de intermediação de serviços deve apresentar um certo grau de auto-gerenciamento. O auto-gerenciamento, permite que detalhes relacionados à manutenção de recursos sejam abstraídos para a camada superior. Uma camada auto-gerenciável é capaz de monitorar seus recursos e se adaptar automaticamente para atender aos serviços demandados.

Este capítulo apresenta o metamodelo proposto para definição da camada de intermediação de serviços para máquinas de execução de modelos baseadas na arquitetura proposta. As construções presentes no metamodelo em questão estão diretamente relacionadas às responsabilidades identificadas para essa camada. O capítulo inicia com uma visão geral do metamodelo e prossegue abordando em detalhes os elemento que o compõem.

4.1 Visão geral

Como observado anteriormente, a camada de intermediação de serviços provê uma interface uniforme para utilização de um conjunto heterogêneo de recursos subjacentes. Um intermediador de serviços atua interceptando as solicitações recebidas por meio de sua interface e determinando quais operações devem ser efetuadas sobre os recursos. Além disso, um intermediador também intercepta eventos provenientes dos recursos por ele monitorados e determina se devem ser sinalizados para a camada superior, se uma adaptação na camada é necessária, ou se alguma outra ação deve ser tomada.

No metamodelo proposto, a definição de um intermediador de serviços se dá pela descrição de como ele deve se comportar em resposta a cada uma das possíveis solicitações que podem vir da camada superior e dos eventos que podem ser gerados pelos

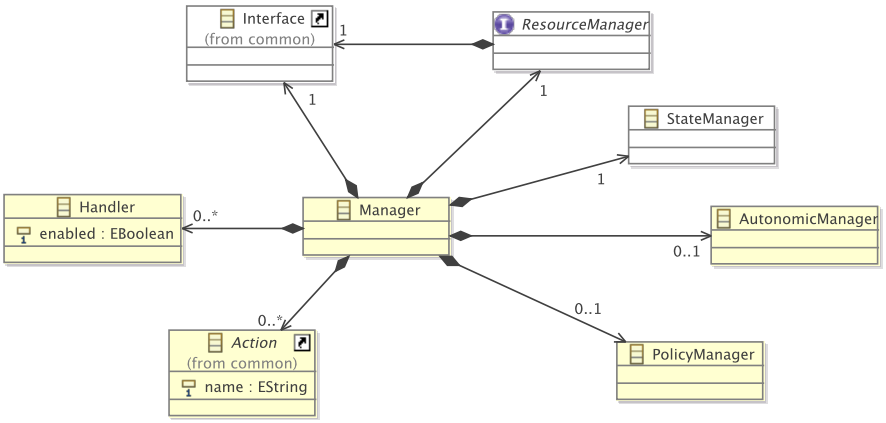


Figura 4.1: *Principais elementos do metamodelo do intermediador de serviços.*

recursos gerenciados. Deste modo, o comportamento de um intermediador de serviços é determinado pela forma como eventos e solicitações recebidos são tratados. Neste trabalho, as solicitações recebidas da camada superior são denominadas *chamadas* e utilizamos o termo *sinal* para designar de forma generalizada um *evento* ou *chamada*. Assim, um sinal pode representar tanto uma solicitação recebida da camada superior quanto um evento gerado por um recurso gerenciado.

Além disso, o metamodelo possui abstrações para descrição de outras características indispensáveis em uma camada de intermediação de serviços. Essas abstrações compreendem a definição de recursos, manutenção de estado, auto-gerenciamento e políticas para seleção de recursos.

Para a construção do metamodelo, foi empregado o metamodelo Ecore, que integra o Eclipse Modeling Framework (EMF) [24]. O emprego do Ecore, que nesse caso atua como um meta-metamodelo, permite o uso de várias ferramentas disponibilizadas pelo EMF que auxiliam na construção, armazenamento e processamento de modelos.

O metamodelo projetado está estruturado em torno de uma classe principal, chamada Manager. Uma instância dessa classe representa um gerenciador de intermediação, que define um escopo para o gerenciamento de recursos e agrupa outros elementos que definem as atribuições específicas da camada de intermediação de serviços.

A Figura 4.1 ilustra os principais elementos que compõem o metamodelo. Nesta figura é possível identificar a classe Manager e os seguintes elementos associados:

- Interface: define as operações disponibilizadas por um gerenciador e os eventos que podem ser gerados por ele.
- Action/Handler: define como sinais serão tratados.
- ResourceManager: define os recursos que serão gerenciados por um determinado Manager, incluindo suas interfaces e como eles são obtidos.

- StateManager: define os tipos de dados que precisam ser mantidos pela camada para prover seus serviços.
- AutonomicManager: define elementos associados ao auto-gerenciamento da camada.
- PolicyManager: define políticas para seleção de recursos e como estas devem ser avaliadas

Uma vez construída uma instância da classe Manager, esta pode ser utilizada como um recurso por outra instância de Manager. Isto possibilita a construção de uma hierarquia de gerenciadores, que podem ser empregados sucessivamente para prover serviços a partir de recursos e de outros gerenciadores de intermediação mais básicos. Além disso, essa característica possibilita a modularização da camada e a reutilização de gerenciadores de intermediação na construção de outros gerenciadores de mais alto nível.

A construção de uma camada de intermediação de serviços se dá a partir da indicação de qual será o seu gerenciador principal que por sua vez pode empregar outros gerenciadores como recursos subjacentes.

As seções seguintes descrevem em detalhes as partes do metamodelo identificadas acima. Como já observado, a abordagem proposta se baseia nas soluções empregadas no desenvolvimento da CVM. Devido a isso, o metamodelo proposto incorpora conceitos independentes de domínio presentes na CVM, e assim sua organização e abstrações guardam semelhanças com aquelas da camada NCB [4].

4.2 Interface

No metamodelo proposto, a interface para utilização de um gerenciador de intermediação é definida por meio de *chamadas* que podem ser realizadas ao gerenciador e *eventos* que podem ser sinalizados por ele. Este tipo de interface segue a mesma abordagem empregada pela CVM para comunicação entre camadas [22]. Dessa forma, a utilização de um gerenciador se dá através da realização de chamadas e tratamento de eventos gerados.

Como a interface de uma camada de intermediação de serviços é definida pela interface do gerenciador principal na hierarquia de gerenciadores, a interação com as camadas superiores ocorre da mesma forma descrita acima. Além disso, as interfaces para utilização dos recursos gerenciados também é descrita da mesma forma, o que possibilita que um gerenciador de intermediação seja tratado como um recurso por parte de outros gerenciadores.

A Figura 4.2 ilustra as classes do metamodelo relacionadas com a descrição de interfaces. A classe Interface é utilizada para descrever a interface de um gerenciador de

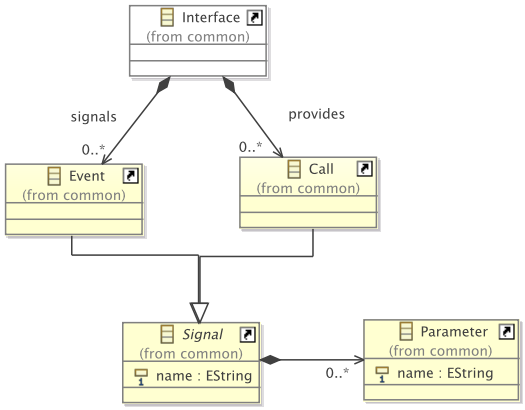


Figura 4.2: *Elementos do metamodelo para descrição de interfaces.*

intermediação ou recurso. Chamadas e eventos que fazem parte da interface são conjuntamente denominados *sinais* e são representados, respectivamente, pelas classes Call e Event. Ambas as classes apresentam como característica comum o fato de possuírem um nome e um conjunto de parâmetros e, por isso, herdam esses atributos da classe Signal. Os parâmetros de um sinal, por sua vez, são identificados por um nome, e são definidos por meio da classe Parameter.

4.3 Tratamento de sinais

Como mencionado na Seção 4.1, o comportamento de uma camada de intermediação de serviços é definido pela forma como esta reage às chamadas recebidas da camada superior e aos eventos sinalizados pelos recursos. O metamodelo proposto disponibiliza construções que permitem definir quais ações devem ser tomadas pela camada em resposta aos sinais recebidos.

O tratamento de sinais na camada de intermediação é definido por meio das classes Signal, Handler e Action. Enquanto os sinais a serem tratados por um gerenciador são descritos como parte da interface do gerenciador e dos recursos gerenciados, os demais elementos (tratadores e ações) são diretamente agrupados no gerenciador. A Figura 4.3 mostra como esses elementos estão organizados no metamodelo.

Uma ação representa uma operação que pode ser executada por um gerenciador, e é definida por meio da classe abstrata Action, que possui as seguintes subclasses:

- CallAction: define uma ação que especifica uma chamada a ser realizada aos recursos gerenciados ou ao próprio gerenciador.
- EventAction: define uma ação que especifica um evento a ser gerado pelo gerenciador.

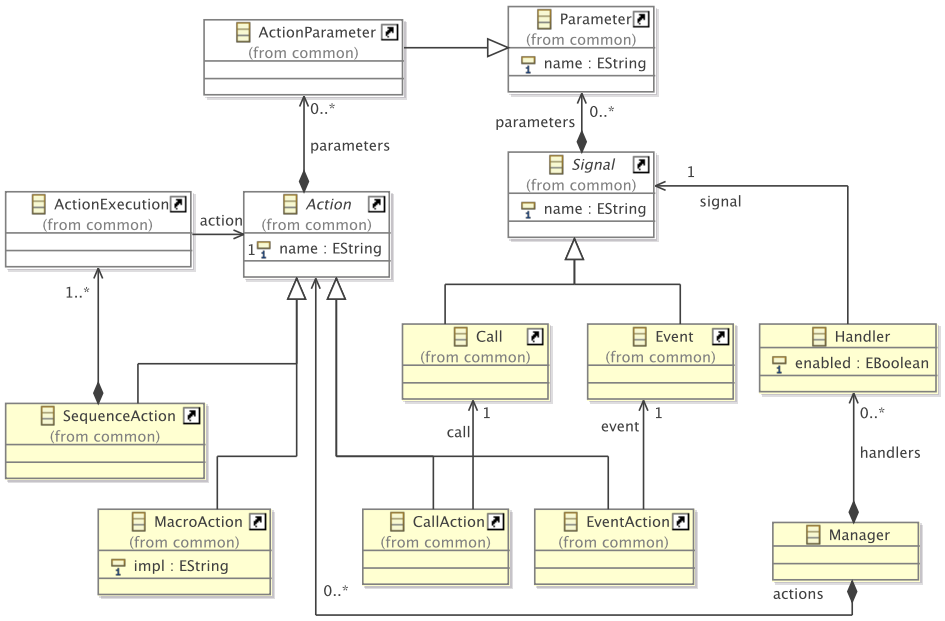


Figura 4.3: *Principais elementos do metamodelo relacionados ao tratamento de sinais .*

- MacroAction: permite ao usuário definir uma classe Java que implementa a ação desejada. Por meio de ações desse tipo, é possível definir ações que implementam uma lógica arbitrária, e executam operações que vão além da intermediação direta de chamadas e eventos.
- SequenceAction: combina uma lista de ações a serem executadas em sequência.

Uma ação também define um contexto necessário para a sua execução por meio de um conjunto de parâmetros. Os parâmetros de uma ação, descritos através da classe ActionParameter, representam as informações necessárias para a execução da operação definida pela ação.

Um tratador de sinais, representado pela classe Handler, é utilizado para indicar a ação que deve ser tomada quando um determinado sinal é identificado. Uma instância de Handler descreve o sinal a ser tratado, se este tratador está habilitado, e a ação a ser tomada.

Além disso, ao definir um tratador, é necessário descrever como serão atribuídos valores aos parâmetros da ação associada. Esta ligação entre o contexto no qual um sinal é identificado e o contexto exigido por uma ação, é definida pela classe ActionExecution. A Figura 4.4 mostra as classes do metamodelo que estão relacionadas com a ligação de contexto.

A classe ActionExecution intermedia a associação entre um tratador de sinal e uma ação, e define um conjunto de associações de parâmetros, descritas através classe ParameterBinding. Uma associação de parâmetro define como será atribuído valor a um

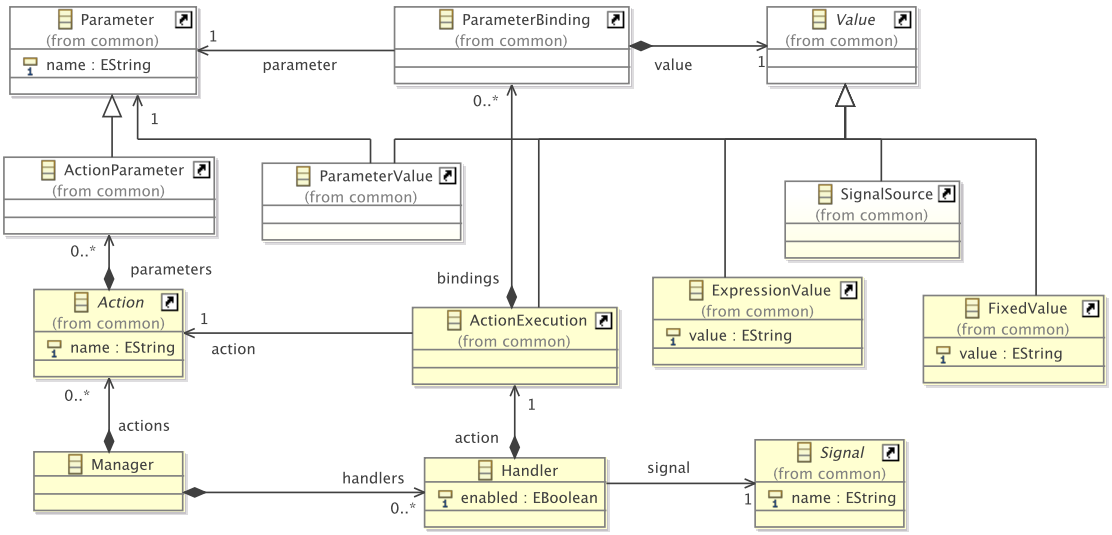


Figura 4.4: *Elementos do metamodelo para associação de parâmetros.*

parâmetro de uma ação. Um parâmetro pode ser associado a diferentes fontes de valores, que são definidas por meio de subtipos do tipo Value.

O metamodelo inclui as classes FixedValue e ParameterValue para definição de fontes com um valor fixo e fontes cujo valor é obtido a partir de um parâmetro do sinal tratado. A classe ExpressionValue, por sua vez, permite que o resultado da avaliação de uma expressão seja utilizado como fonte de valor. Por fim, um outro tipo de fonte de valor, representado pela classe ActionExecution, possibilita que o resultado da execução de uma ação seja atribuído a um parâmetro de uma outra ação. O uso desta fonte de valores permite a definição de tratadores complexos, que utilizam uma combinação de ações para responder a um sinal.

Além de possibilitarem a definição de ações e tratadores de sinal, as construções descritas nesta seção permitem que ações sejam reutilizadas na composição de ações mais complexas, favorecendo a modularização do tratamento de sinais. A utilização dessas construções do metamodelo é melhor ilustrada no Capítulo 6, onde são utilizadas para construir um modelo que representa a camada de intermediação de serviços presente na arquitetura da CVM.

4.4 Recursos

No metamodelo proposto, os recursos gerenciados por uma camada de intermediação de serviços são descritos por meio de um gerenciador de recursos, definido pelo tipo ResourceManager. Um gerenciador de recursos define as interfaces dos recursos gerenciados e como estes podem ser obtidos. A Figura 4.5 apresenta as classes do

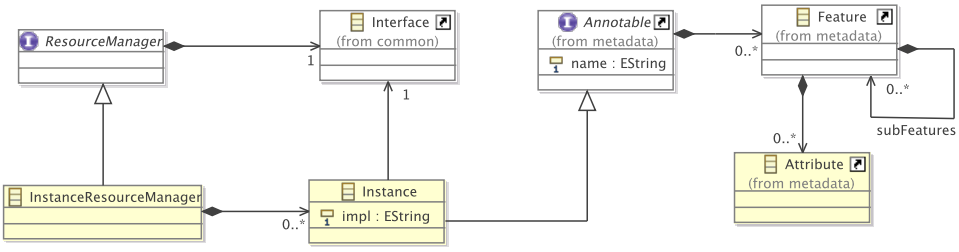


Figura 4.5: *Elementos do metamodelo para descrição dos recursos gerenciados pela camada.*

metamodelo envolvidas na descrição de recursos.

As interfaces dos recursos são descritas através da classe Interface, da mesma forma que a interface de um gerenciador da camada. A obtenção dos recursos, por sua vez, é definida, de acordo com a sua natureza, por meio de classes que são subtipos de ResourceManager. O metamodelo proposto conta com uma classe denominada InstanceResourceManager, que permite a descrição de um conjunto fixo de recursos e suas características. Outras classes que implementam a interface ResourceManager poderiam ser incluídas ao metamodelo para possibilitar a obtenção de recursos de formas mais elaboradas como, por exemplo, através de repositórios de objetos distribuídos.

Uma instância da classe InstanceResourceManager agrupa um conjunto de objetos do tipo Instance. A classe Instance, por sua vez, representa um recurso que pode ser obtido diretamente a partir da instanciação de uma implementação do recurso.

A classe Instance define qual a implementação do recurso que será instanciada em tempo de execução e qual das interfaces descritas pelo ResourceManager correspondente é a interface do recurso. Por fim, a classe Instance implementa a interface Annotable, o que permite que metadados sejam associados aos recursos. Esses metadados podem ser posteriormente utilizados para definição de políticas de seleção de recursos.

4.5 Manutenção de estado

Muitas vezes, o processamento de um determinado sinal pela camada de intermediação de serviços pode depender de fatores que não estão diretamente ligados àquele sinal. A ocorrência de um evento, parâmetros advindos de uma chamada recebida anteriormente, resultados do processamento de outros sinais, entre outros, são informações que podem determinar como um determinado sinal deve ser processado.

Para possibilitar essa variabilidade no processamento de sinais, é preciso que a camada de intermediação de serviços seja capaz de manter dados entre sucessivas ocorrências de sinais. As construções relacionadas à manutenção de estado, presentes no

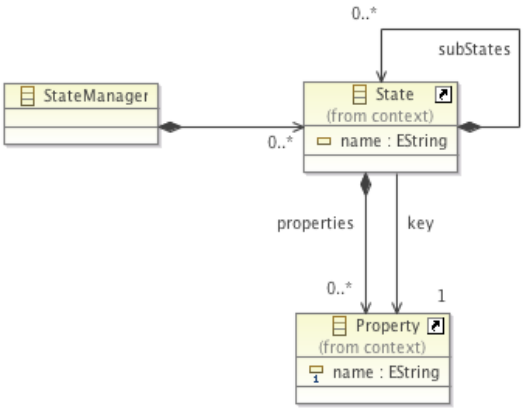


Figura 4.6: *Elementos do metamodelo para descrição dos tipos de dados mantidos pela camada.*

metamodelo proposto, têm como intuito possibilitar a definição de quais tipos de dados precisam ser mantidos pela camada durante a sua execução.

A classe StateManager agrupa os tipos de dados que podem ser mantidos pela camada. Os tipos de dados podem ser descritos através de uma estrutura simples, baseada em atributos e subtipos. Assim como no modelo relacional de dados, a descrição de um tipo de dados também exige a escolha de um atributo chave que identifique unicamente um registro desse tipo.

A Figura 4.6 mostra as classes envolvidas na definição dos tipos de dados a serem mantidos pela camada durante sua execução. A classe State é utilizada para definir um tipo de dados, que agrupa atributos e subtipos. Cada atributo, por sua vez, é definido por meio da classe Property. Os subtipos são definidos a partir da mesma classe State, o que possibilita a definição de tipos de dados compostos.

4.6 Gerenciamento Autônomo

Além de abstrair as diferenças de capacidade entre os recursos existentes, a camada de intermediação de serviços também tem como responsabilidade ocultar da camada superior os detalhes relacionados à dinâmica de utilização dos recursos. Deste modo, para a camada superior, é indiferente o recurso que está sendo utilizado, suas capacidades, como e quando foi selecionado e todos os detalhes envolvidos em sua preparação para realizar as tarefas solicitadas.

Para atender a esta demanda, a camada de intermediação de serviços deve ser capaz de se auto-gerenciar, adaptando-se automaticamente para realizar o serviço solicitado dentro das restrições impostas pelo seu ambiente operacional. O auto-gerenciamento dessa camada envolve o constante monitoramento dos recursos e das solicitações dos

usuários para identificar situações que exigem uma ação e escolher a ação apropriada a ser tomada para que a camada atenda aos serviços solicitados.

Através das abstrações presentes no metamodelo é possível definir o tratamento de eventos gerados pelos recursos e das chamadas realizadas através da interface da camada. Estes mecanismos, associados à manutenção de estado, possibilitam definir como os recursos e solicitações serão monitorados para identificar variados cenários que exigem a execução de uma ação.

Apesar disso, essas abstrações não são apropriadas para a descrição de situações mais complexas, que podem envolver diversos recursos, o estado da camada, dados de chamadas realizadas, entre outros. Com o intuito de facilitar a definição de como se dará o auto-gerenciamento da camada, o metamodelo proposto incorpora um conjunto de abstrações baseadas na arquitetura de computação autônoma proposta pela IBM [28].

Como observado no Capítulo 2, o principal bloco de construção da arquitetura de computação autônoma é o gerenciador autônomo. O gerenciador autônomo implementa as funções MAPE, que permitem monitorar informações sobre recursos gerenciados, analisar essas informações e identificar se uma mudança é necessária. Em caso afirmativo, o gerenciador autônomo ainda cria um plano para realizar a mudança e, em seguida, o executa. A interação entre essas funções é dirigida pela troca de conhecimento, que representa sintomas, solicitações de mudança e planos de mudança.

As abstrações presentes no metamodelo proposto permitem descrever regras que determinam a geração e propagação de conhecimento entre as funções MAPE. Desta forma, ao construir uma camada de intermediação de serviços, devem ser descritas regras para identificação de sintomas, solicitações de mudanças e planos de mudança. Em tempo de execução, o monitor utiliza essas regras para identificar a ocorrência de um sintoma. De forma semelhante o analisador e planejador se baseiam nos sintomas identificados e nas solicitações de mudança geradas para realizarem sua tarefa. A Figura 4.7 ilustra as construções associadas à definição do mecanismo autônomo em um intermediador de serviços.

A classe AutonomicManager agrupa os elementos relacionados ao gerenciamento autônomo de recursos. Essa classe agrupa elementos que descrevem as regras para geração de sintomas, solicitações de mudanças e planos de mudança. A classe Symptom descreve um sintoma a ser monitorado com o intuito de identificar mudanças no contexto da camada. Um sintoma define um conjunto de condições para que este seja identificado. Em tempo de execução, os recursos e o estado da camada são monitorados e as condições de um sintoma avaliadas. Se todas as condições definidas em um sintoma são atingidas uma ocorrência deste sintoma é gerada e passada ao analisador.

As condições agrupadas pela classe Symptom são descritas a partir de expressões. Além das condições, a classe Symptom também define o contexto em que estas

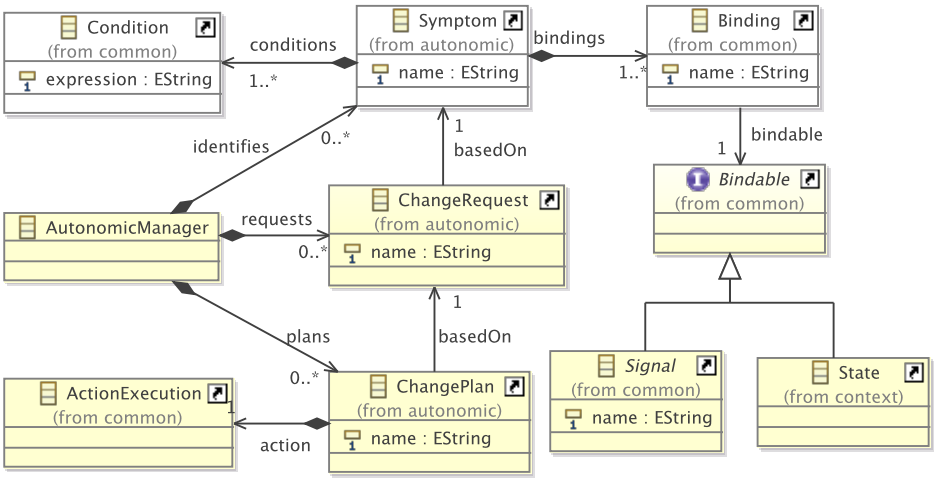


Figura 4.7: *Elementos do metamodelo para descrição do comportamento autônomo.*

expressões serão avaliadas através da classe Binding, que associa um nome (a ser usado na expressão) a um elemento do tipo Bindable. A interface Bindable, por sua vez, é implementada pelas classes Signal e State. Essas abstrações permitem definir condições que envolvam, além de dados de chamadas e eventos, o estado mantido pela camada.

A classe ChangeRequest define uma requisição de mudança que deve ser gerada quando um determinado sintoma é identificado. Associado a uma requisição de mudança podemos ter um plano de mudança definido por instâncias da classe ChangePlan. No metamodelo proposto, um plano de mudança define uma ação a ser executada. Uma ação, por sua vez, pode ser definida por meio dos tipos de ação disponíveis e suas combinações, conforme descrito na Seção 4.3.

4.7 Políticas para seleção de recursos

No metamodelo, políticas são empregadas com o intuito de direcionar a seleção de recursos. As abstrações contempladas pelo metamodelo compreendem, além da descrição de políticas, a definição do momento onde estas devem ser avaliadas e como os resultados dessa avaliação devem ser tratados.

A Figura 4.8 mostra as abstrações relacionadas à utilização de políticas, que se estruturam em torno da classe PolicyManager. Os principais elementos da definição de políticas são representados por meio das classes Policy, PolicyEvaluationContext, PolicyEvaluationPoint, e PolicyEvaluationHandler.

A classe Policy representa uma política, que é definida por meio de um nome e uma condição a ser avaliada sobre um recurso. Uma política também define um valor de negócio que identifica o grau de importância atribuído aos recursos capazes de satisfazer

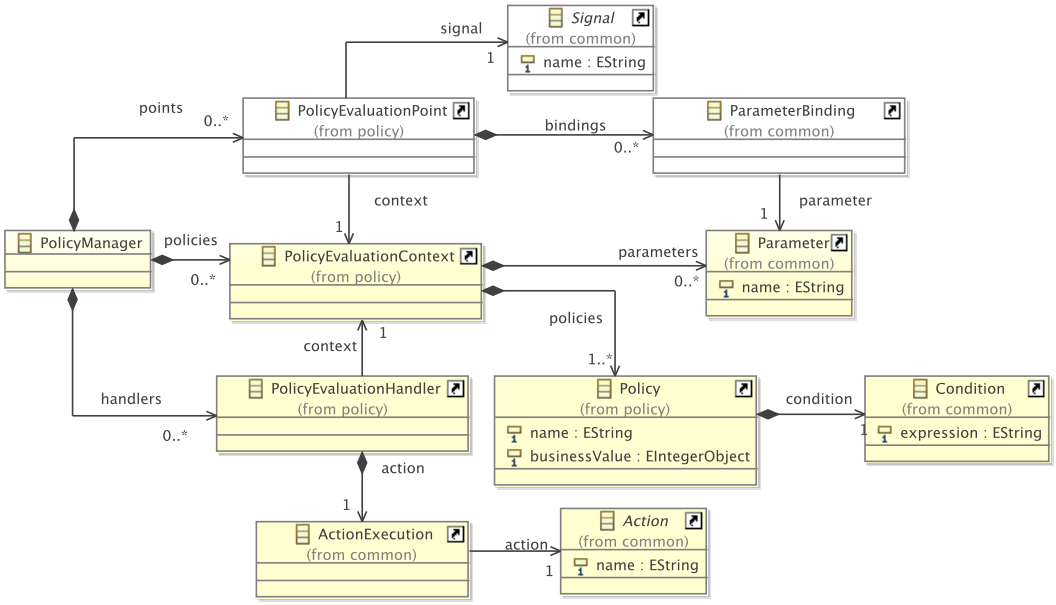


Figura 4.8: *Elementos do metamodelo para descrição de políticas.*

a condição definida. Durante a avaliação das políticas, a seleção de recursos se baseia no seu valor de negócio, acumulado de acordo com a avaliação das políticas definidas.

As políticas são agrupadas por contextos de avaliação de políticas, definidos por meio da classe PolicyEvaluationContext. Um contexto de avaliação define os parâmetros necessários para avaliação das políticas que o integram. Assim, as condições que definem uma política, definidas por expressões, podem envolver metadados definidos sobre os recursos e parâmetros definidos no contexto de avaliação.

Um ponto de avaliação de políticas, definido por meio da classe PolicyEvaluationPoint, especifica um sinal que desencadeia a avaliação de um conjunto de políticas. Por meio deste elemento é possível especificar o momento em que as políticas devem ser avaliadas ou reavaliadas em relação aos recursos. Além disso, um ponto de avaliação de políticas também é responsável por definir como o contexto do sinal é mapeado para o contexto de avaliação de políticas. Esse mapeamento é realizado da mesma forma utilizada na definição de tratadores de sinais, empregando associações de parâmetros descritas por meio da classe ParameterBinding.

Um tratador de avaliação de políticas, representado por meio da classe PolicyEvaluationHandler, define uma ação a ser executada após a avaliação de políticas. Esse tratador tem como função definir como o resultado da avaliação de políticas será utilizado. Um tratador de avaliação de políticas está relacionado a um contexto de avaliação e será acionado sempre que as políticas nesse contexto forem reavaliadas.

4.7.1 Considerações finais

O metamodelo descrito neste capítulo fornece um conjunto de abstrações para a construção de modelos que descrevem uma camada de intermediação de serviços que se enquadra na arquitetura proposta para construção de máquinas de execução de modelos. No Capítulo 5, apresentamos o ambiente de execução desenvolvido para possibilitar a execução de modelos construídos em conformidade com o metamodelo descrito neste capítulo. O Capítulo 6, por sua vez, demonstra como o metamodelo aqui descrito pode ser empregado na construção de um modelo que descreve uma camada de intermediação de serviços equivalente à camada NCB que integra a CVM.

CAPÍTULO 5

Ambiente de execução para o Intermediador de Serviços

Apesar do metamodelo prover abstrações que possibilitam descrever um intermediador de serviços, isto não é suficiente para se obter uma camada executável, capaz de tratar as solicitações recebidas em tempo de execução. Com o intuito de preencher essa lacuna, desenvolvemos um ambiente de execução capaz de carregar um modelo que representa uma camada de intermediação e proceder de acordo com o descrito nesse modelo. Assim, o ambiente de execução desenvolvido implementa a semântica operacional do metamodelo proposto.

O ambiente de execução, desenvolvido na plataforma Java, inclui componentes para a execução de modelos que representam intermediadores de serviços, e uma biblioteca de classes para integração da camada com os recursos a serem gerenciados. Através desses componentes é possível carregar um modelo representado no formato EMF XMI 2.0 e inicializar uma camada de intermediação de serviços de uma máquina de execução de modelos.

Neste capítulo descrevemos a implementação do ambiente de execução, incluindo seus principais componentes, e como eles interagem para prover o comportamento descrito pelo modelo carregado. Na primeira seção apresentamos uma visão geral do ambiente de execução, incluindo o fluxo geral de processamento de sinais e os componentes envolvidos. As seções seguintes descrevem em detalhes a implementação de componentes específicos do ambiente.

5.1 Visão geral

Antes de iniciar a execução de uma camada de intermediação de serviços, as estruturas responsáveis pela realização da camada em tempo de execução precisam ser inicializadas. A inicialização da camada é realizada a partir do processamento de seu modelo, descrito no formato EMF XMI. O ambiente de execução contém um conjunto

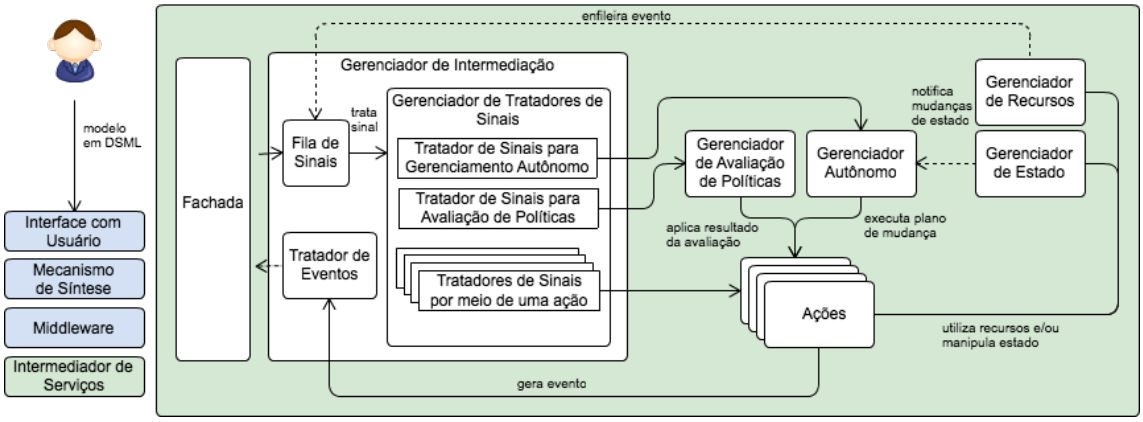


Figura 5.1: *Ambiente de execução para realização de um intermediador de serviços.*

de fábricas de objetos capazes de instanciar, a partir do modelo carregado, os objetos que compõem o ambiente de execução.

No ambiente de execução desenvolvido, um intermediador de serviços está constantemente à espera de chamadas oriundas da camada superior ou de eventos gerados pelos recursos subjacentes. Uma vez identificados, esses sinais são enfileirados para, em seguida, serem processados. Para processar um sinal, o intermediador procura por um tratador de sinais apropriado e encaminha o sinal a esse tratador. De acordo com o modelo em execução, o tratador executa a ação associada ao sinal, podendo interagir com os recursos disponíveis ou manipular o estado mantido pela camada.

A Figura 5.1 ilustra os principais componentes que integram uma camada de intermediação de serviços em tempo de execução. Conforme ilustrado, um modelo de alto nível descrito em uma DSML é processado pelas camadas superiores da máquina de execução de modelos, sendo sucessivamente transformado até chegar à camada de intermediação. Na camada de intermediação, o modelo é recebido como uma sequência de chamadas. Para simplificar a utilização do intermediador de serviços pela camada superior, o ambiente de execução provê uma fachada que permite que a camada superior execute chamadas, e receba notificações de eventos.

O *gerenciador de intermediação* é o principal elemento do ambiente, sendo o responsável por controlar o fluxo de execução. O *gerenciador de intermediação* é o componente responsável por realizar, em tempo de execução, o elemento do metamodelo descrito pela classe Manager. Portanto, em tempo de execução, o contexto de gerenciamento de recursos é definido por um *gerenciador de intermediação*, que pode empregar outros *gerenciadores de intermediação* como recursos subjacentes.

Um *gerenciador de intermediação* serializa o tratamento dos sinais recebidos, que são enfileirados e em seguida processados um a um. O processamento dos sinais é conduzido pelo *gerenciador de tratadores de sinais*, que identifica e executa o *tratador de sinais* apropriado. O *gerenciador de tratadores de sinais* mantém um registro de

tratadores de sinais inicializados de acordo com as definições de tratadores e ações presentes no modelo em execução.

Uma vez identificado o *tratador de sinais* apropriado, a *ação* correspondente é executada. Como ilustrado na Figura 5.1, uma *ação* pode interagir com os recursos por meio do *gerenciador de recursos*, manipular o estado da camada por meio do *gerenciador de estado*, ou gerar eventos para a camada superior por meio do *gerenciador de intermediação*.

Além desses, outros tratadores especializados também são registrados com o objetivo de identificar sinais que podem ser de interesse para as funções de auto-gerenciamento da camada. Esses tratadores interceptam os sinais recebidos e os direcionam para o *gerenciador autônomo* e o *gerenciador de avaliação de políticas*, respectivamente.

O *gerenciador de recursos* mantém referências para os recursos disponíveis para uso da camada, e disponibiliza uma interface para obtenção desses recursos. Além disso, esse gerenciador notifica o *gerenciador de intermediação* sobre eventos gerados pelos recursos. O *gerenciador de estado* disponibiliza uma interface para manipulação de dados, e notifica o *gerenciador autônomo* sobre alterações, que podem disparar o mecanismo de auto gerenciamento da camada.

O *gerenciador autônomo* encapsula as funções MAPE do ciclo de gerenciamento autônomo, que são ativadas pelo *tratador de sinais* correspondente, ou pelo *gerenciador de estado*. A análise dos sinais identificados ou de mudanças no estado da camada pode levar o *gerenciador autônomo* a executar uma *ação*.

De forma similar, o *gerenciador de avaliação de políticas* é ativado pelo *tratador de sinais para avaliação de políticas*, iniciando o processo de avaliação das políticas em relação aos recursos disponíveis. Após a avaliação, uma ação é executada para efetuar chamadas aos recursos ou mudanças nos dados mantidos pelo intermediador, aplicando os resultados da avaliação de políticas.

5.2 Tratamento de sinais

Em uma camada de intermediação de serviços, sinais podem representar chamadas recebidas por meio da interface com a camada superior, ou eventos gerados pelos recursos subjacentes. Os sinais recebidos por um gerenciador da camada de intermediação são tratados por um gerenciador de tratadores de sinais. Este elemento mantém um registro de tratadores de sinais e é responsável por identificar o tratador que responderá a um sinal recebido. Um tratador de sinais, por sua vez, é o responsável por realizar a ação esperada da camada quando da recepção de um sinal.

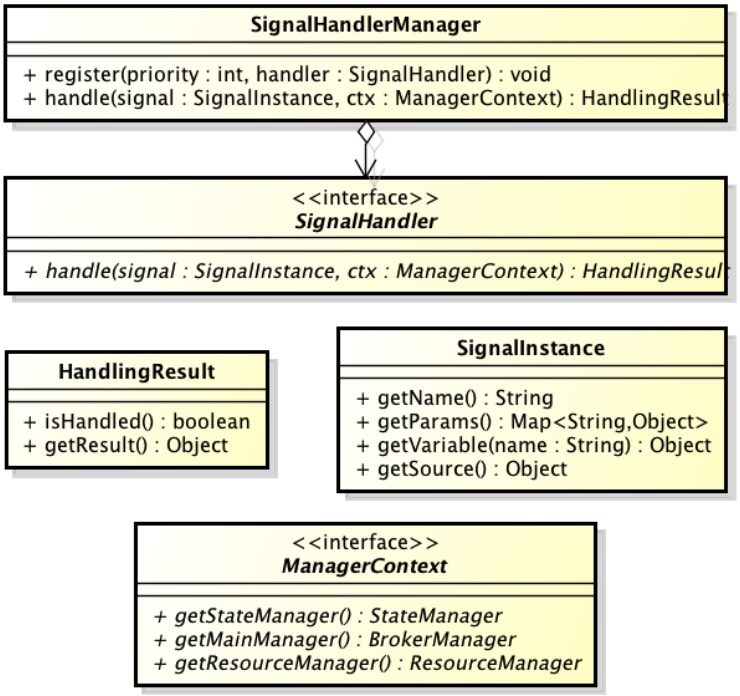


Figura 5.2: *Elementos associados ao tratamento de sinais em tempo de execução.*

O gerenciador de tratadores de sinais é implementado pela classe SignalHandlerManager, que mantém um registro de tratadores de sinais. Um tratador de sinais é representado em tempo de execução pelo tipo SignalHandler. A Figura 5.2 ilustra os tipos associados ao tratamento de sinais.

Cada sinal recebido pelo gerenciador de tratadores é repassado sequencialmente a cada tratador registrado, até que um desses o trate apropriadamente. Neste caso, o tratador retorna ao gerenciador um resultado indicando que o sinal em questão foi tratado. Isto permite que mais de um tratador receba um sinal, mas que uma vez tratado, sua propagação seja interrompida. A ordem de avaliação dos tratadores é estabelecida de acordo com a sua prioridade, definida no momento em que um tratador é registrado junto ao gerenciador de tratadores de sinais.

Um sinal é representado em tempo de execução pela classe SignalInstance, que encapsula a origem, parâmetros e nome do sinal. Ao processar um sinal, um tratador de sinais tem acesso ao contexto de gerenciamento, representado pelo tipo ManagerContext. Através dessa interface, tratadores podem ter acesso aos gerenciadores de intermediação, recursos e estado. O resultado do tratamento de um sinal é representado por meio de um objeto do tipo HandlingResult, que indica se o tratador foi capaz de tratar o sinal em questão e, em caso afirmativo, o resultado desse tratamento.

Como ilustra a Figura 5.3, a classe ActionSignalHandler é uma implementação da interface SignalHandler, utilizada para definir um tratador de sinais que executa uma ação quando um sinal é identificado. Instâncias dessa classe são registradas durante a

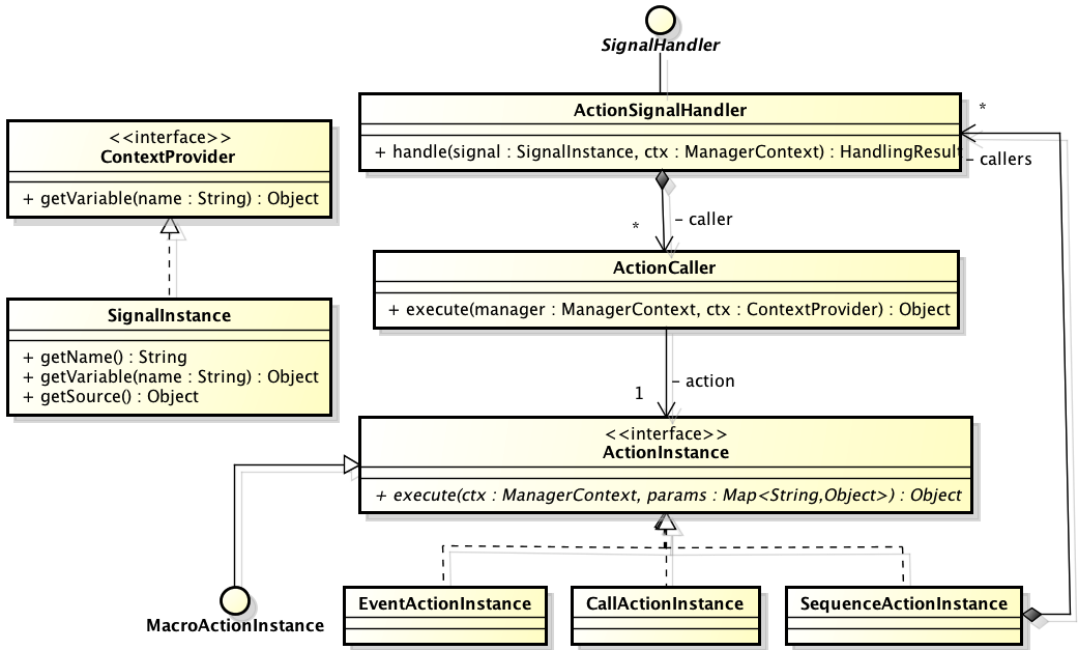


Figura 5.3: *Elementos associados à execução de ações em tempo de execução.*

inicialização da camada, a partir das definições de tratadores existentes no modelo da camada, conforme descrito na Seção 4.3. Ao receber um sinal em tempo de execução, um tratador desse tipo, verifica se o sinal é do mesmo tipo do encontrado em sua definição, e executa a ação correspondente.

Associado a um ActionSignalHandler, há um acionador de ação, implementado pela classe ActionCaller. O acionador é inicializado junto ao tratador, e tem como objetivo realizar o mapeamento do contexto do sinal recebido para o contexto da ação a ser executada. Uma ação, por sua vez, é representada em tempo de execução por um objeto do tipo ActionInstance. A Figura 5.3 também mostra as classes e interfaces relacionadas ao tratamento de sinais por meio da execução de uma ação.

Como já observado, uma ação executa uma determinada operação na camada, que pode envolver a utilização de recursos ou manipulação do estado da camada. Além disso, a execução de uma ação exige um contexto, definido por um conjunto de parâmetros. Essas características podem ser notadas na interface ActionInstance, cujo método execute exige um contexto de gerenciamento, e um conjunto de parâmetros. Por meio do contexto de gerenciamento, a ação pode ter acesso aos gerenciadores de intermediação, recursos e estado. Os parâmetros de uma ação são representados em tempo de execução por um dicionário de nomes e valores.

O acionador de ação, por sua vez, recebe um provedor de contexto, e o utiliza para obter os valores que serão passados como argumentos para uma ação. Um provedor de contexto é um objeto que provê um conjunto de variáveis que podem ser obtidas a partir de seu nome, como pode ser notado na interface ContextProvider. A partir das ligações

de parâmetros definidas no modelo da camada, o acionador de ação é capaz de mapear o contexto provido para os parâmetros esperados pela ação.

Quando uma ação é utilizada para tratar um sinal, o contexto utilizado para a execução da ação é provido pelo próprio sinal. Um sinal é representado em tempo de execução por meio da classe SignalInstance, que implementa a interface ContextProvider. O contexto de um sinal inclui os parâmetros passados juntamente com o sinal e o objeto que gerou esse sinal, identificado pela variável source.

O mapeamento do contexto provido por um sinal para os parâmetros da ação depende do tipo de fonte de valor empregado, conforme descrito na Seção 4.3. Uma fonte de valor define como um valor será obtido do contexto para, em seguida, ser atribuído a um parâmetro da ação que será executada.

Uma fonte de valor fixo, definida em um modelo por meio do tipo FixedValue, retorna o valor fixo definido no modelo em execução e, portanto, independe do sinal recebido. Uma fonte de valor de parâmetro (ParameterValue), por sua vez, obtém o seu valor através do contexto provido pelo sinal. Em tempo de execução, uma busca é feita no contexto do sinal para se obter o valor da variável que representa o parâmetro indicado no modelo. De forma similar, uma fonte de valor de origem do sinal (SignalSource) pode ser utilizada para obter a origem de um sinal, que pode representar um recurso ou a camada superior. A origem de um sinal é obtida por meio de uma consulta ao valor da variável source fornecida pelo contexto do sinal em tratamento.

Enquanto os tipos de fonte de valor citados acima têm o seu comportamento implementado de forma relativamente simples e direta, a implementação de fontes de valor baseadas em expressões, assim como em chamadas a outras ações, é um pouco mais complexa. Os tipos de fontes de valores discutidos acima envolvem, no máximo, uma variável do contexto, enquanto o uso de expressões permite que várias variáveis do contexto provido sejam empregadas para se obter um valor único, que será então passado como um argumento a uma ação.

Na implementação atual do ambiente de execução, as expressões podem ser descritas na linguagem Groovy [27]. Groovy é uma linguagem dinâmica para a plataforma Java que apresenta várias construções que facilitam seu emprego para a descrição de expressões. A linguagem permite o emprego de uma sintaxe simplificada para o acesso a propriedades, utilização de estruturas de dados, e definição de expressões *lambda*. Além disso, essa linguagem pode ser considerada um superconjunto da linguagem Java, pois as construções Java também são construções válidas em Groovy.

Ao definir uma expressão como fonte de valor, é possível empregar todas as variáveis providas pelo contexto provido. Como já observado, quando um sinal é tratado, o contexto consiste dos parâmetros e da origem do sinal. Além do contexto provido, uma expressão também pode ter acesso ao estado da camada por meio de variáveis com o nome

do tipo de dados. Essas variáveis dão acesso a objetos que implementam a interface de um dicionário e permitem a obtenção de um registro de dados de um determinado tipo a partir de seu identificador único. O uso da linguagem Groovy também possibilita que consultas mais complexas sejam realizadas para obtenção de registros. O uso dessas expressões é ilustrado no Capítulo 6, onde são utilizadas na construção de um modelo de intermediador de serviços.

O uso de uma fonte de valor baseada na execução de uma ação tem como objetivo atribuir a um parâmetro de uma ação o resultado da execução de outra ação. Isto é descrito no modelo por meio do tipo ActionExecution, que define, além da ação a ser executada, o mapeamento de contexto. No ambiente de execução, todas as fontes de valores são avaliadas antes da execução de uma ação e, assim sendo, as fontes do tipo ActionExecution são previamente processadas e, só então, os valores obtidos com a execução das ações são passados como argumentos para a ação principal que tratará o sinal em avaliação. A avaliação de uma fonte de valor desse tipo é realizada da mesma forma que a execução de uma ação, por meio de um acionador de ação.

Em tempo de execução, uma ação é representada por um objeto do tipo ActionInstance, uma interface que é implementada pelas classes CallActionInstance, EventActionInstance, e SequenceActionInstance. Além disso, uma subinterface, chamada MacroActionInstance representa uma ação implementada por uma *macro*. A Figura 5.3 mostra a hierarquia do tipo ActionInstance, que são os objetos efetivamente executados pelo acionador de ação após o mapeamento de parâmetros. Além disso, esses elementos estão diretamente associados aos tipos de ação existentes no metamodelo, descritos na Seção 4.3.

A lógica implementada pela classe CallActionInstance é direta, e consiste em enviar uma chamada a um recurso ou ao gerenciador de intermediação, conforme definido no modelo. A classe EventActionInstance atua de forma similar e tem como intuito permitir que a camada gere um evento para as camadas superiores. Em ambos os casos, os parâmetros recebidos pela ação são diretamente utilizados como parâmetros para a chamada ou evento.

Como discutido na Seção 4.3, uma ação implementada por meio de uma *macro* é descrita no modelo da camada por meio da classe MacroAction que integra o metamodelo. Ações desse tipo definem uma classe que implementa a operação a ser realizada. Esta classe deve implementar a interface MacroActionInstance que, por ser uma subinterface de ActionInstance, é acionada da mesma forma que os demais tipos de ação implementados pelo ambiente de execução. No entanto, a lógica implementada por uma *macro* é arbitrária, o que permite definir tipos de ações além dos previstos pelo ambiente de execução. Apesar disso, o contexto para execução dessas ações é o mesmo disponível para os demais tipos de ações, e se limita ao definido na interface ActionInstance.

A classe SequenceActionInstance, por sua vez, encapsula uma série de aciona-

dores de chamadas, que são utilizados para executar, em ordem, cada uma das ações que o compõem, conforme definido no modelo. Ao fim da execução, o resultado da última ação executada é utilizado como resultado da ação como um todo.

De forma geral, um tratador de sinais baseado em uma ação (ActionSignalHandler) recebe um sinal e verifica se é capaz de tratá-lo. Caso afirmativo, o tratador utiliza um acionador de ação (ActionCaller) associado para mapear o contexto do sinal identificado para o contexto da ação (ActionInstance), e em seguida executá-la.

5.3 Recursos

Os recursos utilizados pela camada na realização dos serviços por ela fornecidos, são mantidos em tempo de execução pelo gerenciador de recursos. O gerenciador de recursos mantém um registro dos recursos disponíveis, e provê meios para sua obtenção e manipulação. Além disso, a utilização de recursos em tempo de execução, é mediada por componentes capazes de interagir com os recursos, traduzindo suas construções naquelas utilizadas pela camada.

A classe ResourceManager representa o ponto inicial de acesso aos recursos gerenciados por uma camada de intermediação de serviços. Essa classe mantém um registro de recursos, criado a partir das definições presentes no modelo da camada. A partir dessa classe, é possível obter os recursos registrados, que são representados em tempo de execução pelo tipo Resource.

A interface Resource possui duas implementações concretas: ManagedResource, que media o acesso a um recurso ordinário; e BrokerManager, que representa um outro intermediador de serviços que está sendo utilizado como um recurso. A Figura 5.4 apresenta as principais classes relacionadas ao gerenciamento de recursos em tempo de execução.

A interface Resource é uma extensão das interfaces Touchpoint e Executable. Um *touchpoint* representa uma interface de gerenciamento que disponibiliza uma forma padronizada para sua utilização, independente do tipo de recurso gerenciado. Nesta implementação, Touchpoint é herdeira das interfaces Effector e Sensor e, portanto, um *touchpoint* provê uma interface padrão para realizar chamadas e detectar eventos em um recurso. Além disso, um recurso também é executável, pois também implementa a interface Executable. Ser executável indica que um recurso, em tempo de execução, possui um fluxo próprio de controle, que nesse caso é utilizado para serializar a interação entre a camada e o recurso.

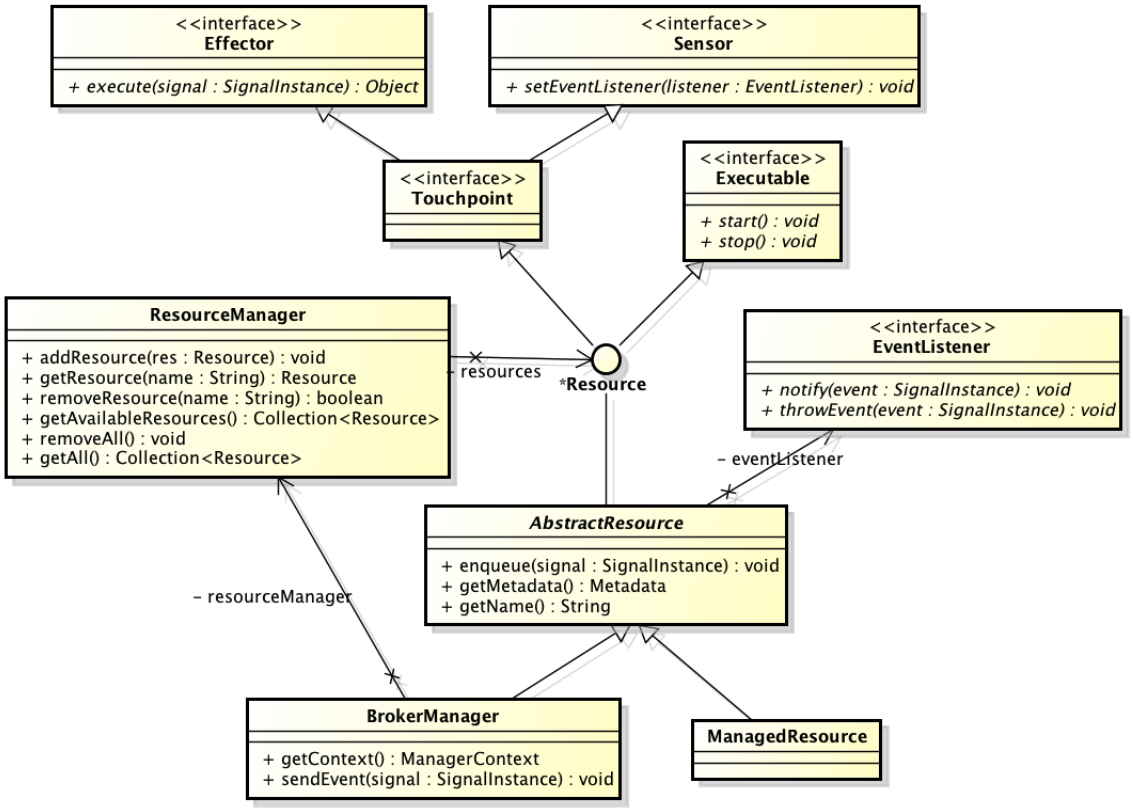


Figura 5.4: *Elementos associados ao gerenciamento de recursos em tempo de execução.*

5.3.1 Integração com os recursos

Um conjunto de classes e interfaces é fornecido para a integração dos recursos com o ambiente de execução. Essa biblioteca tem como função viabilizar a interação entre o ambiente de execução e as implementações dos recursos subjacentes. Esses elementos, que integram o pacote mdvm.sb.adapters, permitem associar a interface dos recursos, descrita no metamodelo, às suas implementações. A Figura 5.5 mostra um diagrama de classes do pacote mencionado.

A integração em questão se aplica aos recursos baseados na instanciação de uma implementação, representados no metamodelo pela classe Instance, descrita na Seção 4.4. Por ser implementado na plataforma Java, o ambiente de execução exige que o recurso também seja implementado nesta plataforma. Apesar disso, é possível utilizar uma implementação que atue apenas como um *wrapper* para a real implementação do recurso, em outra plataforma.

Para que um recurso seja integrado ao ambiente de execução é preciso que um protocolo seja seguido. Antes de tudo, o recurso a ser integrado deve implementar a interface Manageable, que define que o recurso pode ser gerenciado por uma camada de intermediação de serviços. Por meio desta interface, o recurso tem acesso a um notificador de eventos, representado pelo tipo EventNotifier, que deve ser utilizado para sinalizar

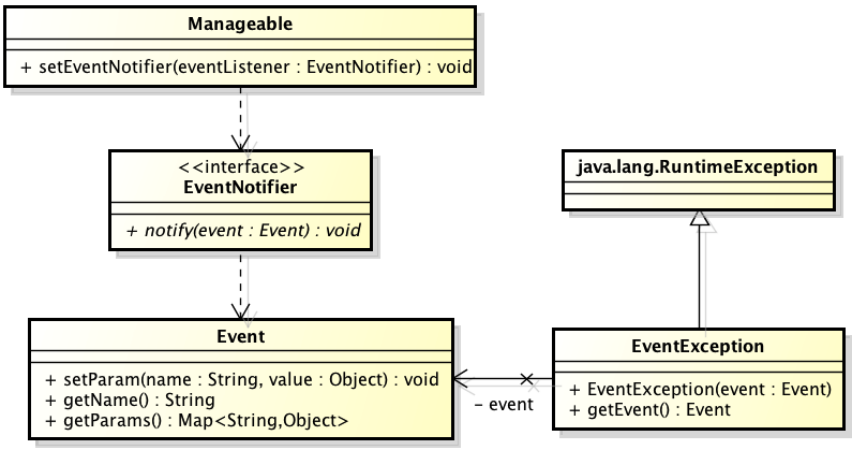


Figura 5.5: *Elementos para integração de recursos ao intermediador de serviços.*

eventos ao intermediador de serviços.

Os eventos a serem sinalizados são representados por meio da classe Event, também disponível no pacote citado. Além dessa classe, a classe EventException pode ser usada para sinalizar a ocorrência de uma exceção no recurso. Ao sinalizar um evento ou exceção, um recurso deve indicar o nome do evento e valores para os seus parâmetros.

Ainda, os métodos que implementam as chamadas providas pelo recurso precisam ser marcados com a anotação @Call. Esta anotação tem como função associar uma chamada provida, descrita no metamodelo, ao método que a implementa. Para isso, a anotação indica o nome da chamada provida associada ao método anotado e a ordem na qual os parâmetros desta chamada devem ser passados ao método. O Código 5.1 apresenta um trecho da implementação de um recurso a ser integrado ao ambiente de execução.

No código em questão, a linha 7 indica que o método sendSchema implementa a chamada SendSchema descrita na interface do recurso e a ordem em que os parâmetros dessa chamada devem ser passados na invocação do método. As linhas 14 a 17 e 23 a 26, ilustram as formas como um recurso pode gerar um evento. Um evento de exceção é considerado de maior prioridade e tratado imediatamente pela camada. Os eventos notificados de forma padrão pelo recurso, utilizando a interface EventNotifier, são serializados pelo gerenciador de intermediação por meio de sua fila de sinais a serem tratados.

Uma vez seguido o protocolo descrito, um intermediador de serviços é capaz de interagir apropriadamente com o recurso, identificando corretamente os eventos por ele gerados, e realizando chamadas aos seus métodos corretos. Esta interação possibilita que o recurso seja gerenciado por uma camada de intermediação de serviços descrita por meio do metamodelo proposto.

Código 5.1 `mdvm.sb.adapters.impl.SmackAdapter`

```
1  public class SmackAdapter implements Manageable {
2      private EventNotifier eventNotifier;
3      public void setEventNotifier(EventNotifier eventNotifier) {
4          this.eventNotifier = eventNotifier;
5      }
6
7      @Call(name = "SendSchema", parameters = {"schema", "participant"})
8      public void sendSchema(String schema, String participant) {
9          if (!isOnline(participant))
10             return;
11         try {
12             sendStringAsFile(participant, "schema", schema);
13         } catch (XMPPException e) {
14             Event event = new Event("SchemaFailed");
15             event.setParam("receiver", participant);
16             event.setParam("schema", schema);
17             throw new EventException(event);
18         }
19     }
20
21     /* ... */
22     private void dealWithSchema(String participant, String schema) {
23         Event event = new Event("SchemaReceived");
24         event.setParam("sender", participant);
25         event.setParam("schema", schema);
26         eventNotifier.notify(event);
27     }
28     /* ... */
29 }
30
```

5.4 Manutenção de estado

A manutenção do estado da camada, em tempo de execução, é realizada por meio do gerenciador de estado. O gerenciador de estado mantém informações sobre os tipos de dados descritos no modelo e os registros de dados associados a esses tipos. A partir do gerenciador, é possível pesquisar, criar, modificar e destruir registros de dados. Além disso, o gerenciador de estado monitora alterações nesses registros e, se necessário, notifica o gerenciador autônomo. As classes que integram o gerenciamento de estado em tempo de execução são ilustradas na Figura 5.6.

A principal classe desse pacote, denominada StateManager, mantém um conjunto de gerenciadores específicos para cada tipo de dados descrito no modelo em execução. Um gerenciador de tipo de dados é implementado pela classe StateTypeManager, e mantém os registros de dados de um determinado tipo de dados. Por meio do gerenciador

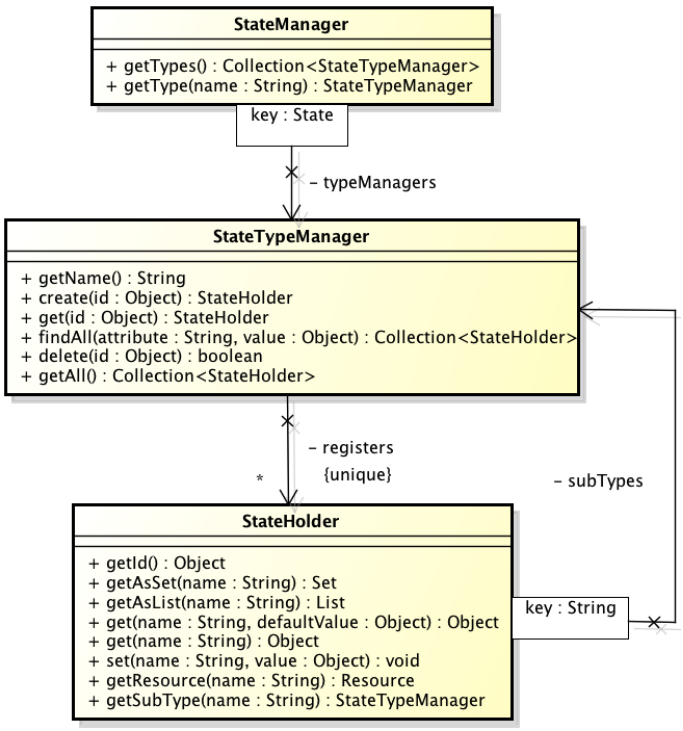


Figura 5.6: *Classes que implementam o gerenciamento de recursos.*

de tipo de dados é possível obter um registro de dados pelo seu identificador, ou realizar uma consulta baseada nos seus demais atributos. Além disso, o gerenciador de tipo de dados também pode ser usado para criar e destruir registros de dados.

Os registros de dados são representados em tempo de execução por instâncias da classe StateHolder. Um StateHolder mantém os dados de um registro, e fornece métodos para acesso e manipulação de seus atributos. Esta classe também é responsável por interceptar a manipulação de um registro com o intuito de identificar mudanças e notificá-las aos componentes interessados.

5.5 Gerenciamento autônomo

A implementação do auto-gerenciamento da camada se baseia na arquitetura de computação autonôma introduzida na Seção 2.3.1. O ambiente de execução inclui um conjunto de classes que implementam as funções de monitoramento (Monitor), análise (Analyzer), planejamento (Planner) e execução (Executor), presentes em um gerenciador autônomo. Além dessas funções, um outro conjunto de classes é utilizado para representar o conhecimento trocado entre essas funções em tempo de execução, incluindo ocorrências de sintomas (SymptomOcurrence), requisições de mudança (ChangeRequestInstance), e planos de mudança (ChangePlanInstance). A Figura 5.7 ilustra as classes que compõem a implementação do mecanismo autônomo.

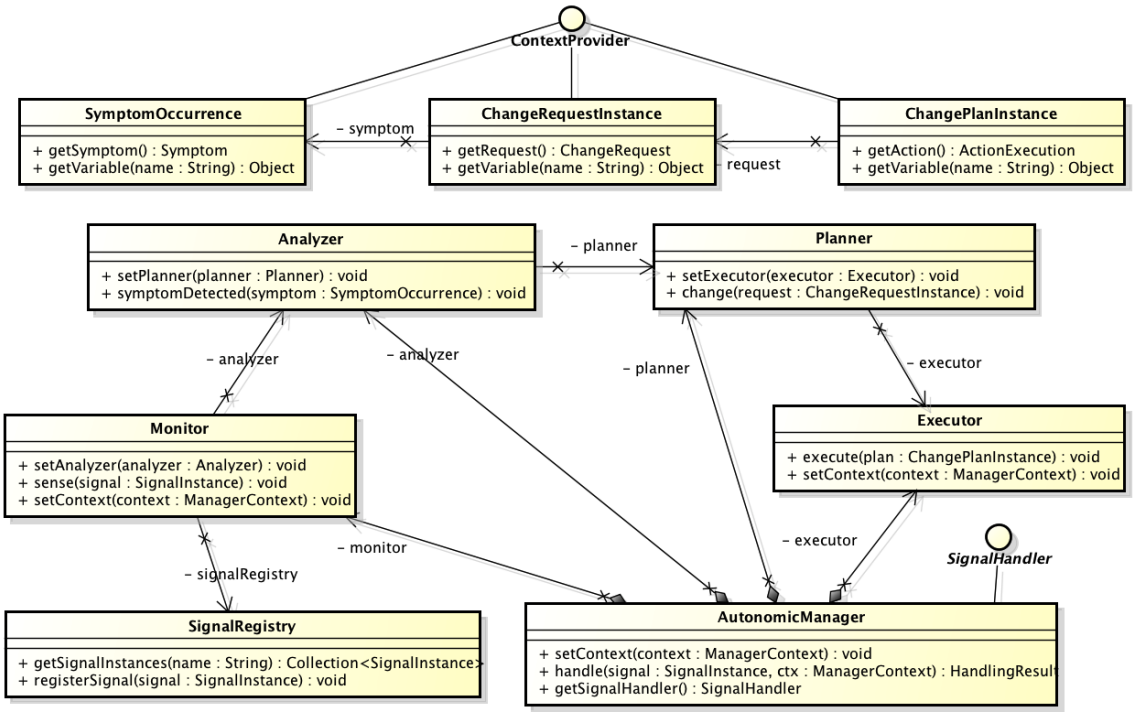


Figura 5.7: *Classes que implementam o mecanismo autônomo da camada.*

Em tempo de execução, o mecanismo autônomo é ativado pelo gerenciador autônomo, implementado pela classe AutonomicManager. O gerenciador autônomo implementa a interface SignalHandler, de forma a interceptar os sinais recebidos pela camada. Apesar disso, o gerenciador autônomo não impede que o sinal continue sendo tratado pelos demais tratadores regularmente. Um sinal intereceptado pelo gerenciador autônomo é encaminhado ao monitor, que avalia se algum sintoma pode ser identificado, e encaminha uma ocorrência para o analisador. O analisador, por sua vez, identifica a mudança associada ao sintoma e envia uma solicitação de mudança para o planejador. Ao receber uma solicitação de mudança, o planejador determina o plano para efetuar a mudança solicitada, e o encaminha ao executor. Este último, por fim, executa a ação definida pelo plano e a executa.

5.5.1 Monitor

Ao receber um sinal, o monitor verifica se algum sintoma, definido no modelo do intermediador, depende desse evento e reavalia as condições desse sintoma. Caso as condições que determinam o sintoma sejam verdadeiras, uma ocorrência de sintoma, representada pela classe SymptomOcurrence, é encaminhada para o analisador. Uma ocorrência de sintoma carrega consigo informações sobre os sinais e registros de dados que culminaram na detecção do sintoma.

As condições de um sintoma são definidas por meio de expressões. Assim como nas fontes de valor, as expressões para definição de uma condição também são descritas em Groovy. A linguagem Groovy possui uma semântica bem definida para coerção de objetos em um tipo booleano, que é empregada na avaliação de uma condição.

O contexto para avaliação das condições é definido no modelo por meio da construção Binding, e pode envolver sinais e tipos de dados, conforme descrito na Seção 4.6. O nome empregado na definição de um elemento do contexto de um sintoma, identifica uma variável que pode ser utilizada em uma expressão para representar o objeto do contexto em tempo de execução.

Quando uma variável é associada a um sinal, essa variável dá acesso a um objeto que apresenta como propriedades os parâmetros do sinal detectado e a origem desse sinal. De forma similar, uma variável associada a um tipo de dados permite obter atributos de um registro de dados desse tipo.

Durante a execução da camada, vários sinais de um mesmo tipo podem ser gerados por diferentes recursos. Além disso, diversos registros de dados de um mesmo tipo podem existir em tempo de execução. Portanto, um mesmo sintoma pode ser detectado em diferentes contextos, gerando várias ocorrências do mesmo sintoma. Ao avaliar os sintomas, o monitor identifica as combinações de ocorrências de um sinal e registros de dados que tornam as expressões de condição verdadeiras e gera uma ocorrência de sintoma para cada uma.

Apesar das condições de um sintoma poderem envolver diversos sinais, esses geralmente não ocorrem em um mesmo instante. O monitor implementado assume que um sinal indica uma mudança de estado e mantém um registro com a última ocorrência de cada tipo de sinal gerado por cada recurso. Esse registro, implementado pela classe SignalRegistry, permite que sintomas que dependem de diversos sinais possam ser identificados.

5.5.2 Analisador

Após a detecção de um sintoma pelo monitor, uma ocorrência de sintoma é repassada ao analisador. O analisador identifica a mudança necessária, de acordo com o mapeamento definido no modelo em execução e cria uma requisição de mudança. A requisição de mudança é representada em tempo de execução pela classe ChangeRequestInstance, que carrega consigo, além de informações sobre a mudança requisitada, a ocorrência de sintoma que gerou a requisição. A implementação do analisador, por meio da classe Analyzer, é bastante simples, pois a associação entre sintoma e requisição de mudança é estabelecida de forma direta no modelo.

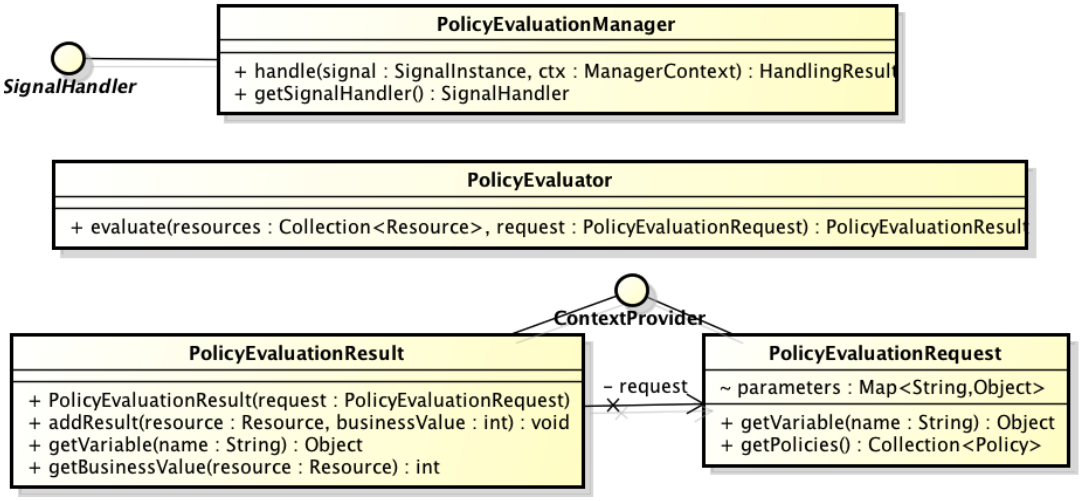


Figura 5.8: *Classes que implementam a avaliação de políticas na camada.*

5.5.3 Planejador

O planejador é implementado pela classe Planner e possui uma implementação bem similar à do analisador. O planejador recebe uma requisição de mudança e identifica o plano de mudança associado. Após identificado o plano, a classe ChangePlanInstance é instanciada com informações do tipo de plano e da requisição de mudança recebida.

5.5.4 Executor

O executor é a última função do ciclo autônomo, tendo como função executar a ação definida no plano de mudança recebido. Ao receber um plano de mudança, o executor obtém a ação definida no plano e cria um acionador para essa ação. O plano de mudança recebido é utilizado como provedor de contexto para a execução da ação. Esse contexto, por sua vez, engloba o contexto da solicitação de mudança e dos sintomas que geraram a mudança. Isto permite que a ação a ser executada tenha seus argumentos associados ao contexto que desencadeou sua execução.

5.6 Políticas

A avaliação de políticas em tempo de execução é conduzida pelo gerenciador de avaliação de políticas, implementado pela classe PolicyEvaluationManager. O gerenciador de avaliação implementa a interface SignalHandler e intercepta os sinais definidos como pontos de avaliação de políticas no modelo em execução. A Figura 5.8 ilustra as classes envolvidas na implementação da avaliação de políticas.

Ao identificar um sinal que define um ponto de avaliação de políticas, o gerenciador de avaliação gera uma requisição de avaliação de políticas, representada pela classe

PolicyEvaluationRequest. Uma requisição de avaliação de políticas inclui as políticas que deverão ser avaliadas e o contexto necessário para avaliação dessas políticas. O mapeamento entre o contexto do sinal que disparou a avaliação de políticas e a requisição de avaliação segue as mesmas regras de mapeamento de parâmetros descritas na Seção 5.2.

Uma requisição de avaliação de políticas é encaminhada ao avaliador de políticas, implementado pela classe PolicyEvaluator. O avaliador de políticas avalia as condições de cada política em relação a cada recurso e identifica o valor total de negócio associado a cada recurso, de acordo com o resultado da avaliação. O valor total de negócio de um determinado recurso é obtido por meio da soma dos valores de negócio atribuídos por cada política cuja condição foi atendida pelo recurso. Os resultados da avaliação são representados por meio da classe PolicyEvaluationResult, que registra os valores de negócio associados a cada recurso. Além disso, o resultado da avaliação de políticas engloba o contexto da requisição de avaliação que gerou esses resultados.

Após a avaliação de políticas, o gerenciador de avaliação identifica os tratadores de avaliação de políticas associados ao grupo de políticas avaliadas. Conforme descrito na Seção 4.7, um tratador de avaliação define uma ação a ser executada após a avaliação de políticas. Ao executar essa ação, o gerenciador de avaliação de políticas utiliza o resultado da avaliação de políticas como provedor de contexto.

5.6.1 Considerações finais

O ambiente de execução apresentado neste capítulo foi construído com o intuito de possibilitar a execução de modelos descritos em conformidade com o metamodelo descrito no Capítulo 4. Os componentes apresentados tem como função realizar o comportamento de uma camada de intermediação de serviços descrita por meio de um modelo, incorporando a semântica operacional do metamodelo. Assim, a partir de um modelo associado ao ambiente de execução, temos uma camada de intermediação de serviços executável, capaz de interagir em tempo de execução com a camada superior e com os recursos subjacentes.

CAPÍTULO 6

Exemplo: Intermediador de Comunicação em Rede

Este capítulo demonstra como o metamodelo proposto e o ambiente de execução fornecido, podem ser utilizados para construir e executar uma camada de intermediação de serviços. Para isto, construímos um modelo, em conformidade com o metamodelo proposto, que descreve o comportamento presente na camada NCB da CVM. Iniciamos este capítulo com uma breve descrição do comportamento da camada NCB, seguido por sua modelagem utilizando as abstrações descritas no Capítulo 4. Ao final do capítulo, descrevemos como o modelo construído foi avaliado, e os resultados obtidos.

6.1 Visão geral

Como mostrado na Seção 2.2, a camada de Intermediação de Comunicação em Rede (*Network Communication Broker*, NCB), é responsável por prover uma interface que abstrai da camada superior os detalhes envolvidos na utilização dos *frameworks* de comunicação, como Skype, Smack, Asterisk etc. A partir da análise do domínio de comunicação e de *frameworks* de comunicação existentes, foi identificado em trabalhos anteriores [10], um conjunto de operações, independentes de tecnologia, capaz de prover o nível de abstração esperado dessa camada. Esse conjunto de operações, que compõe a interface da camada NCB, representa o ponto de partida para a construção de um modelo que defina um comportamento equivalente ao da camada NCB. A Tabela 6.1 lista as operações e eventos que integram a interface da camada NCB.

Para estabelecer uma sessão de comunicação por meio da NCB, as operações disponíveis em sua interface são utilizadas de acordo com um protocolo. Inicialmente, é preciso que o usuário esteja autenticado na camada, o que é feito por meio da operação LoginAll. Uma vez autenticado, o usuário pode enviar e receber esquemas de comunicação por meio da operação SendSchema e do evento SchemaReceived. A troca de esquemas é dirigida pelas camadas superiores e antecede o estabelecimento da comunicação em si, cabendo à camada NCB efetuar a transmissão e recepção de esquemas.

Tabela 6.1: *Interface da camada NCB*

Calls
LoginAll()
LogoutAll()
SendSchema(receivers, schema)
CreateSession(session)
DestroySession(session)
AddParty(session, participant)
RemoveParty(session, participant)
EnableMedium(session, medium)
EnableMediumReceiver(session, medium)
DisableMedium(session, medium)
Events
LoginFailed(framework)
SchemaReceived(sender, schema)
SchemaFailed(receiver, schema)
SessionFailed(session)

Após a negociação de esquemas, uma sessão de comunicação pode ser criada por meio da operação CreateSession. Em seguida, a camada superior pode configurar a sessão de comunicação, determinando seus participantes por meio da operação AddParty, e selecionando o tipo de mídia a ser utilizado através da operação EnableMedium.

Durante sua execução, a camada NCB aguarda que chamadas sejam realizadas pela camada superior às operações disponíveis em sua interface. O processamento de algumas operações, como LoginAll e SendSchema, independe da existência de uma sessão de comunicação e, assim que processadas, geram chamadas correspondentes para os *frameworks* de comunicação. As operações CreateSession, e AddParty, por sua vez, apenas mantêm informações sobre a sessão na camada, até que seja solicitado o estabelecimento da conexão por meio da chamada EnableMedium. A partir desse momento, de acordo com a quantidade de participantes e o tipo de mídia solicitado, a camada seleciona o *framework* mais apropriado. Só então as operações necessárias para estabelecer a comunicação, são efetivamente executadas no *framework* selecionado.

Além de executar as operações solicitadas por meio de sua interface, a camada NCB também interage com os *frameworks* de comunicação gerenciados. Por meio da interface empregada para a interação com os *frameworks* de comunicação, a NCB identifica eventos gerados por esses recursos e efetua chamadas às suas operações. A Tabela 6.2, apresenta a interface empregada para a interação dos recursos gerenciados com a camada NCB. A camada NCB monitora constantemente os eventos gerados pelos *frameworks* de comunicação, podendo esses eventos serem tratados pela camada, ou encaminhados às camadas superiores. Eventos como SchemaFailed, e LoginFailed são diretamente repassados à camada superior, pois a camada NCB não têm conhecimento suficiente para tratá-los.

O evento MediumFailed, por sua vez, inicia o processo de seleção e configuração de um outro *framework* de comunicação que possa ser usado para reestabelecer a comunicação interrompida.

Tabela 6.2: *Interface dos frameworks de comunicação.*

Calls
Login()
Logout()
SendSchema(receivers, schema)
CreateSession(session)
DestroySession(session)
AddParty(session, participant)
RemoveParty(session, participant)
EnableMedium(session, medium)
EnableMediumReceiver(session, medium)
DisableMedium(session, medium)
Events
LoginFailed()
SchemaReceived(sender, schema)
SchemaFailed(receiver, schema)
MediumFailed(medium)

Como é possível observar, a interface da camada NCB e dos recursos por ela utilizada é bastante similar. Isto se deve ao fato de a NCB ter sido projetada para abstrair principalmente os detalhes da seleção, preparação e substituição dos *frameworks* de comunicação.

As seções seguintes descrevem como o comportamento descrito acima pode ser modelado em conformidade com o metamodelo proposto. A modelagem em questão se baseia na análise da implementação existente da camada NCB [4].

6.2 Modelo do Intermediador de Comunicação em Rede

Conforme descrito no Capítulo 4, a definição de uma camada de intermediação de serviços, utilizando o metamodelo proposto, é estruturada em torno de um gerenciador de intermediação da camada, representado no metamodelo pela classe Manager. Ao modelar a camada NCB, foi utilizado apenas um gerenciador de intermediação e, assim sendo, a interface desse gerenciador define a interface da camada NCB, que contém as operações e eventos mencionados na seção anterior.

Ao longo das subseções seguintes, as partes que integram o modelo da camada NCB são descritas por meio de diagramas UML e, quando necessário, por uma notação para descrição de modelos de forma textual. Essa notação, denominada *Human Usable*

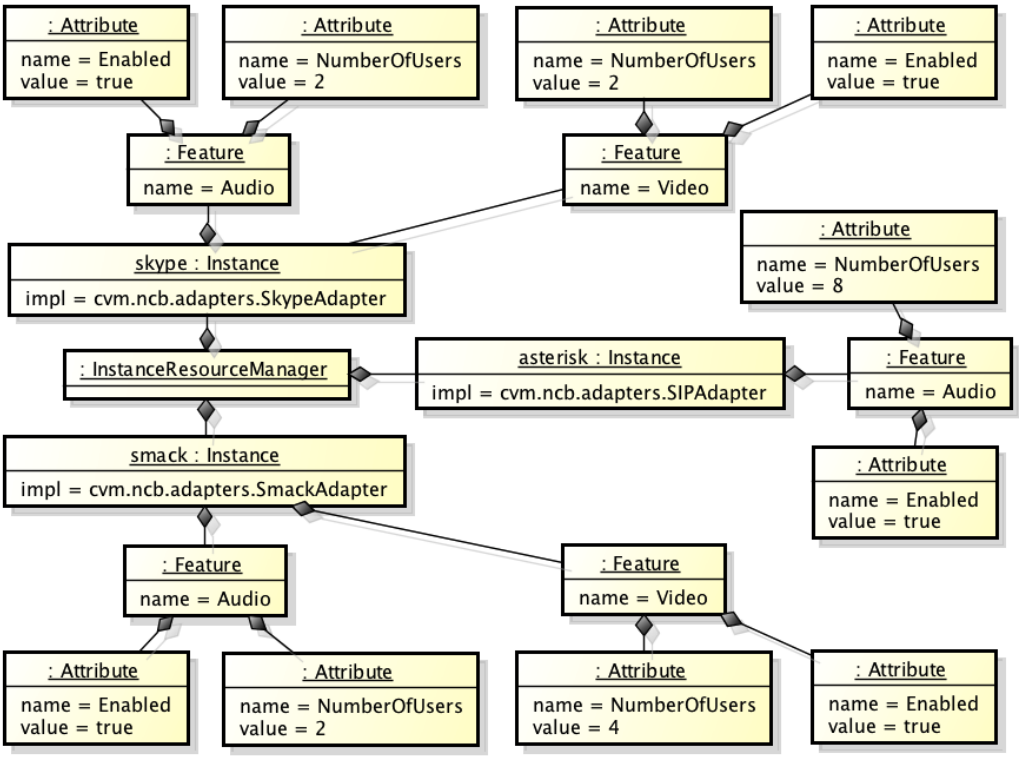


Figura 6.1: *Definição dos* frameworks *de comunicação empregados pela camada NCB.*

Textual Notation (HUTN) [45] também é padronizada pelo OMG e apresenta vantagens na descrição de modelos mais detalhados, cuja representação gráfica pode exigir muito espaço.

6.2.1 Recursos

Para implementar a funcionalidade presente na NCB, foram descritos três recursos no modelo da camada. Estes recursos representam os *frameworks* de comunicação Skype, Smack e Asterisk. A Figura 6.1 mostra o diagrama de objetos do gerenciador de recursos descrito no modelo da NCB, onde os recursos mencionados são registrados. A interface utilizada pelos recursos é omitida nessa figura, mas corresponde à interface descrita na Tabela 6.2.

Cada um dos *frameworks* é implementado por meio de uma classe na plataforma Java, de acordo com o protocolo especificado na Seção 5.3.1.

6.2.2 Estado

Na modelagem da camada NCB, foi descrito apenas um tipo de dados, que é utilizado para manter informações das sessões de comunicação em andamento. O tipo de dados definido, denominado Connection, possui quatro atributos que representam,

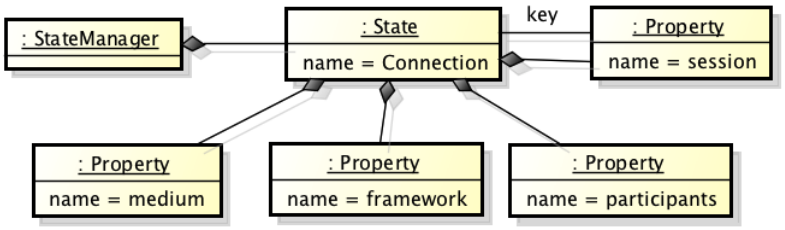


Figura 6.2: *Definição dos tipos de dados empregados pela camada NCB.*

respectivamente, o identificador da sessão, a lista de participantes, o tipo de mídia da comunicação, e o *framework* de comunicação empregado na sessão de comunicação. Ainda, a sessão de comunicação é definida como o atributo chave desse tipo de dados. O diagrama da Figura 6.2 ilustra a parte do modelo da camada que descreve esse tipo de dados.

6.2.3 Tratadores e ações

Para especificar o comportamento da camada em reação aos sinais por ela identificados, foram definidos tratadores para todas as chamadas recebidas da camada superior e dois dos eventos gerados pelos recursos. Os eventos para os quais não foram definidos tratadores são automaticamente encaminhados à camada superior, e representam eventos que a camada NCB não é capaz de tratar. A Tabela 6.3 lista os tratadores definidos, estabelecendo a relação entre sinal e a ação utilizada para tratá-lo.

Tabela 6.3: *Tratadores que integram a definição da camada NCB.*

Call LoginAll	CallAction LoginAllAction
Call LogoutAll	CallAction LogoutAllAction
Call SendSchema	MacroAction SendSchemaAction
Call CreateSession	MacroAction CreateSessionAction
Call DestroySession	MacroAction DestroySessionAction
Call AddParty	MacroAction AddPartyAction
Call RemoveParty	MacroAction RemovePartyAction
Call EnableMedium	MacroAction MediumAction
Call EnableMediumReceiver	MacroAction MediumAction
Call DisableMedium	MacroAction MediumAction
Event LoginFailed	EventAction GenerateLoginFailedEvent
Event MediumFailed	NoAction

Além de estabelecer uma associação entre um sinal e uma ação, um tratador também define como os parâmetros de um sinal são mapeados para os parâmetros de uma ação. O Código 6.1 utiliza a notação HUTN para mostrar a definição do tratador do sinal EnableMedium e a ação MediumAction, utilizada para tratá-lo. Os elementos descritos

Código 6.1 Tratador para a chamada EnableMedium

```
1  Handler {
2    enabled: true
3    signal: Call EnableMedium
4    action: ActionExecution {
5      action: MacroAction MediumAction
6      bindings: ParameterBinding {
7        parameter: Parameter MediumAction.session
8        value: ParameterValue EnableMedium.session
9      }
10     bindings: ParameterBinding {
11       parameter: Parameter MediumAction.medium
12       value: ParameterValue EnableMedium.medium
13     }
14     bindings: ParameterBinding {
15       parameter: Parameter MediumAction.action
16       value: FixedValue "enable"
17     }
18   }
19 }
20 MacroAction MediumAction {
21   impl: cvm.ncb.macros.MediumAction
22   parameters: ActionParameter session
23   parameters: ActionParameter medium
24   parameters: ActionParameter action
25 }
```

neste modelo são instâncias das classes correspondentes no metamodelo. O tratador em questão define que ao receber uma chamada EnableMedium, a camada NCB deve executar a ação MediumAction. Além disso, o tratador define o mapeamento entre os argumentos recebidos junto à chamada EnableMedium e os parâmetros necessários para a execução da ação MediumAction.

A ação empregada no exemplo é baseada em uma *macro* que, por sua vez, é implementada pela classe cvm.ncb.macros.MediumAction. O Código 6.2 mostra como essa *macro* é implementada. Entre as linhas 3 e 6, o identificador da sessão de comunicação é utilizado para obter o *framework* de comunicação utilizado na realização dessa sessão. Isto é realizado por meio de uma consulta ao gerenciador de estado da camada. Em seguida, uma chamada é criada com o auxílio do método createSignal, e encaminhada para ser executada pelo *framework*.

6.2.4 Gerenciamento autônomo

As construções de gerenciamento autônomo da camada NCB têm como intuito identificar falhas nos *frameworks* de comunicação e tomar uma ação para que um novo

Código 6.2 Implementação da ação MediumAction

```
1  public class MediumAction implements MacroActionInstance {
2    public Object execute(ManagerContext ctx, Map<String, Object> params) {
3      Object session = params.get("session");
4      StateHolder connection = ctx.getStateManager()
5          .getType("Connection").get(session);
6      Resource framework = connection.getResource("framework");
7      SignalInstance signal = createSignal(framework, params);
8      framework.enqueue(signal);
9      return null;
10   }
11   private SignalInstance createSignal(Resource resource,
12         Map<String, Object> params) {
13     Object action = params.get("action");
14     if (action.equals("enable"))
15       return resource.createSignal("EnableMedium", params);
16     if (action.equals("receive"))
17       return resource.createSignal("EnableMediumReceiver", params);
18     if (action.equals("disable"))
19       return resource.createSignal("DisableMedium", params);
20     return null;
21   }
22 }
23
```

framework seja selecionado. A definição do gerenciamento autônomo no modelo da camada NCB é apresentada no Código 6.3.

Entre as linhas 2 e 12 do código em questão, é definido um sintoma chamado FrameworkFailed, que indica a ocorrência de uma falha em um *framework*. O sintoma em questão define que o contexto para a avaliação de sua condição está associado a ocorrência de um evento do tipo MediumFailed e a um registro de dados do tipo Connection. Assim, a expressão que define a condição desse sintoma verifica não só se ouve uma falha em um *framework*, mas também se existe alguma sessão de comunicação utilizando o recurso que falhou.

Associado ao sintoma temos solicitação de mudança ChangeFramework, definida entre as linhas 13 e 15. Essa solicitação de mudança é utilizada para ligar um sintoma à um plano de mudança, que é definido nas linhas 16 a 33. O plano de mudança definido indica que a ação ChangeFramework deve ser executada, e os argumentos utilizados na execução dessa ação. Como mencionado na Seção 5.5, o contexto provido por um plano de mudanças abrange o contexto do sintoma que o gerou, permitindo utilizarmos expressões que incluem as variáveis event e conn, que fazem parte do contexto no qual o sintoma FrameworkFailed é identificado.

Código 6.3 Definição dos elementos de gerenciamento autônomo da NCB.

```
AutonomicManager {
  identifies: Symptom FrameworkFailed {
    bindings: Binding event {
      bindable: Event MediumFailed
    }
    bindings: Binding conn {
      bindable: State Connection
    }
    conditions: Condition {
      expression: "event.source == conn.framework && event.medium == conn.medium"
    }
  }
  requests: ChangeRequest ChangeFramework {
    basedOn: Symptom FrameworkFailed
  }
  plans: ChangePlan Plan1 {
    basedOn: ChangeRequest ChangeFramework
    action: ActionExecution {
      action: SequenceAction ChangeFramework
      bindings: ParameterBinding {
        parameter: Parameter ChangeFramework.session
        value: ExpressionValue "conn.session"
      }
      bindings: ParameterBinding {
        parameter: Parameter ChangeFramework.medium
        value: ExpressionValue "event.medium"
      }
      bindings: ParameterBinding {
        parameter: Parameter ChangeFramework.framework
        value: ExpressionValue "event.source"
      }
    }
  }
}
```

6.2.5 Políticas

Conforme descrito na Seção 6.1, durante a configuração de uma sessão de comunicação a camada NCB apenas mantém informações a respeito da sessão, aguardando uma

solicitação para que a comunicação seja estabelecida. A sessão de comunicação é efetivamente estabelecida por meio das chamadas EnableMedium e EnableMediumReceiver, que indicam que a transmissão do tipo de mídia definido deve ser iniciada. No entanto, antes de iniciar essa transmissão é preciso definir qual *framework* de comunicação será utilizado.

A avaliação de políticas na camada NCB tem como intuito selecionar um *framework* de comunicação que melhor atenda às necessidades de uma sessão de comunicação. Assim sendo, a avaliação de políticas é realizada imediatamente antes do estabelecimento de uma conexão. Neste momento, as informações sobre a sessão de comunicação a ser estabelecida já são conhecidos pela NCB e podem ser utilizados para direcionar a seleção do *framework* a ser utilizados.

Para definir este comportamento, de acordo com o metamodelo proposto, as chamadas EnableMedium, e EnableMediumReceiver foram definidas como pontos de avaliação de políticas. Ao definir estas chamadas como pontos de avaliação de políticas, o ambiente de execução é capaz de interceptar essas chamadas e realizar a avaliação de políticas antes que essas chamadas sejam tratadas por suas respectivas ações.

O Código 6.4 mostra a parte do modelo que descreve esse comportamento na camada NCB. O trecho em questão ilustra um contexto de avaliação de políticas (linhas 2 a 18), incluindo algumas das políticas que o compõem, um ponto de avaliação (linhas 19 a 22), e um tratador de avaliação (linhas 23 a 28) que executa a ação UseFramework.

6.3 Avaliação

Uma vez construído um modelo que representa a camada NCB, sua execução junto ao ambiente fornecido foi avaliada. O objetivo da avaliação foi verificar a equivalência do modelo construído com a implementação existente da CVM, e comparar o desempenho de ambos.

Para realizar essa avaliação, foi elaborado um conjunto de cenários de comunicação, que representam uma sequência de interações válidas da NCB com a camada superior ou com os *frameworks* durante a realização de uma comunicação. A execução desses cenários em ambas as implementações foi então utilizada para estabelecer uma comparação entre a implementação original da NCB e a implementação baseada no modelo construído.

Os cenários elaborados para esta avaliação são brevemente descritos abaixo.

- **Cenário 1**. Uma sessão de comunicação por meio de áudio é estabelecida entre dois participantes.

Código 6.4 Definição dos elementos de avaliação de políticas da NCB.

```
PolicyManager {
  context: PolicyEvaluationContext General {
    parameters: Parameter session
    parameters: Parameter medium
    parameters: Parameter numOfParticipants
    policies: Policy Audio01 {
      businessValue: 100
      condition: Condition {
        expression: "Audio.Enabled"
      }
    }
    policies: Policy Audio02 {
      businessValue: 96
      condition: Condition {
        expression: "numOfParticipants <= Audio.NumberOfUsers"
      }
    }
  }
  points: PolicyEvaluationPoint {
    signal: Call EnableMedium
    bindings: /* ... */
  }
  handlers: PolicyEvaluationHandler {
    action: ActionExecution {
      action: MacroAction UseFramework
      bindings: /* ... */
    }
  }
}
```

- **Cenário 2**. Comunicação de áudio é inicialmente estabelecida entre dois participantes. Em seguida é solicitada a inclusão de vídeo na comunicação, o que é feito imediatamente pois o *framework* em uso dispõe dessa capacidade.
- **Cenário 3**. Comunicação de áudio é inicialmente estabelecida entre dois participantes. Após solicitação para que também seja estabelecida a comunicação por meio de vídeo, outro *framework* é selecionado e a substituição do *framework* antigo é realizada de forma transparente pela camada NCB.
- **Cenário 4**. Após estabelecer uma comunicação de áudio entre dois participantes, ocorre uma falha no *framework* em uso. A camada NCB deve efetuar a sua substituição por outro *framework* capaz de realizar a comunicação.
- **Cenário 5**. Após estabelecer uma comunicação de áudio entre dois participantes, ocorre uma falha em todos os *frameworks* disponíveis. A camada NCB deve notificar a falha à camada superior.
- **Cenário 6**. Após receber um esquema de outro participante, uma comunicação de

áudio é estabelecida entre dois participantes. Um evento é gerado para a camada superior indicando o recebimento do esquema. Esta, por sua vez, determina que a camada NCB deve aceitar essa comunicação.

- **Cenário 7**. Ocorre uma falha durante a autenticação com um dos *frameworks* de comunicação. Um evento deve ser gerado para a camada superior.
- **Cenário 8**. Comunicação entre três participantes, onde uma comunicação de áudio é estabelecida entre três participantes.

Os cenários propostos foram inicialmente executados na implementação existente da NCB com o intuito de extrair o comportamento da camada. Para isso, foram configurados interceptadores capazes de registrar as interações entre a camada NCB e os *frameworks* de comunicação. As informações obtidas na execução dos cenários propostos foram então utilizadas na construção de testes automatizados, capazes de executar os cenários elaborados em uma camada de intermediação de serviços e automaticamente verificar se o seu comportamento foi o mesmo obtido na execução com a camada NCB original.

Para execução dos testes automatizados, foi configurado um conjunto de recursos *mock* que simulam o comportamento esperado de um *framework* de comunicação. Objetos *mock* são comumente utilizados na elaboração de testes de software para simular um objeto que não é o alvo direto do teste. Neste caso, o uso de recursos *mock* é particularmente vantajoso, pois permite avaliar a camada de forma independente dos *frameworks* de comunicação.

Cada cenário descreve um conjunto de interações com a camada e verifica se esta se comportou de acordo com o esperado. A Tabela 6.4 demonstra a interação realizada com a camada para o Cenário 4, mostrando como a camada deve interagir com os recursos simulados. A primeira coluna mostra a sequência de chamadas realizadas para a camada NCB, e as demais colunas mostram as chamadas realizadas pela NCB aos recursos *mock*. Neste cenário executamos chamadas ao intermediador para estabelecer uma comunicação de áudio entre dois participantes. A camada NCB escolhe o *Framework 1* e envia para este as chamadas para estabelecer a comunicação. Em seguida, simulamos uma falha no *Framework 1*, fazendo com que a camada NCB automaticamente finalize a comunicação estabelecida utilizando o *Framework 1* e a restabeleça no *Framework 2*.

Tabela 6.4: *Execução do cenário 4*

NCB	*Framework* 1	*Framework* 2
LoginAll()	Login()	Login()
SendSchema("Yali", *schema*)	SendSchema("Yali", *schema*)	SendSchema("Yali", *schema*)
CreateSession("101")	CreateSession("101")	CreateSession("101")
AddParty("101", "Yali")	AddParty("101", "Yali")	AddParty("101", "Yali")
EnableMedium("101", "Audio");	EnableMedium("101", "Audio")	EnableMedium("101", "Audio")
SimulateFailure(Framework 1, "Audio")	DestroySession("101")	

Os testes construídos a partir desses cenários foram utilizados durante toda a construção do modelo descrito na seção anterior para validar sua conformidade com a implementação existente da NCB. Após a construção do modelo e sua validação, esses cenários foram novamente executados em ambas as implementações com o intuito de mensurar e comparar a desempenho de ambas implementações.

Para isso, foram realizadas 100 (cem) execuções de cada cenário e em seguida calculado o tempo médio de execução e o desvio padrão. As execuções foram realizadas em um sistema com as seguintes especificações: Intel Core 2 Duo 2.4 GHz, 4GB 1067 MHz DDR3 RAM, Sistema Operacional Mac OS X 10.6.8 e Oracle JDK 1.6.0_33.

Tabela 6.5: *Tempo de execução dos cenários descritos nas implementações da camada NCB (em ms).*

	NCB original		NCB modelada	
	Tempo médio (ms)	Desvio Padrão	Tempo médio (ms)	Desvio Padrão
Cenário 1	1445,91	105,24	3006,92	614,56
Cenário 2	1351,70	57,94	2839,12	118,40
Cenário 3	1434,29	89,81	3005,04	295,41
Cenário 4	1369,29	51,60	3058,53	149,70
Cenário 5	1900,78	62,43	3316,11	119,30
Cenário 6	1257,97	46,70	2802,38	110,81
Cenário 7	1326,91	70,51	2798,55	119,53
Cenário 8	1297,95	56,25	2795,56	128,46

Os resultados expostos na Tabela 6.5 demonstram que o tempo de execução médio da versão da NCB descrita por meio de um modelo é bem superior ao obtido na versão original. Como esperado, isso se deve em parte ao *overhead* envolvido na carga de um modelo durante a inicialização da camada. Para avaliar a influência da carga inicial do modelo na execução dos cenários em questão, também foi calculado o tempo médio de execução excluindo o tempo de carga do modelo. Os resultados dessa mensuração, que se encontram na Tabela 6.6, demonstram que excluindo-se o tempo de carga do modelo, a versão modelada da camada superior apresenta um tempo médio de execução ligeiramente superior ao obtido com o uso da camada NCB original.

No entanto, mesmo excluindo-se o *overhead* devido ao tempo de carga do modelo, a camada modelada demonstra um tempo médio de execução superior em todos os cenários. Isso se deve em grande parte às operações necessárias para o funcionamento da camada modelada, como a avaliação de expressões, mapeamento de parâmetros, e interação com os recursos por meio de reflexão exigem mais do processamento que a versão original, onde essas operações são desnecessárias. Assim, melhorias significativas no desempenho poderiam ser alcançadas otimizando a implementação do mecanismo de avaliação de expressões e mapeamento de parâmetros.

Tabela 6.6: *Tempo de execução dos cenários excluindo-se o tempo de carga do modelo da camada NCB (em ms).*

	Tempo médio (ms)	Desvio Padrão
Cenário 1	1504,86	77,24
Cenário 2	1580,08	68,10
Cenário 3	1630,58	99,83
Cenário 4	1911,54	33,33
Cenário 5	2422,66	31,22
Cenário 6	1407,88	36,51
Cenário 7	1450,68	37,25
Cenário 8	1528,10	92,29

6.4 Discussão

O uso do metamodelo proposto para a construção de uma camada de intermediação de comunicação em rede reduziu significativamente a quantidade de código a ser escrito. Além disso, o modelo construído para representar a camada também é bem reduzido, o que sugere uma maior produtividade de desenvolvimento. Isso só é possível pois ao incorporar conceitos de auto-gerenciamento de recursos e construções para tratamento de eventos e chamadas, o metamodelo permite que essas funcionalidades sejam reutilizadas, dispensando que sejam codificadas durante a construção de um intermediador. Assim sendo, ao construir uma camada de intermediação, cabe ao desenvolvedor especificar apenas o comportamento específico de domínio em uma linguagem que contém construções relacionadas aos problemas a serem tratados por um intermediador de serviços.

A implementação já existente da camada NCB, que integra a CVM, possui 6777 linhas de código escritas na linguagem de programação Java. Enquanto isso, a camada equivalente, descrita em conformidade com o metamodelo, é composta por um modelo que possui 264 objetos, além de 854 linhas de código que implementam ações definidas por meio de *macros*. Apesar disso, ainda é preciso explorar a construção de camadas de intermediação de serviços para outros domínios para se obter uma compreensão mais ampla dos benefícios que podem ser obtidos pelo uso da abordagem proposta.

CAPÍTULO 7

Trabalhos relacionados

Grande parte dos trabalhos relacionados ao uso de modelos para descrição de máquinas virtuais ou interpretadores para DSMLs está diretamente associado à definição de sua semântica dinâmica. Isto se deve ao fato de a formalização da semântica de linguagens de modelagem ser considerada uma forma de possibilitar a geração automática de ferramentas como interpretadores, compiladores e depuradores para DSMLs [12].

Uma das formas empregadas para formalização da semântica de DSMLs e subsequente geração de interpretadores consiste em estabelecer um mapeamento entre a sintaxe abstrata da DSML e a sintaxe abstrata de uma outra linguagem com semântica bem definida. Para definição do mapeamento são empregados mecanismos de transformação como ATL [30] e QVT [37], ou sistemas de reescrita [1, 54, 32]. Entre essas abordagens, o uso de ancoragem semântica [16, 31] propõe que elementos da DSML sejam mapeados para unidades semânticas definidas por meio de modelos computacionais elementares como máquina de estados finitos, sistemas de eventos discretos, autômatos, entre outros.

Apesar de permitirem formalizar a semântica de virtualmente qualquer linguagem de modelagem, as linguagens para descrição de transformações são bastante genéricas e suas construções não guardam relação direta com a semântica específica relacionada ao processamento de modelos em tempo de execução. Devido a isso, a formalização da semântica utilizando essa estratégia exibe o mesmo problema do uso de linguagens de programação de propósito geral, ou seja, não oferecem construções apropriadas para a resolução de um determinado problema.

Outra abordagem utilizada para formalização da semântica dinâmica propõe a sua incorporação ao metamodelo. Conhecida como *behavior weaving*, essa abordagem promove a extensão do meta-metamodelo para que os metamodelos possam incluir semântica. Nessas abordagens a semântica é geralmente definida por meio de meta-linguagens que se assemelham a linguagens de programação de propósito geral. Portanto, apesar de fornecer meios para integração da semântica operacional ao metamodelo, as construções empregadas para descrição dessa semântica não apresentam ganhos de abstração significativos em relação a abordagens tradicionais. Entre os trabalhos que propõem a integração de semântica ao metamodelo podemos destacar a ferramenta

Kermeta [29], que dispõe de uma linguagem de ações orientada a objetos, e o ambiente de metamodelagem XMF-Mosaic [62] que utiliza a linguagem xOCL (*Executable OCL*).

As abordagens para formalização de semântica são caracterizadas por empregarem representações genéricas que podem ser utilizadas para descrição de semântica dinâmica para qualquer linguagem de modelagem. Assim, apesar de possibilitarem a formalização da semântica de DSMLs e subsequente geração de ferramentas, o nível de abstração oferecido por essas abordagens é similar ao encontrado em linguagens de programação de propósito geral. Enquanto isso, a abordagem proposta neste trabalho propõe o uso de modelos para descrever a semântica de um conjunto delimitado de DSMLs, aquelas voltadas para definição de serviços de alto nível a serem realizados por meio de um conjunto heterogêneo de recursos. Para isso, propomos o uso construções estão diretamente relacionadas à semântica desse conjunto de DSMLs.

Outro grupo de trabalhos relacionados à construção de máquinas de execução de modelos propõe o uso de técnicas e ferramentas tradicionais de desenvolvimento de software orientado a objetos, como padrões de projeto [52, 42], *frameworks* [35, 15] e componentes [47, 20], entre outras, que provêem meios para reutilização de projeto e implementação. Apesar de fornecerem um grande auxílio na construção de interpretadores e outras ferramentas para o processamento de DSLs, essas abordagens se baseiam no uso de linguagens de programação de propósito geral e, assim sendo, suas construções são associadas à plataforma de implementação. A abordagem apresentada guarda semelhanças com essas técnicas por propor uma forma de reutilização dos aspectos comuns à implementação de interpretadores para uma determinada categoria de DSMLs. Além disso, a proposta deste trabalho incorpora essas tecnologias, e pode ser considerado como um *framework* para construção de máquinas de execução de modelos em uma determinada categoria de DSMLs. No entanto, diferente de um *framework*, a especialização da arquitetura proposta se dá por meio da modelagem utilizando construções que estão associadas à interpretação de uma família de DSMLs.

Além dos trabalhos relacionados à formalização de semântica de DSMLs e ao emprego de técnicas de engenharia de software para a construção de máquinas de execução de modelos, existem alguns que se destacam por apresentar mais similaridade com este trabalho. Em [23], os autores propõem uma abordagem dirigida por modelos para criação de *frameworks* de interpretação de modelos que podem ser então utilizados para construir interpretadores para diferentes DSMLs. Um *framework* de interpretação de modelos é construído por meio de uma tecnologia de componentes abstratos e um metamodelo que define construções arquiteturais independentes de domínio, que podem então ser especializadas para construir um interpretador de uma DSML. A abordagem proposta neste trabalho se assemelha à descrita acima na medida em que a arquitetura e metamodelos propostos podem ser considerados como um *framework* de interpretação de

modelos para a construção de DSMLs de uma determinada categoria.

O uso de linhas de produção de linguagens específicas de domínio também é proposto em [59] como uma forma de reutilizar o projeto e implementação de aspectos comuns a diferentes DSLs. O uso de linhas de produção compartilha com a abordagem proposta a idéia de reutilização dos aspectos comuns entre DSMLs de diferentes domínios. No entanto o trabalho citado foca principalmente na composição de DSLs e não trata diretamente da construção de DSMLs. A composição também é proposta em [25] como um meio para construção de DSLs. Esse trabalho compartilha com o nosso a idéia de também aplicar técnicas de MDE e DSLs à construção de máquinas virtuais específicas de domínio. No entanto, as máquinas virtuais são definidas como uma composição de interpretadores de sub-domínios.

Apesar de nossa abordagem apresentar semelhanças com os trabalhos citados, ela se difere pelo fato de não pretender ser totalmente genérica, provendo construções mais expressivas para a construção de máquinas de execução para uma determinada família de DSMLs. Dessa forma, buscamos obter os benefícios do uso de técnicas de MDE e DSMLs no desenvolvimento da plataforma para execução de aplicações definidas por meio de modelos que podem ser criados e modificados em tempo de execução.

CAPÍTULO 8

Conclusões

O emprego de abordagens de MDE, associadas a DSMLs claramente traz vários benefícios para a construção e manutenção de aplicações complexas. Ao fornecer abstrações mais próximas do domínio do problema a ser resolvido, essas abordagens podem possibilitar que aplicações complexas sejam construídas ou ajustadas diretamente pelo usuário final. Apesar disso, a construção de DSMLs e mecanismos capazes de processá-las ainda exige um grande esforço de desenvolvimento. Esses mecanismos são geralmente construídos por especialistas em desenvolvimento de software, que o fazem usando linguagens de programação de propósito geral.

Nesta dissertação, foi apresentada uma abordagem que propõe o uso de modelos para definição de máquinas de execução para DSMLs destinadas à descrição de serviços de alto nível a serem realizados por um conjunto heterogêneo de recursos. A abordagem em questão propõe o uso de uma arquitetura genérica projetada para atender aos requisitos envolvidos na execução de modelos descritos por meio dessas DSMLs e que podem ser criados e modificados em tempo de execução.

A partir dessa arquitetura genérica, propusemos a criação de um metamodelo capaz de capturar os aspectos de cada uma das camadas que a integram e o uso de modelos, em conformidade com o metamodelo, como meio de especialização dessa arquitetura. Neste trabalho, foi construído um metamodelo que captura apenas os aspectos independentes de domínio da camada de intermediação de serviços e um ambiente de execução que provê semântica a este metamodelo. Demonstramos então a especialização desse metamodelo para o domínio de comunicação por meio da construção de um modelo para uma camada de intermediação de comunicação em rede equivalente à camada NCB presente na CVM. Por fim, a camada construída a partir da execução do modelo foi comparada à implementação equivalente que integra a CVM, obtendo resultados compatíveis com o esperado.

Entre as contribuições deste trabalho, podemos citar a aplicação de abordagens de MDE e DSMLs para construção da infraestrutura necessária para a utilização dessas abordagens. Ao utilizarmos modelos não apenas para construção e manipulação de aplicações, mas também para a construção das próprias plataformas de execução de

modelos, caminhamos em direção ao princípio de que todos os aspectos de um sistema são representados e manipulados por modelos, considerado fundamental na engenharia dirigida por modelos [13, 8].

Além disso, o processamento de modelos que podem ser criados e modificados em tempo de execução depende de operações que não estão necessariamente associadas ao domínio de negócio da linguagem de modelagem empregada. A existência de padrões na realização de modelos em tempo de execução sugere a possibilidade de reuso, que pode ser obtido por meio de linguagens de modelagem que capturem esses padrões. Neste trabalho, foram analisadas as operações associadas ao gerenciamento de recursos e intermediação de sua utilização no contexto de modelos que descrevem serviços a serem realizados a partir de um conjunto heterogêneo de recursos. Essas operações foram capturadas em um metamodelo que pode ser utilizado para construção de modelos que definem máquinas de execução específicas para diferentes domínio de negócio.

A abordagem proposta neste trabalho também apresenta limitações. Apesar de propor a construção de modelos que descrevem uma máquina de execução de modelos para uma determinada DSML, o presente trabalho não trata da formalização da ligação entre o metamodelo que descreve a sintaxe abstrata e semântica estática da linguagem e o modelo da máquina de execução que descreve sua semântica operacional. Além disso, o ambiente de execução que dá semântica ao metamodelo proposto para construção de intermediadores de serviço é codificado em uma linguagem de programação de propósito geral, limitando sua flexibilidade.

Por fim, a abordagem proposta se limita à construção de máquinas de execução de modelos voltadas para uma categoria específica de DSMLs para construção de serviços de alto nível que são realizados a partir de um conjunto heterogêneo de recursos disponíveis. Apesar de restringir sua utilização, essa característica permite disponibilizar construções mais apropriadas para a definição de máquinas virtuais para essa categoria de DSMLs.

8.1 Trabalhos futuros

Apesar das contribuições deste trabalho, existem várias áreas que não foram cobertas e necessitam de investigação. Como trabalho futuro, planejamos avançar na extensão do metamodelo para descrição das outras camadas da arquitetura genérica empregada. Esta tarefa, por sua vez, requer a identificação dos aspectos independentes de domínio relacionadas a cada camada e a elaboração de abstrações apropriadas para descrever os aspectos dependentes de domínio na forma de modelos.

Além disso, é preciso avaliar a aplicação da abordagem proposta em outros domínios de aplicação com o intuito de alcançar uma melhor compreensão de sua

aplicabilidade e generalidade. Em particular, pretendemos utilizar o metamodelo proposto neste trabalho para a construção de camadas de intermediação de serviços para os domínios de redes elétricas inteligentes locais e computação ubíqua.

Ademais, mais pesquisas precisam ser conduzidas em direção à integração entre modelos que representam uma máquina de execução e metamodelos que representam as respectivas DSMLs, o que envolve mais estudos na área de combinação de modelos. A possibilidade de adaptação de máquinas de execução de modelos em tempo de execução é uma outra direção de pesquisa que pode apontar para novas possibilidades ainda não exploradas.

Referências Bibliográficas

[1] AGRAWAL, A. **Graph rewriting and transformation (GReAT): a solution for the model integrated computing (mic) bottleneck**. In: *Proceedings 18th IEEE International Conference on Automated Software Engineering*, Oct 2003.

[2] AGRAWAL, D.; LEE, K.-W.; LOBO, J. **Policy-based management of networked computing systems**. *Communications Magazine, IEEE*, 43(10):69 – 75, oct. 2005.

[3] ALLEN, A.; LESLIE, S.; WU, Y.; CLARKE, P.; TIRADO, R. **Self-configuring user-centric communication services**. In: *Systems, 2008. ICONS 08. Third International Conference on*, p. 253 –259, april 2008.

[4] ALLEN, A. A. **Abstractions to Support Dynamic Adaptation of Communication Frameworks for User-Centric Communication**. PhD thesis, Florida International University, 2011.

[5] ALLEN, A. A.; WU, Y.; CLARKE, P. J.; KING, T. M.; DENG, Y. **An autonomic framework for user-centric communication services**. In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, New York, NY, USA, 2009. ACM.

[6] ALLISON, M.; ALLEN, A. A.; YANG, Z.; CLARKE, P. J. **A software engineering approach to user-driven control of the microgrid**. In: *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2011.

[7] BALASUBRAMANIAN, K.; GOKHALE, A.; KARSAI, G.; SZTIPANOVITS, J.; NEEMA, S. **Developing applications using model-driven design environments**. *Computer*, 39(2):33 – 40, feb. 2006.

[8] BÉZIVIN, J. **In search of a Basic Principle for Model-Driven Engineering**. *Novatica – Special Issue on UML (Unified Modeling Language)*, 5(2):21–24, 2004.

[9] BLAIR, G. S.; BENCOMO, N.; FRANCE, R. B. **Models@ run.time**. *IEEE Computer*, 42(10):22–27, 2009.

[10] BOETTNER, P.; GUPTA, M.; WU, Y.; ALLEN, A. A. **Towards policy driven self-configuration of user-centric communication**. In: *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, p. 35:1–35:6, New York, NY, USA, 2009. ACM.

[11] BOUTABA, R.; AIB, I. **Policy-based management: A historical perspective**. *Journal of Network and Systems Management*, 15:447–480, 2007.

[12] BRYANT, B. R.; GRAY, J.; MERNIK, M.; CLARKE, P. J.; FRANCE, R. B.; KARSAI, G. **Challenges and directions in formalizing the semantics of modeling languages**. *Computer Science and Information Systems*, p. 225–253, 2011.

[13] BÉZIVIN, J. **On the unification power of models**. *Software and Systems Modeling*, 4:171–188, 2005.

[14] BÉZIVIN, J. **Model driven engineering: An emerging technical space**. In: Lämmel, R.; Saraiva, J.; Visser, J., editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 de **Lecture Notes in Computer Science**, p. 36–64. Springer Berlin / Heidelberg, 2006.

[15] CAZZOLA, W.; POLETTI, D. **Dsl evolution through composition**. In: *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, p. 6:1–6:6, New York, NY, USA, 2010. ACM.

[16] CHEN, K.; SZTIPANOVITS, J.; NEEMA, S. **Toward a semantic anchoring infrastructure for domain-specific modeling languages**. In: *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, New York, NY, USA, 2005. ACM.

[17] CLARK, T.; SAMMUT, P.; WILLANS, J. S. **Applied metamodelling: a foundation for language driven development.** Ceteva, 2008.

[18] CLARKE, P. J.; HRISTIDIS, V.; WANG, Y.; PRABAKAR, N.; DENG, Y. **A declarative approach for specifying user-centric communication**. In: Smari, W. W.; McQuay, W. K., editors, *CTS*, p. 89–98. IEEE Computer Society, 2006.

[19] CLARKE, P. J.; WU, Y.; ALLEN, A. A.; HERNANDEZ, F.; ALLISON, M.; FRANCE, R. **Towards Dynamic Semantics for Synthesizing Interpreted DSMLs**. In: Mernik, M., editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, p. 242–269. IGI Global, 2012.

[20] CLEENEWERCK, T. **Component-Based DSL Development**. In: Pfenning, F.; Smaragdakis, Y., editors, *Generative Programming and Component Engineering*, volume

2830 de **Lecture Notes in Computer Science**, p. 245–264. Springer Berlin / Heidelberg, 2003.

[21] DAMIANOU, N.; DULAY, N.; LUPU, E.; SLOMAN, M. **The ponder policy specification language**. In: Sloman, M.; Lupu, E.; Lobo, J., editors, *Policies for Distributed Systems and Networks*, volume 1995 de **Lecture Notes in Computer Science**, p. 18–38. Springer Berlin / Heidelberg, 2001.

[22] DENG, Y.; SADJADI, S. M.; CLARKE, P. J.; HRISTIDIS, V.; RANGASWAMI, R.; WANG, Y. **CVM - A communication virtual machine**. *Journal of Systems and Software*, 81(10):1640–1662, 2008.

[23] EDWARDS, G.; MEDVIDOVIC, N. **A methodology and framework for creating domain-specific development infrastructures**. In: *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, p. 168 –177, sept. 2008.

[24] **Eclipse Modeling Framework**. http://www.eclipse.org/modeling/emf/.

[25] ESTUBLIER, J.; VEGA, G.; IONITA, A. **Composing domain-specific languages for wide-scope software engineering applications**. In: Briand, L.; Williams, C., editors, *Model Driven Engineering Languages and Systems*, volume 3713 de **Lecture Notes in Computer Science**, p. 69–83. Springer Berlin / Heidelberg, 2005.

[26] FRANCE, R.; RUMPE, B. **Model-driven development of complex software: A research roadmap**. In: *Future of Software Engineering, 2007. FOSE '07*, p. 37 – 54, may 2007.

[27] **A dynamic language for the Java Platform**. http://groovy.codehaus.org/.

[28] **An architectural blueprint for autonomic computing**. White paper, IBM, June 2006.

[29] JÉZÉQUEL, J.-M.; BARAIS, O.; FLEUREY, F. **Model driven language engineering with kermeta**. In: *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] JOUAULT, F.; ALLILAIRE, F.; BÉZIVIN, J.; KURTEV, I.; VALDURIEZ, P. **ATL: a QVT-like transformation language**. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, p. 719–720, New York, NY, USA, 2006. ACM.

[31] JOUAULT, F.; BÉZIVIN, J.; CONSEL, C.; KURTEV, I.; LATRY, F. **F.: Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages**. In: *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development*, DSPD'06, July 2006.

[32] KARSAI, G.; AGRAWAL, A.; SHI, F.; SPRINKLE, J. **On the use of graph transformations for the formal specification of model interpreters**. 2003.

[33] KELLY, S.; TOLVANEN, J.-P. **Domain-Specific Modeling: Enabling Full Code Generation**. John Wiley & Sons, New Jersey, NY, USA, 2008.

[34] KEPHART, J.; CHESS, D. **The vision of autonomic computing**. *Computer*, 36(1):41 – 50, jan 2003.

[35] KOURIE, D.; FICK, D.; WATSON, B. **Virtual machine framework for constructing domain-specific languages**. *Software, IET*, 3(1):1 –13, february 2009.

[36] KRAMER, J.; MAGEE, J. **Self-managed systems: an architectural challenge**. In: *2007 Future of Software Engineering*, FOSE '07, p. 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[37] KURTEV, I. **State of the art of QVT: A model transformation language standard**. In: Schürr, A.; Nagl, M.; Zündorf, A., editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 de **Lecture Notes in Computer Science**, p. 377–393. Springer Berlin / Heidelberg, 2008.

[38] KURTEV, I.; BÉZIVIN, J.; JOUAULT, F.; VALDURIEZ, P. **Model-based DSL frameworks**. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, New York, NY, USA, 2006. ACM.

[39] LIEBERMAN, H.; PATERNA, F.; KLANN, M.; WULF, V. **End-user development: An emerging paradigm**. In: Lieberman, H.; Paterna, F.; Wulf, V., editors, *End User Development*, volume 9 de **Humana Computer Interaction Series**, p. 1–8. Springer Netherlands, 2006.

[40] LOBO, J.; BHATIA, R.; NAQVI, S. **A policy description language**. In: *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, AAAI '99/IAAI '99, p. 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[41] MELLOR, S.; SCOTT, K.; UHL, A.; WEISE, D. **Model-driven architecture**. In: Bruel, J.-M.; Bellahsene, Z., editors, *Advances in Object-Oriented Information Systems*, volume 2426 de **Lecture Notes in Computer Science**, p. 233–239. Springer Berlin / Heidelberg, 2002.

[42] MERNIK, M.; HEERING, J.; SLOANE, A. M. **When and how to develop domain-specific languages**. *ACM Comput. Surv.*, Dec 2005.

[43] **Microsoft dynamic systems initiative overview**. White paper, Microsoft Corporation, March 2004.

[44] MULLER, P.-A.; FLEUREY, F.; JÉZÉQUEL, J.-M. **Weaving executability into object-oriented meta-languages**. In: Briand, L.; Williams, C., editors, *Model Driven Engineering Languages and Systems*, volume 3713 de **Lecture Notes in Computer Science**, p. 264–278. Springer Berlin / Heidelberg, 2005.

[45] **Human-Usable Textual Notation (HUTN) Specification Version 1.0**. Technical report, OMG, August 2004.

[46] **Meta Object Facility (MOF) Core Specification Version 2.4.1**. Technical report, OMG, August 2011.

[47] PFAHLER, P.; KASTENS, U. **Configuring component-based specifications for domain-specific languages**. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9 - Volume 9*, HICSS '01, p. 9074–, Washington, DC, USA, 2001. IEEE Computer Society.

[48] SCHMIDT, D. **Guest editor's introduction: Model-driven engineering**. *Computer*, 39(2):25 – 31, feb. 2006.

[49] SEIDEWITZ, E. **What models mean**. *Software, IEEE*, 20(5):26 – 32, sept.-oct. 2003.

[50] SELIC, B. **The pragmatics of model-driven development**. *Software, IEEE*, 20(5):19 – 25, sept.-oct. 2003.

[51] SOUSA, G. C. M.; COSTA, F. M.; CLARKE, P. J.; ALLEN, A. **Model-Driven Development of DSML Execution Engines**. In: *Proceedings of the 7th International Workshop on Models@run.time*, Oct 2012.

[52] SPINELLIS, D. **Notable design patterns for domain-specific languages**. *J. Syst. Softw.*, 56(1):91–99, Feb. 2001.

[53] SPRINKLE, J. **Model-integrated computing**. *Potentials, IEEE*, 23(1):28 – 30, feb.-march 2004.

[54] SPRINKLE, J.; AGRAWAL, A.; LEVENDOVSZKY, T.; SHI, F.; KARSAI, G. **Domain model translation using graph transformations**. In: *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, p. 159 – 168, april 2003.

[55] STAHL, T.; VOELTER, M.; CZARNECKI, K. **Model-Driven Software Development: Technology, Engineering, Management**. John Wiley & Sons, 2006.

[56] VAN DEURSEN, A.; KLINT, P.; VISSER, J. **Domain-specific languages: an annotated bibliography**. *SIGPLAN Not.*, June 2000.

[57] WANG, Y.; CLARKE, P. J.; WU, Y.; ALLEN, A.; DENG, Y. **Runtime models to support user-centric communication**. In: *Proceedings of the 3rd International Workshop on Models@runtime*, Sept 2008.

[58] WANG, Y.; WU, Y.; ALLEN, A.; ESPINOZA, B.; CLARKE, P.; DENG, Y. **Towards the operational semantics of user-centric communication models**. In: *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, p. 254 –262, july 2009.

[59] WHITE, J.; HILL, J. H.; GRAY, J.; TAMBE, S.; GOKHALE, A. S.; SCHMIDT, D. C. **Improving domain-specific language reuse with software product line techniques**. *IEEE Software*, 26:47–53, 2009.

[60] WHITE, S.; HANSON, J.; WHALLEY, I.; CHESS, D.; KEPHART, J. **An architectural approach to autonomic computing**. In: *Autonomic Computing, 2004. Proceedings. International Conference on*, p. 2 – 9, may 2004.

[61] WILKINSON, M. **Designing an 'adaptive' enterprise architecture**. *BT Technology Journal*, 24(4):81–92, Oct. 2006.

[62] ÁLVAREZ, J.; EVANS, A.; SAMMUT, P. **Mapping between levels in the metamodel architecture**. In: Gogolla, M.; Kobryn, C., editors, *«UML» 2001 ? The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 de **Lecture Notes in Computer Science**, p. 34–46. Springer Berlin / Heidelberg, 2001.

APÊNDICE A

Modelo do Intermediador de Comunicação em Rede

Neste apêndice são mostrados todos os elementos que compõem o modelo que define o intermediador de serviços descrito no Capítulo 6. Os elementos do modelo em questão são instâncias dos elementos que integram o metamodelo descrito no Capítulo 4 e são descritos utilizando *Human Usable Textual Notation* (HUTN) [45].

Código A.1 Eventos da interface da camada.

```
1   iface: Interface layer {
2     signals: Event LoginFailed {
3       parameters: Parameter framework
4     }
5     signals: Event SchemaReceived {
6       parameters: Parameter sender
7       parameters: Parameter schema
8     }
9     signals: Event SchemaFailed {
10      parameters: Parameter receiver
11      parameters: Parameter schema
12    }
13  }
```

Código A.2 Chamadas da interface da camada.

```
1   iface: Interface layer {
2     provides: Call LoginAll
3     provides: Call LogoutAll
4     provides: Call SendSchema {
5       parameters: Parameter receivers
6       parameters: Parameter schema
7     }
8     provides: Call CreateSession {
9       parameters: Parameter session
10    }
11    provides: Call DestroySession {
12      parameters: Parameter session
13    }
14    provides: Call AddParty {
15      parameters: Parameter session
16      parameters: Parameter participant
17    }
18    provides: Call RemoveParty {
19      parameters: Parameter session
20      parameters: Parameter participant
21    }
22    provides: Call EnableMedium {
23      parameters: Parameter session
24      parameters: Parameter medium
25    }
26    provides: Call EnableMediumReceiver {
27      parameters: Parameter session
28      parameters: Parameter medium
29    }
30    provides: Call DisableMedium {
31      parameters: Parameter session
32      parameters: Parameter medium
33    }
34  }
```

Código A.3 Recursos da camada.

```
1    resourceManager: InstanceResourceManager {
2      instances: Instance Skype {
3        impl: "cvm.ncb.adapters.SkypeAdapter"
4        iface: Interface resource
5        features: Feature Audio {
6          attributes: Attribute Enabled { value: true }
7          attributes: Attribute NumberOfUsers { value: 2 }
8        }
9        features: Feature Video {
10         attributes: Attribute Enabled { value: true }
11         attributes: Attribute NumberOfUsers { value: 2 }
12       }
13     }
14     instances: Instance Smack {
15       impl: "cvm.ncb.adapters.SmackAdapter"
16       iface: Interface resource
17       features: Feature Audio {
18         attributes: Attribute Enabled { value: true }
19         attributes: Attribute NumberOfUsers { value: 2 }
20       }
21       features: Feature Video {
22         attributes: Attribute Enabled { value: true }
23         attributes: Attribute NumberOfUsers { value: 4 }
24       }
25     }
26     instances: Instance Asterisk {
27       impl: "cvm.ncb.adapters.SIPAdapter"
28       iface: Interface resource
29       features: Feature Audio {
30         attributes: Attribute Enabled { value: true }
31         attributes: Attribute NumberOfUsers { value: 8 }
32       }
33     }
34
35   }
36
```

Código A.4 Eventos da interface dos recursos.

```
1    resourceManager: InstanceResourceManager {
2        signals: Event LoginFailed
3        signals: Event SchemaReceived {
4          parameters: sender
5          parameters: schema
6        }
7        signals: Event SchemaFailed {
8          parameters: receiver
9          parameters: schema
10       }
11       signals: Event MediumFailed {
12         parameters: medium
13       }
14     }
15   }
```

Código A.5 Chamadas da interface dos recursos.

```
1   resourceManager: InstanceResourceManager {
2     ifaces: Interface resource {
3       provides: Call Login
4       provides: Call Logout
5       provides: Call SendSchema {
6         parameters: receiver
7         parameters: schema
8       }
9       provides: Call CreateSession {
10        parameters: session
11      }
12      provides: Call DestroySession {
13        parameters: session
14      }
15      provides: Call AddParty {
16        parameters: session
17        parameters: participant
18      }
19      provides: Call RemoveParty {
20        parameters: session
21        parameters: participant
22      }
23      provides: Call EnableMedium {
24        parameters: session
25        parameters: medium
26      }
27      provides: Call EnableMediumReceiver {
28        parameters: session
29        parameters: medium
30      }
31      provides: Call DisableMedium {
32        parameters: session
33        parameters: medium
34      }
35    }
36  }
```

Código A.6 Tipos de dados mantidos pela camada.

```
1     stateManager: StateManager {
2       stateTypes: State Connection {
3         properties: Property session
4         properties: Property medium
5         properties: Property participants
6         properties: Property framework
7         key: Property session
8       }
9     }
10
```

Código A.7 Definição do gerenciamento autônomo.

```
1     autonomicManager: AutonomicManager {
2       identifies: Symptom FrameworkFailed {
3         bindings: Binding event {
4           bindable: Event resource.MediumFailed
5         }
6         bindings: Binding conn {
7           bindable: State Connection
8         }
9         conditions: Condition {
10          expression: "event.source == conn.framework && event.medium == conn.medium"
11        }
12      }
13      requests: ChangeRequest ChangeFramework {
14        basedOn: Symptom FrameworkFailed
15      }
16      plans: ChangePlan Plan1 {
17        basedOn: ChangeRequest ChangeFramework
18        action: ActionExecution {
19          action: SequenceAction ChangeFramework
20          bindings: ParameterBinding {
21            parameter: ActionParameter ChangeFramework.session
22            value: ExpressionValue "conn.session"
23          }
24          bindings: ParameterBinding {
25            parameter: ActionParameter ChangeFramework.medium
26            value: ExpressionValue "event.medium"
27          }
28          bindings: ParameterBinding {
29            parameter: ActionParameter ChangeFramework.framework
30            value: ExpressionValue "event.source"
31          }
32        }
33      }
34    }
35
```

Código A.8 Definição de políticas e sua avaliação.

```
1    policyManager: PolicyManager {
2      context: PolicyEvaluationContext General {
3        parameters: Parameter session
4        parameters: Parameter medium
5        parameters: Parameter numOfParticipants
6        policies: Policy Audio01 {
7          businessValue: 100
8          condition: Condition {
9            expression: "Audio.Enabled"
10         }
11       }
12       policies: Policy Audio02 {
13         businessValue: 96
14         condition: Condition {
15           expression: "numOfParticipants <= Audio.NumberOfUsers"
16         }
17       }
18     }
19     points: PolicyEvaluationPoint {
20       signal: Call layer.EnableMedium
21       bindings: ParameterBinding {
22         parameter: Parameter General.session
23         value: ParameterValue layer.EnableMedium.session
24       }
25       bindings: ParameterBinding {
26         parameter: Parameter General.medium
27         value: ParameterValue layer.EnableMedium.medium
28       }
29       bindings: ParameterBinding {
30         parameter: Parameter General.numOfParticipants
31         value: ExpressionValue "Connection[session].participants.size()"
32       }
33     }
34     handlers: PolicyEvaluationHandler {
35       action: ActionExecution {
36         action: MacroAction UseFramework
37         bindings: ParameterBinding {
38           parameter: ActionParameter UseFramework.session
39           value: ParameterValue General.session
40         }
41         bindings: ParameterBinding {
42           parameter: ActionParameter UseFramework.framework
43           value: ExpressionValue "resources.first()"
44         }
45       }
46     }
47   }
48
```

Código A.9 Tratador e ação da chamada LoginAll.

```
1    actions: CallAction LoginAllAction {
2      call: Call resource.Login
3      target: Expression "Resource.all.findAll { it.available }"
4    }
5    handlers: Handler {
6      signal: Call layer.LoginAll
7      action: ActionExecution {
8        action: CallAction LoginAllAction
9      }
10   }
```

Código A.10 Tratador e ação da chamada LogoutAll.

```
1    actions: CallAction LogoutAllAction {
2      call: Call resource.Logout
3      target: Expression "Resource.all.findAll { it.available }"
4    }
5    handlers: Handler {
6      signal: Call layer.LogoutAll
7      action: ActionExecution {
8        action: CallAction LogoutAllAction
9      }
10   }
```

Código A.11 Tratador e ação da chamada SendSchema.

```
1    actions: MacroAction SendSchemaAction {
2      impl: "cvm.ncb.actions.SendSchemaActionImpl"
3      parameters: ActionParameter receivers
4      parameters: ActionParameter schema
5    }
6    handlers: Handler {
7      signal: Call layer.SendSchema
8      action: ActionExecution {
9        action: MacroAction SendSchemaAction
10       bindings: ParameterBinding {
11         parameter: ActionParameter SendSchemaAction.receivers
12         value: ParameterValue layer.SendSchema.receivers
13       }
14       bindings: ParameterBinding {
15         parameter: ActionParameter SendSchemaAction.schema
16         value: ParameterValue layer.SendSchema.schema
17       }
18     }
19   }
```

Código A.12 Tratador e ação da chamada CreateSession.

```
1    actions: MacroAction CreateSessionAction {
2      impl: "cvm.ncb.actions.CreateSessionActionImpl"
3      parameters: ActionParameter session
4    }
5    handlers: Handler {
6      signal: Call layer.CreateSession
7      action: ActionExecution {
8        action: MacroAction CreateSessionAction
9        bindings: ParameterBinding {
10         parameter: ActionParameter CreateSessionAction.session
11         value: ParameterValue layer.CreateSession.session
12       }
13     }
14
```

Código A.13 Tratador e ação da chamada DestroySession.

```
1    actions: MacroAction DestroySessionAction {
2      impl: "cvm.ncb.actions.DestroySessionActionImpl"
3      parameters: ActionParameter session
4    }
5    handlers: Handler {
6      signal: Call layer.DestroySession
7      action: ActionExecution {
8        action: MacroAction DestroySessionAction
9        bindings: ParameterBinding {
10         parameter: ActionParameter DestroySessionAction.session
11         value: ParameterValue layer.DestroySession.session
12       }
13     }
14   }
```

Código A.14 Tratador e ação da chamada AddParty.

```
1   actions: MacroAction AddPartyAction {
2     impl: "cvm.ncb.actions.AddPartyActionImpl"
3     parameters: ActionParameter session
4     parameters: ActionParameter participant
5   }
6   handlers: Handler {
7     signal: Call layer.AddParty
8     action: ActionExecution {
9       action: MacroAction AddPartyAction
10      bindings: ParameterBinding {
11        parameter: ActionParameter AddPartyAction.session
12        value: ParameterValue layer.AddParty.session
13      }
14      bindings: ParameterBinding {
15        parameter: ActionParameter AddPartyAction.participant
16        value: ParameterValue layer.AddParty.participant
17      }
18    }
19  }
```

Código A.15 Tratador e ação da chamada RemoveParty.

```
1   actions: MacroAction RemovePartyAction {
2     impl: "cvm.ncb.actions.RemovePartyActionImpl"
3     parameters: ActionParameter session
4     parameters: ActionParameter participant
5   }
6   handlers: Handler {
7     signal: Call layer.RemoveParty
8     action: ActionExecution {
9       action: MacroAction RemovePartyAction
10      bindings: ParameterBinding {
11        parameter: ActionParameter RemovePartyAction.session
12        value: ParameterValue layer.RemoveParty.session
13      }
14      bindings: ParameterBinding {
15        parameter: ActionParameter RemovePartyAction.participant
16        value: ParameterValue layer.RemoveParty.participant
17      }
18    }
19  }
```

Código A.16 Tratador e ação da chamada EnableMedium.

```
1    actions: MacroAction MediumAction {
2      impl: "cvm.ncb.actions.MediumActionImpl"
3      parameters: ActionParameter session
4      parameters: ActionParameter medium
5      parameters: ActionParameter action
6    }
7    handlers: Handler {
8      signal: Call layer.EnableMedium
9      action: ActionExecution {
10       action: MacroAction MediumAction
11       bindings: ParameterBinding {
12         parameter: ActionParameter MediumAction.session
13         value: ParameterValue layer.EnableMedium.session
14       }
15       bindings: ParameterBinding {
16         parameter: ActionParameter MediumAction.medium
17         value: ParameterValue layer.EnableMedium.medium
18       }
19       bindings: ParameterBinding {
20         parameter: ActionParameter MediumAction.action
21         value: FixedValue "enable"
22       }
23     }
24   }
```

Código A.17 Tratador da chamada EnableMediumReceiver.

```
1    handlers: Handler {
2      signal: Call layer.EnableMediumReceiver
3      action: ActionExecution {
4        action: MacroAction MediumAction
5        bindings: ParameterBinding {
6          parameter: ActionParameter MediumAction.session
7          value: ParameterValue layer.EnableMediumReceiver.session
8        }
9        bindings: ParameterBinding {
10         parameter: ActionParameter MediumAction.medium
11         value: ParameterValue layer.EnableMediumReceiver.medium
12       }
13       bindings: ParameterBinding {
14         parameter: ActionParameter MediumAction.action
15         value: FixedValue "receive"
16       }
17     }
18   }
```

Código A.18 Tratador da chamada DisableMedium.

```
handlers: Handler {
  signal: Call layer.DisableMedium
  action: ActionExecution {
    action: MacroAction MediumAction
    bindings: ParameterBinding {
      parameter: ActionParameter MediumAction.session
      value: ParameterValue layer.DisableMedium.session
    }
    bindings: ParameterBinding {
      parameter: ActionParameter MediumAction.medium
      value: ParameterValue layer.DisableMedium.medium
    }
    bindings: ParameterBinding {
      parameter: ActionParameter MediumAction.action
      value: FixedValue "disable"
    }
  }
}
```

Código A.19 Tratador e ação do evento LoginFailed.

```
actions: EventAction GenerateLoginFailedEvent {
  event: layer.LoginFailed
  parameters: ActionParameter framework
}
handlers: Handler {
  signal: Event resource.LoginFailed
  action: ActionExecution {
    action: EventAction GenerateLoginFailedEvent
    bindings: ParameterBinding {
      parameter: ActionParameter GenerateLoginFailedEvent.framework
      value: SourceValue
    }
  }
}
```

Código A.20 Tratador do evento MediumFailed.

```
handlers: Handler {
  signal: Event resource.MediumFailed
  action: ActionExecution {
    action: NoAction /* handled by autonomic manager, stops propagation */
  }
}
```

Código A.21 Ação DisableFramework.

```
1    actions: MacroAction DisableFramework {
2      impl: "cvm.ncb.actions.DisableFrameworkImpl"
3      parameters: ActionParameter framework
4    }
5
```

Código A.22 Ação EnqueueEnableMedium.

```
1    actions: CallAction EnqueueEnableMedium {
2      call: layer.EnableMedium
3      parameters: ActionParameter session
4      parameters: ActionParameter medium
5      bindings: ParameterBinding {
6        parameter: Parameter layer.EnableMedium.session
7        value: ParameterValue EnqueueEnableMedium.session
8      }
9      bindings: ParameterBinding {
10       parameter: Parameter layer.EnableMedium.medium
11       value: ParameterValue EnqueueEnableMedium.medium
12     }
13   }
```

Código A.23 Ação ChangeFramework.

```
1    actions: SequenceAction ChangeFramework {
2      parameters: ActionParameter framework
3      parameters: ActionParameter session
4      parameters: ActionParameter medium
5
6      actions: ActionExecution {
7        action: MacroAction DisableFramework
8        bindings: ParameterBinding {
9          parameter: ActionParameter DisableFramework.framework
10         value: ParameterValue ChangeFramework.framework
11       }
12     }
13     actions: ActionExecution {
14       action: CallAction EnqueueEnableMedium
15       bindings: ParameterBinding {
16         parameter: ActionParameter EnqueueEnableMedium.session
17         value: ParameterValue ChangeFramework.session
18       }
19       bindings: ParameterBinding {
20         parameter: ActionParameter EnqueueEnableMedium.medium
21         value: ParameterValue ChangeFramework.medium
22       }
23     }
24   }
```