UNIVERSIDADE FEDERAL FLUMINENSE

THIAGO NASCIMENTO COMICIO

Uma abordagem baseada em contratos de transformação para segurança dirigida a modelos

NITERÓI

2011

UNIVERSIDADE FEDERAL FLUMINENSE

THIAGO NASCIMENTO COMICIO

Uma abordagem baseada em contratos de transformação para segurança dirigida a modelos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Engenharia de Software.

Orientador:
Christiano Braga

NITERÓI

2011

Uma abordagem baseada em contratos de transformação para segurança dirigida a modelos

Thiago Nascimento Comicio

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Prof. D.Sc. Christiano Braga / IC-UFF (Presidente)

Profa. D.Sc. Viviane Torres da Silva / IC-UFF

Prof. D.Sc. Renato Cerqueira / PUC-Rio

Niterói, 10 de janeiro de 2011.

Resumo

Esta dissertação apresenta uma abordagem baseada em contratos de transformação para modelagem, validação e implementação de Segurança Dirigida a Modelos. Nesta abordagem modelos de política de segurança descritos na linguagem de modelagem SecureUML são transformados em código para controle de acesso na linguagem AAC+JAAS. A linguagem SecureUML descreve modelos de políticas de acesso em um sistema e a linguagem AAC+JAAS foi criada para descrever controles de acesso modulares através de conceitos de programação orientada a aspectos e mecanismos presentes no Serviço de Autenticação e Autorização do Java (JAAS). A especificação da transformação entre SecureUML e AAC+JAAS é feita através da técnica de contratos de transformação. Esta técnica especifica a transformação associando as linguagens presentes neste processo em um metamodelo de transformação. Garante-se a validade da transformação através de verificações de invariantes no metamodelo de transformação. No presente trabalho é apresentada a especificação do contrato de transformação entre SecureUML e AAC+JAAS e uma argumentação da sua corretude. Esta especificação foi implementada na ferramenta SecureUMLToAACJAAS, a qual será apresentada juntamente com estudos de caso que a utilizaram para geração de código em AAC+JAAS.

Palavras-chave: Segurança Dirigida a Modelos, Desenvolvimento Dirigido a Modelos, contratos de transformação, SecureUML, JAAS.

Abstract

This work presents an approach based on transformation contracts for Model Driven Security's validation and implementation. In this approach, security policy models described in SecureUML modeling language are transformed into code described in AAC+JAAS modeling language and used for access control. The SecureUML language defines access control policy models and the AAC+JAAS language has been designed to describe modular access control mechanisms by concepts of aspect-oriented programming and the Java Authentication and Authorization Service (JAAS). The transformation contracts technique is used for the transformation specification, defining a transformation metamodel between the modeling languages and a set of invariants to specify the transformation. The transformation correctness is guaranteed by checking the transformation metamodel's invariants. In this work it is presented the transformation contract specification between SecureUML and AAC+JAAS and an argumentation of its correctness. It is also presented the SecureUMLToAACJAAS tool, that implements the transformation contract, and some case studies that make use of it.

Keywords: Model Driven Security, Model Driven Development, transformation contracts, SecureUML, JAAS.

Sumário

1 Introdução **1**

1.1 Objetivos e contribuições 3

1.2 Trabalhos Relacionados 4

1.3 Organização 8

2 Conceitos preliminares **9**

2.1 O processo de compilação de programas 9

2.2 Desenvolvimento dirigido a modelos 12

2.3 Contratos de transformação 22

3 Segurança dirigida por modelos **28**

3.1 Perfil UML de SecureUML+ComponentUML 29

3.2 Metamodelo de SecureUML+ComponentUML 30

4 Aspectos para controle de acesso **36**

4.1 Programação Orientada a Aspectos 36

4.2 Linguagem AAC 38

4.2.1 Metamodelo de AAC 38

4.2.2 Uma instância do metamodelo de AAC 40

4.2.3 Invariantes de AAC 43

5 Serviço de Autorização e Autenticação do Java **45**

5.1 JAAS 45

5.1.1 Autenticação 46
5.1.2 Autorização 47
5.2 O metamodelo de JAAS 49

6 Especificação da transformação de SecureUML+ComponentUML para AAC+JAAS 52

6.1 AAC+JAAS 54
6.1.1 Metamodelo de AAC+JAAS 54
6.1.2 Um modelo descrito em AAC+JAAS 55
6.2 Metamodelo de transformação 58
6.3 Invariantes do metamodelo de transformação 60
6.4 Discussão 68

7 Corretude do contrato de transformação de SecureUML+ComponentUML para AAC+JAAS 70

7.1 Linearização das hierarquias de *roles* e ações em SecureUML 71
7.1.1 Linearização da hierarquia de *roles* 71
7.1.2 Linearização da hierarquia de ações 72
7.1.3 Preservação dos invariantes de *role* e permissões 74
7.1.4 Preservação dos invariantes sobre as relações entre permissões, ações e recursos pelo contrato de transformação 76
7.2 Transformação para AAC+JAAS 79
7.2.1 Transformação de *role* linearizado para AAC+JAAS 79
7.2.2 Transformação de ações linearizadas para AAC+JAAS 80
7.2.3 Preservação dos invariantes sobre a relação entre papéis e permissões pelo contrato de transformação 81
7.2.4 Preservação dos invariantes sobre a relação entre permissões, ações e recursos pelo contrato de transformação 84
7.3 Discussão 89

8 Implementação da transformação de SecureUML+ComponentUML para AAC+JAAS 90

8.1 Implementação 90

8.1.1 Processo de transformação e utilização do código gerado 93

8.1.2 Restrições de autorização e autenticações no código gerado 96

8.1.3 Falhas nas assertivas 98

8.2 Estudos de caso 100

8.2.1 Gerenciador de Reuniões 100

8.2.2 Sistema de atendimento médico domiciliar 103

8.3 Discussão 106

9 Conclusão 108

9.1 Considerações Finais 108

9.2 Trabalhos futuros 109

Referências 111

Apêndice A – Invariantes de SecureUML+ComponentUML 115

A.1 Papel padrão 115

A.2 Hierarquia de papéis 115

A.3 Permissão padrão 116

A.4 Associação entre ações e recursos 116

A.5 Hierarquia de ações 118

Apêndice B – Invariantes de AAC+JAAS 120

B.1 AAC 120

B.2 JAAS 120

Apêndice C – Invariantes do Contrato de Transformação entre SecureUML+ComponentUML e AAC+JAAS 122

C.1 Relação entre elementos de SecureUML+ComponentUML e AAC+JAAS . 122

C.2 Relação entre elementos de SecureUML+ComponentUML e SecureUML-+ComponentUML linearizado 123

C.3 Relação entre elementos de SecureUML+ComponentUML linearizado e AAC+JAAS 123

C.4 SecureUML+ComponentUML linearizado 124

C.4.1 Papel padrão 125

C.4.2 Permissão padrão 125

C.4.3 Associação entre ações e recursos 126

C.5 AAC+JAAS 127

C.5.1 Papel padrão 127

C.5.2 Permissão padrão 127

Apêndice D – Comparação entre os invariantes das linguagens envolvidas na transformação **128**

Capítulo 1

Introdução

O desenvolvimento de software é caracterizado por diversas fases como planejamento, desenho, desenvolvimento, testes e implantação. A fase de desenho tem por objetivo a modelagem de um software. Esta fase representa um ponto importante do processo de desenvolvimento, pois o maior detalhamento e especificação dos modelos que descrevem o sistema favorecem que sejam evitados erros na fase de implementação por parte dos desenvolvedores e, portanto, a necessidade de extensiva manutenção do código do sistema.

Um dos principais artefatos gerados na fase de desenho de um software são os modelos usados para representá-lo. Uma forma padrão de representação de modelos é através da *Unified Modeling Language* (UML) (OMG, 2010), uma linguagem de modelagem de espectro amplo que permite a especificação de sistemas utilizando conceitos de orientação a objetos.

Apesar da melhor especificação e documentação alcançada com a utilização de modelos para guiar o desenvolvimento, não existem garantias de que o sistema será implementado pelos desenvolvedores conforme o especificado.

A abordagem de Desenvolvimento Dirigido a Modelos (DDM) (KLEPPE *et al.*, 2003; CICCHETTI *et al.*, 2007; OMG, 2003) busca diminuir a distância entre a especificação de um sistema e a sua implementação. DDM é uma técnica de desenvolvimento de software que utiliza *modelos* como os principais artefatos do processo de desenvolvimento. Esta técnica realiza verificações e validações nos modelos e a partir deles gera código. O objetivo principal de DDM é elevar o nível de abstração do processo de desenvolvimento de software de maneira que os desenvolvedores passem a trabalhar com modelos ao invés de código. O código criado a partir do modelo é gerado automaticamente, através de *transformações* entre modelos que devem ser verificadas e validadas.

Um conceito importante em DDM é a noção de *domínio* que representa uma classe de modelos. Domínios são representados através dos chamados *metamodelos* que descrevem a linguagem a partir da qual modelos de um domínio são definidos. Estas linguagens podem ser de espectro amplo, ou seja, capazes de representar diferentes domínios, ou de *domínio específico*, quando descrevem um domínio em particular.

Uma vez que os *modelos* se tornam o ponto principal de DDM, sua validação e verificação representam uma importante tarefa. Ao entender uma transformação entre modelos não como rotinas, mas como um modelo de transformação (BÉZIVIN *et al.*, 2006), a verificação do processo de transformação podem utilizar técnicas de validação de modelos. A técnica de contratos de transformação especifica a transformação através da união dos metamodelos de entrada e saída da transformação num metamodelo de transformação. Sobre os metamodelos de origem, destino e de transformação é definido um conjunto de invariantes que garante a conformidade dos modelos com seus metamodelos e da corretude da transformação. Esta técnica foi apresentada inicialmente em (BRAGA, 2010) e continua em aprimoramento. Neste trabalho ela foi utilizada com o objetivo de exercitar a técnica ao descrever uma transformação entre modelos descritos na linguagem SecureUML para a linguagem AAC+JAAS.

Segurança dirigida por modelos (MDS, sigla em inglês) (BASIN *et al.*, 2006) é uma especialização de DDM que utiliza uma linguagem específica de domínio, SecureUML, para a descrição de políticas de controle de acesso. A linguagem SecureUML é uma especialização da UML. Em SecureUML, o engenheiro de segurança pode especificar papéis de usuário e restrições à estes papéis para o acesso de usuários à recursos computacionais, seguindo a abordagem de controle de acesso por papéis (RBAC, sigla em inglês) (FERRAIOLO; KUHN, 1992). Com MDS, o engenheiro de segurança se preocupa em especificar e a analisar políticas de controle de acesso utilizando SecureUML. O código que implementa uma dada política de segurança é gerado automaticamente por um transformador que sabe como produzir código numa plataforma específica para uma dada política de segurança.

Uma forma de modularizar o controle de acesso de um sistema é através da utilização dos conceitos de programação orientada a aspectos. Isso se torna possível na medida em que a verificação de autorização de acesso a um recurso pode ser entendida como um interesse (*concern*) que ocorre antes da execução de um método no sistema. Um exemplo deste uso é a linguagem de modelagem AAC (BRAGA, 2010), que após análise de sua estrutura e invariantes foi modificada neste trabalho.

Outra característica de um controle de acesso é a autenticação. A linguagem Java, por exemplo, possui um serviço padrão para efetuar autenticação e autorização de usuários, o Serviço de Autenticação e Autorização do Java (JAAS)(OAKS, 2001). O JAAS disponibiliza elementos para controle de acesso com autenticação nativa em diversos sistemas com possibilidade de extensão deste mecanismo. Através do estudo dos elementos de JAAS, foi definido neste trabalho uma linguagem de modelagem que o descreve, podendo assim ser utilizado em transformações entre modelos e servir para exercitar as validações deste processo. Com o intuito de aumentar a usabilidade de AAC através de elementos de autenticação, foi criada uma nova linguagem chamada AAC+JAAS.

1.1 Objetivos e contribuições

Este trabalho possui dois objetivos principais. O primeiro deles é analisar e estender a linguagem AAC com a finalidade de aumentar a sua usabilidade através da adição de elementos de autenticação. O segundo objetivo consiste em exercitar a técnica de validação de transformação de modelos chamada contratos de transformação no contexto de políticas de controle de acesso. O foco do seu uso será na especificação de uma transformação que utilize a extensão da linguagem AAC referenciada no primeiro objetivo.

As contribuições desta dissertação consistem nos seguintes pontos:

- Criação de uma nova linguagem de modelagem chamada AAC+JAAS. Esta linguagem representa uma extensão que agrega elementos de autenticação do JAAS à modularização alcançada por conceitos de orientação a aspectos da linguagem AAC. O aumento da usabilidade desta nova linguagem frente a linguagem AAC ocorre ao suprimir a necessidade de implementar mecanismos de autenticação, uma vez que estes já estão presentes na nova linguagem.

- Exercício da técnica de contratos de transformação aplicada ao contexto de controle de acesso. A discussão sobre a sua importância ocorre do fato de esta técnica tornar possível a argumentação da corretude de uma transformação entre modelos. O resultado deste exercício é a especificação de uma transformação de Segurança Dirigida a Modelos culminando na implementação da transformação na forma de uma ferramenta chamada SecureUMLToAACJAAS.

1.2 Trabalhos Relacionados

Este trabalho fez uso da abordagem de programação orientada a aspectos para controle de acesso. O uso de aspectos em segurança está entre os usos mais bem sucedidos no paradigma de Orientação a Aspectos com destaque para (JÜRJENS; HOUMB, 2005; RAY *et al.*, 2004) como representantes do uso de orientação a aspectos no controle de acesso.

A abordagem UMLSec (JÜRJENS; HOUMB, 2005) para desenvolvimento dirigido a modelos é provavelmente a mais próxima da abordagem apresentada neste trabalho. UMLSec é um perfil UML para a especificação de sistemas com segurança crítica, com particular suporte para protocolos de criptografia. O perfil de UMLSec define estereótipos, valores de rótulos (*tagged values*) e restrições que estendem UML permitindo a adição de informações relacionadas a segurança nos diagramas UML. A diferença entre a abordagem apresentada nesta dissertação e a descrita através de UMLSec é que esta última busca a especificação da segurança da troca de mensagens e a forma segura de transferência de dados em um sistema, enquanto que a abordagem desta dissertação consiste no controle de acesso de usuários a recursos controlados definidos por classes e acessados por métodos que compõem estas classes. A abordagem UMLSec se preocupa em definir um modelo de segurança e ainda efetuar verificações de possíveis intrusões, fazendo uso de diferentes diagramas da UML. A abordagem aqui apresentada é mais simplificada com relação a análise da segurança dos modelos, pois o foco é garantir a validação dos modelos envolvidos e do processo de transformação entre as linguagens de modelagem sem que houvessem análises do modelo SecureUML definido pelo analista de segurança. Outra diferença é a forma de validação descrita em UMLSec que utiliza provadores de teoremas utilizando instâncias dos modelos definidos. Conforme será mencionado a seguir, a abordagem desta dissertação é extensível a outras formas de validação e poderia ser complementada através de outras técnicas.

Em (RAY *et al.*, 2004) os autores apresentam a modelagem de segurança através de aspectos e de padrões que facilitem o reuso da modularização em sistemas com restrições de acesso similares. Os autores utilizam programação orientada a aspectos para modelar o controle de acesso através do conceito de *model weaving* para integrar conceitos de interesses de segurança com modelos funcionais. Apesar da relação dos trabalhos estar na utilização de interesses de segurança, na abordagem dos autores é necessário que os analistas de requisitos conheçam a forma de mapeamento do processo de geração da infraestrutura de segurança a partir dos modelos de segurança, similar ao que ocorre na definição de SecureUML. A presente abordagem busca gerar a infraestrutura de segu-

rança de forma transparente utilizando a linguagem AAC+JAAS e efetuando as devidas validações na geração dos modelos e do código e, portanto, focam em pontos diferentes do processo de utilização de aspectos em segurança.

Outros trabalhos como (BLANCO *et al.*, 2009) apresentam a geração de uma arquitetura de segurança baseada em RBAC para domínios específicos através de abordagem MDA, entretanto sem a especificação de um modelo genérico para a política de segurança ou de verificações e validações das transformações utilizadas. A inclusão de mecanismos de validação e especificação aumentam a confiabilidade nos sistemas gerados e auxiliam em seus desenvolvimentos em diversos níveis. Neste trabalho, serão apresentadas argumentações sobre o benefício da utilização da abordagem de contratos de transformação para especificação do processo de transformação entre duas linguagens específicas, reforçando a importância da utilização de validações no processo de desenvolvimento dirigido a modelos.

Em (BASIN *et al.*, 2010), os autores apresentam a utilização de MDS estendendo SecureUML para controle de acesso de sistemas relacionados às suas interfaces GUI. Um novo modelo é gerado a partir dos recursos apresentados nas políticas de controle de acesso e de modelos de interface, no qual o controle de autorização ocorre somente sobre os elementos gráficos da interface que acessam os recursos. Uma vez que os recursos são componentes visuais de uma interface gráfica, esta abordagem se torna mais específica para aplicações com interação visual com o usuário. O presente trabalho apresenta uma infraestrutura de segurança modularizada utilizando aspectos, enquanto que o referido trabalho apresenta um nova extensão da linguagem SecureUML, ou seja, seus objetivos são complementares. Dessa forma, o referido trabalho poderia ser utilizado em conjunto com o trabalho apresentado nesta dissertação. Uma nova linguagem de modelagem seria definida para modelar o controle de acesso a aplicações através de programação orientada a aspectos com o objetivo de modularizar a verificação de políticas de acesso relacionado a componentes de uma interface visual.

A ideia de modelo de transformação é central nesta dissertação. Em (BÉZIVIN *et al.*, 2006), os autores apresentam o conceito de modelo de transformação ao invés de transformações de modelos. Os autores descrevem os benefícios de focar transformações como modelos omitindo detalhes do processo de transformação. Desta maneira modelos são considerados instâncias de metamodelos e transformações podem ser descritas em conformidade com um metamodelo da transformação em específico. Deste ponto de vista, é possível descrever um modelo de transformação formalmente para que seja validado e

verificado. A técnica de contratos de transformação exercitada nesta dissertação define a utilização de um metamodelo de transformação para especificar a transformação e utiliza invariantes OCL para garantir a consistência da transformação e dos modelos envolvidos, permitindo a inclusão de novos mecanismos de validação de modelos ao modelo de transformação. Os passos da transformação não são o foco do processo, mas garantir que a relação entre os elementos das linguagens envolvidas na transformação são atendidas de forma correta gerando modelos de saída com menos possibilidade de erro.

A utilização de invariantes para especificar modelos de transformação é outro ponto chave deste trabalho. A corretude de uma transformação entre modelos tem sido discutida por muitos autores. Em (ANASTASAKIS *et al.*, 2007) o autor enfatiza que uma especificação de transformação de modelos pode ser considerada um tipo especial de modelo e como tal pode ser submetido a técnicas de análise de modelos correntes. De acordo com (CABOT *et al.*, 2009), a corretude de um modelo se torna um ponto importante na medida que defeitos e erros no modelo causarão diretamente defeitos na implementação de sistemas de software devido a aplicação de técnicas de geração de código. Entretanto, notações populares de modelagem não são suficientemente formais para garantir a corretude dos modelos que as utilizam. Dessa forma, um conjunto de técnicas de verificações no nível do modelo são necessárias para garantir a qualidade da especificações de modelos de software.

Algumas abordagens transformam modelos em outras linguagens e formalismos utilizando resolvedores (*solvers*) e provadores de teoremas para validá-los. Em (CABOT *et al.*, 2008), os autores apresentam uma forma de verificar modelos UML/OCL através da sua tradução para um Problema de Satisfação de Restrições (CSP, sigla em inglês). Após a tradução, um resolvedor de restrições é utilizado para verificar se o problema é satisfeito (*satisfiable*) sob um espaço finito de busca de possibilidades. A vantagem deste método é garantir que o modelo e seus invariantes estão bem definidos. Esta abordagem é similar a apresentada como possível extensão deste trabalho através de uma conexão das validações OCL atuais com a abordagem descrita em (BRAGA; HÆUSLER, 2010).

Outro ponto chave na nossa abordagem é o conceito de contratos de transformação. O uso de contratos em transformações de modelos foi apresentada em (BAUDRY *et al.*, 2006) como um dos usos de técnicas de geração de casos de teste para a validação de transformações. Para verificar a consistência das transformações, os autores descrevem a criação de um novo metamodelo adicionando um elemento para conectar os elementos dos metamodelos de origem e destino. Nesta dissertação, a criação de novos elementos é opcional, entretanto, são necessárias conexões entre os elementos dos metamodelos de

entrada e saída que são representadas por associações no metamodelo combinado. Outra diferença é que o uso de contratos não resulta na geração de casos de teste, contudo o objetivo é garantir a consistência do processo de transformação.

Em (CARIOU *et al.*, 2004) os contratos são verificados em três etapas: no modelo de origem, no modelo de destino e no modelo de transformação. Para a transformação, são apresentados dois métodos para verificar a consistência com os contratos de transformação. Na primeira, os elementos do modelo de origem são verificados através de pré-condições e a transformação é verificada através de pós-condições ligando elementos do modelo de origem com o construtor OCL **@pre**. Este construtor permite a comparação entre os elementos do modelo de destino com os elementos do modelo de origem. Neste método, a operação de transformação é descrita em OCL e os metamodelos de origem e destino são os mesmos ou o de destino é muito próximo ao de origem, i.e., representado pelo metamodelo de origem com a adição de alguns elementos novos. O segundo método consiste na criação de dois pacotes no nível dos metamodelos, onde o primeiro contém elementos do metamodelo de origem e o segundo contém elementos do metamodelo de destino. A operação de transformação também é definida em OCL tendo como entrada um pacote para o modelo de origem e resultando em um pacote do modelo de destino. Neste caso, ambos os elementos do modelo de origem e de destino podem ser referenciados na operação de transformação de forma distinta devido aos pacotes.

A diferença entre a presente abordagem e os métodos acima discutidos é o uso de OCL para a operação de transformação, que não é fixado na nossa abordagem, entretanto, houve um intensivo uso de OCL neste trabalho. Pode-se considerar um ponto positivo a utilização de OCL como o padrão não só para a especificação das restrições, mas também para a transformação dos modelos. No entanto, este uso limita a transformação, como no primeiro método, onde os elementos dos metamodelos de origem e de destino devem ser os mesmos. No segundo método, é necessário criar um novo operador OCL para mapear elementos de origem e destino. Em nossa abordagem, o designer deve especificar o metamodelo combinado através da associação entre os elementos de origem e os elementos-alvo de maneira explícita. No entanto, a operação de transformação não se destina a ser expressa com OCL, permitindo que a transformação aconteça de uma forma mais genérica. Além disso, os métodos definidos pelos autores são muito dependentes do metamodelo da UML. O nosso processo é diferente e ocorre sob a premissa de que modelos estão em conformidade com metamodelos, que representam linguagens. A transformação de modelos pode ser vista como transformação entre linguagens, onde os contratos de transformação são invariantes do processo e não apenas suas pré e pós-condições.

1.3 Organização

O restante desta dissertação está organizado da seguinte maneira. No Capítulo 2 são apresentados os conceitos preliminares relacionados a abordagem de Desenvolvimento Dirigido a Modelos adotada neste trabalho utilizando contratos de transformação. No Capítulo 3 é apresentada a abordagem de Segurança Dirigida a Modelos e a linguagem de modelagem SecureUML. No Capítulo 4 é apresentada a linguagem de modelagem AAC utilizada no controle de acesso através de aspectos. No Capítulo 5 é apresentado o *framework* JAAS e uma linguagem de modelagem para descrever parte de seus elementos. No Capítulo 6 é apresentada uma especificação da transformação entre modelos descritos na linguagem SecureUML para a linguagem de modelagem AAC+JAAS. O Capítulo 7 apresenta uma argumentação sobre a corretude da transformação. A implementação da transformação na ferramenta SecureUMLToAACJAAS é discutida no Capítulo 8. Por fim, o Capítulo 9 é destinado aos trabalhos relacionados e às considerações finais desta dissertação.

Capítulo 2

Conceitos preliminares

Os modelos utilizados na abordagem de Desenvolvimento Dirigido a Modelos são descritos por linguagens de modelagem. O processo de transformação de modelos pode então ser descrito como um processo de transformação entre linguagens da mesma forma que o processo de compilação de programas. Uma vez que erros na instância da linguagem de entrada de uma transformação podem ocasionar erros na instância da linguagem de saída gerada, devem ser utilizadas técnicas de verificação e validação em transformações de linguagens. A técnica de contratos de transformação representa uma técnica de validação da transformação entre modelos que tem por objetivo especificar o processo de transformação na forma de um modelo de transformação. Neste capítulo serão apresentados os conceitos do processo de compilação de programas, da abordagem de Desenvolvimento Dirigido a Modelos e da técnica de contratos de transformação utilizadas como base deste trabalho.

2.1 O processo de compilação de programas

Quando os computadores foram criados, eles ocupavam grandes salas e necessitavam de grande quantidade de energia para seu funcionamento. Seu poder computacional não era superior à maioria das máquinas de calcular de hoje e para que os programas fossem executados era necessário que os programadores escrevessem código em linguagem de máquina. Linguagem de máquina é a sequência de *bits* que diretamente controla um processador, fazendo-o somar, comparar e mover dados de um local para outro. À medida que programas maiores foram criados surgiu a necessidade da criação de linguagens de montagem (*assembly*) com mnemônicos que referenciavam as instruções de linguagem de máquina.

A partir da evolução dos computadores e das diferentes arquiteturas criadas, tornam-

se necessárias novas formas para substituir o processo de reescrita dos programas para cada tipo de máquina facilitando a manutenção dos sistemas. Surgem então as linguagens de programação em alto-nível como uma forma de abstrair os passos de um programa que são traduzidos para linguagem de montagem ou de máquina (SCOTT, 2006).

As linguagens de programação de alto-nível podem ser divididas em: linguagens com propósito geral e linguagens específicas de domínio. Uma linguagem de propósito geral pode ser aplicada para escrever programas em vários domínios de aplicação. Por exemplo, a linguagem Java pode ser utilizada para escrever programas de estatística, manipulação de banco de dados, manipulação de arquivos em formatos específicos e para controle de acesso a aplicações. Em contrapartida, uma linguagem específica de domínio é uma linguagem descrita para um específico domínio de aplicação e contém construções que aumentam a sua expressividade neste domínio específico. Por exemplo, a linguagem SQL (*Structured Query Language*) é utilizada apenas para manipulação de banco de dados e a linguagem R é utilizada para aplicações de estatística (MERNIK *et al.*, 2005).

O controle de acesso a um sistema, tema deste trabalho, é um domínio específico e pode ser representado por uma linguagem com construções que auxiliem na definição de programas em tal domínio. Um exemplo de código escrito em uma linguagem específica de domínio para controle de acesso é apresentado na Figura 2.1 para definir uma permissão de acesso de usuário através de um papel a um recurso. Neste exemplo é definida a permissão de execução de métodos de um sistema de arquivo para um papel chamado *papel_a*. A palavra reservada *Role* define um tipo de papel de usuários e o identificador *papel_a* representa o nome do papel, bem como *Resource* define um tipo de recurso a ser acessado e *sistema_arquivos* o nome do recurso. Para definição de uma permissão utiliza-se a construção infixada entre parênteses contendo o nome de um papel, seguido de vírgula e um nome de um recurso.

```
1 Role role_a;
2 Resource filesystem;
3 (role_a, filesystem);
```

Figura 2.1: Código de exemplo para uma linguagem de alto nível

Ao processo de tradução de linguagens de alto-nível para linguagem de montagem ou de máquina se dá o nome de compilação e o programa que executa tal tarefa é chamado de compilador. A Figura 2.2 apresenta o processo de compilação de um programa.

O primeiro passo da compilação é a leitura ou análise léxica, onde o código fonte é lido letra a letra e são construídos *tokens* da linguagem. *Tokens* são categorias de lexemas

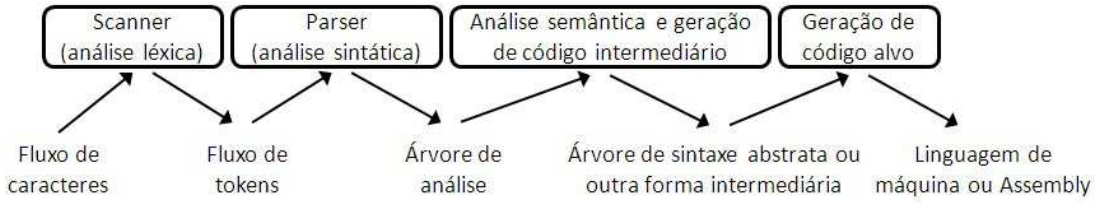

Figura 2.2: Processo de compilação

e os lexemas são as unidades sintáticas de mais baixo nível de uma linguagem. No código apresentado na Figura 2.1, os *tokens* e os lexemas gerados são apresentados na Tabela 2.1.

Lexema	*Token*
Role	palavra-role
papel_a	identificador
Resource	palavra-resource
sistema_arquivo	identificador
(	abre-parênteses
,	vírgula
;	ponto-e-vírgula
}	fecha-chaves

Tabela 2.1: *Tokens* e lexemas gerados na fase de leitura do código de exemplo

Posteriormente, os *tokens* são reunidos pelo analisador sintático em uma árvore de análise (*parse tree*), verificando se o conjunto de *tokens* está bem formado com relação a sintaxe da linguagem na qual ele é descrito.

A sintaxe de uma linguagem de programação é a forma (estrutura) de suas expressões, declarações e blocos de programa (SEBESTA, 2005). Um modo de formalizar a sintaxe de uma linguagem de programação é através da forma de Backus-Naur (BNF, sigla em inglês). BNF é uma metalinguagem, ou seja, uma linguagem utilizada para escrever linguagens de programação que utiliza abstrações para representar estruturas sintáticas.

Na Figura 2.3 é apresentada a descrição de uma linguagem em BNF que define o código apresentado na Figura 2.1. Na especificação da linguagem é definida *declaracao_acesso*, que pode conter definições de papéis, recursos e permissões. Os papéis são definidos pela palavra-reservada *Papel* seguido por um identificador e pelo símbolo de ponto-e-vírgula. Os recursos são definidos pela palavra-reservada *Recurso* seguido por um identificador e pelo símbolo de ponto-e-vírgula. As permissões são formadas por identificadores separados por vírgula entre parênteses.

Recordando a sintaxe de BNF, símbolos não terminais são abstrações compostas de

```
access_statement ::= access_statement_element*
access_statement_element ::=
  role_statement | resource_statement | permission_statement
role_statement ::= "Role" identifier ";"
resource_statement ::= "Resource" identifier ";"
permission_statement ::= "(" identifier "," identifier ")" ";"
identifier ::= "a..z" { "a..z,_,0..9" }*
```

Figura 2.3: Sintaxe em BNF da linguagem de controle de acesso

outras abstrações ou de símbolos terminais; símbolos terminais são lexemas e *tokens*; o operador "::=" define uma regra sintática. Na Figura 2.3 são apresentadas as regras sintáticas com não terminais à esquerda e sequência de terminais e não terminais a direita. O conjunto de caracteres que são separados por aspas ("") representam palavras-reservadas (e.g., *Role*), símbolos especiais (e.g., ","), ou estruturas que definem abstrações como no caso da abstração *identifier*. O símbolo de *pipe* (|) indica alternativas para representar um elemento, por exemplo, a abstração *access_statement_element* é especificada nas abstrações *role_statement*, *resource_statement* e *permission_statement*. Os símbolos de chaves indicam a possibilidade se seus elementos internos serem ou não utilizados e o símbolo "*" indica repetição dos elementos.

Apesar de um programa estar bem formado com relação a uma gramática, não significa que este é um programa válido. A análise semântica visa garantir que um programa está bem formado do ponto de vista de seus tipos. No exemplo apresentado da declaração de acesso da Figura 2.1, não é possível a partir da gramática definida, garantir que a declaração de permissão contém dois identificadores representando um papel e um recurso definidos anteriormente.

Por fim, ocorre a geração de código intermediário e a geração de código de máquina que pode ser executado por um computador.

2.2 Desenvolvimento dirigido a modelos

A *Unified Modeling Language* (UML) (OMG, 2010) é uma linguagem de modelagem de espectro amplo com representações gráficas que permitem aos usuários especificar, visualizar, construir e documentar os componentes de um sistema de software durante as fases de desenho e desenvolvimento. A UML é definida pelo *Object Management Group* (OMG) e descreve diferentes diagramas para representar diferentes visões de um sistema modelado através de objetos. O modelo mais utilizado na UML é o diagrama de classes

que representa as classes de um sistema orientado a objetos. Um sistema gerenciador de arquivos simples, por exemplo, pode ser modelado como uma classe que possui métodos para criar, editar, remover e executar arquivos. A Figura 2.4 apresenta um diagrama de classes representando um sistema gerenciador de arquivos, composto da classe *FileSystem*.

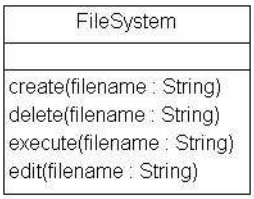

Figura 2.4: Diagrama de classes representando um gerenciador de arquivos

Apesar da melhor especificação e documentação alcançada com a utilização de modelos para guiar o desenvolvimento, não existem garantias que o sistema será implementado pelos desenvolvedores conforme o especificado. O Desenvolvimento Dirigido a Modelos (DDM) é uma abordagem para tornar mais simples o desenvolvimento e manutenção de software que busca diminuir a distância entre a especificação de um sistema e a sua implementação. Seu princípio é aumentar a automação no desenvolvimento de um sistema (KLEPPE *et al.*, 2003). Isso é feito com a utilização de modelos em diferentes níveis de abstração do sistema, onde os modelos são evidenciados como o ponto central do processo de desenvolvimento de software, o que anteriormente era concentrado na codificação (CICCHETTI *et al.*, 2007).

A automação do desenvolvimento é alcançada através de transformações entre modelos que possam ser executadas por um computador. A transformação de modelos tem como definição mais simples a de um processo de conversão de um modelo em outro modelo de um mesmo sistema (OMG, 2003). Normalmente, modelos representados em um maior nível de abstração são transformados em modelos de nível mais baixo, antes que o código possa ser gerado ou o modelo interpretado. Um exemplo é a transformação de diagramas de classe UML em linguagens de alto-nível como Java.

A partir da noção de que modelos são descritos por linguagens, é imediato definir que transformações de modelos relacionam linguagens, como UML e Java. Para tirar vantagem do ganho na interoperabilidade devido ao uso de tais padrões, a definição da sintaxe destas linguagens pode ser descrita como um modelo de classes UML. Este modelo é chamado de metamodelo, pois descreve a sintaxe que modelos bem formados devem seguir. Os modelos bem formados com relação a sintaxe descrita por um metamodelo são considerados instâncias do metamodelo em questão. Um diagrama de classes, por

exemplo, representa uma instância do metamodelo UML.

Um modelo deve estar bem formado em respeito ao seu metamodelo assim como um programa escrito em uma linguagem L deve estar bem formado com relação a sintaxe de L. A Figura 2.5 apresenta o metamodelo para uma linguagem que especifica um controle de acesso simplificado. Qualquer modelo que for escrito a partir dele deve ser visto como uma instância do metamodelo desta linguagem, onde o modelo deve estar bem formado com respeito a este metamodelo.

A metaclasse *Role* define um papel e a metaclasse *Resource* define um recurso controlado. Através das multiplicidades das associações, as duas metaclasses estão relacionadas através de associações com as metaclasses *Permission* que define uma permissão de um papel para executar uma ação sobre um recurso definido pela metaclasse *Action*. Nas associações em UML é possível definir um nome e multiplicidades para cada elemento da associação, que são denominados pontas de associação. A metaclasse *Action* pode ser especializada em ações de leitura dos recursos (*ReadAction*) e ações de escrita nos recursos (*UpdateAction*). Este metamodelo poderia ser descrito em BNF conforme a gramática da Figura 2.6, onde as metaclasses dão origem a não terminais e os terminais são representados naturalmente como objetos do diagrama de objetos, ou seja, instâncias do metamodelo.

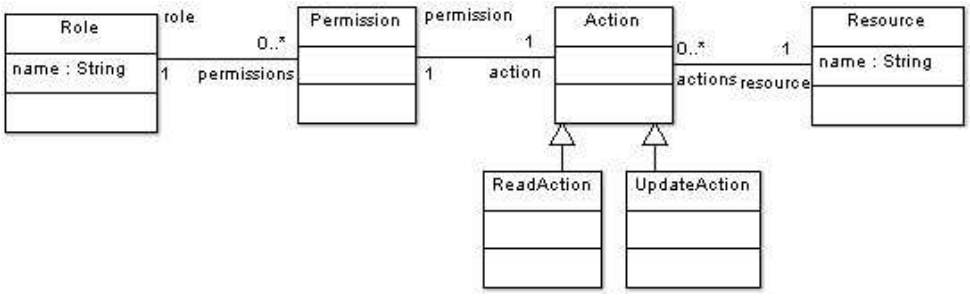

Figura 2.5: Metamodelo de uma linguagem para controle de acesso

```
ModelElement ::= ModelElementDeclaration*
ModelElementDeclaration ::= Role | Resource | RoleResourceAssignment
RoleResourceAssignment ::= "(" Role "," Permission "," Action "," Resource ")"
Action ::= ReadAction | UpdateAction
ReadAction ::= "read" "(" sistema_arquivo ")"
UpdateAction ::= "update" "(" sistema_arquivo ")"
Role ::= papel_a
Resource ::= sistema_arquivo
```

Figura 2.6: Sintaxe em BNF da linguagem do metamodelo para controle de acesso

Uma instância do metamodelo de controle de acesso é apresentada como um modelo

na Figura 2.7. Nele é descrito uma permissão de acesso de leitura de um usuário sob um papel *papel_ a* a um recurso *sistema_ arquivo*.

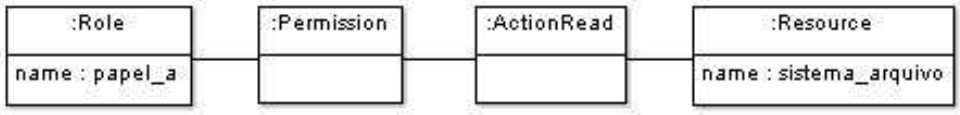

Figura 2.7: Instância do metamodelo de controle de acesso

Uma forma de representar a sintaxe concreta em linguagens de modelagem é através de perfis UML. Um perfil UML é a definição de um conjunto de estereótipos, valores rotulados (*tagged values*), restrições e classes base. O perfil também indica o subconjunto de tipos de elementos que serão utilizados auxiliando os analistas a utilizar somente os elementos específicos para representações de uma determinada área. Em (BOOCH *et al.*, 2005), os autores definem que um perfil UML por ser construído sobre elementos comuns UML não representa uma nova linguagem e pode ser utilizado por ferramentas comuns UML.

Ao representar a sintaxe concreta do metamodelo de controle de acesso, pode-se utilizar um perfil UML específico. A Figura 2.8 representa a política de controle de acesso apresentado na Figura 2.7 descrita neste perfil. As classes com estereótipo $<<Role>>$ representam instâncias da metaclasse *Role* do metamodelo, as classes com estereótipo $<<Resource>>$ representam instâncias da metaclasse *Resource*, as classes de associação com estereótipo $<<Permission>>$ representam instâncias da metaclasse *Permission* e o atributo desta classe de associação está relacionado a um tipo que representa uma ação.

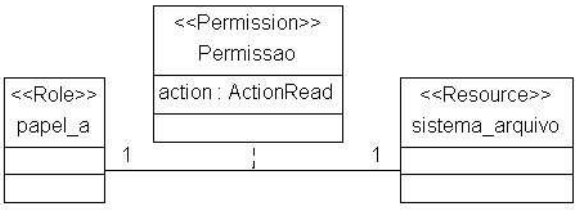

Figura 2.8: Modelo de controle de acesso com perfil UML específico

A partir dos conceitos apresentados de DDM, pode-se desenhar sua relação com os conceitos de linguagens de programação, onde transformações de modelos serão representadas como um processo de compilação de linguagens de programação. A descrição de uma linguagem de modelagem contém uma sintaxe abstrata; uma ou mais sintaxes concretas; um mapeamento entre a sintaxe abstrata e a sintaxe concreta; e uma descrição da semântica (KLEPPE, 2007). Pode-se entender um perfil UML como a sintaxe concreta de uma linguagem de modelagem M, definido como uma extensão de UML. O metamodelo

de M pode ser entendido como a sintaxe abstrata de M. Com esta visão em mente, o primeiro passo que uma transformação de modelos deve executar para transformar um modelo m escrito utilizando um perfil UML que representa uma linguagem de modelagem M é mapear m em uma instância m' do metamodelo de M. Este passo é similar ao que ocorre na fase de *parsing* do código escrito numa linguagem em um compilador. O mapeamento entre os elementos da Figura 2.8 nos elementos da Figura 2.7 poderiam ser descritos pelos seguintes passos:

1. Para cada classe r com estereótipo $<<Role>>$ em m, declare um elemento $\overline{r}$ do tipo *Role* em m', com o valor do atributo *name* de $\overline{r}$ igual ao nome do elemento r;
2. Para cada classe res com estereótipo $<<Resource>>$ em m, declare um elemento $\overline{res}$ do tipo *Resource* em m', com o valor do atributo *name* de $\overline{res}$ igual ao nome do elemento res;
3. Se o atributo a de p é do tipo *ActionRead*, declare um elemento $\overline{a}$ do tipo *ActionRead* em m' e relacione-o a $\overline{p}$;
4. Se o atributo a de p é do tipo *ActionUpdate*, declare um elemento $\overline{a}$ do tipo *ActionUpdate* em m' e relacione-o a $\overline{p}$;
5. Para a conexão de uma permissão p com um papel r, conecte $\overline{p}$ com $\overline{r}$;
6. Para a conexão de uma permissão p com um recurso res, conecte $\overline{a}$ relacionado a $\overline{p}$ com $\overline{res}$;

No processo de transformação, o metamodelo M passa a ser chamado de Ms por representar o metamodelo de entrada da transformação. O próximo passo deste processo consiste na transformação dos elementos de m' do metamodelo Ms em elementos de uma instância n da sintaxe abstrata de um metamodelo Mt, que descreve uma linguagem de saída. Por fim, n passa por um processo de *pretty printing* para gerar uma instância n' da sintaxe concreta de Mt.

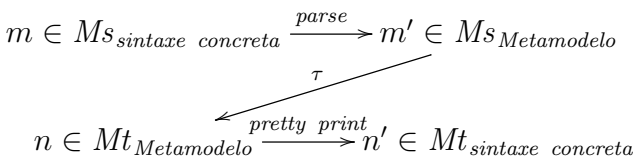

Figura 2.9: Processo de desenvolvimento dirigido a modelos adotado

Neste ponto, o processo de desenvolvimento dirigido a modelos descrito pode ser esquematizado conforme apresentado na Figura 2.9, onde m, m', n, n' são modelos; *Ms* e *Mt* representam as linguagens de entrada e saída de uma transformação de modelos τ; $m \in M$ denota que um modelo m é bem formado com respeito a M; *parse* é o mapeamento de um modelo m escrito de acordo com a sintaxe concreta de uma linguagem de modelagem M em uma versão da sintaxe abstrata m' de m, onde m' é bem formada com respeito ao metamodelo de M; *pretty print* é o mapeamento inverso de *parse*; $M_{Metamodelo}$ representa a sintaxe abstrata para a linguagem de modelagem M, representada como um modelo de classes UML; e $M_{sintaxe\ concreta}$ representa a sintaxe concreta descrita em um formato particular (poderia, por exemplo, ser considerado como a sintaxe concreta de uma linguagem de programação ou um perfil UML para uma linguagem de modelagem), para uma linguagem de programação M.

Não é sempre verdade, contudo, que um modelo bem formado com respeito a um dado metamodelo está em *conformidade com ele*. Por exemplo, podemos estender o metamodelo da linguagem de controle de acesso especializando recursos em entidades e atributos, onde para cada recurso existirá uma ação específica. Para as entidades foi criada a ação *EntityAction* e para os atributos a ação *AttributeAction*. A Figura 2.10 apresenta o metamodelo estendido para esta linguagem. É possível criar um modelo de controle de acesso m bem formado com relação ao metamodelo, onde uma permissão está relacionada a uma ação de entidade, mas que esta ação está relacionada a um atributo. Dessa forma, apesar de bem formado, o modelo não está em conformidade com a linguagem de controle de acesso.

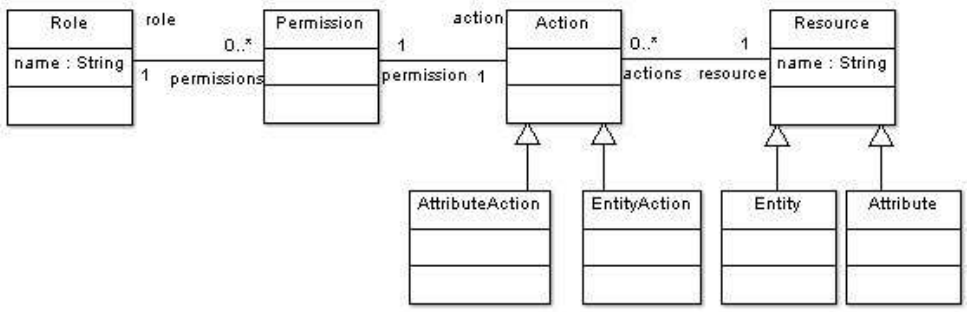

Figura 2.10: Extensão do metamodelo de controle de acesso

O que garante a conformidade de um modelo nesta linguagem é a existência de um *invariante* no metamodelo que especifica que as ações de entidade somente podem estar relacionadas a recursos do tipo *Entity*. Uma vez que este invariante não é satisfeito em um modelo m, este modelo não está em conformidade com o metamodelo que o descreve. A relação de conformidade entre um modelo m e um metamodelo M é dado pela boa

formação de m com respeito a M e a ocorrência de todos os invariantes de M sobre m, assumindo que M seja consistente. Um metamodelo M é considerado consistente, se existe ao menos uma instância que seja bem formada com relação a sua descrição (BRAGA; HÆUSLER, 2010). Neste trabalho assume-se que todos os metamodelos são consistentes.

A relação de conformidade entre um metamodelo e uma instância deste metamodelo é similar ao conceito de verificação de tipos em linguagens de programação. Um programa p sintaticamente correto com respeito a uma linguagem L tem problemas de tipagem caso as regras de tipo em L não sejam satisfeitas em p.

Uma forma de especificar os invariantes de modelos ocorre na utilização da *Object Constraint Language* (OCL)(OMG, 2006). Cada expressão escrita em OCL ocorre sobre tipos definidos nos diagramas UML, por exemplo, classes, interfaces, etc. Desse modo, OCL utiliza aspectos de UML, onde suas expressões adicionam informações que formalizam os modelos orientados a objetos e outros artefatos de modelagem (WARMER; KLEPPE, 2003). OCL é uma linguagem declarativa de alto nível e suas expressões podem ser usadas para especificar operações, ações e propriedades em modelos UML. OCL também pode ser utilizada para especificar restrições em um modelo ou consultas, separadamente dos modelos (OMG, 2006).

Essencialmente, OCL possui construtores para manipular coleções de elementos de modelo tipados em um modelo m, navegando através das relações de m, definindo operações e invariantes em M, onde M é o metamodelo de m. Por exemplo, o invariante *relatesToEntity* apresentado na Figura 2.11 verifica se uma ação de entidade (*EntityAction*) está somente relacionada a recursos do tipo de entidade (*Entity*). Quanto a sintaxe OCL, a palavra-chave *context* define o tipo de objetos aos quais o invariante será aplicado. A palavra-chave *inv* define um invariante. A palavra-chave *self* indica o objeto do contexto definido. A operação booleana *oclIsTypeOf* faz parte dos elementos em OCL e sua finalidade é verificar se um dado elemento é de um tipo específico.

```
1 context EntityAction inv relatesToEntity:
2   self.resource.oclIsTypeOf(Entity);
```

Figura 2.11: Invariante OCL *relatesToEntity*

OCL pode ser utilizada para automaticamente validar modelos UML. Considerando uma implementação de um interpretador OCL, assim como definido em (CLAVEL *et al.*, 2008b), os invariantes de um metamodelo M podem ser aplicados a um modelo m sintaticamente bem formado com respeito a M para garantir a conformidade de m com relação

a M. Dessa forma, antes de aplicar uma transformação de modelos a um dado modelo m, é preciso garantir que m está em conformidade com seu metamodelo M (assim como o apresentado na Figura 2.10), isto é, m é sintaticamente bem formado com respeito a M e todos os invariante em M são satisfeitos em m.

O processo DDM adotado neste trabalho ao se considerar os invariantes é apresentado na Figura 2.12, onde I_M são os invariantes do metamodelo da linguagem de modelagem M e $m \models I_M$ significa que todos os invariantes em I_M são satisfeitos no modelo $m \in M$.

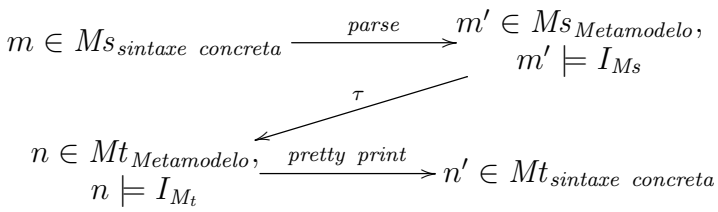

Figura 2.12: Processo de desenvolvimento dirigido a modelos adotado com uso de invariantes

Para ilustrar o processo adotado será utilizada a transformação de um arquivo de políticas de segurança como saída da transformação para o modelo de política de segurança, onde o exemplo da política de controle de acesso de entrada para o sistema de arquivos teria o formato descrito na Figura 2.2.

```
1 Principal papel_a {
2  Permission "Read" sistema_arquivos;
3 }
```

Figura 2.13: Arquivo de política para o controle de acesso do sistema de arquivos

Este arquivo de políticas de segurança é baseado no arquivo de políticas do Sistema de Autenticação e Autorização do Java que será apresentado no Capítulo 5. Em JAAS o elemento *Principal* representa um papel de um usuário ou uma credencial. Os arquivos de política em JAAS relacionam os elementos *Principal* a elementos de permissão através dos elementos de domínios de proteção (*ProtectionDomain*). As permissões descrevem o tipo de acesso que um papel está autorizado a executar sobre um recurso[1].

O metamodelo que define este arquivo de saída é apresentado na Figura 2.14. O tipo *Policy* representa o arquivo de política de acesso. Os elementos do tipo *ProtectionDomain* são os chamados domínios de proteção e são representados pelos blocos gerados no arquivo de política. Os domínios de proteção estão relacionados a elementos do tipo *Principal*

[1] Para ilustração, o metamodelo apresentado é simplificado em relação ao metamodelo do JAAS, pois nos arquivos de política não são especificados os recursos e as ações de acesso aos recursos

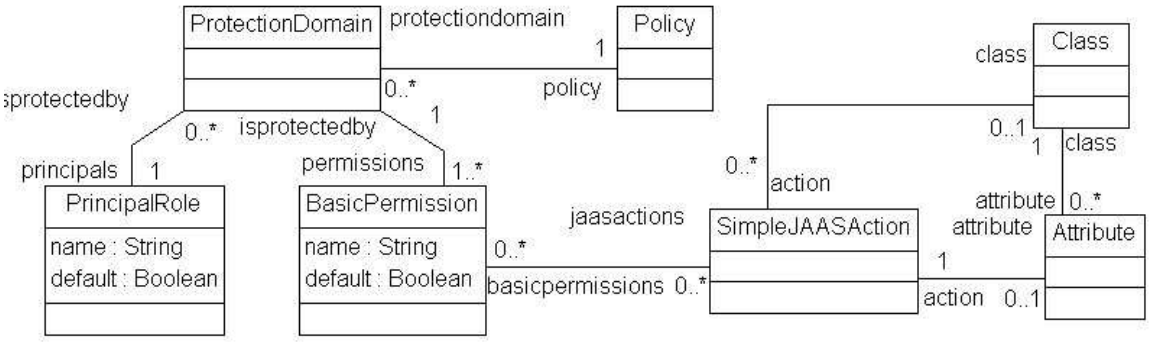

Figura 2.14: Metamodelo de arquivo de políticas de segurança

com elementos de permissão (*BasicPermission*). Uma permissão é relacionada às ações do tipo *SimpleJAASAction*. As ações estão relacionadas a uma classe ou a um atributo. Na Figura 2.15 é apresentado o arquivo de políticas como uma instância do metamodelo da Figura 2.14 transformado a partir do modelo da Figura 2.8.

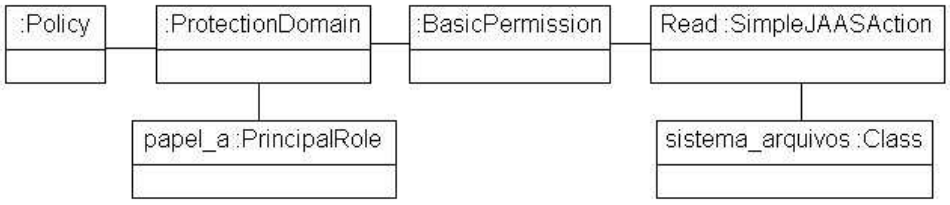

Figura 2.15: Instância do metamodelo do arquivo de políticas de segurança

O processo de *pretty printing* de uma instância deste metamodelo segue os seguintes passos:

1. Gerar um arquivo de política em disco a partir do elemento *pol* do tipo *Policy* ;
2. Para cada domínio de proteção *pd* relacionado a *pol*, escrever o bloco "*Principal* $< nome_principal >$ { *Permission* "$< nome_acao >$" $< nome_recurso >$}", onde
   - $< nome_principal >$ é o nome do *Principal* *p* relacionado ao domínio de proteção *pd*;
   - $< nome_acao >$ é o nome da ação *sja* do tipo *SimpleJAASAction* relacionada a permissão *bp* do *pd*.
   - $< nome_recurso >$ é o nome da classe *c* relacionada a ação *sja* ou o nome da classe *c* seguida por um ponto (".") e pelo nome do atributo *attr* relacionado a ação *sja*.

```
1 context Policy
2 inv existsAPolicy:
3 self.allInstances()->size() = 1
4
5 context SimpleJAASAction
6 inv relatesToAttributeOrClass:
7 self.classe->notEmpty() implies self.atributo->isEmpty() and
8 self.classe->isEmpty() implies self.atributo->notEmpty()
```

Figura 2.16: Invariantes OCL para a linguagem $\text{JAAS}_{Simplificada}$

Seguindo o processo DDM definido na Figura 2.12, a transformação $\tau : Polticasdeacesso \rightarrow JAAS_{simplificado}$ pode ser definida da seguinte forma:

1. É gerado inicialmente um elemento *pol* do tipo *Policy* em n;
2. Os elementos r do tipo *Role* de m' serão transformados em elementos *pr* do tipo *Principal* de n;
3. Os elementos p do tipo *Permission* de m' serão transformados em elementos *bp* do tipo *BasicPermission* em n e a relação de p com r em m' é transformada em um elemento *pd* do tipo *ProtectionDomain* em n relacionado a *bp* e *pr*;
4. Para cada recurso *re* do tipo *Entity* em m' será gerado um elemento c do tipo *Class* em n;
5. Para cada recurso *ra* do tipo *Attribute* em m' será gerado um elemento *at* do tipo *Attribute* em n e a relação entre *ra* e *re* em m' gerará a relação entre *at* e c em n;
6. Para cada ação a do tipo *Action* relacionado a uma permissão p e a um recurso r em m', será gerada uma ação *ja* do tipo *JAASAction* em n relacionada a permissão *bp* transformada a partir de p e a um elemento c do tipo *Class*, caso r seja do tipo *Entity*, ou a um elemento *at* do tipo*Attribute*, caso r seja do tipo *Attribute*.

Antes da geração do código através da execução do *pretty print* no modelo de saída n é necessário verificar os invariantes para a linguagem de saída. Na Figura 2.16 são apresentados dois invariantes de exemplo. O invariante *existsAPolicy* garante que existe um único objeto do tipo *Policy* no modelo, sendo este o objeto que dará origem ao arquivo de políticas. O invariante *relatesToAttributeOrClass* garante que a ação do tipo *SimpleJAASAction* está relacionada a um atributo ou a uma classe, mas nunca aos dois ao mesmo tempo.

2.3 Contratos de transformação

A abordagem de contrato de transformação definida em (BRAGA, 2010) é uma especificação de uma transformação de modelos. Esta é escrita em forma de invariantes que devem ser satisfeitos em um metamodelo de transformação das linguagens de entrada e saída, relacionadas pela transformação de modelos. Um exemplo de invariante para a transformação de modelos $\tau : PoliticasDeAcesso \rightarrow JAAS_{simplificado}$ é que para cada papel r em $m \in PoliticasDeAcesso$ deve existir um *Principal* pr em $\tau(m) \in JAAS_{simplificado}$.

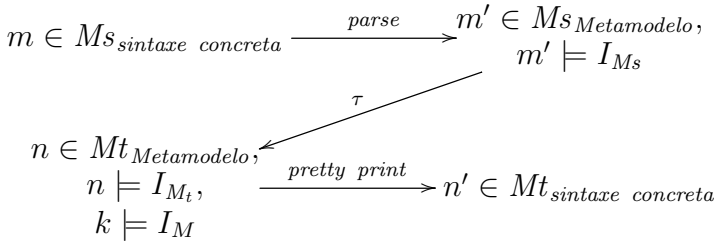

Figura 2.17: Processo de desenvolvimento dirigido a modelos adotado com uso de contratos de transformação

O termo metamodelo de transformação é utilizado para descrever um metamodelo M resultante de uma operação $M_1 \bowtie M_2$ nos metamodelos M_1 e M_2, que estendem os metamodelos M_1 e M_2 por: (i) combinar exclusivamente todos os elementos de M_1 e M_2; (ii) preservar todos os invariantes de M_1 e M_2, ou seja, para todo invariante $\iota \in I_{M_j}$, $j \in \{1, 2\}$, se ι é satisfeito em M_j implica que ι é satisfeito em M e nenhum outro invariante $\kappa \neq \iota$ é satisfeito em M_j como um resultado de $M_1 \bowtie M_2$; e (iii) declarando classes $c \in C$ tais que C é disjunto com as classes de M_1 e M_2, relações $r \in R$, tais que R são disjuntas com as relações de M_1 e M_2, e que r deve relacionar classes de M_1 com M_2 ou classes em C com M_1 ou M_2.

Um contrato de transformação é então definida por um conjunto de invariantes I_M sobre o metamodelo de transformação M, onde $M = M_1 \bowtie M_2$. A abordagem DDM adotada neste trabalho ao se considerar contratos de transformação, pode ser esquematizado conforme a Figura 2.17, onde $M = Ms \bowtie Mt$, $k \in M$, e $k = (m' \bowtie n)$. O modelo k representa uma instância de M e a partir dele será gerado n, através da remoção dos elementos de m'. Por fim, a geração de n' ocorrerá como resultado do processo de *pretty-printing* sobre n.

Para $\tau : Polticasdeacesso \rightarrow JAAS_{simplificado}$ o metamodelo de transformação é definido pela Figura 2.18. Os tipos do metamodelo de entrada são relacionados aos tipos do metamodelo de saída através de invariantes e de associações, por exemplo, a associação

entre *Role* e *PrincipalRole*. Um dos invariantes que garantem a corretude do contrato de transformação é apresentado na Figura 2.19 e garante a correta relação entre atributos do modelo de entrada e do modelo de saída, verificando se eles possuem o mesmo nome e se estão relacionados a ações equivalentes, ou seja, ações do modelo de entrada e do modelo de saída que estão conectadas.

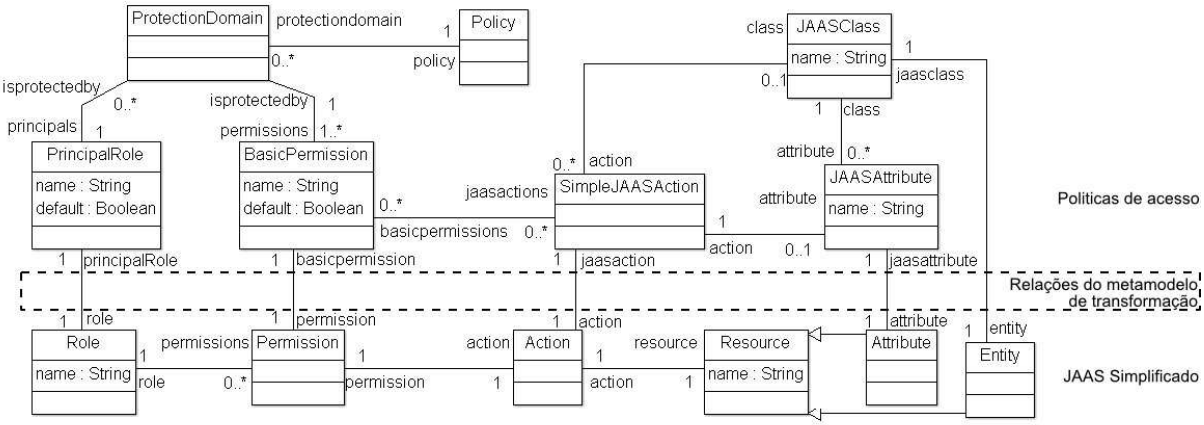

Figura 2.18: Metamodelo da transformação $\tau : Polticasdeacesso \rightarrow JAAS_{simplificado}$

Um contrato de transformação é executado para cada execução de uma transformação de modelos por ele descrita da mesma forma que é executada a verificação dos invariantes de uma metalinguagem. Portanto, transformações de modelos são validadas utilizando contratos de transformação como assertivas ao se utilizar um interpretador OCL para implementação de uma transformação de modelos. Esta abordagem é similar a abordagem de Desenho por Contratos (DbC, sigla em inglês) (MEYER, 1997). Em DbC, programas recebem anotações com assertivas que emitem exceções no caso de falhas. Em contratos de transformação, uma transformação de modelos é finalizada no caso de falha de uma assertiva OCL. Dessa maneira, contratos de transformação fornece os mesmos benefícios para um desenvolvedor de transformações que DbC provê para um programador.

Na Figura 2.20, podem ser identificados três tipos de usuários que se beneficiam do uso de contratos de transformação: designers de linguagens de modelagem, desenvolvedores

```
1 context Attribute
2 inv connectsToJAASAttribute:
3 self.name = self.jaasattribute.name and
4 self.action.jaasaction = self.jaasattribute.action
```

Figura 2.19: Invariante OCL que verifica a relação entre um atributo da entrada e da saída.

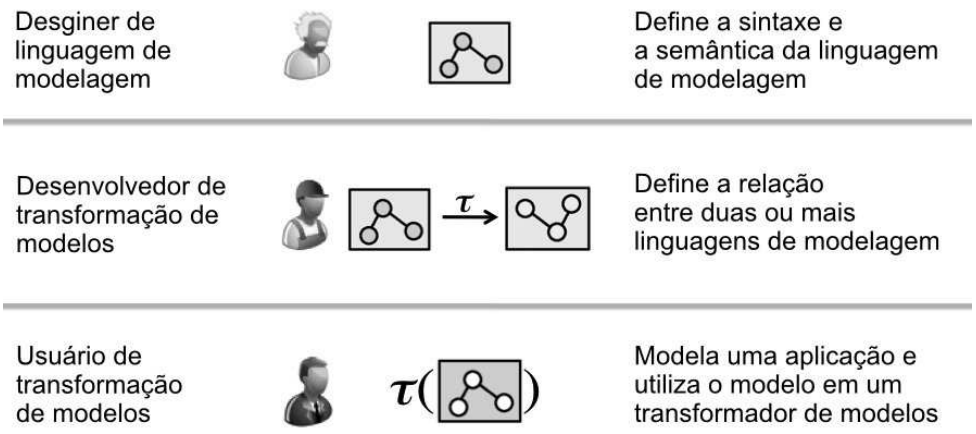

Figura 2.20: Tipos de usuários beneficiados pelo uso de contratos de transformação

de transformações de modelo e usuários de transformações. Estes usuários podem ser divididos em dois ciclos de validação: diretamente relacionados com o desenvolvimento e diretamente relacionado com a utilização do sistema em produção.

Durante o processo de desenvolvimento, os desenvolvedores podem identificar problemas nas fases de *parsing* e de transformação, graças aos invariantes definidos pelos designers referentes às linguagens de modelagem de entrada e saída de uma transformação. Caso ocorram erros na especificação do modelo ou em algum dos invariantes envolvidos na transformação, os desenvolvedores os identificam com a finalidade de informar aos designers sobre este fato. Este objetivo será alcançado desde que o código que implementa a transformação seja correto e tenha sido testado.

Enquanto o transformador estiver em produção, o contrato de transformação pode detectar erros em modelos de usuários e na transformação (no caso de falhas em invariantes de um modelo de transformação). Se um erro no transformador é detectado, ele pode ser informado aos desenvolvedores com a finalidade de o problema seja resolvido na próxima versão do transformador. Considerando estes dois casos, o contrato de transformação e seus invariantes podem aprimorar a qualidade do transformador e o mecanismo de detecção de erros tanto durante o desenvolvimento quanto na fase após a liberação de uma versão.

A partir das definições de contrato de transformação foi criado um padrão de projeto de desenvolvimento de software para transformações baseadas em contratos de transformação descrito na Figura 2.21 tendo como seus co-autores: Christiano Braga, Roberto Menezes, Edson Landim e Cássio Santos. Este padrão foi utilizado na implementação da transformação descrita neste trabalho na forma da ferramenta apresentada no Capítulo 8.

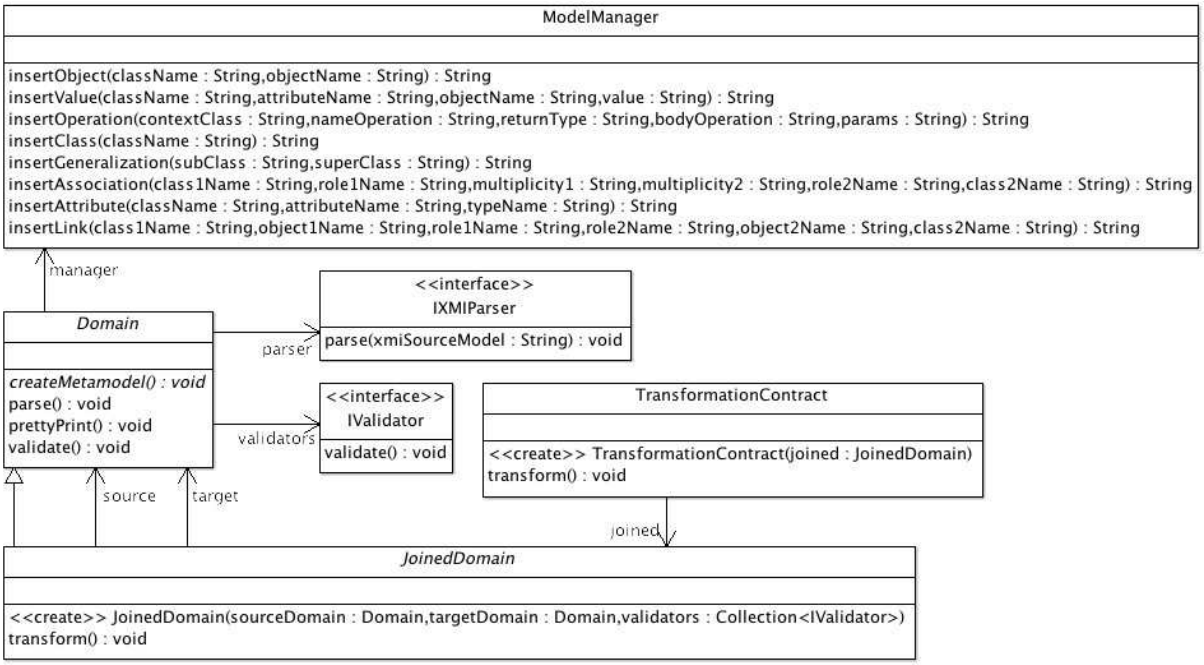

Figura 2.21: Padrão de projeto para contratos de transformação

Com relação às classes que descrevem o padrão de projeto, a classe *Domain* representa o conhecimento sobre um dado domínio, especificando o seu metamodelo, que define a parte estrutural da sintaxe; um ou mais validadores que garantem que uma dada instância carregada no domínio está em conformidade com a linguagem que representa o domínio; um método de *parse*, que utiliza um elemento do tipo *IXMIParser* para ler um arquivo contendo uma instância da sintaxe concreta e a partir dele gerar uma instância em conformidade com a sintaxe abstrata; e o *pretty print*, que é a geração de sintaxe concreta a partir de uma instância da sintaxe abstrata.

A classe *JoinedDomain* representa o domínio formado pela união entre domínios específicos de entrada e saída, contendo conexões entre eles e os passos em conformidade com o contrato de transformação para gerar uma instância do domínio de saída a partir dos elementos do domínio de entrada. Para efetuar as validações, nos domínios foi definida a interface *IValidator* que contém um método para efetuar as validações sobre uma instância do domínio.

A classe *ModelManager* é utilizada pelos domínios para criar em memória os objetos que representam os metamodelos e as instâncias dos metamodelos. No corrente trabalho todas as validações ocorrem através de invariantes OCL aplicadas através de um interpretador OCL, que interagirá com a classe de administração de modelos.

A classe *TransformationContract* define os passos da transformação no método *trans-*

form(), baseados no conceito de contrato de transformação. O diagrama apresentado na Figura 2.22 representa a execução deste método a partir da troca de mensagens com o elemento que representa o domínio combinado de um contrato de transformação.

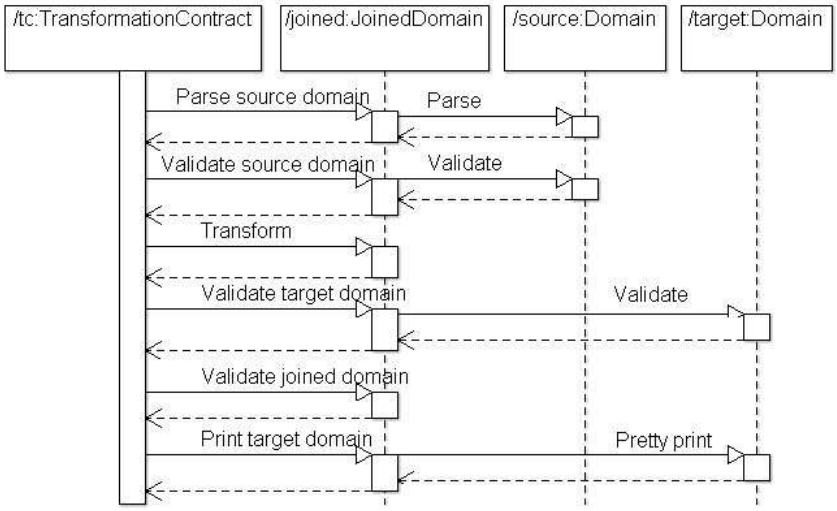

Figura 2.22: Diagrama de mensagens do processo de transformação utilizando contrato de transformação

O primeiro passo descrito no diagrama efetua o processo de *parsing* que gera uma instância do metamodelo de entrada no domínio de entrada. O elemento do tipo *TransformationContract* efetua uma chamada ao método de *parse* do domínio combinado, que designa esta atividade para o seu domínio de entrada. Após a criação desta instância, a sua validação também é feita através do próprio domínio de entrada utilizando validadores específicos que verificam os invariantes OCL do metamodelo de entrada.

O próximo passo representa a transformação, executada a partir do domínio combinado. Neste passo os elementos da instância do metamodelo de entrada são transformados em elementos da instância do metamodelo de saída, onde os elementos das duas instâncias são relacionados através da instância do metamodelo de transformação. Após o processo de transformação, duas validações são efetuadas. Primeiramente, a instância do metamodelo de saída é validada no domínio de saída, a partir da verificação dos invariantes OCL do metamodelo deste domínio. Logo após, a instância do metamodelo de transformação é validada no domínio combinado, para garantir que o passo de transformação está correto conforme o especificado no contrato de transformação.

Após todas as validações, o elemento do tipo *TransformationContract* dispara o processo de *pretty printing* para o domínio combinado, o qual designa o domínio de saída para efetuar o *pretty printing* da instância do metamodelo de saída finalizando o processo.

No Capítulo 3 é apresentado o conceito de Segurança Dirigida a Modelos e a linguagem de modelagem SecureUML. A SecureUML é utilizada para especificar controles de acesso baseados em papéis e representa a linguagem de entrada da transformação descrita neste trabalho.

Capítulo 3

Segurança dirigida por modelos

Requisitos de segurança são normalmente neglicenciados na fase de *design* do desenvolvimento devido a falta de uma metodologia formal. A motivação do conceito de Segurança Dirigida a Modelos (MDS, sigla em inglês) proposto por (BASIN *et al.*, 2006) é a geração de sistemas seguros a partir de modelos relacionados aos requisitos de segurança levantados na fase de análise do ciclo de desenvolvimento de software.

Segurança dirigida por modelos é uma especialização de DDM, pois foca na geração de infraestrutura de segurança através de uma linguagem de modelagem específica para controle de acesso ao invés de utilizar linguagens de modelagem de propósito geral. MDS utiliza a linguagem SecureUML, uma linguagem específica de domínio para a descrição de políticas de controle de acesso. A linguagem SecureUML possui um metamodelo que descreve sua sintaxe abstrata e sua sintaxe concreta é descrito por um perfil UML. Com esta linguagem, o analista de segurança especifica papéis de usuário e restrições à estes papéis para o acesso a recursos computacionais, seguindo a abordagem de controle de acesso por papéis (RBAC, sigla em inglês) (FERRAIOLO; KUHN, 1992). Em MDS, o analista de segurança se preocupa em especificar e analisar políticas de controle de acesso utilizando SecureUML. O código que implementa uma dada política de segurança é gerado automaticamente por um transformador que sabe como produzir código numa plataforma específica para uma dada política de segurança de controle de acesso.

É importante destacar que SecureUML não define quais são os recursos e quais os tipos de ações podem ser executadas sobre os recursos. Dessa maneira, a linguagem deve ser estendida no nível de seu metamodelo (sintaxe abstrata) e com relação a seu perfil UML (sintaxe concreta). A extensão mais utilizada em SecureUML é chamada ComponentUML na qual recursos são estendidos para representar elementos em forma de classes de um

sistema orientado a objetos se dividindo em: entidades (classes controladas), atributos, métodos e pontas de associação. A união de SecureUML e ComponentUML recebe o nome SecureUML+ComponentUML.

3.1 Perfil UML de SecureUML+ComponentUML

SecureUML+ComponentUML possui um perfil UML que anota elementos do diagrama de classes para representar elementos de RBAC. Por exemplo, existe um estereótipo (anotação) sobre classes para representar papéis assim como estereótipos de classe para representar recursos cujo acesso é controlado. A Tabela 3.1 apresenta os estereótipos do perfil UML para SecureUML+ComponentUML.

Estereótipo	Aplicado a	Descrição
<<*Entity*>>	Classe	Indica que esta classe terá o acesso controlado a todos seus elementos e à sua criação e deleção.
<<*Permission*>>	Classe de associação	Indica uma permissão de acesso a um recurso.
<<*Role*>>	Classe	Indica um papel de usuário no sistema controlado.
<<*EntityAction*>>	Atributo	Indica uma ação de criação ou deleção de uma entidade.
<<*EntityAttributeAction*>>	Atributo	Indica uma ação de leitura ou escrita de um atributo de uma entidade.
<<*EntityAssociationEndAction*>>	Atributo	Indica uma ação de leitura ou escrita de uma ponta de associação de uma entidade.
<<*EntityMethodAction*>>	Atributo	Indica uma ação de execução de um método de uma entidade.

Tabela 3.1: Estereótipos do perfil UML de SecureUML+ComponentUML

Para ilustrar o uso do perfil UML de SecureUML+ComponentUML será apresentada um exemplo de política de acesso. O exemplo representa um sistema gerenciador de arquivos que tem a função de criar e executar arquivos. Neste sistema foram levantados os requisitos de segurança e foram identificados dois papéis de usuários. Usuários sob o papel *RoleA* podem criar arquivos e os usuários sob o papel *RoleB* podem efetuar todos os tipos de função do gerenciador de arquivos.

A Figura 3.1 apresenta a modelagem em SecureUML+ComponentUML da política de segurança do gerenciador de arquivos. A entidade *FileSystem* representa o sistema de arquivos e será uma entidade controlada. As entidades são representadas pelo este-

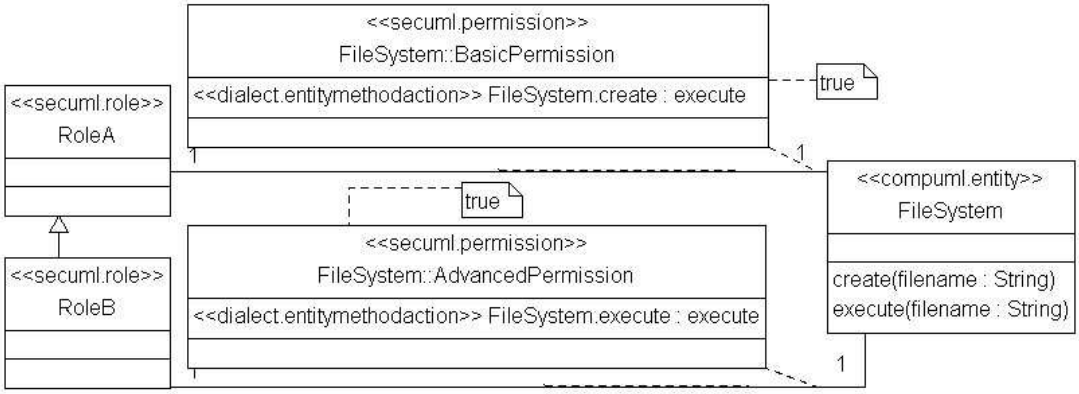

Figura 3.1: Política de segurança para um sistema gerenciador de arquivos

reótipo <<*Entity*>> e seus atributos, métodos e pontas de associação serão identificados como elementos a serem controlados. Os papéis são representados pelo estereótipo <<*Role*>> e as permissões são representadas por classes de associação com o estereótipo <<*Permission*>> relacionadas a um papel e a um recurso. As ações são representadas por atributos das permissões e possuem estereótipos específicos de acordo com a extensão de SecureUML utilizada. Os estereótipos de ações em ComponentUML são: *EntityAction* representando uma ação que ocorre sobre uma entidade; *EntityAttributeAction* para ações que ocorrem sobre atributos; *EntityAssociationEndAction* para ações que ocorrem sobre pontas de associação; e *EntityMethodAction* para ações que ocorrem sobre métodos. A hierarquia de papéis é representada pela relação de generalização do diagrama de classes. Dessa maneira a subclasse de papel (*subrole*) herdará todas as permissões de sua superclasse de papel (*superrole*). Os comentários associados às permissões representam restrições de autorização (*AuthorizationConstraint*) que são descritos em OCL.

3.2 Metamodelo de SecureUML+ComponentUML

O metamodelo de SecureUML é apresentado na Figura 3.2. O tipo *Role* representa papéis de usuários, o tipo *Permission* representa permissões, o tipo *User* representa usuários, o tipo *Resource* representa recursos controlados e o tipo *Action* representa ações de acesso aos recursos controlados. As permissões indicam autorização de acesso sobre ações (*Action*) a serem executadas sobre os recursos (*Resources*). As ações podem ser atômicas (*AtomicAction*) ou compostas por outra ações (*CompositeActions*). Os papéis podem ser organizados de maneira hierárquica, assim como o apresentado no RBAC hierárquico (FERRAIOLO *et al.*, 2003). Além disso, restrições sobre aspectos dinâmicos do sistema podem ser criadas a partir do tipo de restrições de autorização (*Authorization Constraints*).

Por exemplo, “um determinado recurso só pode ser editado se o usuário corrente o criou”.

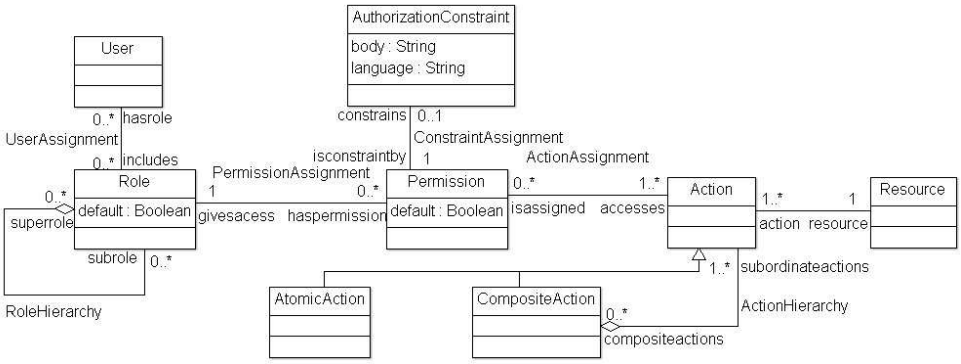

Figura 3.2: Metamodelo da linguagem SecureUML

Na Figura 3.3 são apresentadas as conexões da extensão ComponentUML no metamodelo de SecureUML. Na extensão SecureUML+ComponentUML os recursos são estendidos para receber Entidades (*Entity*), Atributos (*Attributes*), Métodos (*Method*) e Pontas de Associação (*AssociationEnd*). As ações atômicas e compostas devem também ser estendidas para acesso aos recursos da linguagem. Em ComponentUML um atributo pode, por exemplo, ser acessado pelas ações atômicas de leitura (*AtomicRead*) e escrita (*AtomicUpdate*) e pela ação composta de acesso total sobre o atributo (*AttributeFullAccess*). A Tabela 3.2 apresenta os recursos e as ações relacionadas a seu acesso na linguagem ComponentUML, onde as ações sublinhadas são ações composta.

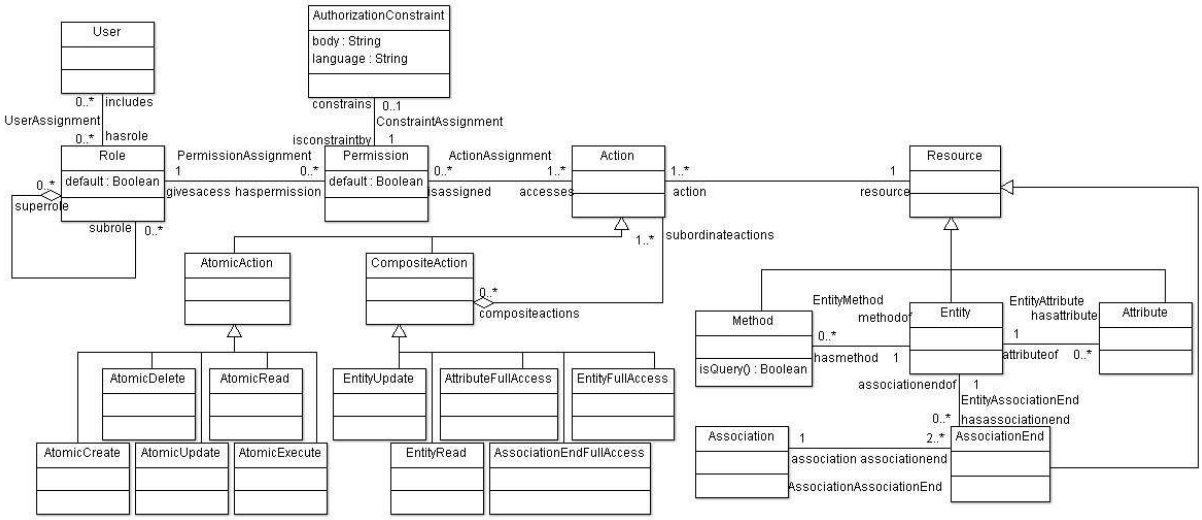

Figura 3.3: Extensão do metamodelo de SecureUML com ComponentUML

Conforme descrito pelo processo DDM adotado neste trabalho, o primeiro passo que uma transformação de modelos deve executar para transformar um modelo m escrito utilizando um perfil UML que representa uma linguagem de modelagem M é mapear m em

Recurso	Ação
Entity	create, read, update, delete, full access
Attribute	read, update, full access
Method	execute
AssociationEnd	read, update, full access

Tabela 3.2: Relação entre recursos e ações de acesso em ComponentUML (BASIN *et al.*, 2009)

uma instância do metamodelo de M. Portanto, para executar a transformação de modelos SecureUML para outros modelos é necessário transformar $m \in Ms_{Perfil\ UML\ SecureUML}$ em $m' \in Ms_{Metamodelo\ SecureUML}$.

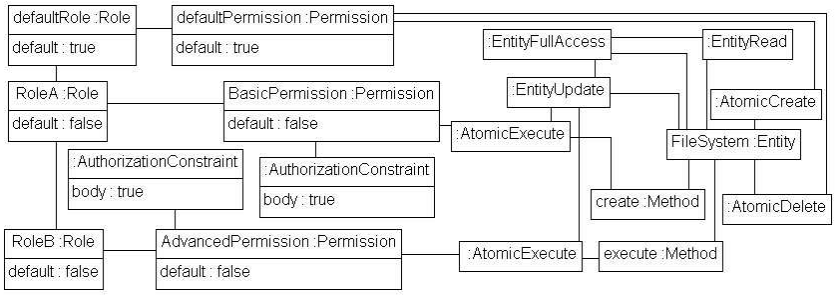

Figura 3.4: Política de segurança de arquivos como instância do metamodelo SecureUML

A Figura 3.4 apresenta a instância m' do modelo m da política de segurança do acesso ao gerenciador de arquivos da Figura 3.1 gerado a partir dos passos descritos no Apêndice C de (BASIN *et al.*, 2009). O primeiro passo da criação do modelo abstrato consiste na criação de um papel especial chamado *DefaultRole*. Este papel é definido de modo que todo usuário no sistema esteja relacionado a ele e que todo papel herde as suas permissões. Isso garante que um usuário no sistema estará sempre relacionado a algum papel, mesmo que este não tenha permissão de acesso a nenhuma ação. Também é criada uma permissão especial chamada *DefaultPermission* que está relacionada somente ao papel *DefaultRole*, e portanto, todos os papéis herdam esta permissão.

Em um modelo SecureUML, se um recurso não está relacionado explicitamente a permissões de acesso, indica que este recurso pode ser acessado por qualquer usuário. Define-se este acesso pela relação da permissão *DefaultPermission* com as ações de acesso ao recurso. A medida que novas permissões são relacionadas às ações de acesso de um recurso, as ações perdem a relação com a permissão *DefaultPermission* e somente usuários sob papéis relacionados às novas permissões terão acesso ao recurso.

Para cada recurso de m é gerado uma instância deste recurso em m' juntamente com

todas as ações relacionadas a seu acesso. As ações atômicas após criadas são conectadas a permissão *DefaultPermission*, garantindo que certas ações possam ser executadas por qualquer usuário, caso estas não estejam relacionadas a nenhuma permissão específica. Para cada entidade e em m é gerada uma instância $\overline{e}$ do tipo *Entity* em m' e as ações correspondentes ao seu acesso: $efa(\overline{e})$ do tipo *EntityFullAccess* que representa o controle total sobre a entidade; $er(\overline{e})$ do tipo *EntityRead* que representa o acesso de leitura aos atributos e pontas de associação da entidade; $eu(\overline{e})$ do tipo *EntityUpdate* que representa o acesso de escrita aos atributos e pontas de associação; $ac(\overline{e})$ do tipo *AtomicCreate* que representa o acesso a criação de uma entidade; e $ad(\overline{e})$ do tipo *AtomicDelete* que representa a ação de remoção da entidade. Inicialmente as ações atômicas $ac(\overline{e})$ e $ad(\overline{e})$ são conectadas a permissão *DefaultPermission*. Para cada método me de e em m será gerada uma instância $\overline{me}$ de *Method* que estará relacionada a $\overline{e}$ e ações $ae(\overline{me})$ do tipo *AtomicExecute*. Os mesmos passos serão executados para os outros recursos com a geração das suas ações específicas.

Nota-se em m' que algumas ações não estão relacionadas a permissão especial *DefaultPermission*. Isto é devido ao mapeamento das permissões de m para m', onde somente os papéis e subpapéis relacionados a uma permissão terão acesso às ações a ela associadas. No mapeamento, para cada permissão p em m é gerada uma permissão $\overline{p}$ em m' e para cada relação entre uma permissão p e um papel r em m é gerada uma relação entre $\overline{r}$ e $\overline{p}$ em m'. Para cada ação a relacionada a uma permissão p e um recurso res em m será gerada uma relação entre $\overline{a}$ e $\overline{p}$ e serão removidas as relações entre $\overline{a}$ e a permissão especial *DefaultPermission*, caso $\overline{a}$ seja uma ação atômica. Por exemplo, a ação do tipo *AtomicExecute* relacionada a instância *create* de *Method* em m' não está relacionada a permissão *DefaultPermission*, pois em m a permissão *BasicPermission* está relacionada a ação de execução do método *create* da entidade *FileSystem*.

Os passos do algoritmo foram criados de maneira a garantir que o modelo m' gerado é *bem formado* com relação a sintaxe abstrata de SecureUML. Entretanto, o algoritmo não garante que certas propriedades de tipos da linguagem são atendidas, ou seja, que o modelo está em *conformidade* com a linguagem SecureUML. Para garantir tal conformidade, a verificação de invariantes sobre m' deve ser realizada, onde

$$m' \in Ms_{Metamodelo\ \ SecureUML}$$

$$m' \models I_{Ms}$$

Esta notação indica que um modelo m' está em conformidade com a linguagem SecureUML, pois, além de bem formado, m' atende aos invariantes I_{Ms} da linguagem que o

descreve.

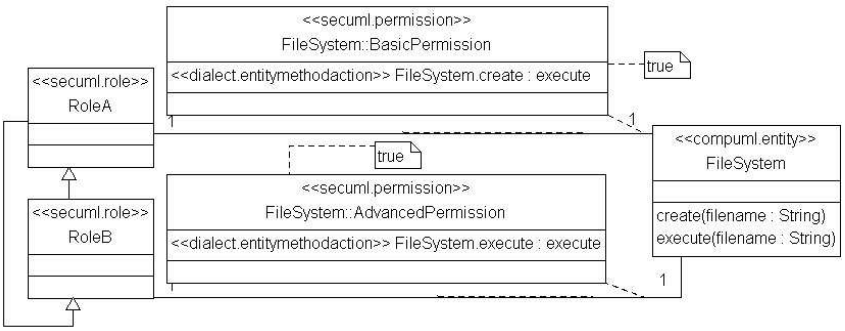

Figura 3.5: Política de segurança de arquivos não conforme com o metamodelo SecureUML+ComponentUML

A Figura 3.5 apresenta um modelo bem formado com relação a Ms, mas que não está em conformidade com o metamodelo SecureUML+ComponentUML devido ao invariante *noCyclesinRoleHierarchy* apresentado na Figura 3.6. Este invariante verifica a presença de ciclos em hierarquias de roles através da verificação para cada role r, se r não está incluído no fecho transitivo da relação *superRole*, no metamodelo SecureUML, que representa todos os papéis de sua hierarquia.

O invariante *noCyclesinRoleHierarchy* utiliza duas operações OCL para calcular o fecho transitivo de papéis: *superrolePlus* e *superrolePlusOnSet*. A operação *superrolePlusOnSet* executa tal cálculo através de uma chamada recursiva em cada elemento da coleção resultante da relação *superRole* para cada classe r. Quanto a sintaxe OCL, o significado informal dos construtores OCL restantes no exemplo são o seguinte: *forAll* itera sobre os elementos de uma dada coleção verificando um predicado; *excludes* verifica se uma dada coleção não contém um determinado elemento; *collects* cria uma coleção de objetos que satisfaçam um determinado predicado; *asSet* gera um conjunto a partir de uma coleção; e *including* inclui um elemento em uma dada coleção.

Os demais invariantes de SecureUML+ComponentUML são listados no Apêndice A e estão relacionados principalmente a verificação de relações entre ações e recursos, relações entre ações e a permissão especial *DefaultPermission* e a relação entre as ações compostas e atômicas. Erros encontrados por estas verificações podem ser gerados pela inconsistência da implementação do algoritmo de mapeamento entre as sintaxes concreta e abstrata.

Uma vez que o engenheiro de segurança está convencido de que seu modelo de controle de acesso de fato representa o que ele deseja, depois de validá-lo com as verificações OCL e eventualmente fazer consultas sobre o modelo particular que está projetando, ele pode então gerar código automaticamente invocando um transformador de modelos. No

```
1 context Role inv noCyclesinRoleHierarchy:
2   self.superrole->forAll(r|r.superrolePlus()->excludes(self))
3
4 context Role::superrolePlus():Set(Role) body:
5   self.superrolePlusOnSet(self.superrole)
6
7 context Role::superrolePlusOnSet(rs:Set(Role)):Set(Role) body:
8   if rs.superrole->exists(r|rs->excludes(r))
9   then self.superrolePlusOnSet(rs
10    ->union(rs.superrole)->asSet())
11  else rs->including(self)
12  endif
```

Figura 3.6: Invariante OCL noCyclesinRoleHierarchy

Capítulo 6 será apresentado um modelo de saída para uma transformação de SecureUML+ComponentUML e a especificação da transformação em questão.

Capítulo 4

Aspectos para controle de acesso

Em um sistema que utiliza controle de acesso, a autorização pode facilmente ser identificada como um comportamento que ocorre antes do acesso ao recurso controlado e, portanto, ser mapeada para uma pré-condição. Em (BRAGA, 2010), o autor apresenta a linguagem Aspectos para Controle de Acesso (AAC, sigla em inglês) que define uma infraestrutura baseada em aspectos para controlar o acesso a recursos. Neste trabalho foi executada uma análise sobre a linguagem AAC levando a uma reestruturação de sua sintaxe abstrata e seus invariantes. Neste capítulo serão apresentados os conceitos básicos de Programação Orientada a Aspectos, o metamodelo da linguagem AAC, sua sintaxe concreta representada por código AspectJ e os invariantes relacionados a esta linguagem.

4.1 Programação Orientada a Aspectos

Programação Orientada a Aspectos (AOP, sigla em inglês)(KICZALES *et al.*, 2001) é uma técnica para melhorar a separação de interesses (*concerns*) em software, onde os aspectos proveem mecanismos para capturar interesses que não possam ser decompostos do resto do sistema, no entanto, em muitos casos estes interesses são secundários com relação às funcionalidades do sistema. Por exemplo, o controle de acesso a um recurso não representa a parte funcional de um sistema, entretanto, as verificações de autorização não podem ser separadas das ações de acesso a estes recursos.

Implementações em AOP são normalmente relacionadas a linguagens de Orientação a Objetos, como é o caso de AspectJ, uma extensão de orientação a aspectos para a linguagem orientada a objetos Java. Os programas definidos nesta extensão utilizam classes para a parte funcional do sistema e a elas são adicionados aspectos para capturar interesses que ortogonalizam a estrutura da classe. Nesta seção são apresentados os conceitos

de AOP através de exemplos em AspectJ.

Um conceito chave desta abordagem são os chamados pontos de junção (*join points*), considerados pontos bem definidos na execução de um programa e podem ser representados como nós em tempo de execução de um grafo de chamadas a um objeto. Estes nós incluem pontos nos quais um objeto recebe uma chamada a um de seus métodos e pontos nos quais um campo de um objeto é referenciado. As arestas são relações de controle sobre o fluxo de chamadas entre os nós.

O conjunto dos pontos de junção são chamados de *pointcuts*, no qual as formas de correspondência aos pontos de junção em tempo de execução são definidas pelos designadores de *pointcut* (*pointcut designators*). A Figura 4.1 apresenta um designador de *pointcut* para o método *create()* da classe *FileSystem* do exemplo do gerenciador de arquivos da Figura 3.1. A palavra-chave *call* indica a declaração de uma interceptação a chamada de um método. A assinatura definida indica que a chamada será referente ao método *create* da classe *FileSystem* com um parâmetro do tipo *String* .

```
1 call(void FileSystem.create(String))
```

Figura 4.1: Código AspectJ de um designador de *pointcut*

Na Figura 4.2 é apresentado o código do *pointcut PCcreate*, definido pela palavra-chave *pointcut*, contendo o ponto de junção já apresentado. Neste *pointcut* é definida a instância do tipo *FileSystem* que será interceptada através do atributo *self* e o parâmetro passado para a chamada do método através do atributo *filename*.

```
1 pointcut PCcreate(FileSystem self,String filename):
2   call(void FileSystem.create(String)) && target(self) && args(filename);
```

Figura 4.2: Código AspectJ de um *pointcut*

Relacionados aos *pointcuts* são definidos os adendos (*advices*), que são construções parecidas com métodos utilizadas para adicionar comportamentos adicionais aos *pointcuts*. Os adendos podem representar execuções que ocorrem em diferentes momentos para cada ponto de junção do *pointcut*. Alguns exemplos de adendos são os que ocorrem antes, durante ou depois do ponto definido pelo *pointcut*.

O trecho de código da Figura 4.3 apresenta a definição de um adendo que ocorre antes (*before advice*) da execução dos pontos de junção definidos pelo *pointcut PCcreate*. A palavra-chave *before* declara o *before advice* e na sua assinatura são informados os atributos dos elementos utilizados pelo *pointcut* e o nome do *pointcut* ao qual está referenciado.

Por fim, é definido o bloco de comandos que será executado em todos os pontos de junção do *pointcut* indicado.

```
1 before (FileSystem self,String filename):PCcreate(self,filename){
2  //passos a serem executados antes do método create
3 }
```

Figura 4.3: Código AspectJ de um adendo

A união dos pontos de junção, *pointcuts* e adendos constituem um aspecto, que são unidades modulares de implementação das ortogonalizações de um sistema. Em AspectJ, os aspectos são definidos por declarações similares a definições de classes e contém definições de *pointcuts* e dos adendos relacionados aos *pointcuts*. Na Figura 4.4 é apresentado o código de um aspecto chamado *FileSystem_ ACAspect*, contendo o *pointcut PCcreate* e um *before advice* relacionado a este *pointcut*.

```
1 public aspect FileSystem_ACAspect {
2  pointcut PCcreate(FileSystem self,String filename):
3    call(void FileSystem.create(String)) && target(self) && args(filename)
       ;
4  before (FileSystem self,String filename):PCcreate(self,filename){
5    //passos a serem executados antes do método create
6  }
7 }
```

Figura 4.4: Código AspectJ de um aspecto

4.2 Linguagem AAC

Com intuito de formalizar o controle de acesso através da utilização de aspectos, foi criada a linguagem de modelagem AAC, na qual são definidos tipos que representam parte dos conceitos de orientação a objetos relacionados a outros tipos para controle de acesso a recursos de um sistema orientado a objetos.

4.2.1 Metamodelo de AAC

A sintaxe abstrata de AAC é definida a partir do metamodelo apresentado na Figura 4.5, onde aspectos são representados pelo tipo *Aspect*, *pointcuts* são representados pelo tipo *Pointcut* e os *before advices* são representados pelo tipo *BeforeAdvice*. Nesta linguagem foi definido somente o adendo do tipo *BeforeAdvice*, devido ao fato de este ser suficiente para verificar as autorizações de acesso a chamadas de métodos.

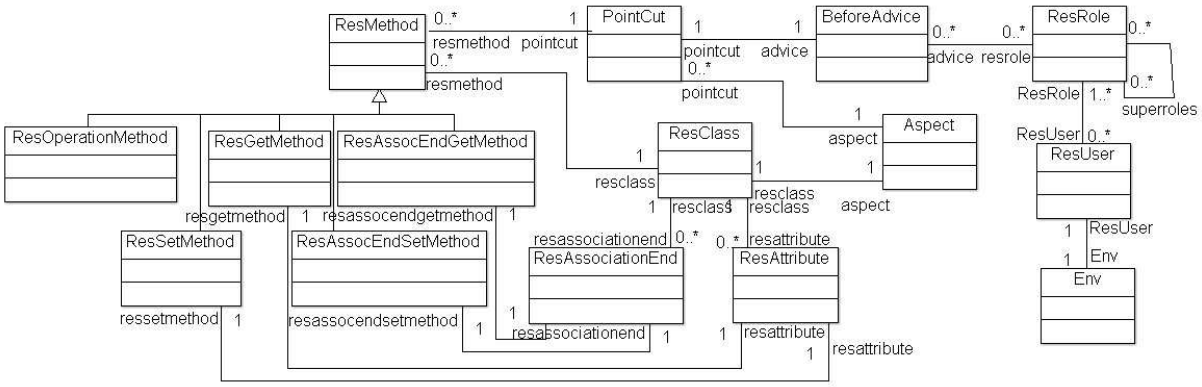

Figura 4.5: Metamodelo da linguagem AAC

Conforme já descrito, os aspectos são utilizados juntamente com linguagens orientadas a objetos e atuam sobre classes destas linguagens. Em AAC, os aspectos são aplicados sobre tipos que representam classes abstratas definindo *pointcuts* sobre métodos por ela definidos. As classes abstratas são representadas pelo tipo *ResClass*. Os atributos das classes abstratas são representados pelo tipo *ResAttribute* e as pontas de associação pelo tipo *ResAssociationEnd*. O acesso dos *pointcuts* ocorre em AAC sobre métodos e toda forma de acesso será definido por tipos que representam métodos. Estes se dividem em: (i) métodos para acesso a atributos definidos pelo tipo *ResSetMethod* para escrita de atributos e *ResGetMethod* para leitura de atributos; (ii) métodos para acesso às pontas de associação definidos pelo tipo *ResAssocEndSetMethod* para escrita nas pontas de associação e pelo tipo *ResAssocEndGetMethod* para leitura das pontas de associação; (iii) operações.

A representação dos usuários e dos papéis dos usuários se faz necessária para o controle de acesso e por isso também possuem tipos definidos em AAC. O usuário que é autenticado no sistema gera uma instância do tipo *ResUser*, onde seus papéis são representados pelo tipo *ResRole*. O tipo *Env* representa a classe de ambiente que deve ser estendida para a instanciação do usuário de acordo com o sistema de autenticação que for utilizado.

O metamodelo de AAC apresentado neste trabalho possui modificações em comparação com o metamodelo apresentado em (BRAGA, 2010). As modificações ocorreram após a análise estrutural da linguagem tendo como resultado a adição dos seguintes tipos ao metamodelo da linguagem: usuários do sistema (*ResUser*), pontas de associação (*ResAssociationEnd*) e os métodos de acesso às pontas de associação (*ResAssociationEndGetMethod* e *ResAssociationEndSetMethod*). O tipo *ResAssociationEnd* e os tipos a ele relacionados tornam a linguagem mais completa com a possibilidade de verificar as pontas de associação de um recurso controlado, enquanto que o tipo *ResUser* evidencia um tipo utilizado na sintaxe concreta, mas que não era representado na sintaxe abstrata.

Além disso, foram adicionadas novas relações entre os tipos. A relação entre o tipo que representa um atributo controlado (*ResAttribute*) e os métodos que representam seu acesso (*ResGetMethod* e *ResSetMethod*) foi adicionada garantindo que o atributo sempre estará relacionado a um único método de acesso de cada tipo. Este mesmo conceito foi aplicado novamente criando uma relação entre os tipos de pontas de associação. Com o intuito de evidenciar que a permissão de acesso de um papel de usuário ocorrerá através de verificações presentes em instâncias do tipo *BeforeAdvice*, uma nova relação foi adicionada entre *BeforeAdvice* e *ResRole*. A adição da metaclasse que representa um usuário gerou as seguintes relações: com o tipo *ResRole*, representando os papéis ao qual um usuário está associado, e com o tipo *Env*, o qual recebe o usuário atualmente autenticado no sistema.

4.2.2 Uma instância do metamodelo de AAC

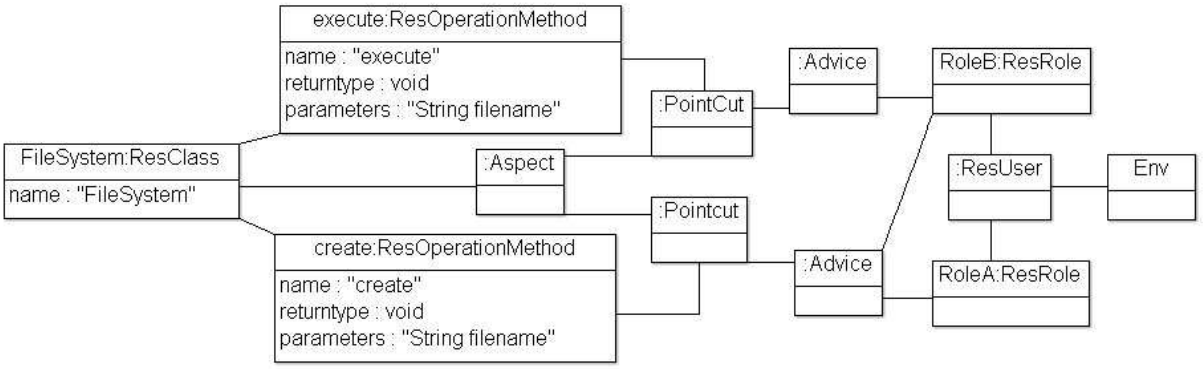

Figura 4.6: Política de segurança de arquivos como instância do metamodelo AAC

A Figura 4.6 apresenta a política de segurança de acesso ao gerenciador de arquivos como uma instância do metamodelo da AAC. A sintaxe concreta desta instância é apresentada nas Figuras 4.7, 4.8, 4.9, 4.10 e 4.11 como resultado do processo de *pretty printing* que gera código-fonte.

Para modularizar o acesso ao exemplo do gerenciador de arquivos apresentado como um modelo de política de segurança SecureUML, a classe abstrata que o define representa o recurso a ser controlado. Se ocorrer modificações na lógica interna aos métodos do sistema, estas não necessitarão de mudanças na definição da classe abstrata. A Figura 4.7 apresenta a classe abstrata para o gerenciador de arquivos. Caso o acesso não seja autorizado, exceções serão emitidas o que é evidenciado através da declaração *throws Exception* em cada método.

Para representar papéis, será criada a classe *Role*, onde cada papel será representado por uma classe que herda desta classe. A Figura 4.8 apresenta a classe *Role*, que contém

```
1 abstract class FileSystem {
2   abstract void create(String filename) throws Exception;
3   abstract void execute(String filename) throws Exception;
4 }
```

Figura 4.7: Classe abstrata do recurso de Sistema de Arquivos

uma lista de papéis da sua hierarquia definida pela lista *superRoles*. O método *verifyHierarchy* verifica se um determinado *role* faz parte da hierarquia de papéis do objeto *Role* corrente. Este método é necessário devido a ocorrência de hierarquia múltiplas de papéis ser possível em AAC.

```
1 public abstract class Role {
2   protected List<Class<? extends Role>> superRoles;
3   public abstract void fillSuperRoles();
4   public boolean verifyHierarchy(Class<? extends Role> role) {
5     if (superRoles.contains(role)) return true;
6     return false;
7   }
8 }
```

Figura 4.8: Classe representando um papel

A Figura 4.9 apresenta a classe que representa o papel *RoleA*. Esta classe herda a classe *Role* e adiciona os tipos de classe que estão na sua hierarquia de papéis na lista *superRoles*, que neste caso é representado pela classe *RoleA* e *DefaultRole*.

```
1 public class RoleA extends Role {
2   @Override
3   public void fillSuperRoles() {
4     this.superRoles.add(DefaultRole.class);
5     this.superRoles.add(RoleA.class);
6   }
7 }
```

Figura 4.9: Classe do papel *RoleA* do controle ao sistema de arquivos

Para efetuar as verificação de autorização, é preciso saber qual o papel do usuário que está autenticado no sistema. Para tal tarefa, classe do tipo *User* contém um atributo do tipo *Role* que representa o papel do usuário autenticado no sistema. O usuário atual do sistema é registrado na classe de ambiente *Env* apresentada na Figura 4.10, que possui os métodos necessários para autenticação de um usuário no sistema. Estes métodos devem ser implementados pelos desenvolvedores, já que a autenticação de usuários não é contemplada em AAC.

Para controle do acesso aos métodos da classe *FileSystem*, é criado um aspecto, com

```
1 public class Env {
2   private static User userRole;
3   public Env(User user) {
4     userRole = user;
5   }
6   public static User getUsrRole() {
7     return userRole;
8   }
9   //métodos de autenticação e geração do objeto userRole
10 }
```

Figura 4.10: Classe de ambiente que controla o usuário autenticado no sistema

pointcuts para interceptar as chamadas aos métodos *create* e *execute*. Seguindo o modelo de política de segurança SecureUML para o gerenciador de arquivos, os usuários sob o papel *RoleA* podem acessar o método *create()*. Em AAC este acesso ocorre através de uma verificação no *before advice* relacionado ao *pointcut* que define um ponto de junção para este método.

Na Figura 4.11 é apresentado o *pointcut PCCreate* que intercepta chamadas ao método *create* e o *before advice* para este *pointcut*. No *before advice* é verificado se o usuário autenticado no sistema (representado pelo elemento *caller*) possui em sua hierarquia a classe *RoleA*. Por exemplo, um usuário sob o papel *RoleB* poderia efetuar tal acesso. Caso o usuário não esteja sob um papel que tenha *RoleA* em sua hierarquia, então uma exceção é gerada e a execução do método não ocorrerá.

```
1 public aspect FileSystem_ACAspect {
2   pointcut PCcreate( FileSystem self , String filename):
3     call(void FileSystem.create(String)) && target(self) && args(filename)
        ;
4   before (FileSystem self,String filename) throws Exception:PCcreate(self
      ,filename){
5     User caller=Env.getUsrRole();
6     if(caller.verifyRoleHierarchy(RoleA.class)) return;
7     throw new Exception("O usuário não possui permissão para este acesso")
        ;
8   }
9 }
```

Figura 4.11: Parte do código do Aspecto para controle de acesso sobre a classe FileSystem

Este exemplo de controle de acesso representa a sintaxe concreta da linguagem AAC, mas não apresenta todos os tipos de AAC, como é o caso de acesso a atributos e pontas de associação. Devido ao requisito do acesso ocorrer através de métodos pelos *pointcuts*, para cada atributo e ponta de associação devem ser gerados métodos de leitura e escrita a esses elementos nas classes abstratas.

4.2.3 Invariantes de AAC

A estrutura do metamodelo de AAC possui algumas limitações que deverão ser verificadas através de invariantes. De forma análoga ao apresentado em SecureUML deve ser verificada a hierarquia de papéis, que em AAC é representado por elementos do tipo *ResRoles*. Outra verificação ocorre sobre a consistência do ciclo das relações entre os elementos do tipo *Aspect*, *Pointcut*, *ResMethod* e *ResClass*. A Figura 4.12 apresenta um erro nesta relação, pois o elemento do tipo *Pointcut* relacionado a instância *Method* de *ResOperationMethod* deveria estar relacionado ao elemento do tipo *Aspect* da classe que contém *Method*.

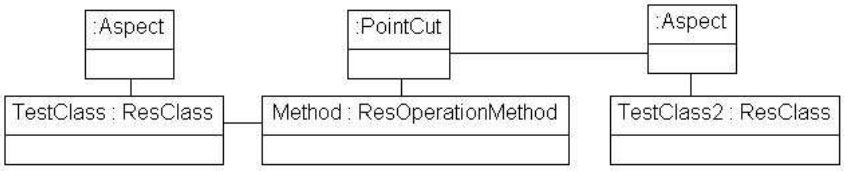

Figura 4.12: Instância de AAC com erro na relação entre aspectos e métodos

Na Figura 4.13 é apresentado o invariante *targetsResClassAspectPointcut* que atua sobre elementos do tipo *ResMethod*. Neste invariante é verificado se o elemento do tipo *ResClass* relacionado a um *ResMethod* é o mesmo elemento que é alcançado através da relação com *Pointcut* e *Aspect* por este *ResMethod*.

```
1 context ResMethod inv targetsResClassAspectPointcut:
2   self.resclass = self.pointcut.aspect.resclass
```

Figura 4.13: Código OCL da relação entre *Aspect*, *PointCut*, *ResMethod* e *ResClass*

Métodos de acesso aos tipos que representam atributos (*ResAttributes*) e pontas de associação (*ResAssociationEnd*) também podem ser mal-formados. A Figura 4.14 ilustra este erro , pois os métodos de acesso ao elemento *a* do tipo *ResAttribute* tem de estar relacionados a classe *TestClass*. Em compensação, *setA* do tipo *ResSetMethod* está relacionado a classe *TestClass2*.

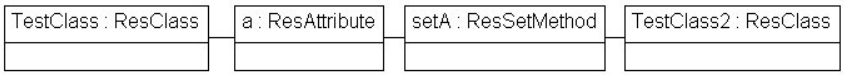

Figura 4.14: Instância de AAC com erro na relação entre atributos e métodos de acesso

Na Figura 4.15 é apresentado o invariante *targetsResClassSetMethod* que atua sobre elementos do tipo *ResSetMethod*. Neste invariante é verificado se o elemento do tipo *Res-*

Class relacionado ao *ResSetMethod* é o mesmo elemento que está relacionado ao atributo do tipo *ResAttribute* acessado pelo *ResSetMethod*.

```
1 context ResSetMethod inv targetsResClassSetMethod:
2   self.resclass = self.resattribute.resclass
```

Figura 4.15: Código OCL da relação entre *ResClass*, *ResSetMethod* e *ResAttribute*

Comparando-se ao trabalho apresentado em (BRAGA, 2010), novos invariantes foram adicionados na definição da linguagem AAC. Os invariantes *targetsResClassSetMethod* e *targetsResClassGetMethod* foram criados após a inclusão das relações entre o tipo *ResAttribute* e os tipos que representam métodos de acesso a este tipo. A finalidade destes invariantes é garantir a integridade das relações entre os tipos relacionados a um *ResAttribute* e o tipo *ResClass* que representa uma classe controlada. O mesmo princípio é aplicado para os tipos relacionados a pontas de associação dando origem aos invariantes *targetsResClassAssocEndGetMethod* e *targetsResClassAssocEndSetMethod*. Por fim, o invariante *targetsResClassAspectPointcut*, explicado anteriormente, foi adicionado para assegurar a relação correta entre os tipos *Aspect*, *PointCut*, *ResMethod* e *ResClass*. Os detalhes destes invariantes são apresentados na Seção B.1.

Capítulo 5

Serviço de Autorização e Autenticação do Java

No Capítulo 4 foi apresentada a linguagem AAC para modularização da autorização de um controle de acesso que é parametrizada pela autenticação. Um exemplo de *framework* de autorização com suporte a autenticação é o chamado Serviço de Autorização e Autenticação do Java (JAAS) que será apresentado neste capítulo.

5.1 JAAS

O Serviço de Autorização e Autenticação do Java (JAAS) (OAKS, 2001) é o serviço de autenticação e autorização da tecnologia de segurança do Java (*Java Security*). JAAS possui dois propósitos (ORACLE, 2006): (i) autenticação de usuários, para determinar de forma segura quem está executando o código Java, independentemente do código representar uma aplicação, *applet*, *bean*, ou *servlet*; (ii) autorização de usuários para realizar as ações requisitadas.

JAAS implementa uma versão em Java para Módulos de Autenticação Plugáveis (PAM, sigla em inglês), um mecanismo para integrar múltiplos esquemas de autenticação de baixo nível em uma Interface de Programação de Aplicações (API, sigla em inglês). A autenticação no JAAS é feita de forma modular, permitindo que as aplicações permaneçam independentes das tecnologias de autenticação. As novas tecnologias podem ser conectadas a uma aplicação sem a necessidade de efetuar modificações no código da aplicação.

O controle de acesso no Java é construído sobre quatro conceitos: códigos-fonte, permissões, políticas e domínios de proteção. Os códigos-fonte (*code source*) são um encapsu-

lamento da localização de certas classes de aplicações ou serviços do Java. O código-fonte é um simples objeto que reflete a URL da qual uma classe foi carregada e as chaves (caso exista alguma) que serão utilizadas para assinar estas classes.[1] As permissões (*permissions*) são um encapsulamento de uma requisição para desempenhar uma particular operação. Os domínios de proteção (*protection domain*) são um encapsulamento da relação entre códigos-fonte, permissões e os elementos do tipo *PrincipalRole*, que representam papéis ou grupos de usuários.

5.1.1 Autenticação

As aplicações em JAAS estabelecem o processo de autenticação instanciando um objeto de contexto de *login* (*LoginContext*). O contexto de *login* referencia um arquivo de configuração para determinar quais as técnicas de autenticação, no JAAS chamadas de módulos de *login* (*LoginModules*), serão utilizadas para efetuar o *login* do usuário no sistema. Os módulos de *login* podem requisitar informações do usuário para efetuar a autenticação, como nome de usuário e senha, voz, ou impressões digitais. Outros utilizam a autenticação nativa do sistema operacional no qual a aplicação está sendo executada.

O contexto de *login* (*LoginContext*) possui os usuário autenticados em um sistema controlado. Um usuário ao se autenticar é representado por um objeto *Subject*. Este objeto contém uma série de objetos *PrincipalRole* e credenciais. Um objeto *PrincipalRole* pode representar tipos diferentes de informações relevantes do usuário, e.g., nome do usuário, grupos dos quais faz parte, nome do *host* no qual o usuário foi autenticado, entre outras. As credenciais dos usuários podem ser definidas como um objeto qualquer que tenha informação relevante para a autenticação do usuário, por exemplo, chaves criptográficas ou certificados de segurança.

Um aplicativo JAAS pode utilizar mais de um módulo de autenticação, definidos através do arquivo de configuração. Diz-se então que os módulos de autenticação são empilháveis, pois adicionam novos elementos do tipo *PrincipalRole* ao elemento do tipo *Subject* a medida que vão sendo utilizados na autenticação.

A Figura 5.1 apresenta um exemplo de arquivo de configuração dos módulos de autenticação do JAAS. Neste arquivo estão definidas duas formas de autenticação: *Login1* e *Login2*. Na forma de autenticação denominada *Login1* é utilizado apenas um módulo de

[1] Este conceito não será utilizado neste trabalho, pois do ponto de vista de controle de acesso a usuários não importa de onde a classe foi carregada e se ela está assinada. Este tipo de controle está em um nível de abstração mais baixo, relacionado a segurança da parte de arquitetura e infraestrutura da aplicação.

autenticação, o *SampleLoginModule*. Já na forma *Login2* são utilizados quatro módulos de autenticação.

```
1 Login1 {
2  sample.SampleLoginModule required debug=true;
3 };
4
5 Login2 {
6  sample.SampleLoginModule required;
7  com.sun.security.auth.module.NTLoginModule sufficient;
8  com.foo.SmartCard requisite debug=true;
9  com.foo.Kerberos optional debug=true;
10 };
```

Figura 5.1: Exemplo de arquivo de configuração de *LoginModule* do JAAS

A modularização do JAAS ocorre pela forma como são aplicadas e verificadas as permissões de um usuário e os módulos de *login* utilizados na autenticação. A autenticação é definida por um arquivo indicando as formas de *login* e os módulos que serão necessários para a autenticação do usuário. A política de segurança contendo as permissões relacionadas a cada usuário é definida através de outro arquivo. Para executar uma aplicação de forma segura é necessário utilizar o administrador de segurança do Java (*SecurityManager*) e indicar a ele quais serão os arquivos de configuração e de *login* a serem utilizados.

5.1.2 Autorização

Os arquivos de política definem os domínios de proteção relacionando um elemento do tipo *PrincipalRole* a uma ou mais permissões. Na Figura 5.2 é apresentado um arquivo de política para o exemplo do gerenciador de arquivos. Neste exemplo são definidos dois elementos do tipo *PrincipalRole*: *RoleA* e *RoleB*.

A palavra reservada *grant* define quais as permissões estão associadas a um *PrincipalRole*. O principal do tipo *RoleA* está relacionado às permissões *DefaultPermission_JAASPermission* e a permissão *BasicPermission_JAASPermission*. As permissões são definidas como classes que estendem a classe abstrata *Permission*.

Para que se verifique a relação entre *PrincipalRole* e *BasicPermission* é necessária a criação de uma ação privilegiada, definida por uma classe que estenda a interface *PrivilegedAction* do JAAS. Entende-se por ação privilegiada em JAAS, aquela ação que pode ser executada somente por usuários que possuam privilégios específicos concedidos pela associação de um *Principal* e um *Permission*. Ao ser executado o método *run()* de uma classe que representa uma ação privilegiada é informado qual o *Subject* executará a ação. As

```
1 grant Principal br.uff.mda.jaas.security.auth.Role "RoleA" {
2  permission permission.BasicPermission_JAASPermission "";
3 };
4
5 grant Principal br.uff.mda.jaas.security.auth.Role "RoleB" {
6  permission permission.BasicPermission_JAASPermission "";
7  permission permission.AdvancedPermission_JAASPermission "";
8 };
```

Figura 5.2: Código do um arquivo de políticas do JAAS para o sistema de arquivos

classes que implementam a interface *PrivilegedAction* verificam junto ao *SecurityManager* se um determinada permissão está atrelada a um *PrincipalRole* do *Subject* corrente. Na Figura 5.3 é apresentado o código da ação de execução do método de criação de arquivos no gerenciador de arquivos. O objeto de *SecurityManager* do sistema é utilizado a partir do método *getSecurityManager()* da classe *System* e verifica se a permissão *BasicPermission_ JAASPermission* está associada ao *Subject* enviado para a execução desta ação.

```
1 public class FileSystemcreateMethodAtomicExecute implements
     PrivilegedAction {
2  public Boolean run(){
3    SecurityManager sm = System.getSecurityManager();
4    sm.checkPermission(new BasicPermission__JAASPermission("
        BasicPermission_JAASPermission"));
5  }
6 }
```

Figura 5.3: Código da ação privilegiada *FileSystemcreateMethodAtomicExecute*

Para execução da ação privilegiada, o método estático *Subject.doAsPrivileged()*[2] é executado e tem como parâmetros o *Subject* que irá executar a ação e a própria ação. Na Figura 5.4 é apresentado o código da classe *FileSystem* que representa o recurso do sistema de arquivos. Antes da execução dos passos do método ocorre a verificação se o elemento *actualSubject* que representa o usuário autenticado no sistema tem a permissão de executar a ação privilegiada *FileSystemcreateMethodAtomicExecute*.

As Figuras 5.5 e 5.6 apresentam execuções de criação de um novo arquivo através do sistema gerenciador de arquivos. Na Figura 5.5 é apresentada a criação com sucesso do arquivo "a.txt", onde um objeto da classe *FileSystem* executa a ação privilegiada *FileSystemcreateMethodAtomicExecute* passando um objeto *actualSubject* do tipo *Subject*. Este objeto possui algum elemento do tipo *PrincipalRole* relacionado a permissão *Ba-*

[2] O método Subject.doAsPrivileged() é um método interno ao JAAS e o seu fluxo de execução não faz parte do escopo deste trabalho.

```
1 public class FileSystem {
2   public create(String filename){
3     //Verifica antes da execução a permissão de acesso
4     if(Subject.doAsPrivileged(actualSubject, new
          FileSystemcreateMethodAtomicExecute())){
5       //executa passos do método
6     } else {
7       //emite erro de acesso não permitido
8     }
9   }
10 }
```

Figura 5.4: Código para execução de uma ação privilegiada

sicPermission_JAASPermission no arquivo de políticas da Figura 5.2, por exemplo, um elemento do tipo *PrincipalRole* com identificador *RoleA* ou *RoleB*.

Já na Figura 5.6 a criação não ocorre devido a uma tentativa de acesso não autorizado. Neste caso o processo é o mesmo, entretanto, o objeto *actualSubject* não possui nenhum elemento do tipo *PrincipalRole* autorizado a executar a ação privilegiada *FileSystemcreateMethodAtomicExecute*.

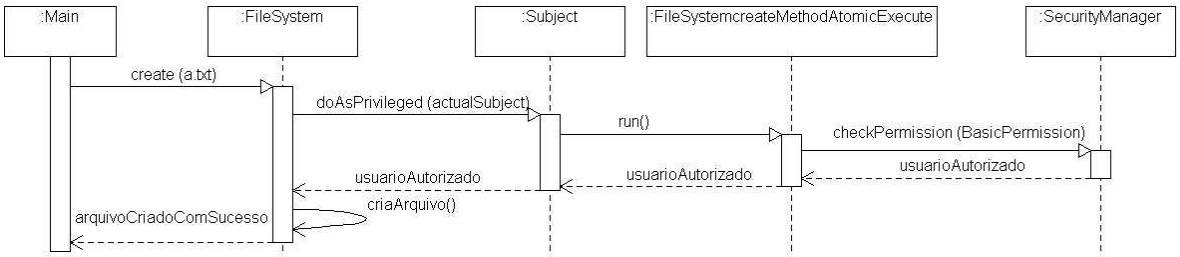

Figura 5.5: Criação de arquivo com sucesso através da classe FileSystem

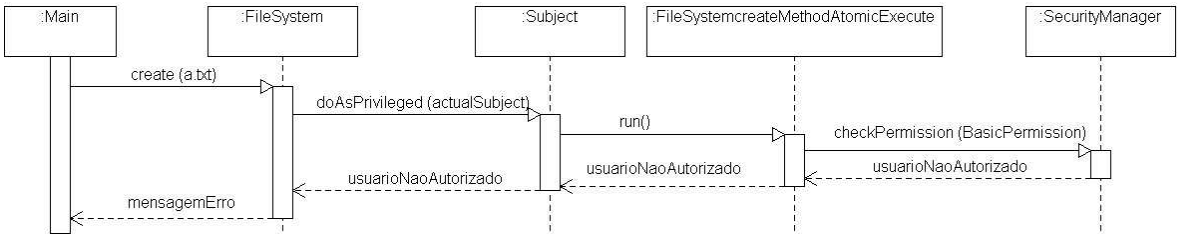

Figura 5.6: Falha na criação de arquivo através da classe FileSystem

5.2 O metamodelo de JAAS

Com a finalidade de utilizar JAAS como parte de um processo DDM, parte da autenticação e autorização do JAAS foi modelada como o metamodelo apresentado na Figura 5.7.

A metaclasse *PrincipalRole* representa um tipo que implementará a interface *Principal* no sistema controlado utilizando JAAS. O tipo de domínio de proteção (*ProtectionDomain*) relaciona os tipos de papéis e de permissões. O tipo *BasicPermission* herda de uma permissão do JAAS representada pelo tipo *JAASPermission*. O tipo *JAASAction* representa uma ação privilegiada e implementa a interface *PrivilegedAction* do JAAS. O tipo *PolicyFile* representa o arquivos de políticas que contém os domínios de proteção.

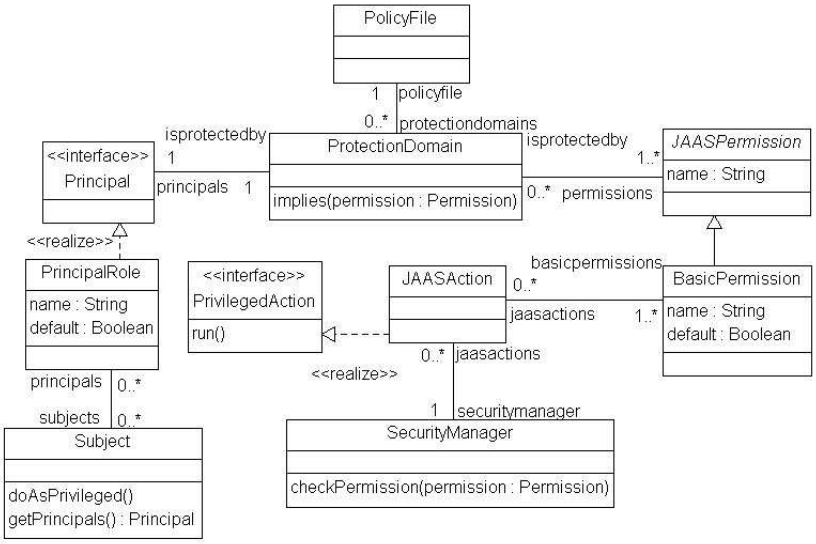

Figura 5.7: Metamodelo representando parte de JAAS

A Figura 5.8 apresenta a política de segurança do acesso ao gerenciador de arquivos como uma instância do metamodelo de JAAS. O conjunto de códigos-fonte e arquivos de configuração e de políticas apresentados nas seções 5.1.1 e 5.1.2 formam a sintaxe concreta de JAAS e representam o resultado do processo de *pretty print* da instância deste modelo.

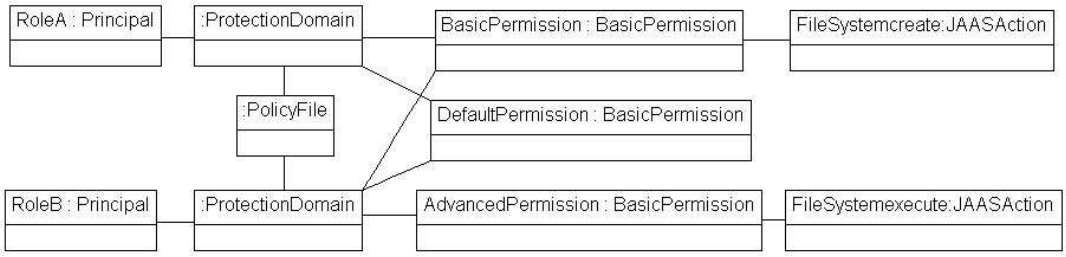

Figura 5.8: Instância do metamodelo de JAAS do sistema de arquivos

Cada instância do metamodelo de JAAS representa um arquivo de políticas de acesso do JAAS, classes de permissões e classes de ações privilegiadas. Um exemplo de arquivo de políticas do JAAS é apresentado na Figura 5.2, onde domínios de proteção relacionam os identificadores de elementos do tipo *PrincipalRole* com permissões do tipo *BasicPermission*. As permissões, que no metamodelo são elementos do tipo *BasicPermission*, são

representadas como classes que estendem a classe *BasicPermission* do JAAS. A Figura 5.9 apresenta o código da classe *BasicPermission_JAASPermission* que representa uma permissão no JAAS. Para as ações privilegiadas serão geradas classes que implementam a interface *PrivilegedAction*, como a ação *FileSystemcreateMethodAtomicExecute* apresentada na Figura 5.3. Para cada relação entre uma ação privilegiada e uma permissão, será gerado uma verificação na ação privilegiada de acesso a permissão através do método *chekpermission()* elemento do tipo *SecurityManager* do sistema.

```
1 public class BasicPermission__JAASPermission extends BasicPermission {
2   public BasicPermission__JAASPermission(String name) {
3     super(name);
4   }
5 }
```

Figura 5.9: Código da permissão *BasicPermission_JAASPermission*

O metamodelo gerado para JAAS não define o tipo de recurso a ser controlado, mas a geração da infraestrutura de autorizações e de ações privilegiadas que verificam se um determinado *principal* está relacionado a uma permissão.

A estrutura do metamodelo proposto para JAAS é simples e não possui restrições de tipos como os apresentados para SecureUML+ComponentUML ou AAC. Uma restrição que existe em JAAS é que um sistema controlado utiliza apenas um único arquivo de políticas de autorização. Dessa maneira no metamodelo criado existirá apenas um elemento representando o arquivos de políticas *JAASPolicy* e todos os domínios de proteção estarão relacionados somente a este elemento único. Na Figura 5.10 é apresentado o invariante *existsOnlyOnePolicyFile* que verifica se existe apenas um elemento representando o arquivo de políticas em JAAS. A relação entre domínios de proteção e o arquivo de políticas é garantido pela estrutura do metamodelo e do invariante *existsOnlyOnePolicyFile*.

```
1 context PolicyFile inv existsOnlyOnePolicyFile:
2   PolicyFile.allInstances()->size() = 1
```

Figura 5.10: Código OCL que garante que existe apenas um arquivo de políticas em JAAS

JAAS será utilizado neste trabalho juntamente com AAC, onde além de prover os mecanismos de autenticação, modificará a autorização provida por AAC. Este é o tema do Capítulo 6.

Capítulo 6

Especificação da transformação de SecureUML+ComponentUML para AAC+JAAS

A linguagem SecureUML+ComponentUML pode ser utilizada por Analistas de Segurança na fase de levantamentos de requisitos e na fase de concepção do controle de acesso de um sistema de software. Os modelos de políticas de segurança descritos nesta linguagem auxiliam no processo de desenvolvimento e documentação da implementação de políticas de segurança de um sistema com acesso controlado. Estes modelos podem ser construídos em qualquer editor UML equipado com o perfil UML, que representa esta linguagem de modelagem para construção de políticas de segurança.

Seguindo a abordagem de Desenvolvimento Dirigido a Modelos, os modelos SecureUML+ComponentUML são considerados artefatos vivos no processo de desenvolvimento e a partir deles é gerada infraestrutura de segurança para controle de acesso. Neste trabalho é apresentada uma infraestrutura de segurança baseada na linguagem AAC, que utiliza características de JAAS para complementar os processos de autenticação e autorização. A linguagem será chamada AAC+JAAS e representará a saída do processo de transformação a partir de um modelo SecureUML+ComponentUML.

Este trabalho utiliza a técnica de contratos de transformação apresentada na Seção 2.3 para especificação do processo de transformação entre modelos. O processo DDM adotado para a transformação entre SecureUML+ComponentUML e AAC+JAAS seguirá o processo apresentado na Figura 6.1, onde $M = M_{SecureUML+ComponentUML} \bowtie M_{AAC+JAAS}$, $k \in M$, e $k = (m \bowtie n)$.

Um modelo m descrito através do perfil UML definido para SecureUML+Compo-

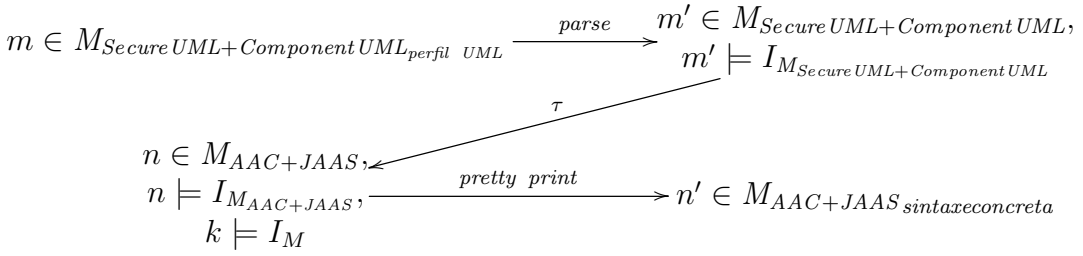

Figura 6.1: Processo MDD da transformação entre SecureUML+ComponentUML e AAC+JAAS

nentUML passará por um processo de *parsing* no qual gerará um modelo m', instância do metamodelo de SecureUML+ComponentUML, onde invariantes são utilizados para garantir que m' está em conformidade com SecureUML+ComponentUML.

Após as verificações dos invariantes, o modelo m' será transformado em um modelo n, instância do metamodelo de AAC+JAAS, por meio de uma transformação τ. Durante a transformação é gerado um modelo k, instância do metamodelo de transformação M. O metamodelo de transformação M especifica a transformação τ relacionando os tipos de SecureUML+ComponentUML e AAC+JAAS e descrevendo um conjunto de invariantes I_M.

Os invariantes I_M garantem que k está em conformidade com M e que a transformação foi executada corretamente. Em seguida, é verificado se n está em conformidade com AAC+JAAS através dos invariantes desta linguagem descritos por $I_{M_{AAC+JAAS}}$. Por fim, ocorre o processo de *preety printing* que gera a infraestrutura de códigos-fonte e arquivos de política de segurança para controle de acesso em AAC+JAAS.

Neste capítulo serão apresentados a linguagem AAC+JAAS, o metamodelo de transformação, e, por fim, os invariantes do metamodelo de transformação de SecureUML+ComponentUML para AAC+JAAS. A linguagem AAC+JAAS será apresentada na Seção 6.1 através de sua sintaxe concreta, do metamodelo que descreve a linguagem e seus invariantes. O metamodelo de transformação que relaciona os metamodelos SecureUML+ComponentUML e AAC+JAAS será apresentado na Seção 6.2 e os invariantes do metamodelo de transformação que garantem a correta relação entre seus tipos serão apresentados na Seção 6.3.

6.1 AAC+JAAS

No Capítulo 4 foi apresentada a linguagem AAC que modulariza a autorização de acesso a recursos controlados representados como classes abstratas. Para esta modularização, AAC utiliza elementos de Programação Orientada a Aspectos através da extensão AspectJ que representa aspectos na linguagem Java. Em AAC não existem mecanismos de autenticação e a autorização depende da implementação dos desenvolvedores e da instanciação de tipos para representar os papéis dos usuários autenticados.

O JAAS, apresentado no Capítulo 5, provê mecanismos de autenticação e autorização para Java de forma modular com relação a autenticação. Entretanto, a modularização de JAAS não se estende a autorização.

Como fruto deste trabalho decidiu-se unir as duas formas de controle de acesso em uma nova linguagem de modelagem chamada AAC+JAAS, que passa a agregar a modularização de autenticação do JAAS e a da autorização através de aspectos de AAC juntamente com os mecanismos de autorização de JAAS.

6.1.1 Metamodelo de AAC+JAAS

A linguagem AAC implementa a autorização de determinados papéis para executar ações sobre recursos controlados através de aspectos. Os papéis são gerados através da implementação da autenticação do sistema por parte dos desenvolvedores.

O metamodelo utilizado para descrever JAAS não define recursos controlados, mas a partir da autenticação do JAAS, determinados *principals* terão acesso a certas permissões através de um arquivo de política contendo domínios de proteção. A verificação de uma determinada permissão para acesso ocorre em ações privilegiadas.

Busca-se na união entre AAC+JAAS atrelar a modularização de AAC com a autenticação de JAAS. Para manter a modularização de AAC utiliza-se elementos referentes a aspectos e para manter a autenticação de JAAS utiliza-se os elementos para autorização gerados a partir da autenticação.

O metamodelo que representa AAC+JAAS é apresentado na Figura 6.2 e contém uma união disjunta dos tipos de AAC e JAAS, exceto os tipos *ResRole* e *ResUser* do metamodelo de AAC que foram substituídos por *PrincipalRole* definido em JAAS. A conexão entre os dois metamodelos ocorre através da ligação entre os elementos *BeforeAdvice* do metamodelo de AAC e *JAASAction* do metamodelo de JAAS.

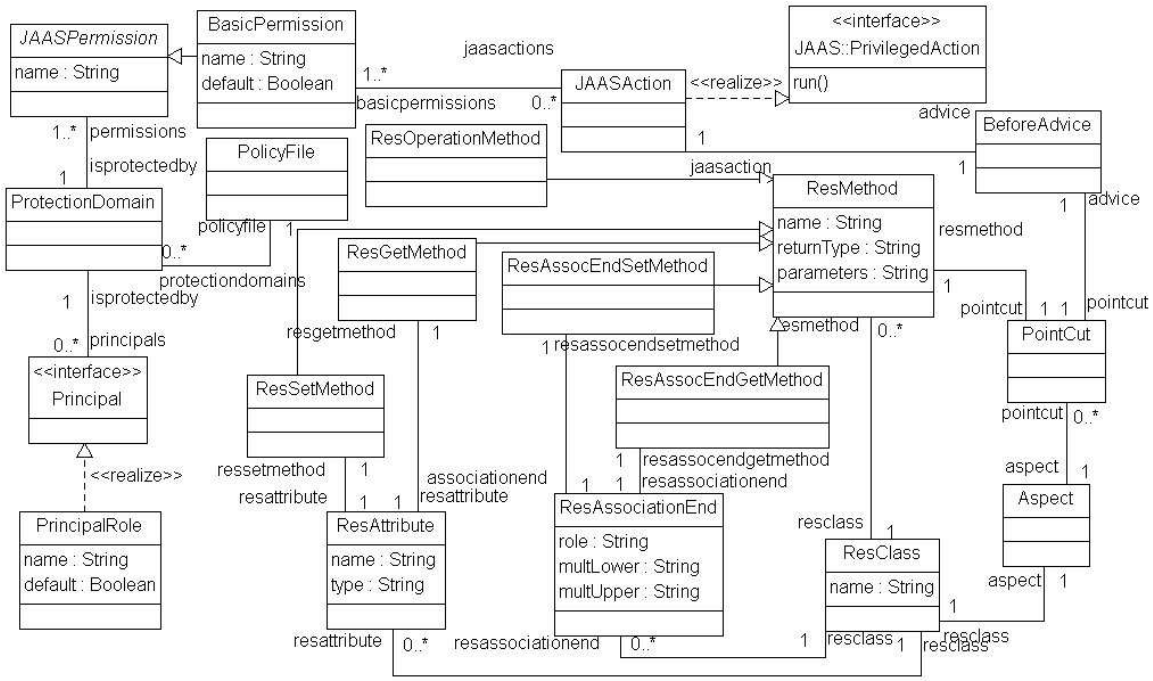

Figura 6.2: Metamodelo AAC+JAAS

Em AAC+JAAS os recursos controlados serão representados por uma classe abstrata definida pelo tipo *ResClass* e os elementos que a compõe assim como em AAC, pois o metamodelo definido para JAAS não possui o conceito de recurso controlado. Em AAC, um aspecto controla um tipo de dados, representado pela classe abstrata. Em AAC+JAAS serão mantidas as metaclasses referentes a aspectos e aos elementos que os compõem em AAC: *Aspect*, *Pointcut* e *BeforeAdvice*.

A modularização das autorizações em AAC+JAAS ocorre através do uso de aspectos, no entanto, a verificação de permissão de acesso ocorre através da utilização de elementos do tipo *JAASAction*, que representam ações privilegiadas no JAAS, ao invés de verificar diretamente as permissões sobre os papéis dos usuários, como ocorria em AAC.

Para utilizar a autenticação do JAAS, a metaclasse que representava papéis em AAC deixa de existir em AAC+JAAS, sendo substituída pelo metaclasse *PrincipalRole* definido em JAAS no Capítulo 5. No entanto, existe uma diferença entre os dois tipos, pois *PrincipalRole* não possui a hierarquia de papéis que era definida em AAC.

6.1.2 Um modelo descrito em AAC+JAAS

A Figura 6.3 apresenta a instância do metamodelo da AAC+JAAS da política de segurança do acesso ao gerenciador de arquivos. Os códigos-fonte e os arquivos de configuração

e política apresentados a seguir representam uma instância da sintaxe abstrata em forma da sintaxe concreta de AAC+JAAS e são o resultado do processo de *pretty printing* da instância deste modelo em sua sintaxe concreta.

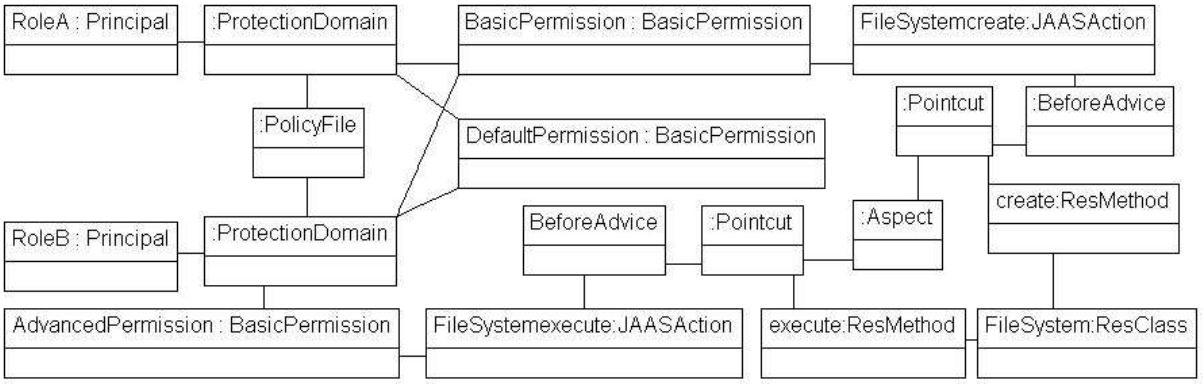

Figura 6.3: Política de segurança de arquivos como instância do metamodelo AAC+JAAS

A Figura 6.4 apresenta novamente a classe abstrata para o exemplo do gerenciador de arquivos, apresentado no Capítulo 5, que permanece inalterada na nova linguagem.

```
1 abstract class FileSystem {
2   abstract void create(String filename) throws Exception;
3   abstract void execute(String filename) throws Exception;
4 }
```

Figura 6.4: Classe abstrata do recurso de Sistema de Arquivos em AAC+JAAS

A classe *Env* de AAC é estendida e passa a declarar um método que retorna um objeto do tipo *SubjectJAAS*. A classe *Env* estendida é apresentada na Figura 6.5. O tipo *SubjectJAAS* representa uma interface para elementos do tipo *Subject* e auxilia na modularizaração do acesso às diversas formas de autenticação do JAAS. Este tipo será abordado no Capítulo 8 juntamente com a biblioteca *JAASLib*, que representa uma interface com os tipos *PrincipalRole* relacionados a cada tipo de autenticação. A relação de cada *PrincipalRole* com as permissões do tipo *BasicPermission* em AAC+JAAS ocorre através dos arquivos de política de JAAS.

```
1 public class Env {
2   private static SubjectJAAS subject;
3   public Env(SubjectJAAS subjectTmp) {
4     subject = subjectTmp;
5   }
6   public static SubjectJAAS getSubject() { return subject; }
7 }
```

Figura 6.5: Classe de ambiente que controla o usuário autenticado no sistema AAC+JAAS

No exemplo do sistema de gerenciador de arquivos, o arquivo de políticas apresentado na Figura 6.6 define *RoleA* com acesso às permissões *DefaultPermission_ JAASPermission* e *BasicPermission_ JAASPermission* e o papel *RoleB* com acesso à permissão *Advanced-Permission_ JAASPermission* e às permissões de *RoleA*. Caso ocorram modificações na relação entre papéis e permissões será somente necessário modificar o arquivo de políticas do JAAS.

```
1 grant Principal br.uff.mda.jaas.security.auth.Role "RoleA" {
2  permission permission.DefaultPermission_JAASPermission "";
3  permission permission.BasicPermission_JAASPermission "";
4 };
5
6 grant Principal br.uff.mda.jaas.security.auth.Role "DefaultRole" {
7  permission permission.DefaultPermission_JAASPermission "";
8 };
9
10 grant Principal br.uff.mda.jaas.security.auth.Role "RoleB" {
11  permission permission.DefaultPermission_JAASPermission "";
12  permission permission.BasicPermission_JAASPermission "";
13  permission permission.AdvancedPermission_JAASPermission "";
14 };
```

Figura 6.6: Código do um arquivo de políticas do AAC+JAAS para o gerenciador de arquivos

A Figura 6.7 apresenta o código de parte do aspecto que controla o acesso a execução do método *create()* do recurso *FileSystem* para o exemplo do gerenciador de arquivos. A variável *caller* recebe um elemento do tipo *Subject* representando o usuário autenticado no sistema. A partir deste usuário é feita a tentativa de execução da ação privilegiada *FileSystemcreateMethodAtomicExecute*.

```
1 public aspect FileSystem_ACAspect {
2  pointcut PCcreate( FileSystem self , String filename):
3    call(void FileSystem.create(String)) && target(self) && args(filename)
       ;
4  before (FileSystem self,String filename) throws Exception:PCcreate(self
     ,filename){
5    Subject caller=Env.getSubject();
6    if (Subject.doAsPrivileged(caller.getSubject(), new
       FileSystemcreateMethodAtomicExecute(self, filename), null))
7     return;
8    else
9     throw new Exception("Usuário não possui permissão para efetuar este
        acesso");
10  }
11 }
```

Figura 6.7: Parte do código do Aspecto para controle de acesso sobre a classe FileSystem em AAC+JAAS

A ação do tipo *JAASAction* utilizada em AAC+JAAS recebe em seu construtor o recurso controlado para possíveis verificações de restrições dinâmicas e implementa a interface *PrivilegedAction* do JAAS. O seu método *run()* efetuará a verificação ao ser executado pelo método abstrato *doAsPrivileged()* da classe *Subject*.

A Figura 6.8 apresenta o código da ação *FileSystemcreateMethodAtomicExecute* do tipo *JAASAction* que verifica a permissão de acesso ao método *create()* da classe controlada *FileSystem*. A ação recebe em seu construtor o recurso controlado *resource* do tipo *FileSystem* e os parâmetros do método *create* deste recurso, representando uma extensão do código apresentado em 5.3.

```
1 public class FileSystemcreateMethodAtomicExecute implements
    PrivilegedAction {
2  FileSystem resource;
3  String filename;
4  public FileSystemcreateMethodAtomicExecute(FileSystem resource, String
     filename) {
5   this.resource = resource;
6   this.filename = filename;
7  }
8  public Boolean run(){
9   SecurityManager sm = System.getSecurityManager();
10  try{
11   sm.checkPermission(new BasicPermission__JAASPermission("
       BasicPermission__JAASPermission"));
12  }catch (Exception e) {
13   return false;
14  }
15 }
16 }
```

Figura 6.8: Código da ação privilegiada para verificação de acesso ao método *create()* da classe *FileSystem*

A estrutura do metamodelo de AAC+JAAS permite certas construções que não são permitidas na sua definição e deverão ser verificadas através de invariantes. Os invariantes de AAC+JAAS são apresentados no Apêndice B e serão representados pelos invariantes de AAC em conjunto com os invariantes de JAAS, exceto os invariantes que incindem sobre os elementos de AAC retirados de AAC+JAAS. Um exemplo é a retirada do invariante que verificava a existência de ciclos em elementos do tipo *ResRole*.

6.2 Metamodelo de transformação

O metamodelo de transformação de um contrato de transformação é formado pela união disjunta das metaclasses da linguagem de entrada e das metaclasses da linguagem de

saída, além de associações entre as metaclasses das duas linguagens. O metamodelo de transformação entre as linguagens SecureUML+ComponentUML e AAC+JAAS seguirá este princípio e incluirá todas as metaclasses dos metamodelos das duas linguagens. Nesta seção é apresentada a relação entre as metaclasses no metamodelo de transformação.

As permissões na linguagem de entrada SecureUML+ComponentUML tem relação direta com elementos de permissão na linguagem de saída AAC+JAAS. Cada permissão de SecureUML+ComponentUML representada por um elemento do tipo *Permission* terá uma associação para uma permissão de AAC+JAAS, representada por um elemento do tipo *BasicPermission*. Cada restrição de autorização de SecureUML+ComponentUML, representada por um elemento do tipo *AuthorizationConstraint*, terá uma associação para uma restrição de AAC+JAAS, representada por um elemento do tipo *BasicPermissionConstraint*.

Os papéis na linguagem de entrada SecureUML+ComponentUML tem relação com tipos que representam papéis na linguagem de saída AAC+JAAS. Cada papel de SecureUML+ComponentUML, representado por um elemento do tipo *Role*, terá uma associação para um papel de AAC+JAAS, representada por um elemento do tipo *PrincipalRole* junto com um elemento de domínio de proteção representado pelo elemento do tipo *ProtectionDomain*. A Figura 6.9 destaca a relação entre as ações das duas linguagens do ponto de vista dos *roles* e permissões. Na parte inferior da Figura 6.9 temos os elementos do metamodelo de SecureUML+ComponentUML relacionados aos elementos do metamodelo de AAC+JAAS na parte superior.

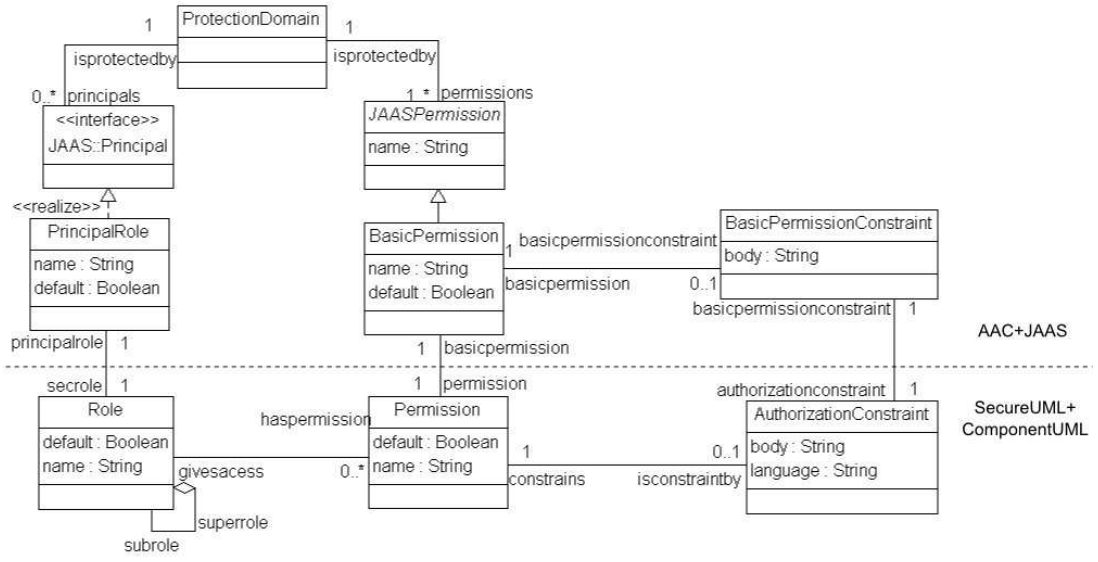

Figura 6.9: Relação entre papéis e permissões no metamodelo de transformação

Os recursos controlados na linguagem de entrada SecureUML+ComponentUML tem

uma associação com elementos da linguagem de saída AAC+JAAS, conforme é apresentado na Figura 6.10. Na parte inferior da Figura 6.10 estão as metaclasses representando recursos do metamodelo de entrada representadas por *Entity*, *Attribute*, *AssociationEnd* e *Method*. Cada metaclasse SecureUML+ComponentUML+ComponentUML na Figura 6.10 possui uma contraparte em AAC+JAAS dentre as metaclasses *ResClass*, *ResAttribute*, *ResAssociationEnd* e *ResOperationMethod*, numa bijeção trivial.

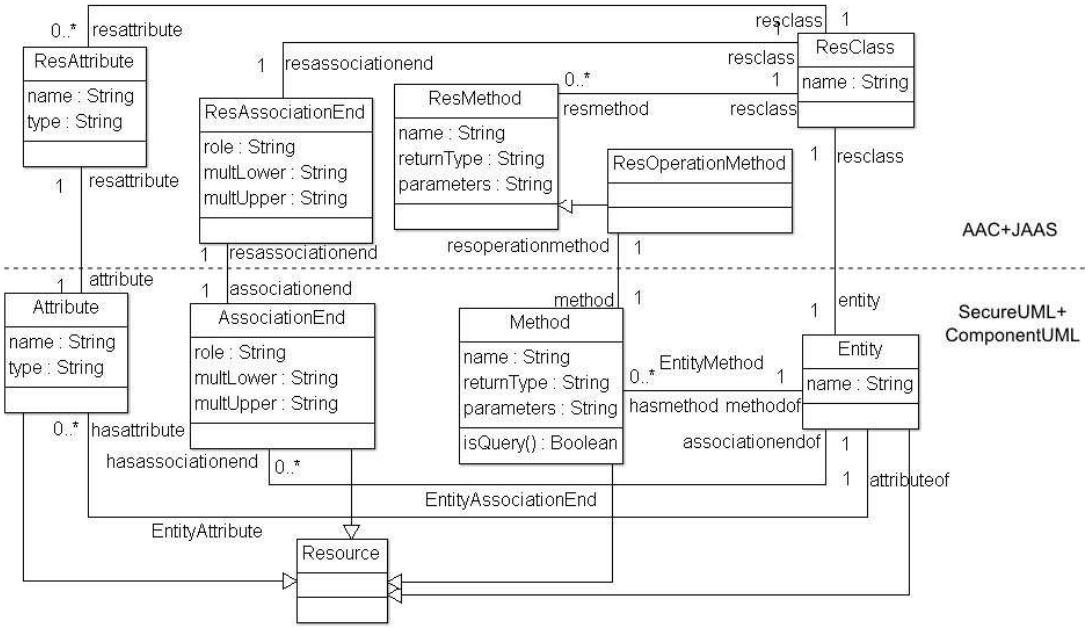

Figura 6.10: Relação entre recursos no metamodelo de transformação

As ações na linguagem de entrada SecureUML+ComponentUML tem relação com tipos da linguagem de saída AAC+JAAS. Cada ação atômica de SecureUML+ComponentUML representado por um elemento *AtomicAction* terá uma associação para uma ação de AAC+JAAS representada por um tipo *JAASAction*. A Figura 6.11 destaca a relação entre os papéis e permissões das duas linguagens. Na parte inferior temos as metaclasses do metamodelo de entrada relacionados às metaclasses do metamodelo de saída na parte superior.

6.3 Invariantes do metamodelo de transformação

Apenas as associações no metamodelo de transformação não garantem a correspondência de um elemento de AAC+JAAS com um elemento SecureUML+ComponentUML. Por exemplo, os valores dos metaatributos de um elemento da linguagem SecureUML+ComponentUML podem ser diferentes dos valores dos metaatributos do elemento correspondente

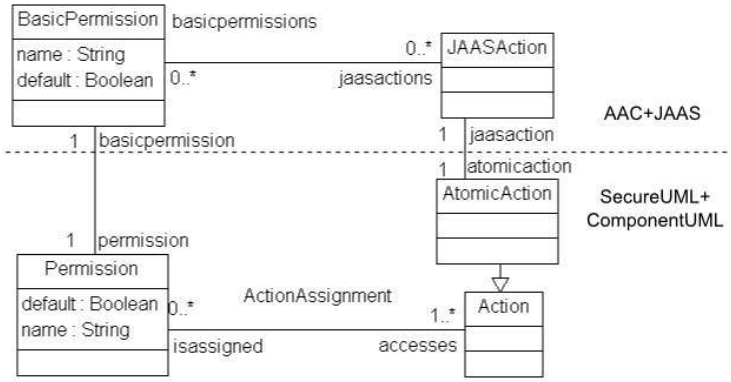

Figura 6.11: Relação entre ações de acesso a recurso no metamodelo de transformação

em AAC+JAAS. Para garantir a corretude da relação entre os tipos das duas linguagens foram criados invariantes no contrato de transformação de SecureUML+ComponentUML para AAC+JAAS para validar a relação estrutural do metamodelo de transformação.

Os invariantes OCL apresentados na Figura 6.12 garantem a correta relação entre os tipos que representam permissões em SecureUML+ComponentUML e AAC+JAAS no metamodelo de transformação. O invariante *targetsPermission* atua sobre todo elemento $bp \in BasicPermission$ de AAC+JAAS verificando se existe um elemento $p \in Permission$ de SecureUML+ComponentUML relacionado a bp através dos valores equivalentes dos metaatributos destes elementos. Caso p esteja relacionado a uma restrição de autorização $ac \in AuthorizationConstraint$ de SecureUML+ComponentUML, implica que o elemento bp deve estar relacionado a um elemento $bpc \in BasicPermissionConstraint$ em AAC+JAAS.

```
1 context BasicPermission inv targetsPermission:
2 self.permission->notEmpty() and self.name = self.permission.name and
3   self.default = self.permission.default and
4   self.permission.isconstraintby->notEmpty() implies
5     self.basicpermissionconstraint->notEmpty() and
6     self.basicpermissionconstraint = self.permission.isconstraintby.basicpermissionconstraint
7
8 context Permission inv targetsBasicPermission:
9 self.basicpermission->notEmpty()
```

Figura 6.12: Invariantes OCL do contrato de transformação da relação entre *Permission* e *BasicPermission*

A relação entre os tipos que representam restrições de autorização de SecureUML+ComponentUML e de AAC+JAAS é garantida pelo invariante OCL *targetsAuthorizationConstraint* apresentado na Figura 6.13. Este invariante garante que os metaatributos dos tipos relacionados possuem os mesmos valores.

```
1 context BasicPermissionConstraint inv targetsAuthorizationConstraint:
2 self.authorizationconstraint->notEmpty() and self.body = self.authorizationconstraint.body
3
4 context AuthorizationConstraint inv targetsBasicPermissionConstraint:
5 self.basicpermissionconstraint->notEmpty()
```

Figura 6.13: Invariantes OCL do contrato de transformação da relação entre *AuthorizationConstraint* e *BasicPerimissionConstraint*

A relação entre papéis no metamodelo de transformação é representada pela relação entre tipos *Role* de SecureUML+ComponentUML e *PrincipalRole* de AAC+JAAS. A correta relação entre estes tipos deve garantir, além da relação entre os metaatributos de cada tipo, a correta relação entre as permissões de cada linguagem, ou seja, as permissões concedidas a um papel em SecureUML+ComponentUML devem ser mantidas em AAC+JAAS. O Capítulo 7 argumenta sobre esta corretude. O passo que foi chamado de linearização da hierarquia de papéis é necessário pois em SecureUML+ComponentUML existe o conceito de hierarquia de papéis, o que não ocorre em AAC+JAAS, e influencia diretamente na relação entre os tipos *Role* e *Permission*.

Para a relação entre recursos controlados nas duas linguagens foram criados os seguintes invariantes apresentados na Figura 6.14: *AttributeTargetsResAttribute*, que garante a correta relação entre atributos nas duas linguagens; *AssociationEndTargetsResAssociationEnd*, que garante a correta relação entre as pontas de associação nas duas linguagens; *MethodTargetsResOperationMethod*, que garante a correta relação entre os métodos nas duas linguagens; e *EntityTargetsResClass*, que garante a correta relação entre as entidades nas duas linguagens.

Destaca-se o invariante *EntityTargetsResClass* que efetua outras verificações além da relação entre os metaatributos dos elementos do tipo *Entity* de SecureUML+ComponentUML+ComponentUML e *ResClass* de AAC+JAAS. Este invariante verifica também a relação entre os atributos, pontas de associação e métodos relacionados ao tipo *Entity* com os tipos correspondentes em AAC+JAAS . Por exemplo, considerando que um elemento $e \in Entity$ de SecureUML+ComponentUML+ComponentUML está relacionado a um elemento $rc \in ResClass$ de AAC+JAAS, se e está relacionado a um elemento $a \in Attribute$ de SecureUML+ComponentUML, então a deve estar relacionado a um elemento $ra \in ResAttribute$ de AAC+JAAS relacionado a rc, gerando um ciclo correto. O mesmo ocorre para pontas de associação e métodos.

As ações de acesso a recursos controlados em SecureUML+ComponentUML também possuem hierarquias que se relacionam às permissões de acesso da linguagem através da re-

```
1 context Attribute inv AttributeTargetsResAttribute:
2   self.name = self.resattribute.name and
3   self.type = self.resattribute.type
4
5 context AssociationEnd inv AssociationEndTargetsResAssociationEnd:
6   self.role = self.resassociationend.role and
7   self.multLower = self.resassociationend.multLower and
8   self.multupper = self.resassociationend.multUpper
9
10 context Method inv MethodTargetsResOperationMethod:
11   self.name = self.resoperationmethod.name and
12   self.returnType = self.resoperationmethod.returnType and
13   self.parameters = self.resoperationmethod.parameters
14
15 context Entity inv EntityTargetsResClass:
16   self.name = self.resclass.name and
17   self.hasattribute = self.resclass.resattribute.attribute and
18   self.hasassociationend = self.resclass.resassociationent.assocationend and
19   self.hasmethod = self.resclass.resmethod.method
```

Figura 6.14: Invariantes OCL da relação entre recursos no metamodelo de transformação

lação entre os tipos *Permission* e *Action*. Assim como na relação de papéis do metamodelo de transformação é necessária a execução de um passo de linearização e posteriormente a transformação que, juntamente com os invariantes do contrato de transformação de SecureUML+ComponentUML para AAC+JAAS, garantirão a correta transformação as relações das permissões e das ações em SecureUML+ComponentUML em AAC+JAAS.

Além das relações com permissões, as ações estão relacionadas aos recursos controlados em SecureUML+ComponentUML e esta relação deve ser garantida através dos tipos de AAC+JAAS. Para falarmos da relação entre ações e recursos é necessário entender como ocorre o acesso aos recursos controlados em cada linguagem, tanto em SecureUML+ComponentUML quanto em AAC+JAAS.

Na Figura 6.15 é apresentada a relação de dependência entre os recursos controlados da linguagem SecureUML+ComponentUML e as ações atômicas que efetuam o acesso a estes recursos. Esta relação de dependência é garantida pelos invariantes descritos para a relação entre recursos e ações em SecureUML+ComponentUML . Somente serão consideradas as ações atômicas, pois os elementos do tipo *JAASAction* de AAC+JAAS estão relacionados somente a estas ações. As ações compostas de SecureUML+ComponentUML não serão utilizadas após o passo de linearização da hierarquia de ações.

Os invariantes de SecureUML+ComponentUML *targetsAnEntity*, *targets* e *targetsAMethod* relacionados aos elementos do tipo *AtomicCreate*, *AtomicDelete*, *AtomicUpdate*, *AtomicRead* e *AtomicExecute* garantem a relação entre as ações e os recursos em Secure-

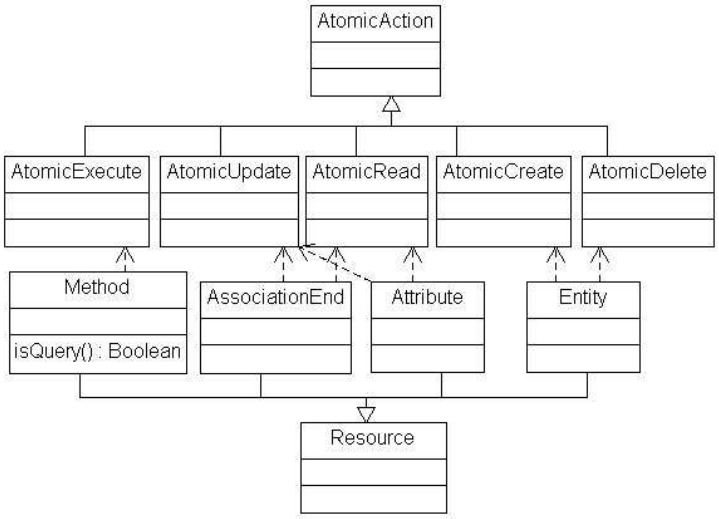

Figura 6.15: Dependência entre ações atômicas e recursos na linguagem SecureUML+ComponentUML

UML+ComponentUML. Cada tipo de ação só pode estar relacionada a recursos específicos. A Figura 6.16 apresenta o invariante *targetsAMethod*, onde o elemento do tipo *AtomicExecute* somente pode estar relacionado a recursos do tipo *Method*.

```
1 context AtomicExecute inv targetsAMethod:
2   self.resource.oclIsTypeOf(Method)
```

Figura 6.16: Invariante OCL de SecureUML+ComponentUML *targetsAMethod* que garante a correta relação entre os elementos do tipo *AtomicExecute* e *Method*

Outros invariantes de SecureUML+ComponentUML são também utilizados para garantir a relação entre ações e recursos. É o caso dos invariantes chamados *areAccessedBy* que garantem que cada recurso somente será acessado pelos tipos de ações dos quais tem relações de dependência. A Figura 6.17 apresenta o invariante *areAccessedBy*, que garante que um elemento do tipo *Method* só pode estar relacionado a uma ação do tipo *AtomicExecute*.

```
1 context Method inv areAccessedBy:
2 self.action->size() = 1 and self.action->exists(a|a.oclIsTypeOf(AtomicExecute))
```

Figura 6.17: Invariante OCL *areAccessedBy* que garante a correta relação entre os elementos dos tipos *Method* e *AtomicExecute*

Já o acesso aos recursos em AAC+JAAS é garantido pela estrutura da linguagem, onde cada recurso é acessado por métodos específicos. Na Figura 6.18 é apresentado parte do metamodelo de AAC+JAAS, com destaque para o acesso dos recursos que ocorre através de um ou mais métodos descritos por tipos que estendem o tipo *ResMethod* e tem

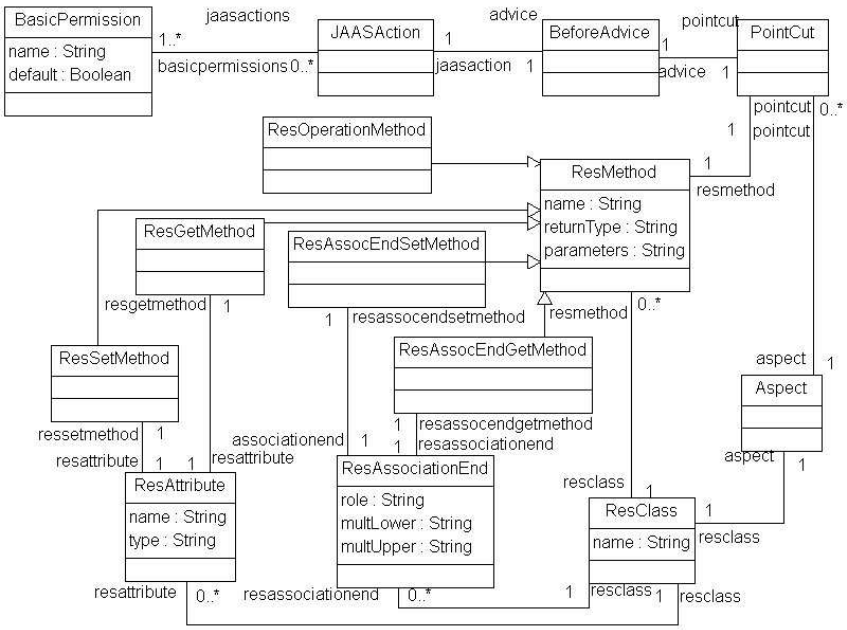

Figura 6.18: Acesso aos recursos eme AAC+JAAS

associações diretas aos recursos acessados. Um atributo está associado a dois métodos de acesso: *ResGetMethod* para ler o atributo e *ResSetMethod* para alterá-lo. Uma ponta de associação está associada a dois métodos de acesso: *ResAssocEndGetMethod* para ler o atributo e *ResAssocEndSetMethod* para alterá-la. O elemento do tipo *ResOperationMethod* representa um método que não é do tipo *getter* ou *setter* e será a forma de acesso a métodos controlados. Cada método está associado a um único *JAASAction* através dos elementos dos tipos *Pointcut* e *BeforeAdvice*. Portanto, o acesso a um recurso ocorrerá se houver uma permissão *BasicPermission* associada a um método através de um elemento do tipo *JAASAction*, onde esta relação será sempre única.

Uma vez explicadas as formas de acesso de cada linguagem podemos garantir a correta relação entre os elementos de tipos que representam ações no metamodelo de transformação com o uso de invariantes. Cada ação SecureUML+ComponentUML do tipo *AtomicExecute* se relacionará à um tipo AAC+JAAS do tipo *ResOperationMethod*. Cada ação SecureUML+ComponentUML do tipo *AtomicRead* se relacionará à um elemento do tipo *ResGetMethod* ou *ResAssociationEndGetMethod* de AAC+JAAS, dependendo do recurso ao qual ela está associada. Cada ação SecureUML+ComponentUML do tipo *AtomicUpdate* se relacionará à um elemento do tipo *ResSetMethod* ou *ResAssociationEndSetMethod* de AAC+JAAS, dependendo do recurso ao qual ela está associada.

As ações do tipo *AtomicDelete* e *AtomicCreate* não possuem correspondentes em AAC+JAAS. Uma ação do tipo *AtomicCreate* representa a criação de uma entidade

controlada, entretanto, não é possível efetuar verificações em um método utilizando *before advices* em AspectJ. Já uma ação do tipo *AtomicDelete* representa a ação de apagar uma entidade controlada, entretanto, não é possível criar um *pointcut* em AspectJ para destrutores de classes controladas. Como solução, pode-se definir no diagrama SecureUML+ComponentUML métodos que representem estas ações na entidade controlada.

Para a relação entre *AtomicUpdate* e *ResSetMethod* ou *ResSetAssociationEndMethod* foi criado o invariante *targetsUpdateResMethodsFromResAttributeOrResAssociationEnd* apresentado na Figura 6.19 que atua sobre elementos do tipo *AtomicUpdate* e garante que o elemento relacionado a ele no metamodelo de AAC+JAAS é do tipo *ResSetMethod* ou *ResSetAssociationEndMethod*. Ainda é necessário verificar que um elemento do tipo *ResSetMethod* se relaciona com um *AtomicUpdate* e de maneira análoga com *ResSetAssociationEndMethod*.

```
1 context AtomicUpdate inv targetsUpdateResMethodsFromResAttributeOrResAssociationEnd:
2 self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResSetMethod)
3   or self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResAssocEndSetMethod)
```

Figura 6.19: Invariante OCL *targetsUpdateResMethodsFromResAttributeOrResAssociationEnd* que garante a correta relação entre os elementos dos tipos *AtomicUpdate* e *ResSetMethod* ou *ResAssocEndSetMethod*

Para a relação entre *ResSetMethod* e *AtomicUpdate* foram criados no metamodelo de transformação os invariantes *targetsAtomicUpdateFromResSetMethod* e *ResAttributeTargetsAttributeThroughResSetMethod* que são apresentados na Figura 6.20. O invariante *targetsAtomicUpdateFromResSetMethod* atua sobre elementos do tipo *ResSetMethod* e garante que elementos a eles relacionados no metamodelo de transformação são elementos do tipo *AtomicUpdate*. O invariante *ResAttributeTargetsAttributeThroughResSetMethod* garante que um elemento do tipo *ResAttribute* de AAC+JAAS está relacionado a um elemento do tipo *Attribute* de SecureUML+ComponentUML através da relação entre os acessos de cada elemento no metamodelo de transformação. Os dois invariantes são complementares ao invariante *targetsUpdateResMethodsFromResAttributeOrResAssociationEnd*.

Os invariantes do metamodelo de transformação que garantem a relação entre *AtomicUpdate* e *ResAssociationEndSetMethod* são apresentados na Figura 6.21 de forma análoga aos invariantes da relação entre *AtomicUpdate* e *ResSetMethod*.

Ainda de forma análoga, ocorre a relação entre *AtomicRead* e *ResGetMethod* ou *ResAssocEndGetMethod* no metamodelo de transformação. Os invariantes desta relação são

```
1 context ResSetMethod inv targetsAtomicUpdateFromResSetMethod:
2 self.pointcut.advice.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResAttribute inv ResAttributeTargetsAttributeThroughResSetMethod:
5 self.ressetmethod.pointcut.advice.jaasaction.action.resource = self.attribute
```

Figura 6.20: Invariantes OCL que garantem a correta relação entre os elementos dos tipos *ResSetMethod* e *AtomicUpdate*

```
1 context ResAssocEndSetMethod inv targetsAtomicUpdateFromResSetAssociationEndMethod:
2 self.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResAssociationEnd inv
      ResAssociationEndTargetsAssociationEndThroughResSetAssociationEndMethod:
5 self.resassociationendsetmethod.pointcut.advice.jaasaction.action.resource = self.associationend
```

Figura 6.21: Invariantes OCL que garantem a correta relação entre os elementos dos tipos *ResAssocEndSetMethod* e *AtomicUpdate*

apresentados no Apêndice C. Para a equivalência entre *AtomicExecute* e *ResOperationMethod* foram criados no metamodelo de transformação os invariantes *targetsResOperationMethod* e *targetsAtomicExecute* apresentados na Figura 6.22. O invariante *targetsResOperationMethod* atua sobre elementos do tipo *AtomicExecute* de SecureUML+ComponentUML e garante que o elemento relacionado a ele é do tipo *ResOperationMethod* de AAC+JAAS. O invariante *targetsAtomicExecute* atua sobre elementos do tipo *ResOperationMethod* de AAC+JAAS e garante que elementos relacionados a ele no metamodelo de transformação são elementos do tipo *AtomicExecute* de SecureUML+ComponentUML.

```
1 context AtomicExecute inv targetsResOperationMethod:
2 self.jaasaction.resmethod.oclIsTypeOf(ResOperationMethod)
3
4 context ResOperationMethod inv targetsAtomicExecute:
5 self.jaasaction.action.oclIsTypeOf(AtomicExecute)
```

Figura 6.22: Invariantes OCL que garantem a correta relação entre os elementos dos tipos *ResOperationMethod* e *AtomicExecute*

Com a verificação dos invariantes do metamodelo de transformação garante-se que a transformação de um modelo SecureUML+ComponentUML em modelo AAC+JAAS ocorreu corretamente. No Capítulo 7 será argumentada a corretude da transformação especificada juntamente com o passo de linearização dos elementos de papéis e ações.

6.4 Discussão

Neste capítulo foi apresentada a linguagem AAC+JAAS. Esta linguagem foi definida com o objetivo de controlar o acesso a recursos através da modularização alcançada por elementos de orientação a aspectos utilizando a autenticação fornecida pelo Serviço de Autenticação e Autorização do Java.

Esta nova linguagem representa uma extensão da linguagem de modelagem AAC, apresentada inicialmente em (BRAGA, 2010) e que após uma análise de seus elementos e relações sofreu algumas alterações conforme apresentado no Capítulo 4. A linguagem AAC controla a autorização de um sistema, entretanto, possui a carência de um mecanismo de autenticação na sua sintaxe concreta. Nesta linguagem, o desenvolvedor é o responsável por acoplar um mecanismo de autenticação no código gerado a partir de sua sintaxe concreta.

Em busca de acoplar mecanismos de autenticação em uma nova linguagem de modelagem para geração de uma infraestrutura padronizada foram analisadas alguns sistemas de segurança. O primeiro sistema de segurança analisado foi o Spring Security (MULARIEN, 2010) que fornece autenticação e autorização sobre aplicações desenvolvidas na plataforma Spring (WALLS; BREIDENBACH, 2005) de maneira personalizável em diversos níveis. A plataforma Spring é construída sobre a linguagem Java, portanto, o Spring Security não constitui uma plataforma padrão de segurança sobre a linguagem Java. Outra alternativa foi representada pelo sistema de segurança nativo ao ASP.Net (GASTER *et al.*, 2002), que possui construções específicas para controle de acesso baseados em papéis. Contudo, devido a não haver construções que incorporassem os conceitos de orientação a aspectos em ASP.Net no momento desta análise, esta alternativa foi descartada. A partir da premissa de geração de uma infraestrutura padronizada e extensível a partir da linguagem AAC, decidiu-se por utilizar o mecanismo de segurança JAAS, descrito no Capítulo 5. Este sistema de segurança é nativo a linguagem Java, o que facilita a extensão de AAC com utilização de AspectJ para os elementos de orientação a aspectos.

O JAAS apresenta algumas peculiaridades, como é o caso da descrição da relação entre os *principals* e as permissões através de arquivos não criptografados, ou ainda a falta de transparência na forma como são feitas as verificações de autorização. Vale ainda destacar que JAAS é um padrão que sofreu poucas modificações desde o seu lançamento. Devido a estes fatores, pode-se buscar no futuro a utilização de outros mecanismos de autenticação e da geração de novas linguagens de modelagem mais robustas.

Com relação a AAC+JAAS, apesar de esta ser uma extensão de AAC, ela não representa uma extensão conservativa, pois não possui o conceito de hierarquia de papéis e ações. Este fato é importante com relação a transformações, pois não só novos elementos devem ser adicionados, mas também a forma estrutural é diferente.

A partir da criação da linguagem AAC+JAAS foi possível descrever a transformação entre modelos de SecureUML para esta nova linguagem. A transformação é descrita pela técnica de contratos de transformação que especifica a transformação através de um metamodelo de transformação que conecta os elementos das linguagens utilizadas na transformação. Sobre o metamodelo de transformação são definidos invariantes que garantem que as propriedades dos invariantes da linguagem de entrada serão mantidos na linguagem de saída. Esta técnica é focada na especificação da transformação da mesma forma que se busca especificar um sistema orientado a objetos através de diagramas UML. A fase de especificação da técnica de contratos de transformação não visa a forma como ocorrerá a transformação, mas o resultado do processo, o que permite uma visão em alto nível da transformação, melhor planejamento do processo de transformação, antecipar possíveis problemas na conexão entre elementos das linguagens e geração de testes para validar a transformação.

Capítulo 7

Corretude do contrato de transformação de SecureUML+ComponentUML para AAC+JAAS

A transformação através de contratos de transformação deve garantir que o modelo de saída seja equivalente ao modelo de entrada. Nesta seção será apresentada uma argumentação de que o modelo de entrada SecureUML+ComponentUML terá um modelo equivalente como saída representado na linguagem AAC+JAAS. Isso ocorre devido a preservação dos invariantes de SecureUML+ComponentUML na transformação para AAC+JAAS, especificado pelo contrato de transformação.

A técnica de contratos de transformação apresentada neste capítulo é composta pelo metamodelo de transformação e de invariantes sobre este metamodelo. No metamodelo de transformação elementos do metamodelo de entrada e saída são relacionados através de associações entre suas metaclasses. Os invariantes sobre o metamodelo de transformação restringem as relações entre as metaclasses dos metamodelos referenciados pela transformação especificando *o que* a transformação deve produzir.

Para descrever a corretude da transformação, o modelo de SecureUML+ComponentUML será transformado em um modelo em uma linguagem de modelagem equivalente, retirando algumas associações que representam hierarquias do metamodelo de SecureUML+ComponentUML, o qual será chamado de SecureUML+ComponentUML com as hierarquias de papéis e ações compostas linearizadas. A transformação dos modelos em SecureUML+ComponentUML com hierarquias linearizadas para modelos em AAC+JAAS ocorre então de maneira direta e logo é possível descrever os invariantes do contrato de transformação trivialmente. Neste capítulo, toda referência a contrato de transformação será restrita ao contrato entre SecureUML+ComponentUML e AAC+JAAS.

7.1 Linearização das hierarquias de *roles* e ações em SecureUML

O metamodelo de SecureUML+ComponentUML apresentado novamente na Figura 7.1 possui duas associações que representam hierarquias na sua estrutura. A primeira relação é a composição entre papéis, representados pela associação *RoleHierarchy* entre elementos *Role*. A segunda relação é a composição entre ações, representados pela associação *ActionHierarchy* entre elementos *CompositeAction* e *Action*.

7.1.1 Linearização da hierarquia de *roles*

Com relação a hierarquia de papéis, um *role* herdará as permissões para execução das ações de todos os seus *superroles*, ou seja, do fecho transitivo da relação *RoleHierarchy*. Entretanto, na linguagem AAC+JAAS, não existe a ideia de hierarquia de papéis.

Antes de transformar um modelo SecureUML+ComponentUML num modelo AAC+JAAS as hierarquias de *roles* e de ações serão linearizadas. Isso significa que para cada papel em SecureUML+ComponentUML, existirá um elemento equivalente em AAC+JAAS contendo todas as permissões do fecho transitivo da hierarquia de papéis, garantindo que o acesso de um usuário em SecureUML+ComponentUML será preservado em AAC+JAAS. A mesma ideia se aplica à hierarquia de ações, onde a permissão de acesso através de uma ação a um recurso deve ser preservado em AAC+JAAS após a transformação.

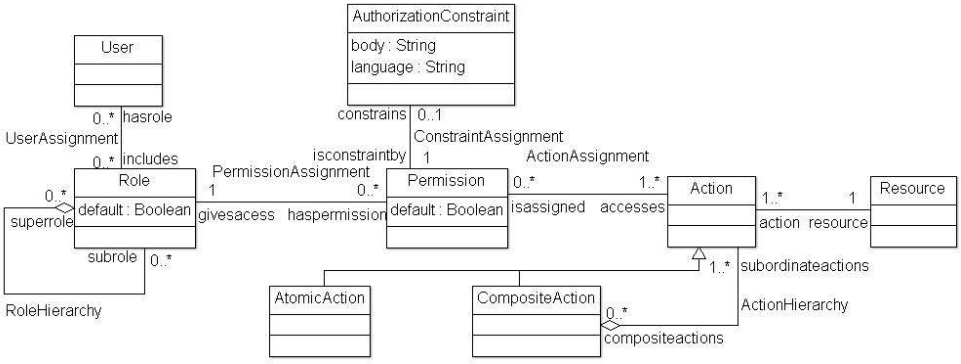

Figura 7.1: Metamodelo da linguagem SecureUML

O processo de linearização de *roles* é definido como uma transformação sobre a hierarquia de papéis de um $role \in Role$ de SecureUML+ComponentUML transformando-o em $role' \in Role'$ em SecureUML+ComponentUML com a hierarquia linearizada, onde *role'* não têm relações de *superrole* e *subrole* e está relacionado a todas os elementos

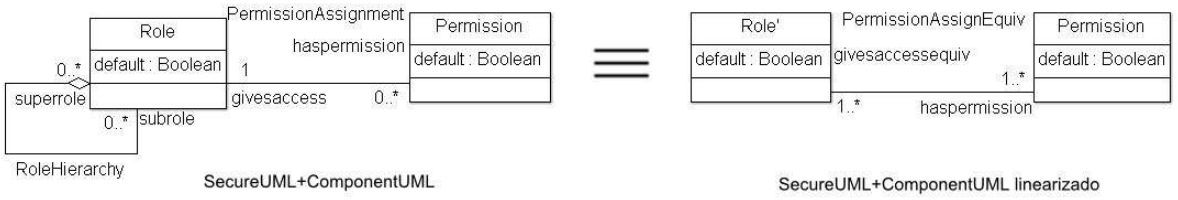

Figura 7.2: Equivalência estrutural do metamodelo SecureUML+ComponentUML e SecureUML+ComponentUML linearizada com relação a hierachia de papéis

$p \in Permission$ do fecho-transitivo de *superrole* de *role*. Além disso, *role'* deve ter todos os metaatributos contendo os mesmos valores de *role*. Esta equivalência estrutural é apresentada na Figura 7.2, onde a relação do tipo *Role'* com *Role* é exibida.

O invariante *targetsRoleFlattened* apresentado na Figura 7.3 especifica a linearização de papéis no contrato de transformação. Este invariante verifica que para cada $role \in Role$ existe um único $role' \in Role'$, o qual não possui hierarquia de papéis. O invariante também garante que o elemento *role'* terá os valores de seus metaatributos iguais aos valores dos metaatributos de *role* e que as permissões associadas a um *role'* definidas pela ponta de associação *haspermission* são as mesmas que o conjunto de permissões do fecho-transitivo de *role* gerados pela operação *allPermissions()*. Esta verificação valerá para todos os elementos do tipo *Role*.

```
1 context Role inv targetsRoleFlattened:
2 Role'.allInstances()->select( role' |
3   role'.name = self.name and
4   role'.default = self.default
5   role'.haspermission = self.allPermissions())
6   ->size() = 1
```

```
1 context Role::allPermissions():Set(Permission) body:
2 self.superrolePlus().haspermission->asSet()
```

Figura 7.3: Invariantes da relação entre *role* e *role'* no contrato de transformação

7.1.2 Linearização da hierarquia de ações

De forma análoga, a linguagem SecureUML+ComponentUML define que uma ação subordinada (*subordinatedaction*) herdará as permissões de todos as ações que a compõe, ou seja, do fecho transitivo da relação *ActionHierarchy*. A linearização da hierarquia de ações faz com que um $aa \in AtomicAction$ seja transformado em $aa' \in AtomicAction$ em SecureUML+ComponentUML. Para tanto, *aa'* deixa de ter a composição de hierarquia de ações e fica relacionada a todos os elementos $p \in Permission$ do fecho-transitivo da

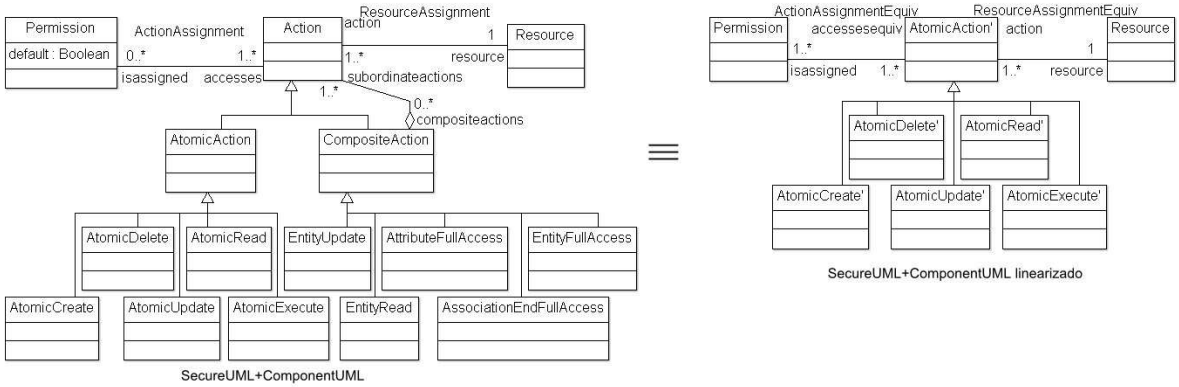

Figura 7.4: Equivalência estrutural do metamodelo SecureUML+ComponentUML e SecureUML+ComponentUML com com relação a hierachia de ações linearizada.

associação *subordinatedaction* de *aa*. Além disso, *aa′* deve ter todos os metaatributos com os mesmos valores de *aa*. Esta equivalência estrutural é apresentada na Figura 7.4, onde *AtomicAtion′* de SecureUML+ComponentUML linearizado equivalente a *AtomicAction* de SecureUML+ComponentUML é exibido.

O invariante *targetsActionFlattened* apresentado na Figura 7.5 especifica a linearização de ações. Este invariante verifica que para cada *aa* ∈ *AtomicAction* de SecureUML+ComponentUML existe um único *aa′* ∈ *AtomicAction′* de SecureUML+ComponentUML linearizado sem hierarquia de ações. O invariante também garante que *aa′* terá os valores de seus metaatributos iguais aos valores dos metaatributos de *aa* e que as permissões associadas a um *aa'* definidas pela ponta de associação *isassigned* são as mesmas que o conjunto de permissões do fecho-transitivo de *aa* gerados pela operação *allAssignedPermissions()*. O invariante também deve verificar que a relação com o recurso de *aa* foi mantido em *aa′*.

```
1 context AtomicAction inv targetsActionFlattened:
2 AtomicAction'.allInstances()->select( aa' |
3   aa'.name = self.name and
4   aa'.isassigned = self.allAssignedPermissions() and
5   aa'.resource = self.resource)
6   ->size() = 1
```

```
1 context Action::allAssignedPermissions():Set(Permission) body:
2 self.compActionPlus()->collect(a|a.isassigned)->asSet()
```

Figura 7.5: Invariantes da relação entre *AtomicAction* e *AtomicAction'* no contrato de transformação

7.1.3 Preservação dos invariantes de *role* e permissões

Para garantir que os invariantes de SecureUML+ComponentUML são preservados em AAC+JAAS, precisamos garantir que eles são preservados inicialmente em SecureUML+ComponentUML com as hierarquias linearizadas. Nesta subseção será verificada a relação dos invariantes de SecureUML+ComponentUML com os invariantes do contrato de transformação das relações entre *Role* e *Permission* e na Seção 7.1.4 será verificada a relação de *Action* com *Permission* e *Resource*.

Após a apresentação da preservação dos invariantes de SecureUML+ComponentUML na transformação para SecureUML+ComponentUML com as hierarquia linearizadas será abordado na Seção 7.2 a corretude da transformação de SecureUML+ComponentUML com as hierarquias linearizadas para AAC+JAAS.

Os invariantes relacionados às permissões de SecureUML+ComponentUML não sofrem mudanças, a menos dos nomes das metaclasses, quando for o caso. Estes invariantes são apresentados na Figura 7.6. O invariante *defaultPermissionsAssignedToDefaultRole* garante que a permissão *DefaultPermission* somente está relacionada a papéis que são do tipo *DefaultRole*, ambos representados pelo atributo *default* contendo o valor booleano verdadeiro (*true*). Os invariantes *existsADefaultPermission* e *constraintByTrue* não serão explicados, pois não afetam a transformação para SecureUML+ComponentUML linearizada.

```
1 context Permission
2
3 inv existsADefaultPermission:
4 self.allInstances()->select(p|p.default)->size() = 1
5
6 inv defaultPermissionAssignedToDefaultRole:
7 self.default implies self.givesaccess->forAll(r|r.default)
8
9 inv constraintByTrue:
10 self.default implies self.isconstraintby.body = ''true''
```

Figura 7.6: Invariantes OCL referentes ao papel *DefaultRole*

Com relação aos papéis, já foi abordada a presença de um papel especial e único chamado *DefaultRole* nas instâncias de SecureUML+ComponentUML e que todos os papéis em uma instância SecureUML+ComponentUML devem ter este papel especial como um papel na sua hierarquia. A Figura 7.7 apresenta invariantes do metamodelo SecureUML+ComponentUML referente ao *DefaultRole*. O invariante *existsADefaultRole* garante que existe um único papel especial *DefaultRole* e o invariante *allRolesInheritFromDefaultRole*

garante que todos os papéis tem o papel *DefaultRole* no seu fecho-transitivo de hierarquia de papéis.

```
1 context Role
2
3 inv existsADefaultRole:
4 self.allInstances()->select(r|r.default)->size() = 1
5
6 inv allRolesInheritFromDefaultRole:
7 self.superrolePlus()->exists(r|r.default)
```

Figura 7.7: Invariantes OCL referentes ao papel *DefaultRole* de SecureUML+ComponentUML

Após a transformação para SecureUML+ComponentUML linearizado, esta última conterá um papel especial que será chamado de *DefaultRoleFlattened*, representando o papel especial *DefaultRole* de SecureUML+ComponentUML linearizado. O invariante *existsADefaultRole* de SecureUML+ComponentUML terá um invariante equivalente no contrato de transformação chamado *existsADefaultRoleFlattened* que verifica a existência de *DefaultRoleFlattened*. Em SecureUML+ComponentUML linearizada será mantida a premissa de que todo usuário ao ser autenticado sem papéis específicos estará relacionado ao papel padrão *DefaultRoleFlattened* com acesso a permissão *DefaultPermission*.

O invariante *allRolesInheritFromDefaultRole* não será mais válido, pois os papéis linearizados não terão hierarquia de papéis. Como já mencionado, a hierarquia de papéis em SecureUML+ComponentUML representa que um *subrole* está relacionado as permissões do seu fecho-transitivo de hierarquia de papéis. Sabe-se pelo invariante de SecureUML+ComponentUML *defaultPermissionAssignedToDefaultRole* que o papel *DefaultRole* somente está relacionado a permissão *DefaultPermission*. Devido a linearização, todos os papéis linearizados devem estar relacionados a permissão *DefaultPermission*, o que é garantido pelo invariante *defaultPermissionAssignedToAllRoleFlattened* do contrato de transformação. Tanto o invariante *existsADefaultRoleFlattened*, quanto o invariante *defaultPermissionAssignedToAllRoleFlattened* são apresentados na Figura 7.8.

```
1 context Role'
2
3 inv existsADefaultRoleFlattened:
4 self.allInstances()->select(role'|role'.default)->size() = 1
5
6 inv defaultPermissionAssignedToAllRoleFlattened:
7 self.haspermission->includes(p|p.default)
```

Figura 7.8: Invariantes OCL referentes ao papel *DefaultRole* em SecureUML+ComponentUML linearizado

O último invariante de SecureUML+ComponentUML relativo a papéis, já citado anteriormente, é apresentado na Figura 7.9 com o nome *noCyclesinRoleHierarchy*. Este invariante garante que não ocorra a presença de ciclos na hierarquia de papéis. Este invariante não é válido em SecureUML+ComponentUML linearizada, uma vez que não existem hierarquias de papéis nesta linguagem.

```
1 context Role inv noCyclesinRoleHierarchy:
2 self.superrole->forAll(r|r.superrolePlus()->excludes(self))
```

Figura 7.9: Invariante OCL referente a ciclo na hierarquia de papéis em SecureUML

7.1.4 Preservação dos invariantes sobre as relações entre permissões, ações e recursos pelo contrato de transformação

Nesta subseção será verificada a preservação dos invariantes de SecureUML+ComponentUML sobre a relação entre permissões e ações, entre ações e recursos e entre ações e ações após o processo de linearização. Para cada invariante visitado será apresentada uma explicação do porquê da preservação dos invariantes e da eventual necessidade de se especificar um invariante associado no contrato de transformação.

Conforme apresentado na Seção 7.1.3, os invariantes de SecureUML+ComponentUML sobre permissões não sofrem alterações no contrato de transformação, mas a relação entre as ações e as permissões precisa ser verificada após a linearização. Os invariantes de SecureUML+ComponentUML *existsAPermission* e *overridingDefaultPermission* apresentados na Figura 7.10 atuam sobre ações atômicas e precisam ser revistos.

O invariante de SecureUML+ComponentUML *existsAPermission* especifica que toda ação atômica precisa estar relacionada a pelo menos uma permissão. Este invariante é verificado novamente no contrato de transformação com o nome *existsAPermissionInFlattened* sobre $aa' \in AtomicAction'$, pois a propriedade de que toda ação atômica está relacionada a pelo menos uma permissão deve ser mantida após a linearização.

Em contrapartida, o invariante de SecureUML+ComponentUML *overridingDefaultPermission* verifica que se uma ação atômica possui mais de uma permissão na hierarquia de ações, nenhuma delas seria a permissão *DefaultPermission*. Com a linearização das ações atômicas de SecureUML+ComponentUML, a relação com as permissões devem se manter equivalentes e consequentemente este invariante será mantido e verificado no contrato de transformação com o nome *overridingDefaultPermissionInFlattened*.

O invariante de SecureUML+ComponentUML *nonDefaultPermission* garante que nenhuma ação composta é relacionada a permissão *DefaultPermission*. Entretanto, este invariante perde o sentido já que as ações compostas não fazem parte de SecureUML+ComponentUML linearizada.

```
1 context AtomicAction
2
3 inv existsAPermission:
4 self.allAssignedPermissions()->notEmpty()
5
6 inv overridingDefaultPermission:
7 self.allAssignedPermissions()->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

```
1 context AtomicAction'
2
3 inv existsAPermissionInFlattened:
4 self.isassigned->notEmpty()
5
6 inv overridingDefaultPermissionInFlattened:
7 self.isassigned->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

Figura 7.10: Invariantes OCL referentes ao papel *DefaultRole* em SecureUML+ComponentUML linearizado

Os invariantes de SecureUML+ComponentUML apresentados na Seção A.4 são referentes a correta relação entre ações e os recursos controlados. Os invariantes relacionados a ações compostas perdem o sentido devido a linearização, como por exemplo o invariante *targetsAnEntity* que garante que toda ação do tipo *EntityFullAccess* está relacionado a recursos do tipo *Entity*. Entretanto, as relações entre ações atômicas e recursos devem ser verificadas devido a preservação da especialização de ações atômicas após a linearização.

A especialização de ações atômicas indica a forma de acesso a um tipo de recurso controlado e por isso é preservada. Os invariantes de SecureUML+ComponentUML *targetsAMethod* e o correspondente *flattenedTargetsAMethod* no contrato de transformação para SecureUML+ComponentUML linearizado são apresentados na Figura 7.11. Os demais invariantes da relação entre ações e recursos são apresentados na Seção C.4.3.

```
1 context AtomicExecute inv targetsAMethod:
2 self.resource.oclIsTypeOf(Method)
```

```
1 context AtomicExecute' inv flattenedTargetsAMethod:
2 self.resource.oclIsTypeOf(Method)
```

Figura 7.11: Invariantes OCL referentes a relação entre a ações atômicas em SecureUML+ComponentUML e SecureUML+ComponentUML linearizado

Os invariantes de SecureUML+ComponentUML com o nome *areAccessedBy* verificam se os recursos estão relacionados corretamente a ações relacionadas ao seu tipo, tanto atômicas, quanto compostas. Após a linearização é necessário garantir que os recursos continuam a ser corretamente acessados por ações atômicas. Para o recurso do tipo *Entity* o invariante de SecureUML+ComponentUML *areAccessedBy* é apresentado na Figura 7.12 junto com o seu equivalente *areAccessedByFlattened* em SecureUML+ComponentUML linearizado. No invariante *areAccessedBy* é verificado que o recurso do tipo *Entity* é acessado por cinco ações: *EntityFullAccess*, *EntityRead*, *EntityUpdate*, *AtomicCreate* e *AtomicDelete*. Após a linearização, conforme o invariante *areAccessedByFlattened*, um recurso do tipo *Entity* será acessado somente pelos dois tipos de ações atômicas: *AtomicCreate* e *AtomicDelete*.

```
1 context Entity inv areAccessedBy:
2 self.action->size() = 5 and
3 self.action->exists(a|a.oclIsTypeOf(EntityFullAccess)) and
4 self.action->exists(a|a.oclIsTypeOf(EntityUpdate)) and
5 self.action->exists(a|a.oclIsTypeOf(EntityRead)) and
6 self.action->exists(a|a.oclIsTypeOf(AtomicCreate)) and
7 self.action->exists(a|a.oclIsTypeOf(AtomicDelete))
```

```
1 context Entity inv areAccessedByFlattened:
2 self.action->size() = 3 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicCreate)) and
4 self.action->exists(a|a.oclIsTypeOf(AtomicDelete))
```

Figura 7.12: Invariante OCL referentes a relação entre o recurso *Entity* e as suas ações

Por fim, os invariantes de SecureUML+ComponentUML que verificam a correta relação entre as ações compostas e ações atômicas apresentados na Seção A.5 perdem também a validade devido a linearização das ações atômicas. Um exemplo é o invariante *containsSubactions* apresentado na Figura 7.13 que verifica se as ações compostas do tipo *AttributeFullAccess* estão relacionadas corretamente às suas ações subordinadas que devem ser do tipo *AtomicUpdate* e *AtomicRead*.

```
1 context AttributeFullAccess inv containsSubactions:
2 self.subordinatedactions =
3   self.resource.action->select(a|a.oclIsTypeOf(AtomicUpdate))
4   ->union(self.resource.action->select(a|a.oclIsTypeOf(AtomicRead)))
```

Figura 7.13: Invariante OCL que verifica a relação entre os tipos de ações compostas e ações atômicas

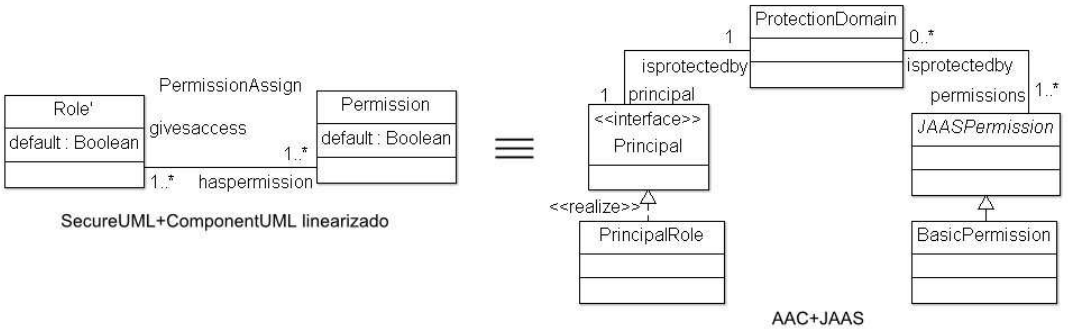

Figura 7.14: Relação de equivalência estrutural do metamodelo SecureUML+ComponentUML com a hierarquia de papéis linearizada e o metamodelo AAC+JAAS

7.2 Transformação para AAC+JAAS

Após a execução da transformação para SecureUML+ComponentUML linearizada é possível executar a transformação para a linguagem de saída AAC+JAAS. Cada papel linearizado $role'$ de SecureUML+ComponentUML será transformado em um elemento $pr \in PrincipalRole$ de AAC+JAAS e cada ação atômica linearizada $aa' \in AtomicAction'$ é transformada num elemento $ja \in JAASAction$ de AAC+JAAS. A Figura 7.14 apresenta as relações entre o metamodelo de SecureUML+ComponentUML com a hierarquia de papéis linearizada e o metamodelo AAC+JAAS.

7.2.1 Transformação de *role* linearizado para AAC+JAAS

Conforme apresentado no Capítulo 6, o metamodelo de transformação relaciona os elementos de papéis e permissões dos dois metamodelos de acordo com o apresentado na Figura 7.15. Inicialmente as permissões $p \in Permission$ de SecureUML+ComponentUML são transformadas em permissões $bp \in BasicPermissions$ de AAC+JAAS, o que é garantido pelo invariante $PermissionIsConnectedToBasicPermission$ do contrato de transformação apresentado na Figura 7.16. Este invariante garante também que as permissões relacionadas tem o mesmo valor para seus metaatributos.

A transformação entre papéis $role' \in Role'$ de SecureUML+ComponentUML com hierarquia linearizada para $pr \in PrincipalRole$ de AAC+JAAS ocorre de forma intuitiva através do metamodelo de transformação, onde pr além de receber o valor dos metaatributos de $role'$ para seus metaatributos, está relacionado a todas as permissões $bp \in BasicPermission$ que são conectadas a todas as permissões $p \in Permission$ relacionadas a $role'$. Esta afirmação é garantida pelo invariante $RoleFlattenedIsConnectedTo-$

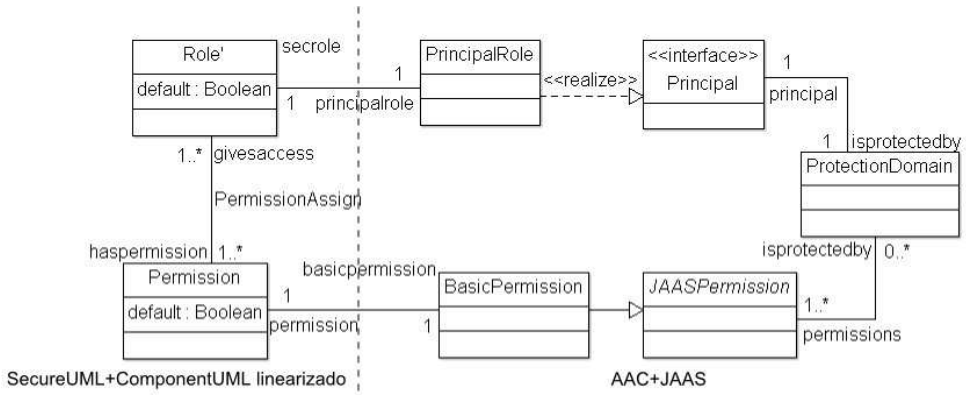

Figura 7.15: Relação entre papéis e permissões no metamodelo de transformação

```
1 context BasicPermission inv PermissionIsConnectedToBasicPermission:
2 self.name = self.permission.name and self.default = self.permission.default
```

Figura 7.16: Invariante OCL da relação entre permissões no contrato de transformação

PrincipalRole do contrato de transformação, apresentado na Figura 7.17. Este invariante verifica se cada $pr \in PrincipalRole$ de AAC+JAAS está relacionado a um $role' \in Role'$. São verificados também se os valores dos metaatributos *name* e *default* são iguais entre pr e $role'$, onde todas as permissões bp relacionadas a pr através de seu domínio de proteção pd estão relacionadas a todas a permissões p de $role'$.

```
1 context PrincipalRole inv RoleFlattenedIsConnectedToPrincipalRole:
2 self.name = self.secrole.name and
3   self.default = self.secrole.default and
4   self.isprotectedby.permissions.permission->asSet() = self.secrole.haspermission->asSet()
```

Figura 7.17: Invariante OCL da relação entre papéis no contrato de transformação

7.2.2 Transformação de ações linearizadas para AAC+JAAS

Para a transformação de ações, $aa' \in AtomicAction'$ de SecureUML+ComponentUML linearizado será transformado em um elemento $ja \in JAASAction$ em AAC+JAAS. A Figura 7.18 apresenta a equivalência estrutural entre o metamodelo de SecureUML+-ComponentUML com a hierarquia de ações linearizada e o metamodelo AAC+JAAS.

Conforme apresentado no Capítulo 6, o metamodelo de transformação relaciona os elementos de ações e permissões dos dois metamodelos de acordo com o apresentado na Figura 7.19.

A transformação entre ações $aa' \in AtomicAction'$ de SecureUML+ComponentUML linearizado para $ja \in JAASAction$ em AAC+JAAS ocorre de forma intuitiva através

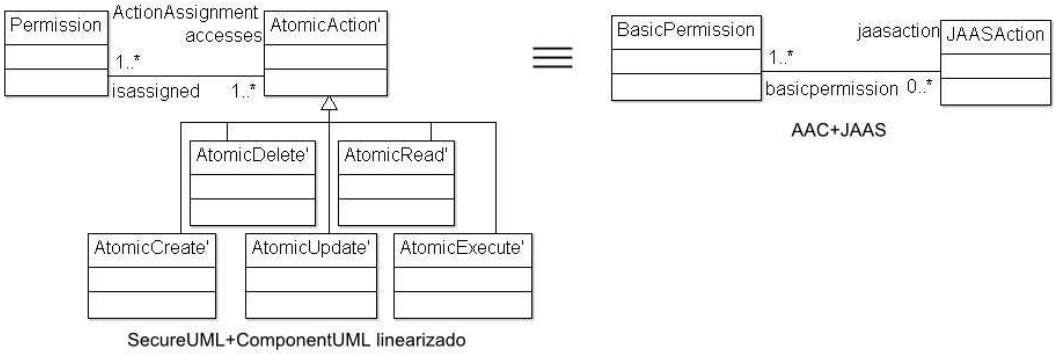

Figura 7.18: Relação de equivalência estrutural do metamodelo SecureUML+ComponentUML com a hierarquia de ações linearizada e o metamodelo AAC+JAAS

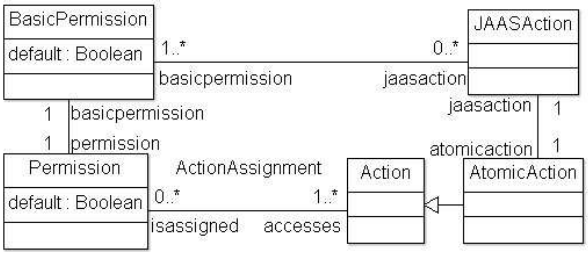

Figura 7.19: Relação entre ações e permissões no metamodelo de transformação

do metamodelo de transformação, onde *ja* está relacionada a todas as permissões $bp \in BasicPermission$ que são conectadas a todas as permissões $p \in Permission$ relacionadas a $role'$. Esta afirmação é garantida pelo invariante *AtomicActionFlattenedIsConnectedToJAASAction* do contrato de transformação apresentado na Figura 7.20. Este invariante verifica que cada $ja \in JAASAction$ de AAC+JAAS está relacionado a um $aa' \in AtomicAction'$, onde todas as permissões *bp* relacionadas a *ja* estão relacionadas a todas a permissões p de aa'.

```
1 context JAASAction inv AtomicActionFlattenedIsConnectedToJAASAction:
2 self.basicpermission.permission = self.atomicaction.isassigned
```

Figura 7.20: Invariante OCL da relação entre ações no contrato de transformação

No Capítulo 6 foi apresentada a relação de ações e recursos e nas subseções será apresentado que esta relação será garantida em AAC+JAAS de acordo com a sua estrutura.

7.2.3 Preservação dos invariantes sobre a relação entre papéis e permissões pelo contrato de transformação

Nesta subseção é verificada a preservação dos invariantes da transformação de SecureUML+ComponentUML linearizado para AAC+JAAS do ponto de vista da relação entre

papéis e permissões. Para cada invariante será apresentada uma explicação do porquê da preservação dos invariantes e da possível necessidade de se utilizar um invariante no contrato de transformação. Na linguagem AAC+JAAS existirá um elemento equivalente a *DefaultRoleFlattened* de SecureUML+ComponentUML linearizado e outro equivalente a *DefaultPermission*. Um usuário ao se autenticar no sistema em AAC+JAAS será automaticamente relacionado ao principal *DefaultPrincipalRole* e pelos invariantes garante-se que o *DefaultPrincipalRole* está relacionado a permissão *DefaultBasicPermission*.

Primeiramente, os invariantes de SecureUML+ComponentUML linearizado apresentados são relacionados às suas permissões. Estes invariantes terão equivalentes no contrato de transformação relacionados a linguagem AAC+JAAS.

O invariante *existsADefaultPermission* verifica que no modelo de SecureUML+ComponentUML linearizado haverá apenas uma única permissão do tipo *DefaultPermission*, definida pelo valor verdadeiro de seu atributo *default*. Para esta permissão, haverá uma permissão equivalente em AAC+JAAS também contendo o atributo *default* com valor verdadeiro. O invariante *existsADefaultBasicPermission* do contrato de transformação garante que existirá uma permissão única que tenha o seu atributo *default* com valor verdadeiro, a qual será chamado neste trabalho por *DefaultBasicPermission*. Os invariantes *existsADefaultPermission* de SecureUML+ComponentUML linearizado e *existsADefaultBasicPermission* de AAC+JAAS são apresentados na Figura 7.21.

```
1 context Permission inv existsADefaultPermission:
2 self.allInstances()->select(p|p.default)->size() = 1
3
4 context BasicPermission inv existsADefaultBasicPermission:
5 self.allInstances()->select(p|p.default)->size() = 1
```

Figura 7.21: Invariantes OCL do contrato de transformação da equivalência entre *DefaultPermission* e *DefaultBasicPermission*

O invariante *defaultPermissionsAssignedToDefaultRole* de SecureUML+ComponentUML linearizado garante que a permissão *DefaultPermission* somente está relacionada a papéis que são do tipo *DefaultRoleFlattend*, definido pelo valor verdadeiro de seu atributo *default*. Em AAC+JAAS existe também a figura de um papel padrão, chamado de *DefaultPrincipalRole* que está relacionado a permissão *DefaultBasicPermission* de AAC+JAAS. O invariante *defaultBasicPermissionAssignedToDefaultPrincipalRole* de AAC+JASS garante que o elemento *DefaultBasicPermission* estará relacionado ao elemento *DefaultPrincipalRole*. Os invariantes *defaultPermissionAssignedToDefaultRole* de SecureUML+ComponentUML linearizado e *defaultBasicPermissionAssignedTo-*

DefaultPrincipalRole de AAC+JAAS são apresentados na Figura 7.22.

```
1 context Permission inv defaultPermissionAssignedToDefaultRole:
2 self.default implies self.givesaccess->forAll(role'|role'.default)
3
4 context BasicPermission inv defaultBasicPermissionAssignedToDefaultPrincipalRole:
5 self.default implies self.isprotectedby.principal->forAll(pr|pr.default)
```

Figura 7.22: Invariantes OCL do contrato de transformação da equivalência entre a relação dos papéis padrões e das permissões padrões

O invariante *constraintByTrue* de SecureUML+ComponentUML linearizado garante que o elemento do tipo *AuthorizationConstraint* que está relacionado a permissão *DefaultPermission* terá seu corpo com o valor "true". Isso significa que ações relacionadas à *DefaultPermission* podem ser realizadas por qualquer usuário. Em AAC+JAAS também existe a possibilidade de serem incluídas restrições aos elementos que representam permissões através de elementos do tipo *BasicPermissionConstraint*. Por isso, a permissão *DefaultBasicPermission* é relacionada a uma restrição sempre válida, para que as ações a ela relacionada sejam executadas por qualquer usuário, assim como em SecureUML+ComponentUML linearizado. O invariante *basicpermissionconstraintByTrue* de AAC+JAAS garante esta propriedade e é apresentado na Figura 7.23 juntamente com o invariante *constraintByTrue* de SecureUML+ComponentUML linearizado.

```
1 context Permission inv constraintByTrue:
2 self.default implies self.isconstraintby.body = ''true''
3
4 context BasicPermission inv basicpermissionconstraintByTrue:
5 self.default implies self.basicpermissionconstraint.body = ''true''
```

Figura 7.23: Invariantes OCL do contrato de transformação da equivalência entre as restrições padrões de autorização

Para a equivalência entre os papéis em SecureUML+ComponentUML linearizada e AAC+JAAS é necessário verificar a equivalência dos invariantes de papéis nas duas linguagens. Em SecureUML+ComponentUML linearizada, o papel *DefaultRoleFlattend* é o papel ao qual todos os usuários estarão relacionados ao serem autenticados no sistema sem nenhum papel específico onde sua existência é garantida pelo invariante *existsADefaultRoleFlattened*. Em AAC+JAAS também haverá um elemento representando um papel do tipo *PrincipalRole* que será chamado de *DefaultPrincipalRole*, sendo definido pelo valor verdadeiro contido em seu atributo *default* e ao qual todos os usuários em AAC+JAAS ao se autenticarem estarão relacionados. A existência do elemento *DefaultPrincipalRole* em AAC+JAAS é garantido pelo invariante *existsADefaultPrincipalRole* apresentado na

Figura 7.24 juntamente com o invariante *existsADefaultRoleFlattened* de SecureUML+-ComponentUML linearizada.

```
1 context Role' inv existsADefaultRoleFlattened:
2 self.allInstances()->select(r|r.default)->size() = 1
3
4 context PrincipalRole inv existsADefaultPrincipalRole:
5 self.allInstances()->select(pr|pr.default)->size() = 1
```

Figura 7.24: Invariantes OCL referentes ao papel padrão em SecureUML+ComponentUML linearizado e AAC+JAAS

Após linearização da hierarquia de papéis, a permissão *DefaultPermission* foi relacionada a todos os papéis linearizados, o que é garantido pelo invariante *defaultPermissionAssignedToAllRoleFlattened* de SecureUML+ComponentUML linearizado. Em AAC+JAAS, mantendo a equivalência entre os invariantes, todos os papéis do tipo *PrincipalRole* serão relacionados a permissão *DefaultBasicPermission*. O invariante *defaultBasicPermissionAssignedToAllPrincipalRole* no contrato de transformação, garante que todos os elementos que representam papéis através do tipo *PrincipalRole* em AAC+JAAS estarão relacionados a permissão *DefaultBasicPermission*. Este invariante de AAC+JAAS, juntamente ao invariante *defaultPermissionAssignedToAllRoleFlattened* de SecureUML+ComponentUML linearizado são apresentados na Figura 7.25.

```
1 context Role' inv defaultPermissionAssignedToAllRoleFlattened:
2 self.haspermission->includes(p|p.default)
3
4 context PrincipalRole inv defaultBasicPermissionAssignedToAllPrincipalRole:
5 self.isprotectedby.permissions->includes(bp|bp.default)
```

Figura 7.25: Invariantes OCL referentes a relação entre papéis e a permissão padrão em SecureUML+ComponentUML linearizado e AAC+JAAS

7.2.4 Preservação dos invariantes sobre a relação entre permissões, ações e recursos pelo contrato de transformação

Nesta subseção será verificada a preservação dos invariantes de SecureUML+ComponentUML linearizada em AAC+JAAS sobre a relação entre permissões e ações e entre ações e recursos. Para cada invariante visitado será apresentada uma explicação do porquê da preservação dos invariantes e da possível necessidade de se utilizar um invariante no contrato de transformação.

O invariante de SecureUML+ComponentUML linearizado *existsAPermission* especifica que toda ação atômica precisa estar relacionada a pelo menos uma permissão o que

garante que cada recurso terá ao menos uma permissão para cada tipo de acesso das ações. Seu equivalente em AAC+JAAS é representado pelo invariante *existsABasicPermission*, o qual garante que toda ação do tipo *JAASAction* estará relacionada a pelo menos uma permissão do tipo *BasicPermission*, preservando a relação entre as permissões e ações em AAC+JAAS. Os invariantes *existsAPermission* de SecureUML+ComponentUML linearizado e *existsABasicPermission* de AAC+JAAS são apresentados na Figura 7.26.

```
1 context AtomicAction' inv existsAPermission:
2 self.allAssignedPermission()->notEmpty()
3
4 context JAASAction inv existsABasicPermission:
5 self.basicpermission->notEmpty()
```

Figura 7.26: Invariantes OCL referentes ao papel *DefaultRole* em SecureUML+ComponentUML linearizado

Para garantir que nenhuma ação relacionada a mais de uma permissão, poderá estar relacionada a permissão *DefaultPermission* existe o invariante *overridingDefaultPermission* de SecureUML+ComponentUML linearizado. Esta verificação será preservada em AAC+JAAS com o invariante *overridingDefaultBasicPermission* que garante que nenhuma ação do tipo *JAASAction* que estiver relacionada a mais de uma permissão do tipo *BasicPermission* poderá estar relacionada a permissão *DefaultBasicPermission*. Os invariantes *overridingDefaultPermission* de SecureUML+ComponentUML linearizado e *overridingDefaultBasicPermission* de AAC+JAAS são apresentados na Figura 7.27.

```
1 context AtomicAction' inv overridingDefaultPermission:
2 self.isassigned->forAll(p1, p2| p1<>p2 implies not(p1.default))
3
4 context JAASAction inv overridingDefaultBasicPermission:
5 self.basicpermission->forAll(bp1, bp2| bp1<>bp2 implies not(bp1.default))
```

Figura 7.27: Invariantes OCL referentes ao papel *DefaultRole* em SecureUML+ComponentUML linearizado

Os invariantes de SecureUML+ComponentUML linearizado apresentados na Seção C.4.3 são referentes a correta relação entre ações e os recursos. Esta relação é intrínseca a estrutura de AAC+JAAS, pois conforme apresentado no Capítulo 6, as ações de AAC+JAAS estão relacionadas a métodos que acessam diretamente os recursos controlados através de associações nesta linguagem. Para mostrar a preservação do acesso a cada recurso serão apresentados os invariantes *areAccessedByFlattened* e *flattenedtargets* de SecureUML+ComponentUML linearizado e posteriormente será apresentada a estrutura que representa o acesso em AAC+JAAS.

```
1 context AtomicUpdate' inv flattenedtargets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context AtomicRead' inv flattenedtargets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context Attribute inv areAccessedByFlattened:
2 self.action->size() = 2 and
3   self.action->exists(a|a.oclIsTypeOf(AtomicRead')) and
4   self.action->exists(a|a.oclIsTypeOf(AtomicUpdate'))
```

Figura 7.28: Invariantes OCL referentes a relação entre a ações e recurso de atributo em SecureUML+ComponentUML

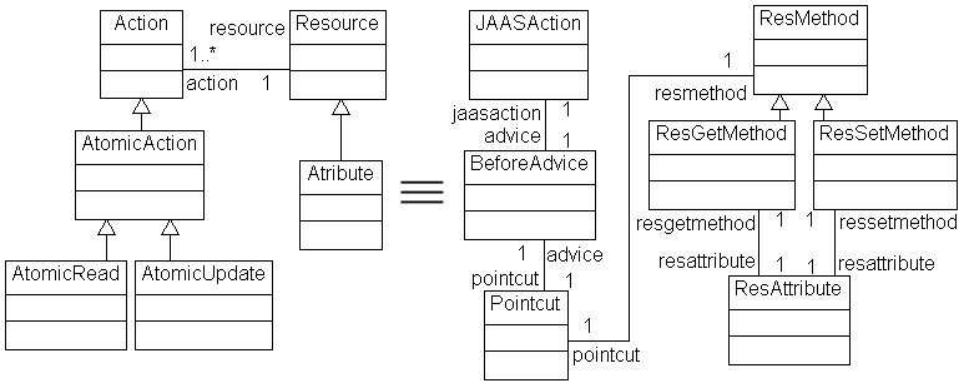

Figura 7.29: Comparação da relação de acesso a um atributo em SecureUML+ComponentUML linearizado e AAC+JAAS

O correto acesso a atributos de SecureUML+ComponentUML linearizado através das ações é garantido pelos invariantes *areAccessedByFlattened* e *flattenedtargets* apresentados na Figura 7.28. O invariante *areAccessedByFlattened* que atua sobre atributos garante que todo atributo será acessado pelas ações atômicas dos tipos *AtomicRead'* e *AtomicUpdate'*. Para as ações atômicas dos tipos *AtomicUpdate'* e *AtomicRead'* existem os invariantes *flattenedtargets* que garantem que estas ações ou acessam um recurso representado por um atributo ou por uma ponta de associação. Na Figura 7.29 é apresentada a equivalência estrutural dos acessos a um atributo, tanto em SecureUML+ComponentUML com hierarquia linearizada, quanto em AAC+JAAS. Em AAC+JAAS, o acesso a um atributo ocorre diretamente pelos métodos *ResGetMethod* e *ResSetMethod*. Estes métodos são relacionados a ação *JAASAction* através dos elementos *Pointcut* e *BeforeAdvice*.

O correto acesso a pontas de associação de SecureUML+ComponentUML linearizada através das ações ocorre de forma análoga ao acesso aos atributos e é garantido pelos invariantes *areAccessedByFlattened* e *flattenedtargets* apresentados na Figura 7.30. O invariante *areAccessedByFlattened* que atua sobre pontas de associação garante que toda

```
1 context AtomicUpdate' inv flattenedtargets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context AtomicRead' inv flattenedtargets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context Associationend inv areAccessedByFlattened:
2 self.action->size() = 2 and
3   self.action->exists(a|a.oclIsTypeOf(AtomicRead')) and
4   self.action->exists(a|a.oclIsTypeOf(AtomicUpdate'))
```

Figura 7.30: Invariantes OCL referentes a relação entre a ações e recurso de atributo em SecureUML+ComponentUML

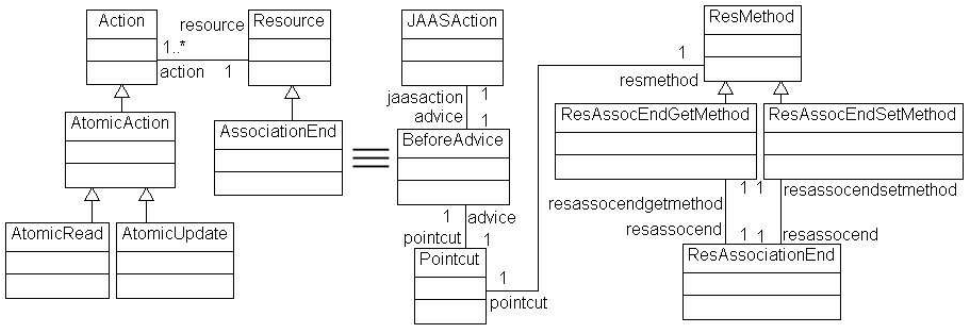

Figura 7.31: Comparação da relação de acesso a uma ponta de associação em SecureUML+ComponentUML linearizada e AAC+JAAS

ponta de associação será acessada pelas ações atômicas *AtomicRead'* e *AtomicUpdate'*. Para as ações atômicas *AtomicUpdate'* e *AtomicRead'* existem os invariantes *flattenedtargets* que garantem que estas ações ou acessam um recurso representado por um atributo ou por uma ponta de associação. Na Figura 7.31 é apresentada a equivalência estrutural dos acessos às pontas de associação. Em AAC+JAAS, o acesso a uma ponta de associação ocorre diretamente pelos métodos *ResAssocEndGetMethod* e *ResAssocEndSetMethod*. Estes métodos são relacionados a ação *JAASAction* através dos elementos *Pointcut* e *BeforeAdvice*.

```
1 context AtomicExecute' inv targetsAMethod:
2 self.resource.oclIsTypeOf(Method)
```

```
1 context Method inv areAccessedBy:
2 self.action->size() = 1 and self.action->exists(a|a.oclIsTypeOf(AtomicExecute'))
```

Figura 7.32: Invariantes OCL referentes a relação entre a ações e recurso de atributo em SecureUML+ComponentUML

O correto acesso aos métodos de SecureUML+ComponentUML linearizado através

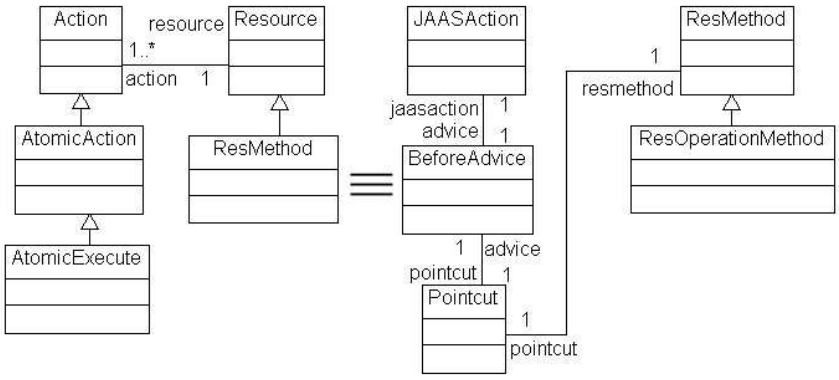

Figura 7.33: Comparação da relação de acesso a um método em SecureUML+ComponentUML linearizado e AAC+JAAS

das ações é garantido pelos invariantes *areAccessedByFlattened* e *flattenedtargetsAMethod* apresentados na Figura 7.32. O invariante *areAccessedByFlattened* que atua sobre métodos garante que todo método será acessado pela ação atômica *AtomicExecute'*. Para a ação atômica *AtomicExecute'* existe o invariante *flattenedtargetsAMethod* que garante que esta ação somente acessa um recurso representado por um método. Na Figura 7.33 é apresentada a equivalência estrutural dos acessos. Em AAC+JAAS, o acesso a um método ocorre diretamente no elemento *ResOperationMethod*. Este método é relacionado a ação *JAASAction* através dos elementos *Pointcut* e *BeforeAdvice*.

O acesso a entidades ocorre através das ações atômicas *AtomicCreate'* e *AtomicDelete'* conforme descrito pelos invariantes da Figura 7.34. Estes acessos de criação e deleção não ocorrem em AAC+JAAS, conforme já descrito anteriormente, o que gera a necessidade que as classes controladas tenham métodos específicos para criação e deleção de entidades.

```
1 context AtomicCreate' inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AtomicDelete' inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context Entity inv areAccessedByFlattened:
2 self.action->size() = 2 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicCreate')) and
4 self.action->exists(a|a.oclIsTypeOf(AtomicDelete'))
```

Figura 7.34: Invariantes OCL referentes a relação entre a ações e recurso de entidade em SecureUML+ComponentUML

7.3 Discussão

Neste capítulo foi apresentada uma argumentação da corretude da transformação entre as linguagens SecureUML+ComponentUML e AAC+JAAS. Esta argumentação ocorreu sobre a técnica de contratos de transformação e utilizou um passo de linearização da hierarquia de papéis e de ações, tornando o processo de transformação simples e intuitivo.

A argumentação se baseou em verificar que as propriedades de tipos de uma instância da linguagem SecureUML+ComponentUML são mantidas na linguagem AAC+JAAS. As verificações foram feitas através da criação de invariantes na linguagem AAC+JAAS garantindo as mesmas propriedades da linguagem SecureUML+ComponentUML, exprimidas nesta última através da estrutura da linguagem e dos invariantes definidos em seu metamodelo.

Ao invés de utilizar o passo de linearização, a transformação poderia ser executada em apenas um passo. Entretanto, os invariantes do contrato de transformação que representam as propriedades de SecureUML+ComponentUML seriam mais complexos devido as diferenças na definição de papéis e ações nas linguagens envolvidas na transformação. Esta complexidade poderia ocasionar erros na implementação da transformação.

A utilização da técnica de contratos de transformação se mostrou eficiente na especificação e identificação de erros nos vários níveis de usuários beneficiados, entretanto, esta técnica é centrada nas definições criadas pelos designers de linguagens de modelagem e, caso seja mal especificada, pode gerar modelos incorretos. Além disso, o escopo adotado com esta técnica visa a verificação de instâncias na sintaxe abstrata e a especificação da transformação entre as sintaxes abstratas de linguagens de modelagem, permitindo que erros ocorram na sintaxe concreta devido a transformação entre a sintaxe abstrata e concreta das linguagens envolvidas na transformação. Como solução, poderiam ser utilizadas outras técnicas de validação de modelos juntamente com a técnica de contratos de transformação.

Por fim, é importante ressaltar que além das validações empíricas realizadas com os estudos de caso apresentados no Capítulo 8, argumentou-se no presente capítulo sobre a equivalência de modelos SecureUML e AAC+JAAS de maneira complementar. Apesar de informal, foi apresentada a preservação das características estruturais de modelos nestas duas linguagens especificas envolvidas na transformação. Propõe-se como um trabalho futuro a formalização da transformação entre SecureUML e AAC+JAAS.

Capítulo 8

Implementação da transformação de SecureUML+ComponentUML para AAC+JAAS

A transformação especificada pelo contrato de transformação entre SecureUML+ComponentUML e AAC+JAAS foi implementada na ferramenta SecureUMLToAACJAAS. Esta ferramenta recebe como entrada um modelo representando uma política de segurança SecureUML+ComponentUML e gera como saída código em AspectJ e JAAS.

Este capítulo apresenta a implementação da ferramenta como aplicação do padrão de projeto apresentado no Capítulo 2 e apresenta como o código gerado por ela é utilizado em um sistema controlado. Por fim, são apresentados alguns estudos de caso de políticas de segurança SecureUML+ComponentUML e o código gerado pela ferramenta.

8.1 Implementação

A ferramenta SecureUMLToAACJAAS é, essencialmente, uma implementação do contrato de transformação descrito no Capítulo 6. Esta ferramenta foi criada a partir da utilização do padrão de projeto definido para contratos de transformação apresentado na Seção 2.3. A Figura 8.1 apresenta a forma como o padrão foi aplicado na implementação da ferramenta.

Para representar os domínios das linguagens definidas pelo contrato de transformação, a classe abstrata *Domain* foi estendida para conter a definição de cada linguagem de modelagem. Para a validação de cada domínio, foram criadas referências para validadores representados por classes que implementam a interface *IValidator*. A classe *SecureUML-*

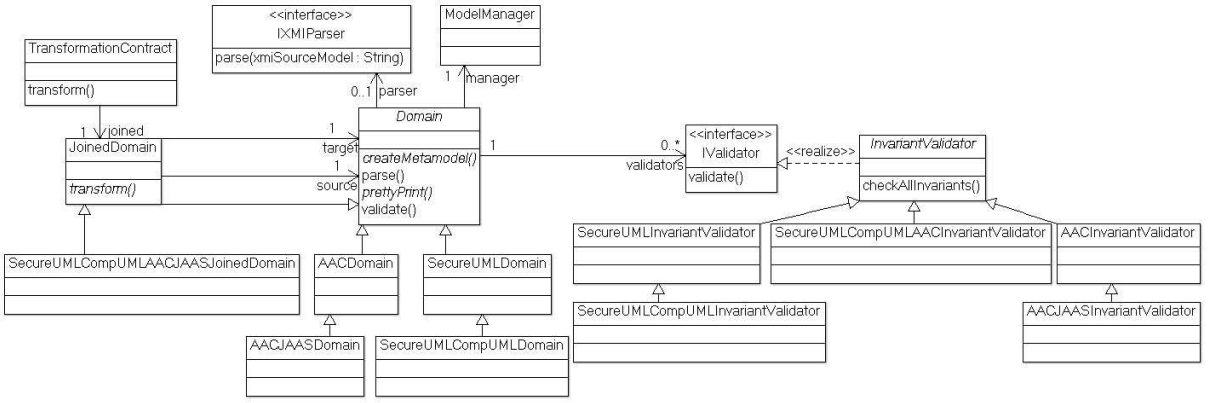

Figura 8.1: Extensão da arquitetura de transformação na ferramenta SecureUMLToAAC-JAAS

Domain estende a classe de domínio *Domain* e representa a linguagem SecureUML. A classe *SecureUMLCompUMLDomain* estende a classe *SecureUMLDomain* e representa a linguagem SecureUML+ComponentUML. A classe *AACDomain* também estende a classe de domínio *Domain* e representa a linguagem AAC. A classe *AACJAASDomain* estende a classe *AACDomain* e representa a linguagem AAC+JAAS, a linguagem de destino da transformação. Note que a noção de extensão conservativa de linguagens é capturada pela relação de herança entre as classes que estendem *Domain*.

A classe *JoinedDomain* representa a linguagem associada a um metamodelo de transformação. Este domínio combinado conterá a definição do metamodelo de transformação, sendo este o responsável pela adição das relações nos metamodelos de entrada e saída, e pela aplicação da transformação, através da chamada ao método *transform()*. A classe *SecureUMLCompUMLAACJoinedDomain* representa o domínio composto da união disjunta dos domínios SecureUML+ComponentUML e AAC+JAAS. Assim como os domínios representando as linguagens de modelagem, o domínio combinado terá validadores para garantir a corretude das instâncias geradas nestes domínios.

Neste trabalho foram aplicadas validações OCL. O padrão de projeto de contratos de transformação permite, no entanto, a conexão da implementação de um transformador de modelos em outros validadores como, por exemplo, um validador de consistência de modelos, descrito em (BRAGA; HÆUSLER, 2010).

A classe *InvariantValidator* implementa a interface de validação *IValidator* fornecendo métodos para utilização de um validador para invariantes OCL. Nesta classe foi utilizado o EOS (CLAVEL *et al.*, 2008a), um interpretador OCL que tem a capacidade de criar diagramas de classes e objetos em memória e executar sobre eles as verificações e operações OCL. Todas as validações serão executadas através de invariantes OCL e

dessa forma todas as classes de validação estenderão a classe *InvariantValidator*. A classe *SecureUMLCompUMLInvariantValidator* estende a classe *SecureUMLInvariantValidator* e representa o validador do domínio de SecureUML+ComponentUML. Esta classe conterá todos os invariantes que garantem que uma instância está em conformidade com SecureUML+ComponentUML. A classe *AACJAASInvariantValidator* estende a classe *AACInvariantValidator* e representa o validador do domínio de AAC+JAAS. A classe *SecureUMLCompUMLAACJAASInvariantValidator* representa o validador do domínio combinado e conterá os invariantes do metamodelo de transformação e do contrato de transformação.

Com relação aos domínios, a sua estrutura deve ser composta da estrutura do metamodelo que o define. A representação dos metamodelos em cada domínio é feita através de classes que contém os mesmos atributos e relacionamentos contidos no metamodelo dos domínios. Por exemplo, a classe *SecureUMLCompUMLDomain* está relacionada a classes que representam todos os tipos contidos no metamodelo de SecureUML+ComponentUML. A classe que representa o domínio combinado entre SecureUML+ComponentUML e AAC+JAAS terá acesso a todas as classes do metamodelo de transformação através das classes *SecureUMLCompUMLDomain* e *AACJAASDomain*.

O processo de transformação com o contrato de transformação entre SecureUML+ComponentUML e AAC+JAAS seguirá os mesmos passos para os domínios SecureUML+ComponentUML e AAC+JAAS que os passos apresentados de forma genérica no diagrama de mensagens da Figura 2.22 da Seção 2.3.

Durante o processo de *parsing* é gerada a instância do metamodelo de SecureUML+ComponentUML. Esta instância será representada no transformador por objetos das classes relacionadas ao domínio de SecureUML+ComponentUML. Para efetuar as verificações OCL, é utilizado o interpretador OCL que terá uma replicação da instância do metamodelo gerada em memória. O interpretador OCL é utilizado pelo domínio de SecureUML+ComponentUML, através de seu validador, para verificar os invariantes do metamodelo correspondente.

A transformação definida pelo domínio combinado gerará a instância do metamodelo de AAC+JAAS e a relacionará com a instância de SecureUML+ComponentUML. Este passo ocorrerá novamente em forma de objetos e da instância replicada no interpretador OCL. O interpretador OCL é então utilizado pelo domínio combinado, através de seu validador, para verificar os invariantes do metamodelo correspondente. Com a geração da instância de AAC+JAAS, o domínio correspondente pode efetuar as verificações dos

invariantes e disparar o processo de *pretty printing*.

O processo de *pretty printing* é executado a partir do método *prettyprint()* de alguns tipos do domínio AAC+JAAS que efetuam a geração do código através do percorrimento de uma árvore. Por exemplo, o método *prettyprint()* de um elemento do tipo *ResClass* executa em cascata a geração de seus atributos, métodos e pontas de associação. Estes por sua vez, gerarão as ações a eles atreladas.

8.1.1 Processo de transformação e utilização do código gerado

O processo de transformação é antecedido pela criação dos modelos através de uma ferramenta para construção de modelos UML com suporte a criação de perfis. Um exemplo deste tipo de ferramenta é a ArgoESS (The UCMaude Group, 2009), baseada na ferramenta ArgoUML. ArgoESS possui o perfil UML de SecureUML+ComponentUML pré-carregado e utiliza a biblioteca EOS para efetuar consultas e verificações OCL a medida que os modelos são construídos.

Após o término da construção do modelo e das verificações que o Analista de Segurança considerar necessárias, um arquivo XMI[1] deve ser gerado através da funcionalidade de exportação da ferramenta ArgoESS. O transformador fornece atualmente suporte para o XMI produzido pela ferramenta ArgoESS.

A ferramenta SecureUMLToAACJAAS é executada e recebe como parâmetro o caminho do arquivo XMI. A partir deste ponto, são executados os passo da transformação do contrato de transformação. Caso algum invariante falhe em algum dos passos, o processo é interrompido e uma mensagem é exibida ao usuário.

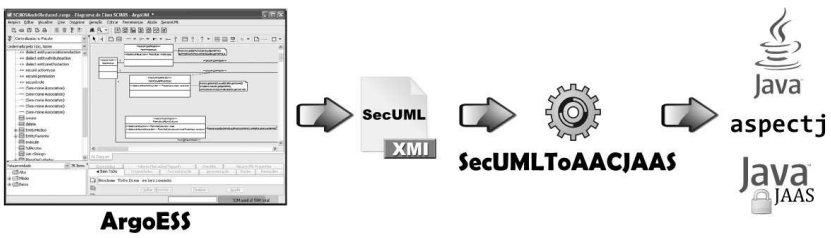

Figura 8.2: Processo de transformação

O resultado da transformação é um conjunto de códigos fonte e arquivos de configuração que representam uma instância da sintaxe concreta de AAC+JAAS. Para cada classe

[1] O formato XMI é um formato definido pela OMG baseado no formato XML, que descreve modelos UML ou modelos representados por perfis UML.

controlada *ResClass* será gerado um pacote de mesmo nome que recebe um arquivo de código Java com uma classe abstrata e um arquivo AspectJ contendo o código do aspecto que irá controlar o acesso à classe e aos elementos que a compõe. Para as permissões serão geradas classes Java que estendem a classe *BasicPermission* no pacote denominado *permission*. Para as ações privilegiadas de acesso aos recursos, serão geradas classes que implementam a interface *PrivilegedAction* no pacote *action*. Além disso, dois arquivos de configuração são gerados: *jaas.policy*, que contém a política de segurança JAAS relacionando os papéis e as permissões, e *jaas.config*, que contém a relação de contextos de autenticação suportados pela transformação. A Figura 8.3 apresenta a estrutura gerada para o modelo de segurança SecureUML+ComponentUML do gerenciador de arquivos da Figura 3.1.

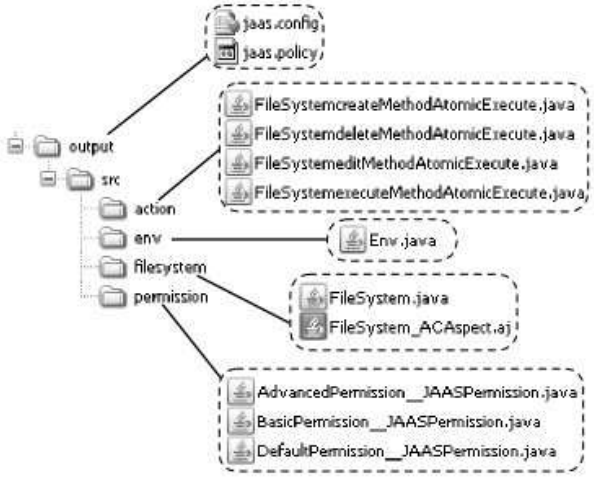

Figura 8.3: Estrutura de pastas e arquivos gerados pelo transformador para o exemplo do gerenciador de arquivos

Após a transformação, os desenvolvedores podem utilizar o código gerado em seu sistema copiando os arquivos de classes abstratas e aspectos e referenciando-os no código de seu projeto. As classes abstratas devem ser estendidas por novas classes ou classes já existentes na parte funcional do sistema a fim de representar os recursos controlados. Neste processo de extensão das classes é necessário verificar a necessidade de que sejam renomeados os pacotes das classes no sistema final. Por fim, todo o projeto deve ser recompilado utilizando o compilador de AspectJ. Note que não é necessário alterar cada método ou ação controlada de forma a garantir as pré-condições de controle de acesso a um método. Todo o código de controle de acesso está encapsulado dentro do aspecto e sua execução é automática. A Figura 8.4 apresenta a especialização da classe abstrata *FileSystem* do gerenciador de arquivos apresentado no Capítulo 5.

Com relação a autenticação, os elementos do tipo *Principal* relacionados a cada usuá-

```
1 public class FileSystemConcrete extends FileSystem {
2    @Override
3    public void create(String filename) {
4       File f = new File(filename);
5       f.createNewFile();
6    }
7    @Override
8    public void execute(String filename) {
9       Runtime r = Runtime.getRuntime();
10      r.exec(filename);
11   }
12 }
```

Figura 8.4: Classe especializada do gerenciador de arquivos

rio são diferentes para cada plataforma de autenticação devido a uma restrição do JAAS. Por exemplo, considerando um usuário no sistema NT relacionado a um grupo chamado *Teste* e um usuário no Unix relacionado a um grupo *Teste*. No sistema NT, o elemento do tipo *Subject* gerado na autenticação de JAAS deste usuário estará relacionado a um *Principal* do tipo *NTSidGroupPrincipal* contendo o código de identificação do grupo *Teste* no sistema NT (por exemplo, S-1-5-32-544). Enquanto que no sistema Unix, *Subject* estaria relacionado a um *Principal* do tipo *UnixNumericGroupPrincipal* contendo o código de identificação do grupo *Teste* no sistema Unix (por exemplo, 105). Para tornar o código gerado mais genérico, foi criada uma biblioteca chamada *JAASLib*, que cria uma abstração sobre os elementos do tipo *Principal* de JAAS fornecendo suporte para alguns módulos de autenticação. A biblioteca *JAASLib* será tema da Seção 8.1.2.

Para autenticação no JAAS é utilizada a classe *AuthUtil* da biblioteca *JAASLib*, que receberá o nome do arquivo de políticas, o nome do arquivo de configuração de autenticação e o nome do contexto de *login* a ser utilizado. A Figura 8.5 apresenta o código utilizado no inicio da execução do gerenciador de arquivos. O arquivo de políticas de segurança *jaas.policy* e de configuração de autenticação *jaas.config* são gerados pelo transformador, onde este último por não fazer parte da transformação é gerado de forma genérica, contendo um contexto de *login* chamado *JAASSystem*.

A Figura 8.6 apresenta o arquivo genérico de autenticação gerado pela ferramenta SecureUMLToAACJAAS. Nele é definido um contexto de *login* com nome *JAASSystem* e são incluídas três opções de módulos de *login*. O primeiro representa um módulo experimental que utiliza arquivos XML para autenticação. O segundo representa o módulo de autenticação de sistemas Unix. O último módulo de autenticação é o módulo de autenticação nativo de sistemas NT. Para utilizar o arquivo gerado é necessário que o desenvolvedor retire o comentário de uma das linhas.

```
1 public class Main {
2  public static void main(String[] args) throws Exception {
3    // try to authenticate a user
4    LoginContext lc = AuthUtil
5       .authenticateUser("jaas.policy", "jaas.config", "JAASSystem");
6    // gets its subject
7    Subject subject = lc.getSubject();
8    new Env(new SubjectJAAS(subject));
9  }
10 //outros métodos da inicialização da aplicação
11 }
```

Figura 8.5: Classe de inicialização da aplicação com a execução da autenticação por módulos de *login*

```
1 JAASSystem{/** Configuração de login para a aplicação JAAS FileSystem **/
2 //br.uff.mda.jaas.security.authentication.loginmodule.XMLLoginModule
    required;
3 //com.sun.security.auth.module.UnixLoginModule required;
4 //com.sun.security.auth.module.NTLoginModule required;
5 };
```

Figura 8.6: Arquivo de configurações do JAAS para autenticação

Por fim, é necessário indicar que a aplicação utiliza controle de acesso e que a execução da aplicação utilizará o administrador de segurança do JAAS (*SecurityManager*). Para tal finalidade é preciso utilizar alguns parâmetros na linha de código de execução do sistema. A Figura 8.7 apresenta o comando para executar o sistema de gerenciador de arquivos após o seu empacotamento. O parâmetro *-Djava.security.manager* indica que este é um aplicativo protegido e que deve ter sua segurança gerenciada pelo administrador de segurança do Java. O parâmetro *-Djava.security.policy=jaas.policy* indica que o arquivo de políticas a ser verificado pelo administrador de segurança terá o nome *jaas.policy*, que é o arquivo de políticas gerado pelo transformador.

```
1 java -Djava.security.manager -Djava.security.policy=jaas.policy -jar
    filessystem.jar
```

Figura 8.7: Comando para execução da aplicação do gerenciador de arquivos

8.1.2 Restrições de autorização e autenticações no código gerado

Apesar da proposta do JAAS de ser uma plataforma genérica de autenticação e autorização, os tipos que implementam a interface *Principal* do *framework* JAAS são relacionados a cada mecanismo de autenticação, onde não são fornecidos métodos para acesso a cer-

tas informações dos usuários. O mecanismo de autenticação nativo do Windows, por exemplo, popula o elemento do tipo *Subject* de um usuário com elementos do tipo *NTSidGroupPrincipal* representando os grupos aos quais o usuário está relacionado localmente à um *host* ou domínio no qual o usuário foi autenticado previamente. Entretanto, não é possível acessar o nome do grupo ao qual o usuário está associado, somente o código do identificador único de segurança do NT, chamado de SID (*Security Identifier*). O mesmo ocorre no sistema Unix, onde os grupos do usuário são representados por elementos do tipo *UnixNumericGroupPrincipal* que contém identificadores dos grupos aos quais o usuário está relacionado, mas que também não tem informações sobre o nome do grupo.

O JAAS pode ser estendido para receber outros tipos de mecanismos de autenticação que não os padrões e nativos e por isso foi definida então uma abstração para tornar o código gerado independente do tipo de dados representando a informação do grupo fornecido pelo JAAS.

Este foi um passo necessário devido a forma como SecureUML+ComponentUML define os papéis dos usuário e as restrições de autorizações. Os papéis são representados por um identificador que pode ser transformado no nome de grupos de usuários nos diversos sistemas. Contudo, os elementos do tipo *Principal* não trazem o nome de um elemento, mas seu código interno. As restrições fazem uso de informações que existem em tempo de execução, por exemplo, o nome do usuário que está autenticado no sistema. Esta informação é interna a um tipo diferente de *Principal* em cada sistema.

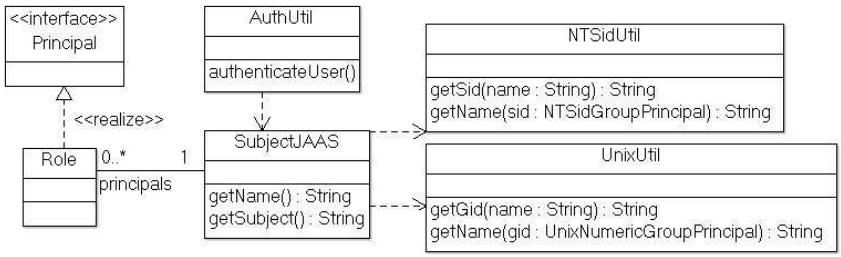

Figura 8.8: Biblioteca para autenticação na aplicação gerada pelo transformador SecureUMLToAACJAAS

A abstração criada é representada pela utilização da biblioteca *JAASLib*. Esta biblioteca foi criada contemplando o processo de autenticação nativa do JAAS nos sistemas NT e Unix, podendo ser estendida para outras plataformas. O detalhamento das classes desta biblioteca é apresentado na Figura 8.8.

Ao iniciar uma aplicação controlada, o desenvolvedor deve indicar qual a forma de autenticação. Para facilitar este processo, a classe *AuthUtil* efetua a autenticação de acordo

com o arquivo de configuração gerado pelo transformador. No arquivo de configuração são descritos os mecanismos de autenticação, ou como chamado no JAAS, os módulos de *login* (*LoginModule*).

A autenticação em JAAS gera um objeto do tipo *Subject* para representar um usuário autenticado no sistema contendo uma série de elementos do tipo *Principal*. O objeto do tipo *Subject* é passado para um elemento do tipo *SubjectJAAS*, que utiliza as classes *NTSidUtil* para conversão de *Subjects* em sistemas Windows e a classe *UnixUtil* para conversão de *Subjects* em sistemas Unix.

O elemento do tipo *SubjectJAAS* passa a conter o nome do usuário e um conjunto de objetos do tipo *Role*. Os objetos *Role* representam o nome dos papéis relacionados aos usuários em cada sistema e são extraídos a partir dos códigos provenientes dos elementos do tipo *Principal*. Caso os usuários do transformador utilizem outro tipo de autenticação, a biblioteca deve ser estendida e a classe *SubectJAAS* deve utilizar outros métodos para geração dos elementos do tipo *Role*.

8.1.3 Falhas nas assertivas

Conforme descrito no processo de transformação, caso ocorram falhas nas fases descritas no contrato de transformação, a ferramenta SecureUMLToAACJAAS interrompe o processo de transformação e exibe um erro indicando qual ou quais invariantes falharam antes de prosseguir para a próxima fase.

A falha do invariante pode representar não só um modelo que não está em conformidade a algum dos metamodelos gerados fornecido pelo usuário da ferramenta, mas uma falha de implementação por parte dos desenvolvedores. Portanto, a técnica de contratos de transformação representa além de uma técnica de validação do processo de transformação entre modelos, uma técnica de auxílio no desenvolvimento das implementações das transformações por ela especificadas.

Todo o processo de desenvolvimento da ferramenta SecureUMLToAACJAAS foi auxiliado pela utilização de contratos de transformação, pois devido a falhas nas verificações de invariantes, problemas foram identificados e puderam ser corrigidos sem que fosse verificado o código gerado.

Um exemplo é o erro identificado durante a integração do analisador (*parser*) de arquivos que representam o modelo de políticas de segurança SecureUML+ComponentUML no formato XMI. O invariante de SecureUML+ComponentUML chamado *overridingDe-*

faultPermission gerou falhas no momento em que modelos contendo associações entre entidades controladas foram utilizados na transformação. A Figura 8.9 apresenta este invariante, que garante que um elemento do tipo *AtomicAction*, representando uma ação atômica para acesso a recursos na linguagem SecureUML+ComponentUML, não pode estar relacionado a permissão *DefaultPermission*, caso esteja relacionado a mais de uma permissão.

```
1 context AtomicAction inv overridingDefaultPermission:
2 self.allAssignedPermission()->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

Figura 8.9: Invariante OCL *overridingDefaultPermission*

A falha do invariante indica que existiam problemas no modelo ou na ferramenta, entretanto, verificações sobre o modelo garantiram sua boa formação e o erro se encontrava na ferramenta. Devido a fase em que o invariante *overridingDefaultPermission* é verificado, concluiu-se que o erro se encontrava no processo de geração da instância da sintaxe abstrata da linguagem SecureUML+ComponentUML a partir do modelo de sintaxe concreta de SecureUML+ComponentUML representado pelo XMI.

A causa do erro na ferramenta era uma inversão nos passos descritos no algoritmo utilizado para geração da instância da sintaxe abstrata descrito em (BASIN *et al.*, 2009). Primeiramente, todos os recursos controlados devem ser convertidos, gerando as ações de acesso a cada recurso. Posteriormente, as permissões são convertidas e relacionadas a cada ação, onde as ações perdem a relação com o elemento *DefaultPermission*. Os passos invertidos geravam relações entre elementos de ação do tipo *AtomicAction* e o elemento de permissão *DefaultPermission*, além das permissões já relacionadas aos elementos do tipo *AtomicAction*. Sem a verificação do contrato de transformação pela ferramenta, o erro seria propagado e provavelmente a sua detecção ocorreria somente na execução dos sistemas conectados a infraestrutura de segurança gerada.

A Figura 8.10 apresenta uma parte da instância da sintaxe abstrata gerada a partir do modelo de um gerenciador de reuniões com relações errôneas entre as ações atômicas e o elemento *DefaultPermission*. A política de segurança e o sistema gerado para o gerenciador de reuniões serão apresentados na Seção 8.2.

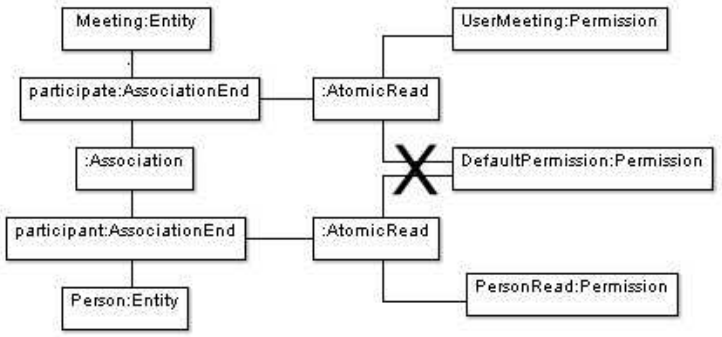

Figura 8.10: Instância do metamodelo SecureUML+ComponentUML com conexões erradas entre os elementos

8.2 Estudos de caso

Nesta seção serão apresentados dois exemplos de aplicação da transformação de modelos de política SecureUML+ComponentUML para a infraestrutura de segurança em AAC+JAAS. O primeiro exemplo representa um gerenciador de reuniões apresentado em (BASIN *et al.*, 2009) e o segundo um sistema de atendimento médico domiciliar apresentado em (CARVALHO *et al.*, 2010).

8.2.1 Gerenciador de Reuniões

Neste exemplo será apresentada uma política de segurança para o diagrama de classes de um gerenciador de reuniões apresentado na Figura 8.11. O gerenciador de reuniões possui um cadastro de pessoas, salas e reuniões. As pessoas são representadas pela classe *Person*, as salas pela classe *Room*, e as reuniões pela classe *Meeting*. Cada reunião possui uma pessoa como seu responsável, representado pela ponta de associação *owner*, e é composta por dois ou mais participantes, representado pela ponta de associação *participants*.

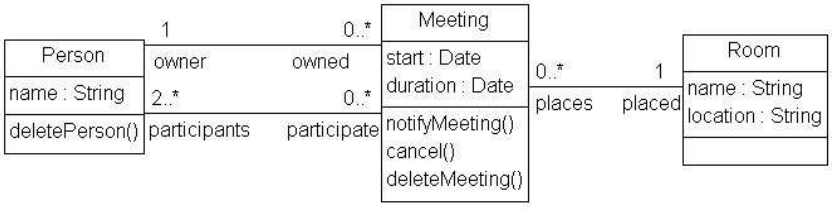

Figura 8.11: Classes de um sistema gerenciador de reuniões

Para este sistema foram identificados três perfis de acesso que serão representados por papéis de usuários. O primeiro perfil pode criar e acessar as reuniões em modo de leitura no sistema. Este perfil também pode modificar, apagar, cancelar e enviar notificações das reuniões das quais é responsável. O papel que representa este perfil será

chamado de *SystemUser*. O segundo perfil terá os mesmos acessos permitidos para o papel *SystemUser*, entretanto, ele poderá cancelar e notificar qualquer reunião do sistema. O papel que descreve este perfil será chamado de *Supervisor*. Para o controle das pessoas cadastradas no sistema haverá outro papel, chamado *SystemAdministrator*.

O diagrama SecureUML+ComponentUML gerado para este exemplo é apresentado na Figura 8.12. Este diagrama é baseado no exemplo apresentado em (BASIN *et al.*, 2009), com a adição da permissão de leitura da entidade *Person* pelo papel *SystemUser*, sem o qual não seria possível adicionar participantes às reuniões criadas.

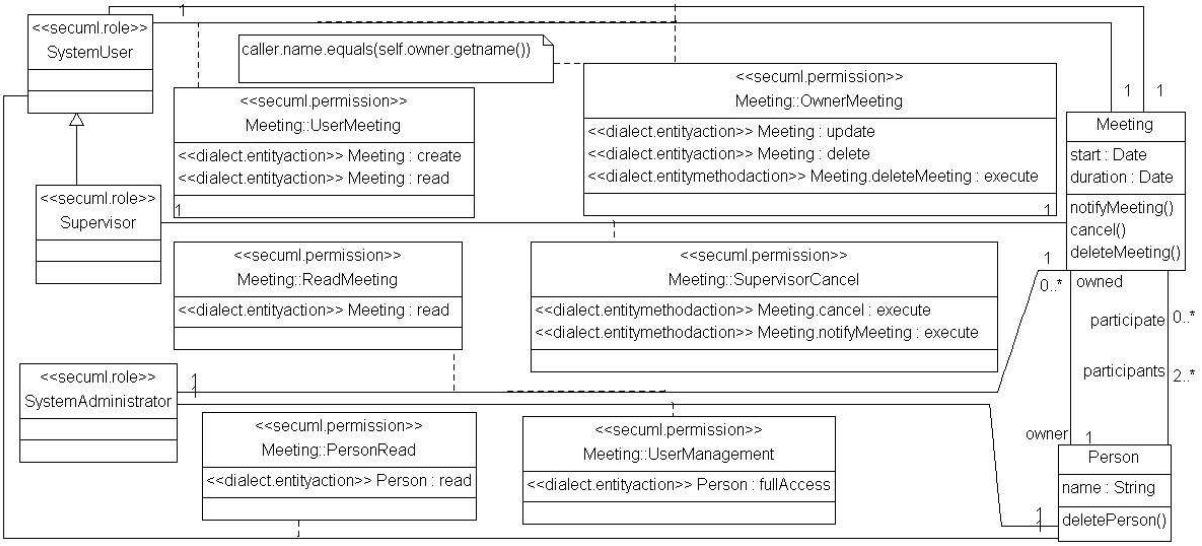

Figura 8.12: Modelo de política de segurança SecureUML+ComponentUML para o gerenciador de reuniões

Este exemplo difere do exemplo do gerenciador de arquivos do Capítulo 4 com relação a associações entre entidades controladas e restrições de autorização. As associações entre entidades controladas gerarão em AAC+JAAS elementos do tipo *ResAssociationEnd* que são representados por pontas de associação. Na sintaxe concreta, *ResAssociationEnd* são representadas por atributos nas classes abstratas que representam entidades controladas em AAC+JAAS. Para acesso às pontas de associação são gerados os elementos do tipo *ResAssocEndSetMethod* e *ResAssocEndGetMethod* que são representados na sintaxe concreta por métodos para leitura e escrita das pontas de associação. Na Figura 8.13 é apresentada parte da classe abstrata representando uma reunião, com destaque para as pontas de associação e para o método de notificação de uma reunião (*notifyMeeting()*).

Para a classe de reuniões será gerado o aspecto chamado *Meeting_ACAspect*. Parte do aspecto é apresentado na Figura 8.14 com destaque para o *pointcut* que intercepta as chamadas para o método *notifyMeeting()*. A verificação de acesso ao *pointcut* ocorre no

```
1  public abstract class Meeting implements Serializable {
2  // association end
3  protected List<Person> participants; protected Person owner;
4  // methods
5  public List<Person> getparticipants() throws Exception {return
       participants; }
6  public void setparticipants(List<Person> participants) throws Exception
       { this.participants = participants; }
7  public abstract void notifyMeeting () throws Exception;
8  //Os outros elementos foram omitidos para ilustração
9 }
```

Figura 8.13: Código em Java para classe abstrata que representa reuniões

before advice através da ação *MeetingnotifyMeetingMethodAtomicExecute*.

```
1 public aspect Meeting_ACAspect {
2  pointcut PCnotifyMeeting(Meeting self): call(void Meeting.notifyMeeting
       ()) && target( self );
3  before(Meeting self, Person owner) throws Exception:PCnotifyMeeting(
       self ){
4    SubjectJAAS caller=Env.getSubject();
5    if (Subject.doAsPrivileged(caller.getSubject(),new
         MeetingnotifyMeetingMethodAtomicExecute(self), null))
6       return;
7    else throw new Exception("O usuário não possui autorização.");
8  }
9 }
```

Figura 8.14: Código AspectJ para a classe Meeting em AAC+JAAS

Na ação *MeetingnotifyMeetingMethodAtomicExecute* ocorrerá a verificação da permissão de acesso ao método *notifyMeeting()* que pode ser acessado por dois papéis diferentes. Para o acesso a este método pelo papel *SystemUser* é utilizada a permissão *OwnerMeeting*, que possui uma restrição associada, definido por um elemento do tipo *AuthorizationConstraint*.

As restrições são representadas como um comentário associado a permissão no modelo SecureUML+ComponentUML. Esta restrição garante que para os usuários sob o papel *SystemUser* somente os responsáveis por uma reunião podem executar o método de notificação. A Figura 8.15 apresenta o código da ação *MeetingnotifyMeetingMethodAtomicExecute*, onde a verificação da permissão *OwnerMeeting* ocorre na linha 8 e a verificação da restrição a ela associada ocorre na linha 10. Usuários sob o papel *Supervisor* tem acesso ao método *notifyMeeting()* através da permissão *SupervisorCancel*, que não possui nenhuma restrição a ela atrelada. A verificação desta última permissão ocorre na linha 12. É importante destacar que o conteúdo de uma restrição deve ser escrito na forma de

uma expressão booleana na linguagem Java, pois a ferramenta inclui o conteúdo da restrição diretamente no código da ação. SecureUML+ComponentUML permite a definição da linguagem das restrições através de seu metamodelo, onde nos modelos comumente utiliza-se a linguagem OCL para descrever as restrições. Entretanto, para evitar o uso de um transformador de código OCL para Java, decidiu-se por utilizar neste trabalho a escrita das restrições diretamente a linguagem de destino.

```
1 public class MeetingnotifyMeetingMethodAtomicExecute implements
    PrivilegedAction<Boolean> {
2  Meeting resource;
3  public Boolean run(){
4    SubjectJAAS caller = Env.getSubject();
5    SecurityManager sm = System.getSecurityManager();
6    //permissions
7    try{
8      sm.checkPermission(new OwnerMeeting__JAASPermission(""));
9      //authorization constraints
10     if(caller.getName().equals(resource.getowner().getname())) return
         true;
11   }catch (Exception e) {} try{
12     sm.checkPermission(new SupervisorCancel__JAASPermission(""));
13   }catch (Exception e) {}
14   return false;
15 }
16 }
```

Figura 8.15: Código em Java para a ação de acesso ao método *notifyMeeting()*

8.2.2 Sistema de atendimento médico domiciliar

Neste exemplo será apresentado um protótipo baseado no Sistema Computacional Inteligente de Assistência Domiciliar à Saúde (SCIADS) (CARVALHO *et al.*, 2010). Através da documentação do sistema (ERTHAL, 2010) e de testes na aplicação[2] foi definida uma política de segurança para uma parte do sistema. As entidades identificadas para o protótipo são apresentadas na Figura 8.16. Neste sistema, um paciente possui um plano de cuidados e é atendido por um ou mais médicos. Com relação ao tratamento dos pacientes, os médicos podem adicionar atividades ou medicamentos em forma de prescrições nos planos de cuidados dos pacientes que atende.

Os papéis identificados para este protótipo representam médicos e pacientes do sistema. O papel de paciente pode ler somente seu próprio plano de cuidados e as prescrições que dele fazem parte. O papel de médico pode ler o plano de cuidados e criar prescrições para os pacientes aos quais atende. Um usuário sob o papel médico pode modificar e

[2] Disponível em `http://sites.google.com/site/sistemadetelesaude/home/prot`.

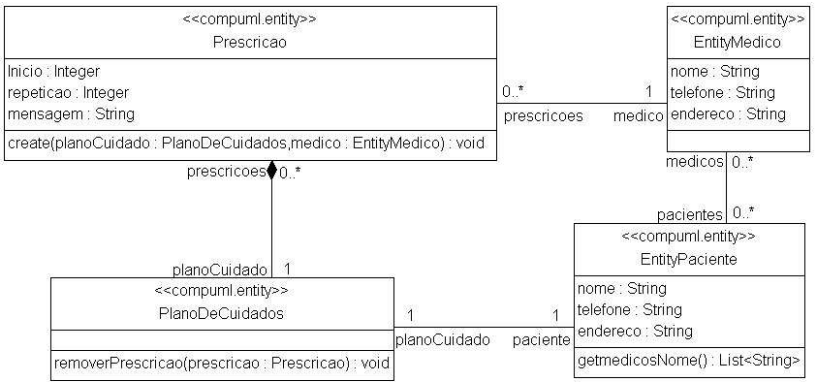

Figura 8.16: Classes de um Sistema Computacional Inteligente de Assistência Domiciliar à Saúde

excluir as prescrições que ele próprio criou, mas não as prescrições efetuadas por outros médicos. O diagrama SecureUML+ComponentUML gerado para este exemplo é apresentado na Figura 8.17, onde um papel de paciente é definido pelo elemento *RolePaciente* e o papel de médico é definido pelo elemento *RoleMedico*.

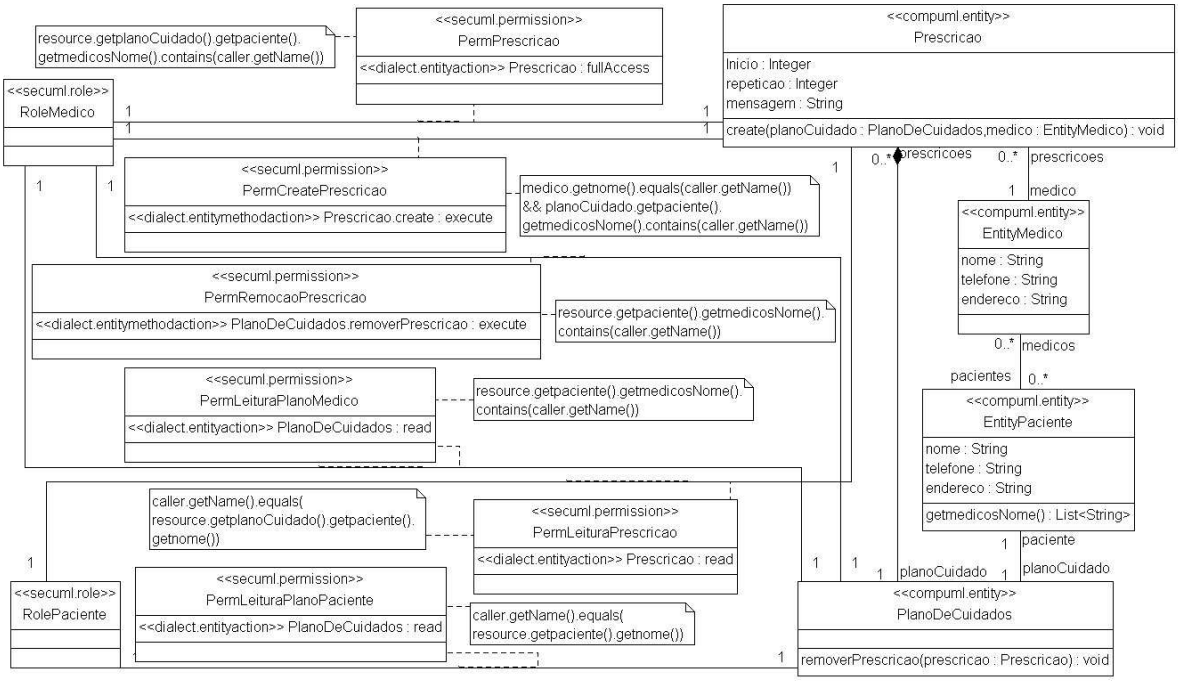

Figura 8.17: Modelo de política de segurança SecureUML+ComponentUML do SCIADS

Este exemplo apresenta mais restrições relacionadas as permissões devido a sua complexidade. Por exemplo, o plano de cuidados estará relacionado a três permissões definidas no modelo SecureUML+ComponentUML: *PermLeituraPlanoMedico*, *PermLeituraPlanoPaciente* e *PermRemocaoPrescricao*. Usuários sob o papel médico tem acesso de leitura ao plano de cuidados de seus pacientes através da permissão *PermLeituraPlanoMedico*.

Usuários sob o papel *Paciente* tem acesso de leitura ao seu próprio plano de cuidados através da permissão *PermLeituraPlanoPaciente*. Os médicos podem remover prescrições dos planos de cuidados dos pacientes, caso a prescrição tenha sido criada pelo próprio médico que está executando a ação de remoção. A Figura 8.18 apresenta o modelo SecureUML+ComponentUML com destaque para a entidade de Plano de Cuidados e as permissões a ela relacionadas.

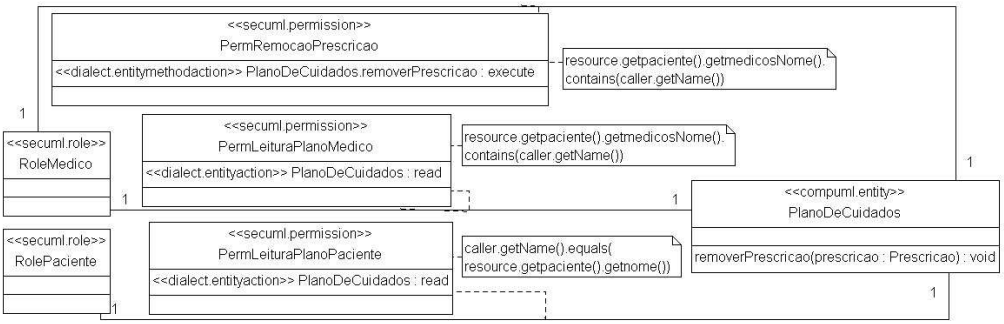

Figura 8.18: Modelo de política de segurança SecureUML+ComponentUML do elemento de Plano de Cuidados do SCIADS

As restrições de autorização indicam as verificações que devem ocorrer em tempo de execução, por exemplo, se o médico atende a um paciente, a restrição *resource.getpaciente().getmedicosNome().contains(caller.getName())* será verdadeira. A variável especial *caller* é definida em SecureUML, com a finalidade de representar o usuário autenticado no sistema. Para simplificar o modelo foi utilizado o nome dos médicos e pacientes nas verificações, contudo em um caso real seria indicado a utilização de um identificador único dos elementos.

```
1 public class PlanoDeCuidadosprescricoesAssociationEndAtomicRead
    implements PrivilegedAction<Boolean> {
2  PlanoDeCuidados resource;
3  public Boolean run(){
4  SubjectJAAS caller = Env.getSubject();
5  SecurityManager sm = System.getSecurityManager();
6  try{
7    sm.checkPermission(new PermLeituraPaciente__JAASPermission(""));
8    if(caller.getName().equals(resource.getpaciente().getnome()))
9      return true;
10 }catch (Exception e) {} try{
11   sm.checkPermission(new PermLeituraMedico__JAASPermission(""));
12   if(resource.getpaciente().getmedicosNome().contains(caller.getName()))
13     return true;
14 }catch (Exception e) { }
15 return false;
16 }
```

Figura 8.19: Código em Java para a ação de acesso ao método *notifyMeeting()*

Após a geração da infraestrutura de segurança, as verificações das permissões ocorrerão nas diversas ações que acessam a entidade de plano de cuidados. Na ação *PlanoDeCuidadosprescricoesAssociationEndAtomicRead* ocorrerá a verificação da permissão de acesso a ponta de associação das prescrições de um plano de cuidados em JAAS. Cada uma das permissões é verificada, juntamente com as restrições. Caso alguma delas seja verdadeira, então o acesso será autorizado. A verificação de leitura do plano de cuidados por parte dos pacientes é executada na linha 7 e a verificação da restrição para esta permissão ocorre na linha 8. A verificação de leitura do plano de cuidados por parte dos médicos é executada na linha 11 e a verificação da restrição para esta permissão ocorre na linha 12.

8.3 Discussão

Neste capítulo foi apresentada a implementação da transformação entre SecureUML+ComponentUML e AAC+JAAS na implementação da ferramenta SecureUMLToAACJAAS. Esta ferramenta utiliza o padrão de projeto apresentado na Seção2.3 e implementa a transformação utilizando o contrato de transformação apresentado no Capítulo 6.

A ferramenta SecureUMLToAACJAAS tem a limitação de aceitar somente o formato XMI da ferramenta ArgoESS para os arquivos de entrada. Além disso, o código gerado pela ferramenta necessita da biblioteca JAASLib, descrita na Seção 8.1.2. A utilização da biblioteca JAASLib facilita a autenticação em diferentes sistemas, mas torna o código dependente desta biblioteca, portanto, menos genérico.

No presente momento, ao ocorrer algum erro no processo de transformação, a ferramenta SecureUMLToAACJAAS exibe uma mensagem de erro e gera um arquivo de log com informações dos invariantes executados com sucesso e do invariante que gerou a falha. Apesar de utilizar um mecanismo de verificação da transformação, existe a possibilidade de geração de sistemas incorretos devido a utilização de modelos de políticas de segurança não condizentes aos requisitos de um sistema. Este fato poderia sucitar nos usuários do transformador a falsa impressão que a transformação ocorreu de maneira incorreta. Para solucionar tal questão poderiam ser utilizados mecanismos que auxiliem nas verificações do comportamento do sistema com relação a política definida no modelo SecureUML, por exemplo, mecanismos de *traceability* (AIZENBUD-RESHEF *et al.*, 2006). Um mecanismo de *traceability* busca apresentar uma relação entre diferentes produtos do processo de desenvolvimento de um sistema e poderia ser gerado pela transformador para

indicar quais permissões do modelo SecureUML não foram atendidas ao ocorrer uma falha de autorização no sistema gerado com código em AAC+JAAS.

Neste capítulo também foram apresentados estudos de casos de sistemas gerados com código resultante da ferramenta SecureUMLToAACJAAS complementando o Capítulo 7 referente a argumentação da corretude da transformação no que diz respeito à validação do presente trabalho. Os estudos de caso apresentam as validações de maneira empírica ao utilizar transformações de diferentes tamanhos e complexidades entre as linguagens SecureUML e AAC+JAAS.

Ainda com relação aos estudos de caso, devido a forma como a linguagem SecureUML é definida, os recursos controlados nos sistemas serão representados como classes, seus atributos e métodos. Vale lembrar que é possível definir recursos controlados que representem elementos específicos, por exemplo, classes de interface de um serviço ou elementos de uma interface gráfica, desde que estes sejam modelados como uma classe, ou melhor dizendo, uma entidade de um modelo SecureUML que gerará uma classe abstrata após a transformação.

Capítulo 9

Conclusão

9.1 Considerações Finais

Neste trabalho foi apresentada a especificação e uma implementação de Segurança Dirigida a Modelos baseada em contratos de transformação para modelagem de políticas de controle de acesso e geração automatizada de código modular para um sistema que necessita de controle de acesso. Esta implementação utiliza a linguagem SecureUML para modelar as políticas de segurança, onde através de transformações gera infraestrutura de código de segurança utilizando elementos de JAAS e de orientação a aspectos para modularizar o controle de acesso. A estrutura de aspectos combinada a JAAS foi especificada numa linguagem de modelagem chamada AAC+JAAS e garante certa modularização no controle de acesso das aplicações que a utilizarem.

O processo de desenvolvimento seguiu a abordagem de Desenvolvimento Dirigido a Modelos focado na especificação e validação das transformações baseadas na técnica de contratos de transformação. Na abordagem de Desenvolvimento Dirigido a Modelos, os modelos são artefatos vivos que são utilizados para geração de novos modelos e código ao invés de representarem apenas parte da documentação no processo de desenvolvimento. No entanto, é necessário que sejam utilizadas técnicas de validação e verificação dos modelos pois a existência de inconsistências nos modelos gera inconsistências nos sistemas por eles descritos.

Definindo modelos como instâncias de linguagens, fez-se um paralelo entre o processo de transformação de modelos com o processo de transformação entre linguagens de programação, onde a verificação utilizando invariantes de metamodelos pode ser considerada como a verificação de semântica estática de linguagens de programação.

Um contrato de transformação é representado por um conjunto de invariantes que especificam a correta relação dos metamodelos de entrada e saída. Neste trabalho foi definido um contrato de transformação a partir do metamodelo de transformação e um conjunto de invariantes para a relação das linguagens SecureUML+ComponentUML e AAC+JAAS. Com o intuito de validação da especificação e da técnica de contratos de transformação, foi apresentado um passo de linearização de alguns elementos para tornar direta e intuitiva a relação entre os elementos de cada metamodelo. Este passo serviu para diminuir a complexidade dos invariantes do contrato de transformação.

A ferramenta SecureUMLToAACJAAS foi desenvolvida com o uso do contrato de transformação referido. A especificação do contrato de transformação e a garantia da sua corretude necessitaram de maior empenho antes da implementação da ferramenta. No entanto, este empenho gerou como benefício a verificação e localização de erros durante todo o processo de desenvolvimento da ferramenta. Outro ponto importante a destacar é que o processo de transformação ao ser representado como um modelo, pode ser estendido para outras formas de validação de modelos.

Conclui-se neste trabalho a importância da aplicação técnicas de validação e verificação em transformações de modelo, em especial da técnica de contratos de transformação. Com o auxílio destas técnicas aumenta-se a confiança na corretude de um transformador de modelos. A especificação define o que deve ser gerado a partir de um modelo auxiliando na correta implementação de ferramentas de transformação de modelos.

9.2 Trabalhos futuros

Para trabalhos futuros propõe-se a utilização de novas formas de validação a serem aplicadas na transformação, maior integração com ambientes de criação de modelos e a utilização de contratos de transformação com linguagens de transformação de modelos.

As validações OCL não garantem que um modelo é válido sob qualquer aspecto, por exemplo, se ele é consistente. Outras técnicas de validação se tornam necessárias como a técnica proposta em (BRAGA; HÆUSLER, 2010). Através desta técnica é possível racionalizar se um modelo é consistente, por exemplo, validar a consistência de um modelo SecureUML+ComponentUML antes de aplicar transformações.

Com o objetivo de aumentar a usabilidade do transformador, propõe-se que este seja integrado a ambientes de criação de modelos UML com suporte a perfis SecureUML+ComponentUML e validação OCL. Após a criação de um modelo de política, o analista de

segurança, em um mesmo ambiente, validaria o modelo criado e o transformaria através do transformador descrito nesta dissertação para o código de infraestrutura de segurança. Um ambiente que contém estes elementos é o Eclipse EMF (*Eclipse Modeling Framework*), que possui ferramentas para criação de modelos UML com perfis e validação OCL.

A não ocorrência de falhas nos invariantes do processo de transformação indica que o modelo utilizado está em conformidade com o metamodelo de SecureUML+Component-UML. Entretanto, não significa que o analista de segurança tenha criado um modelo de acordo com as especificações a ele fornecidas. Com o objetivo de auxiliar estes usuários do processo de transformação, poderia ser utilizado um mecanismo de *traceability* no código gerado. Este mecanismo seria responsável por informar, quando ativado, qual permissão do modelo de políticas de segurança não foi atendida ao ocorrer uma falha no acesso a um recurso. Dessa maneira, o analista de segurança poderia facilmente identificar qual especificação foi modelada de forma incorreta ou que não foi atendida.

Por fim, os contratos de transformação poderiam ser utilizados juntamente com linguagens específicas para transformação de modelos, como é o caso da linguagem QVT (*Query View Transformations*) definida pela OMG. Esta nova forma de aplicação de contratos de transformação produzirá uma comparação com o padrão de projetos e a implementação apresentados nesta dissertação com relação a modularidade da transformação e a capacidade de aplicar esta técnica de especificação utilizando outras formas de transformação de modelos.

Referências

AIZENBUD-RESHEF, N.; NOLAN, B. T.; RUBIN, J.; SHAHAM-GAFNI, Y. Model traceability. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 45, p. 515–526, July 2006. ISSN 0018-8670. Disponível em: `http://dx.doi.org/10.1147/sj.453.0515`.

ANASTASAKIS, K.; BORDBAR, B.; KÜSTER, J. Analysis of model transformations via alloy. In: BAUDRY A. FAIVRE, S. G. A. P. B. (Ed.). *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)*. Berlin/Heidelberg: Springer, 2007. p. 47–56.

BASIN, D.; CLAVEL, M.; DOSER, J.; EGEA, M. Automated analysis of security-design models. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 51, n. 5, p. 815–831, 2009. ISSN 0950-5849.

BASIN, D.; DOSER, J.; LODDERSTEDT, T. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 15, n. 1, p. 39–91, 2006. ISSN 1049-331X.

BASIN, D. A.; CLAVEL, M.; EGEA, M.; SCHLÄPFER, M. Automatic generation of smart, security-aware gui models. In: MASSACCI, F.; WALLACH, D. S.; ZANNONE, N. (Ed.). *ESSoS*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 5965), p. 201–217. ISBN 978-3-642-11746-6.

BAUDRY, B.; DINH-TRONG, T.; MOTTU, J.-M.; SIMMONDS, D.; FRANCE, R.; GHOSH, S.; FLEUREY, F.; TRAON, Y. L. Model transformation testing challenges. *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, Bilbao, Spain, July 2006. Disponível em: `http://www.irisa.fr/triskell/publis/2006/baudry06b.pdf`.

BÉZIVIN, J.; BüTTNER, F.; GOGOLLA, M.; JOUAULT, F.; KURTEV, I.; LINDOW, A. Model Transformations? Transformation Models! In: *9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*. Berlin/Heidelberg: Springer, 2006. v. 4199/2006, p. 440–453. Disponível em: `http://dx.doi.org/10.1007/11880240_31`.

BLANCO, C.; GUZMÁN, I. G. R. de; FERNÁNDEZ-MEDINA, E.; TRUJILLO, J.; PIATTINI, M. Applying an mda-based approach to consider security rules in the development of secure dws. In: *ARES*. [S.l.]: IEEE Computer Society, 2009. p. 528–533.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. [S.l.]: Addison-Wesley Professional, 2005. Hardcover. ISBN 0321267974.

BRAGA, C. A transformation contract to generate aspects from access control policies. *Journal of Software and Systems Modeling*, Springer, Berlin Heidelberg, 2010. DOI: 10.1007/s10270-010-0156-x.

BRAGA, C.; HÆUSLER, E. H. Lightweight analysis of access control models with description logic. *Innovations in Systems and Software Engineering*, v. 6, p. 115–123, 2010.

CABOT, J.; CLARISÓ, R.; RIERA, D. Verification of UML/OCL class diagrams using constraint programming. In: *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. Washington, DC, USA: IEEE Computer Society, 2008. p. 73–80. ISBN 978-0-7695-3388-9.

CABOT, J.; CLARISÓ, R.; RIERA, D. Verifying UML/OCL operation contracts. In: LEUSCHEL, M.; WEHRHEIM, H. (Ed.). *IFM*. Berlin, Heidelberg: Springer-Verlag, 2009. (Lecture Notes in Computer Science, v. 5423), p. 40–55. ISBN 978-3-642-00254-0.

CARIOU, E.; MARVIE, R.; SEINTURIER, L.; DUCHIEN, L. OCL for the specification of model transformation contracts. In: PATRASCOIU, O. (Ed.). *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal*. [S.l.]: University of Kent, 2004. p. 69–83.

CARVALHO, S.; ERTHAL, M.; MARELI, D.; SZTAJNBERG, A.; COPETTI, A.; LOQUES, O. Monitoramento remoto de pacientes em ambiente domiciliar. In: *XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - Salão de Ferramentas*. Gramado, RS, Brasil: [s.n.], 2010.

CICCHETTI, A.; RUSCIO, D. D.; PIERANTONIO, A. A metamodel independent approach to difference representation. *Journal of Object Technology*, v. 6, n. 9, p. 165–185, 2007.

CLAVEL, M.; EGEA, M.; ANGEL, G. de D. M. Building an efficient component for OCL evaluation. *ECEASST*, v. 15, 2008.

CLAVEL, M.; EGEA, M.; DIOS, M. A. G. de. Eceasst building an efficient component for ocl evaluation. *ECEASST*, v. 15, 2008.

ERTHAL, M. Protótipo de monitoramento remoto de pacientes em ambiente domiciliar – manual de utilização – versão 3. 2010. Disponível em: `http://sites.google.com/site/sistemadetelesaude/downloads/Manual_de_utilizacao.pdf`. Acesso em: 10 dez. 2010.

FERRAIOLO, D.; KUHN, R. Role-based access control. In: *In 15th NIST-NCSC National Computer Security Conference*. Baltimore, USA: [s.n.], 1992. p. 554–563.

FERRAIOLO, D. F.; CHANDRAMOULI, R.; AHN, G.-J.; GAVRILA, S. I. The role control center: features and case studies. In: *SACMAT*. [S.l.]: ACM, 2003. p. 12–20. ISBN 1-58113-681-1.

GASTER, B.; KENT, D.; SEVEN, D.; SABBADIN, E.; CONWAY, R.; BASIURA, R.; LAKSHMINARAYANAN, S.; SIVAKUMAR, S. *ASP.NET Security*. 1st. ed. [S.l.]: Peer Information, 2002. 400 p. ISBN 1861006209.

JÜRJENS, J.; HOUMB, S. H. Dynamic secure aspect modeling with UML: From models to code. In: BRIAND, L. C.; WILLIAMS, C. (Ed.). *MoDELS*. Berlin Heidelberg: Springer, 2005. (Lecture Notes in Computer Science, v. 3713), p. 142–155. ISBN 3-540-29010-9.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of aspectj. In: KNUDSEN, J. L. (Ed.). *ECOOP*. Berlin Heidelberg: Springer, 2001. (Lecture Notes in Computer Science, v. 2072), p. 327–353. ISBN 3-540-42206-4.

KLEPPE, A. G. A language description is more than a metamodel. In: *Fourth International Workshop on Software Language Engineering, Nashville, USA*. Grenoble, France: [s.n.], 2007. ISBN not assigned.

KLEPPE, A. G.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 37, p. 316–344, December 2005. ISSN 0360-0300. Disponível em: `http://doi.acm.org/10.1145/1118890.1118892`.

MEYER, B. *Object-Oriented software construction*. 2nd.. ed. [S.l.]: Prentice Hall, 1997.

MULARIEN, P. *Spring Security 3*. Packt Publishing, 2010. ISBN 1847199747. Disponível em: `https://www.packtpub.com/spring-security-3/book`.

OAKS, S. *Java Security*. 2. ed. Sebastopol, CA, USA: O'Reilly Media, 2001. ISBN 0596001576.

OMG. *MDA Guide Version 1.0.1*. [S.l.], June 2003. Disponível em: `http://www.omg.org/docs/omg/03-06-01.pdf`.

OMG. *Object Constraint Language Specification, version 2.0*. [S.l.], June 2006. Disponível em: `http://fparreiras/papers/OCLSpec.pdf`.

OMG. *OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3*. [S.l.], 2010. Disponível em: `http://www.omg.org/spec/UML/2.3/Infrastructure/PDF`.

ORACLE. Java™Authentication and Authorization Service (JAAS) Reference Guide. `http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html`. 2006.

RAY, I.; FRANCE, R. B.; LI, N.; GEORG, G. An aspect-based approach to modeling access control concerns. *Information & Software Technology*, v. 46, n. 9, p. 575–587, 2004.

SCOTT, M. L. *Programming Language Pragmatics, Third Edition*. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0126339511.

SEBESTA, R. W. *Concepts of Programming Languages*. 7. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321312511.

The UCMaude Group. Manual de utilização da ferramenta argoess (argoess howto). 2009. Disponível em: `http://maude.sip.ucm.es/eos/downloads/argoess-howto.html`. Acesso em: 1 dez. 2010.

WALLS, C.; BREIDENBACH, R. *Spring in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2005. ISBN 1932394354.

WARMER, J.; KLEPPE, A. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321179366.

APÊNDICE A – Invariantes de SecureUML+ComponentUML

Os invariantes deste apêndice foram apresentados no trabalho de (BASIN *et al.*, 2009) e garantem a corretude estrutural de instâncias de SecureUML+ComponentUML.

A.1 Papel padrão

Os invariantes a seguir garantem que o modelo contém um papel padrão com a semântica desejada para SecureUML; em particular, qualquer usuário está sob, pelo menos, o papel padrão.

```
1 context Role
2
3 inv existsADefaultRole:
4 self.allInstances()->select(r|r.default)->size() = 1
5
6 inv allRolesInheritFromDefaultRole:
7 self.superrolePlus()->exists(r|r.default)
```

```
1 context User
2
3 inv allUsersAssignedDefaultRole:
4 self.hasrole->exists(r|r.default)
```

A.2 Hierarquia de papéis

Os invariantes a seguir garantem que a hierarquia de papéis é acíclica.

```
1 context Role
2
```

```
3 inv noCyclesinRoleHierarchy:
4 self.superrole->forAll(r|r.superrolePlus()->excludes(self))
```

A.3 Permissão padrão

Os invariantes a seguir garantem que os modelos contém uma permissão padrão com a semântica proposta em SecureUML. Em particular, a permissão padrão deve somente estar relacionada ao papel padrão e deve estar relacionado a somente ações atômicas. Além disso, as ações atômicas são relacionadas a pelo menos uma permissão. Caso sejam relacionadas a mais de uma permissão, nenhuma delas será a permissão padrão.

```
1 context Permission
2
3 inv existsADefaultPermission:
4 self.allInstances()->select(p|p.default)->size() = 1
5
6 inv defaultPermissionAssignedToDefaultRole:
7 self.default implies self.givesaccess->forAll(r|r.default)
8
9 inv constraintByTrue:
10 self.default implies self.isconstraintby.body = ''true''
```

```
1 context CompositeAction
2
3 inv nonDefaultPermission:
4 self.allAssignedPermission()->forAll(p|not(p.default))
```

```
1 context AtomicAction
2
3 inv existsAPermission:
4 self.allAssignedPermission()->notEmpty()
5
6 inv overridingDefaultPermission:
7 self.allAssignedPermission()
8   ->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

A.4 Associação entre ações e recursos

Os invariantes a seguir garantem que as ações referenciam corretamente os recursos.

```
1 context AtomicCreate inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AtomicDelete inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AtomicUpdate inv targets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd
```

```
1 context AtomicRead inv targets:
2 self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context AtomicExecute inv targetsAMethod:
2 self.resource.oclIsTypeOf(Method)
```

```
1 context EntityFullAccess inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context EntityRead inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context EntityUpdate inv targetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AttributeFullAccess inv targetsAnAttribute:
2 self.resource.oclIsTypeOf(Attribute)
```

```
1 context AssociationEndFullAccess inv targetsAnAssociationEnd:
2 self.resource.oclIsTypeOf(AssociationEnd)
```

Os invariantes a seguir garantem que os recursos estão relacionados corretamente às ações.

```
1 context Entity
2
3 inv areAccessedBy:
4 self.action->size() = 5 and
5   self.action->exists(a|a.oclIsTypeOf(EntityFullAccess)) and
6   self.action->exists(a|a.oclIsTypeOf(EntityUpdate)) and
7   self.action->exists(a|a.oclIsTypeOf(EntityRead)) and
8   self.action->exists(a|a.oclIsTypeOf(AtomicCreate)) and
9   self.action->exists(a|a.oclIsTypeOf(AtomicDelete))
```

```
1 context Attribute
2
3 inv areAccessedBy:
4 self.action->size() = 3 and
5   self.action->exists(a|a.oclIsTypeOf(AttributeFullAccess)) and
6   self.action->exists(a|a.oclIsTypeOf(AtomicRead)) and
7   self.action->exists(a|a.oclIsTypeOf(AtomicUpdate))
```

```
1 context Method
2
3 inv areAccessedBy:
4 self.action->size() = 1 and
5   self.action->exists(a|a.oclIsTypeOf(AtomicExecute))
```

```
1 context Associationend
2
3 inv areAccessedBy:
4 self.action->size() = 3 and
5   self.action->exists(a|a.oclIsTypeOf(AssociationEndFullAccess)) and
6   self.action->exists(a|a.oclIsTypeOf(AtomicRead)) and
7   self.action->exists(a|a.oclIsTypeOf(AtomicUpdate))
```

A.5 Hierarquia de ações

Os invariantes a seguir garantem que a composição das ações compostas está correta com relação aos tipos das ações atômicas.

```
1 context EntityFullAccess
2
3 inv containsSubactions:
4 self.subordinatedactions =
5   self.resource.action
6   ->select(a|a.oclIsTypeOf(EntityUpdate))
7   ->union(self.resource.action
8   ->select(a|a.oclIsTypeOf(EntityRead)))
9   ->union(self.resource.action
10  ->select(a|a.oclIsTypeOf(AtomicCreate)))
11  ->union(self.resource.action
12  ->select(a|a.oclIsTypeOf(AtomicDelete)))
```

```
1 context EntityRead
```

```
2
3 inv containsSubactions:
4 self.subordinatedactions =
5   self.resource.oclAsType(Entity).hasattribute.action
6   ->select(a|a.oclIsTypeOf(AtomicRead))
7   ->union(self.resource.oclAsType(Entity)
8   .hasassociationend.action
9   ->select(a|a.oclIsTypeOf(AtomicRead)))
10  ->union(self.resource.oclAsType(Entity).hasmethod
11  ->select(me|me.isQuery).action
12  ->select(a|a.oclIsTypeOf(AtomicExecute)))
```

```
1 context EntityUpdate
2
3 inv containsSubactions:
4 self.subordinatedactions =
5   self.resource.oclAsType(Entity).hasattribute.action
6   ->select(a|a.oclIsTypeOf(AtomicUpdate))
7   ->union(self.resource.oclAsType(Entity)
8   .hasassociationend.action
9   ->select(a|a.oclIsTypeOf(AtomicUpdate)))
10  ->union(self.resource.oclAsType(Entity).hasmethod
11  ->select(me|not(me.isQuery)).action
12  ->select(a|a.oclIsTypeOf(AtomicExecute)))
```

```
1 context AttributeFullAccess
2
3 inv containsSubactions:
4 self.subordinatedactions = self.resource.action
5   ->select(a|a.oclIsTypeOf(AtomicUpdate))
6   ->union(self.resource.action
7   ->select(a|a.oclIsTypeOf(AtomicRead)))
```

```
1 context AssociationEndFullAccess
2
3 inv containsSubactions:
4 self.subordinatedactions = self.resource.action
5   ->select(a|a.oclIsTypeOf(AtomicUpdate))
6   ->union(self.resource.action
7   ->select(a|a.oclIsTypeOf(AtomicRead)))
```

APÊNDICE B – Invariantes de AAC+JAAS

Os invariantes deste apêndice garantem a corretude estrutural de instâncias de AAC+JAAS.

B.1 AAC

Os invariantes desta seção garantem a corretude estrutural de instâncias de AAC+JAAS relacionados a AAC.

Os invariantes a seguir garantem que os métodos de acesso em AAC estão relacionados corretamente as classes que contém os recursos controlados.

```
1 context ResGetMethod inv targetsResClassGetMethod:
2   self.resclass = self.resattribute.resclass
```

```
1 context ResSetMethod inv targetsResClassSetMethod:
2   self.resclass = self.resattribute.resclass
```

```
1 context ResAssocEndGetMethod inv targetsResClassAssocEndGetMethod:
2   self.resclass = self.resassociationend.resclass
```

```
1 context ResAssocEndSetMethod inv targetsResClassAssocEndSetMethod:
2   self.resclass = self.resassociationend.resclass
```

O invariante a seguir garante que os métodos de acesso em AAC estão relacionados corretamente aos

```
1 context ResMethod inv targetsResClassAspectPointcut:
2   self.resclass = self.pointcut.aspect.resclass
```

B.2 JAAS

Os invariantes desta seção garantem a corretude estrutural de instâncias de AAC+JAAS relacionados a JAAS.

O invariante a seguir garante haverá apenas um arquivo de políticas de JAAS nas instâncias de modelos AAC+JAAS.

```
1 context PolicyFile inv existsOnlyOnePolicyFile:
2   PolicyFile.allInstances()->size() = 1
```

APÊNDICE C – Invariantes do Contrato de Transformação entre SecureUML+ComponentUML e AAC+JAAS

Os invariantes deste apêndice garantem a corretude estrutural do metamodelo de transformação.

C.1 Relação entre elementos de SecureUML+ComponentUML e AAC+JAAS

```
1 context AtomicUpdate inv targetsUpdateResMethodsFromResAttributeOrResAssociationEnd:
2 self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResSetMethod)
3   or self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResAssocEndSetMethod)
```

```
1 context ResSetMethod inv targetsAtomicUpdateFromResSetMethod:
2 self.pointcut.advice.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResAttribute inv ResAttributeTargetsAttributeThroughResSetMethod:
5 self.ressetmethod.pointcut.advice.jaasaction.action.resource = self.attribute
```

```
1 context ResAssocEndSetMethod inv targetsAtomicUpdateFromResSetAssociationEndMethod:
2 self.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResAssociationEnd inv
      ResAssociationEndTargetsAssociationEndThroughResSetAssociationEndMethod:
5 self.resassociationendsetmethod.pointcut.advice.jaasaction.action.resource = self.associationend
```

```
1 context AtomicExecute inv targetsResOperationMethod:
2 self.jaasaction.resmethod.oclIsTypeOf(ResOperationMethod)
3
```

```
4 context ResOperationMethod inv targetsAtomicExecute:
5 self.jaasaction.action.oclIsTypeOf(AtomicExecute)
```

C.2 Relação entre elementos de SecureUML+ComponentUML e SecureUML+ComponentUML linearizado

```
1 context Role inv targetsRoleFlattened:
2 Role'.allInstances()->select( role' |
3   role'.isFlattened() and
4   role'.name = self.name and
5   role'.default = self.default
6   role'.haspermission = self.allPermissions())
7   ->size() = 1
```

```
1 context AtomicAction inv targetsActionFlattened:
2 AtomicAction'.allInstances()->select( aa' |
3   aa'.name = self.name and
4   aa'.isassigned = self.allAssignedPermissions() and
5   aa'.resource = self.resource)
6   ->size() = 1
```

C.3 Relação entre elementos de SecureUML+ComponentUML linearizado e AAC+JAAS

Invariante que atua sobre *PrincipalRole*.

```
1 context PrincipalRole inv RoleFlattenedIsConnectedToPrincipalRole:
2 self.name = self.secrole.name and
3   self.default = self.secrole.default and
4   self.isprotectedby.permissions.permission->asSet() = self.secrole.haspermission->asSet()
```

Invariante que atua sobre *JAASAction*.

```
1 context JAASAction inv AtomicActionFlattenedIsConnectedToJAASAction:
2 self.atomicaction.isFlattened() and
3   self.permissions.permission = self.atomicaction.isassigned
```

Invariante que atua sobre *ResAttribute* e *ResAssocEndSetMethod*.

```
1 context AtomicUpdate inv targetsUpdateResMethodsFromResAttributeOrResAssociationEnd:
2 self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResSetMethod)
3   or self.jaasaction.advice.pointcut.resmethod.oclIsTypeOf(ResAssocEndSetMethod)
```

Invariantes que atuam sobre *ResAttribute*.

```
1 context ResSetMethod inv targetsAtomicUpdateFromResSetMethod:
2 self.pointcut.advice.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResGetMethod inv targetsAtomicReadFromResGetMethod:
5 self.pointcut.advice.jaasaction.action.oclIsTypeOf(AtomicRead)
6
7 context ResAttribute inv ResAttributeTargetsAttributeThroughResSetMethod:
8 self.ressetmethod.pointcut.advice.jaasaction.action.resource = self.attribute
```

Invariantes que atuam sobre *ResAssocEndSetMethod*.

```
1 context ResAssocEndSetMethod inv targetsAtomicUpdateFromResSetAssociationEndMethod:
2 self.jaasaction.action.oclIsTypeOf(AtomicUpdate)
3
4 context ResAssocEndGetMethod inv targetsAtomicReadFromResGetAssociationEndMethod:
5 self.jaasaction.action.oclIsTypeOf(AtomicRead)
6
7 context ResAssociationEnd inv
      ResAssociationEndTargetsAssociationEndThroughResSetAssociationEndMethod:
8 self.resassociationendsetmethod.pointcut.advice.jaasaction.action.resource = self.associationend
```

Invariantes que atuam sobre *ResOperationMethod*

```
1 context AtomicExecute inv targetsResOperationMethod:
2 self.jaasaction.resmethod.oclIsTypeOf(ResOperationMethod)
3
4 context ResOperationMethod inv targetsAtomicExecute:
5 self.jaasaction.action.oclIsTypeOf(AtomicExecute)
```

C.4 SecureUML+ComponentUML linearizado

Nesta seção são apresentados os invariantes que são equivalentes aos invariantes de SecureUML+ComponentUML.

C.4.1 Papel padrão

Os invariantes a seguir garantem que o modelo contém um papel padrão com a semântica desejada para SecureUML; em particular, qualquer usuário está sob, pelo menos, o papel padrão.

```
1 context Role'
2
3 inv existsADefaultFlattenedRole:
4 self.allInstances()->select(r'|r'.default)->size() = 1
5
6 inv defaultPermissionAssignedToAllRoleFlattened:
7 self.haspermission->includes(p|p.default)
```

C.4.2 Permissão padrão

Os invariantes a seguir garantem que os modelos contém uma permissão padrão com a semântica proposta em SecureUML. Em particular, a permissão padrão deve somente estar relacionada ao papel padrão e deve estar relacionado a somente ações atômicas. Além disso, as ações atômicas são relacionadas a pelo menos uma permissão. Caso sejam relacionadas a mais de uma permissão, nenhuma delas será a permissão padrão.

```
1 context Permission
2
3 inv existsADefaultPermission:
4 self.allInstances()->select(p|p.default)->size() = 1
5
6 inv defaultPermissionAssignedToDefaultFlattenedRole:
7 self.default implies self.givesaccess->forAll(r'|r'.default)
8
9 inv constraintByTrue:
10 self.default implies self.isconstraintby.body = ''true''
```

```
1 context AtomicAction'
2
3 inv existsAPermission:
4 self.givesaccess->notEmpty()
5
6 inv overridingDefaultPermission:
7 self.givesaccess->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

C.4.3 Associação entre ações e recursos

Os invariantes a seguir garantem que as ações referenciam corretamente os recursos.

```
1 context AtomicCreate' inv flattenedTargetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AtomicDelete' inv flattenedTargetsAnEntity:
2 self.resource.oclIsTypeOf(Entity)
```

```
1 context AtomicUpdate' inv flattenedTargets:
2 self.resource.oclIsTypeOf(Attribute)
3 or self.resource.oclIsTypeOf(AssociationEnd
```

```
1 context AtomicRead' inv flattenedTargets:
2 self.resource.oclIsTypeOf(Attribute)
3 or self.resource.oclIsTypeOf(AssociationEnd)
```

```
1 context AtomicExecute' inv flattenedTargetsAMethod:
2 self.resource.oclIsTypeOf(Method)
```

Os invariantes a seguir garantem que os recursos estão relacionados corretamente às ações linearizadas.

```
1 context Entity inv areAccessedByFlattened:
2 self.action->size() = 2 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicCreate')) and
4 self.action->exists(a|a.oclIsTypeOf(AtomicDelete'))
```

```
1 context Attribute inv areAccessedByFlattened:
2 self.action->size() = 2 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicRead')) and
4 self.action->exists(a|a.oclIsTypeOf(AtomicUpdate'))
```

```
1 context Method inv areAccessedByFlattened:
2 self.action->size() = 1 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicExecute'))
```

```
1 context Associationend inv areAccessedByFlattened:
2 self.action->size() = 2 and
3 self.action->exists(a|a.oclIsTypeOf(AtomicRead')) and
4 self.action->exists(a|a.oclIsTypeOf(AtomicUpdate'))
```

C.5 AAC+JAAS

Nesta seção são apresentados os invariantes que são equivalentes aos invariantes de SecureUML+ComponentUML linearizado.

C.5.1 Papel padrão

```
1 context PrincipalRole
2
3 inv existsADefaultPrincipalRole:
4 self.allInstances()->select(pr|pr.default)->size() = 1
5
6 inv defaultBasic%\-%Permission%\-%Assigned%\-%ToAllPrincipal%\-%Role:
7 self.isprotectedby.permissions->includes(bp|bp.default)
```

C.5.2 Permissão padrão

```
1 context BasicPermission
2
3 inv existsADefaultBasicPermission:
4 self.allInstances()->select(p|p.default)->size() = 1
5
6 inv defaultBasic%\-%Permission%\-%Assigned%\-%To%\-%Default%\-%Principal%\-%Role:
7 self.default implies self.isprotectedby.principal->forAll(pr|pr.default)
8
9 inv basicpermissionconstraintByTrue:
10 self.default implies self.basicpermissionconstraint.body = ''true''
```

```
1 context JAASAction
2
3 inv existsABasicPermission:
4 self.basicpermission->notEmpty()
5
6 inv overridingDefaultPermission:
7 self.permissions->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

APÊNDICE D – Comparação entre os invariantes das linguagens envolvidas na transformação

Neste apêndice é apresentada uma tabela comparativa entre os invariantes de SecureUML+ComponentUML, SecureUML+ComponentUML linearizado e AAC+JAAS.

SecureUML+ComponentUML	Contrato de Transformação	AAC+JAAS
Papel padrão (*DefaulRole*)	Papel padrão (*DefaulRoleFlattened*)	Papel padrão (*DefaulPrincipalRole*)
context Role **inv** existsADefaultRole: self.allInstances()−>select(r|r.default) −>size() = 1 **inv** allRolesInheritFromDefaultRole: self.superrolePlus()−>exists(r|r. default)	**context** Role' **inv** existsADefaultRoleFlattened: self.allInstances()−>select(role'|role'. default)−>size() = 1 **inv** defaultPermissionAssignedTo- AllRoleFlattened: self.haspermission−>includes(p|p. default)	**context** PrincipalRole **inv** existsADefaultPrincipalRole: self.allInstances()−>select(pr|pr. default)−>size() = 1 **inv** defaultBasicPermissionAssigned- ToAllPrincipalRole: self.isprotectedby.permissions−> includes(bp|bp.default)
context User **inv** allUsersAssignedDefaultRole: self.hasrole−>exists(r|r.default)	Não são considerados usuários na transformação de SecureUML+ComponentUML para SecureUML+ComponentUML linearizado	Não são considerados usuários na transformação de SecureUML+ComponentUML para AAC+JAAS
Hierarquia de papéis	Não possui hierarquia de papéis	Não possui hierarquia de papéis
context Role **inv** noCyclesinRoleHierarchy: self.superrole−>forAll(r|r.superrolePlus ()−>excludes(self))		
Permissão padrão (DefaultPermission).	Permissão padrão (DefaultPermission).	Permissão padrão (DefaultBasicPermission).

context Permission **inv** existsADefaultPermission: self.allInstances()−>select(p|p.default) −>size() = 1 **inv** defaultPermissionAssignedToDefaultRole : self.default implies self.givesaccess−> forAll(r|r.default) **inv** constraintByTrue: self.default implies self.isconstraintby. **body** = "true"	**context** Permission **inv** existsADefaultPermission: self.allInstances()−>select(p|p.default) −>size() = 1 **inv** defaultPermissionAssignedToDefaultRoleFlattened : self.default implies self.givesaccess−> exists(r|r.default) **inv** constraintByTrue: self.default implies self.isconstraintby. **body** = ''true''	**context** BasicPermission **inv** existsADefaultBasicPermission: self.allInstances()−>select(p|p.default) −>size() = 1 **inv** defaultBasicPermissionAssignedTo- DefaultPrincipalRole: self.default implies self.isprotectedby. principal−>forAll(pr|pr.default) **inv** basicpermissionconstraintByTrue: self.default implies self. basicpermissionconstraint.**body** = ''true''
context CompositeAction **inv** nonDefaultPermission: self.allAssignedPermission()−>forAll(p |not(p.default))	Não possui ações compostas	Não possui ações compostas
context AtomicAction **inv** existsAPermission: self.allAssignedPermission()−> notEmpty() **inv** overridingDefaultPermission: self.allAssignedPermission()−>forAll(p1, p2| p1<>p2 implies not(p1. default))	**context** AtomicAction' **inv** existsAPermissionInFlattened: self.isassigned−>notEmpty() **inv** overridingDefaultPermission- InFlatttened: self.isassigned−>forAll(p1, p2| p1<> p2 implies not(p1.default))	**context** JAASAction **inv** existsABasicPermission: self.basicpermission−>notEmpty() **inv** overridingDefaultPermission: self.permissions−>forAll(p1, p2| p1 <>p2 implies not(p1.default))
Associação entre recursos e ações		
context AtomicCreate **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	**context** AtomicCreate' **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	Não utilizado em AAC+JAAS
context AtomicDelete **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	**context** AtomicDelete' **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	Não utilizado em AAC+JAAS
context AtomicUpdate **inv** targets: self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)	**context** AtomicUpdate' **inv** targets: self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)	Garantido pela estrutura de AAC+JAAS
context AtomicRead **inv** targets: self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)	**context** AtomicRead' **inv** targets: self.resource.oclIsTypeOf(Attribute) or self.resource.oclIsTypeOf(AssociationEnd)	Garantido pela estrutura de AAC+JAAS
context AtomicExecute **inv** targetsAMethod: self.resource.oclIsTypeOf(Method)	**context** AtomicExecute' **inv** targetsAMethod: self.resource.oclIsTypeOf(Method)	Garantido pela estrutura de AAC+JAAS

context EntityFullAccess **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	Não possui hierarquia de ações	Não possui hierarquia de ações
context EntityRead **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	Não possui hierarquia de ações	Não possui hierarquia de ações
context EntityUpdate **inv** targetsAnEntity: self.resource.oclIsTypeOf(Entity)	Não possui hierarquia de ações	Não possui hierarquia de ações
context AttributeFullAccess **inv** targetsAnAttribute: self.resource.oclIsTypeOf(Attribute)	Não possui hierarquia de ações	Não possui hierarquia de ações
context AssociationEndFullAccess **inv** targetsAnAssociationEnd: self.resource.oclIsTypeOf(AssociationEnd)	Não possui hierarquia de ações	Não possui hierarquia de ações
Relação entre recurso e ações.		
context Entity **inv** areAccessedBy: self.action−>size() = 5 and self.action−>exists(a|a.oclIsTypeOf(EntityFullAccess)) and self.action−>exists(a|a.oclIsTypeOf(EntityUpdate)) and self.action−>exists(a|a.oclIsTypeOf(EntityRead)) and self.action−>exists(a|a.oclIsTypeOf(AtomicCreate)) and self.action−>exists(a|a.oclIsTypeOf(AtomicDelete))	**context** Entity **inv** areAccessedBy: self.action−>size() = 3 and self.action−>exists(a|a.oclIsTypeOf(AtomicCreate')) and self.action−>exists(a|a.oclIsTypeOf(AtomicDelete'))	Garantido pela estrutura de AAC+JAAS
context Attribute **inv** areAccessedBy: self.action−>size() = 3 and self.action−>exists(a| a.oclIsTypeOf(AttributeFullAccess)) and self.action−>exists(a|a.oclIsTypeOf(AtomicRead)) and self.action−>exists(a|a.oclIsTypeOf(AtomicUpdate))	**context** Attribute **inv** areAccessedBy: self.action−>size() = 2 and self.action−>exists(a|a.oclIsTypeOf(AtomicRead')) and self.action−>exists(a|a.oclIsTypeOf(AtomicUpdate'))	Garantido pela estrutura de AAC+JAAS
context Method **inv** areAccessedBy: self.action−>size() = 1 and self.action−>exists(a|a.oclIsTypeOf(AtomicExecute))	**context** Method **inv** areAccessedBy: self.action−>size() = 1 and self.action−>exists(a|a.oclIsTypeOf(AtomicExecute'))	Garantido pela estrutura de AAC+JAAS

context Associationend **inv** areAccessedBy: self.action−>size() = 3 and self.action−>exists(a| a.oclIsTypeOf(AssociationEndFullAccess)) and self.action−>exists(a|a.oclIsTypeOf(AtomicRead)) and self.action−>exists(a|a.oclIsTypeOf(AtomicUpdate))	**context** Associationend **inv** areAccessedBy: self.action−>size() = 2 and self.action−>exists(a|a.oclIsTypeOf(AtomicRead)) and self.action−>exists(a|a.oclIsTypeOf(AtomicUpdate))	Garantido pela estrutura de AAC+JAAS
Hierarquia de ações	Não possui hierarquia de ações	Não possui hierarquia de ações
context EntityFullAccess **inv** containsSubactions: self.subordinatedactions = self. resource.action −>select(a|a.oclIsTypeOf(EntityUpdate)) −>union(self.resource.action −>select(a|a.oclIsTypeOf(EntityRead))) −>union(self.resource.action −>select(a|a.oclIsTypeOf(AtomicCreate))) −>union(self.resource.action −>select(a|a.oclIsTypeOf(AtomicDelete)))	-	-
context EntityRead **inv** containsSubactions: self.subordinatedactions = self.resource.oclAsType(Entity). hasattribute.action −>select(a|a.oclIsTypeOf(AtomicRead)) −>union(self.resource.oclAsType(Entity) .hasassociationend.action −>select(a|a.oclIsTypeOf(AtomicRead))) −>union(self.resource.oclAsType(Entity).hasmethod −>select(me|me.isQuery).action −>select(a|a.oclIsTypeOf(AtomicExecute)))	-	-

context EntityUpdate **inv** containsSubactions: self.subordinatedactions = self.resource.oclAsType(Entity). hasattribute.action −>select(a|a.oclIsTypeOf(AtomicUpdate)) −>union(self.resource.oclAsType(Entity) .hasassociationend.action −>select(a|a.oclIsTypeOf(AtomicUpdate))) −>union(self.resource.oclAsType(Entity).hasmethod −>select(me|not(me.isQuery)).action −>select(a|a.oclIsTypeOf(AtomicExecute)))	-	-
context AttributeFullAccess **inv** containsSubactions: self.subordinatedactions = self. resource.action −>select(a|a.oclIsTypeOf(AtomicUpdate)) −>union(self.resource.action −>select(a|a.oclIsTypeOf(AtomicRead)))	-	-
context AssociationEndFullAccess **inv** containsSubactions: self.subordinatedactions = self. resource.action −>select(a|a.oclIsTypeOf(AtomicUpdate)) −>union(self.resource.action −>select(a|a.oclIsTypeOf(AtomicRead)))	-	-

Tabela D.1: Tabela comparativa entre os invariantes do Contrato de Transformação