MARCELO SHINJI HIGASHIYAMA

JACOWEB-ABC: INTEGRAÇÃO DO MODELO DE CONTROLE DE ACESSO $UCON_{ABC}$ NO CORBASEC

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

CURITIBA

2005

MARCELO SHINJI HIGASHIYAMA

JACOWEB-ABC: INTEGRAÇÃO DO MODELO DE CONTROLE DE ACESSO $UCON_{ABC}$ NO CORBASEC

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: *Metodologia e Técnicas de Computação*

Orientador: Prof. Dr. Lau Cheuk Lung

CURITIBA

2005

Higashiyama, Marcelo Shinji

JaCoWeb-ABC: Integração do Modelo de Controle de Acesso $UCON_{ABC}$ no CORBASec. Curitiba, 2005. 111 p.

Dissertação de Mestrado – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1. $UCON_{ABC}$ 2. CORBASec 3. XACML 4. Controle de Acesso. I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada

À minha noiva,
aos nossos pais,
e aos meus irmãos.

Agradecimentos

Agradeço ao meu orientador Lau pela sua excelente orientação e paciência, acreditando em mim e agindo da maneira correta na fase final desta dissertação.

Agradeço ao Rafael Obelheiro pela grande ajuda durante a montagem de minha proposta, pois o resultado final deste trabalho se deu a muitas de suas sugestões.

Agradeço ao amigo Marcos Pchek Laureano, que me incentivou a iniciar com o meu mestrado, assim como outros amigos e colegas que tive contato durante este período.

E agradeço de coração à minha noiva Clara, pelo carinho, paciência e compreensão diante de minhas ausências para que eu pudesse concluir este trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

AC	*Access Control*
ABC	*Authorization, oBligation and Condition (Autorização, oBrigação e Condição)*
ACL	*Access Control List*
ATT(S)	*Atributos do Sujeito*
ATT(O)	*Atributos do Objeto*
CORBA	*Common Object Request Broker Architecture*
CORBASec	*CORBA Security Services*
COSS	*Common Object Services Specification*
DRM	*Digital Right Management*
DAC	*Discretionary Access Control*
DCOM	*Distributed Component Object Model*
EBAC	*Expression Based Access Control*
IDL	*Interface Definition Language*
LDAP	*Lightweight Directory Access Protocol*
MAC	*Mandatory Access Control*
OMA	*Object Management Architecture*
OMG	*Object Management Group*
ORB	*Object Request Broker*
PDP	*Policy Decision Point*
PEP	*Policy Evaluate Point*
RAD	*Resource Access Decision*
RBAC	*Role Based Access Control*
UCON	*Usage Control*
SAML	*Security Assertion Markup Language*
SSL	*Secure Socket Layer*
XACML	*eXtensible Access Control Model Language*
XEBACML	*eXtensible Expression Based Access Control Model Language*
XML	*eXtensible Markup Language*

Resumo

A arquitetura JaCoWeb-ABC é uma extensão da especificação CORBASec que aplica o modelo de controle de acesso $UCON_{ABC}$ à sua camada de segurança. O JaCoWeb-ABC define um controle de acesso configurável, que trabalha com políticas de autorização, obrigação e decisão. Ele é capaz de adaptar os modelos tradicionais como o MAC, DAC e RBAC, assim como outros modelos existentes, a partir de políticas definidas pelo administrador do sistema. Estas políticas de segurança podem ser definidas de duas formas, sendo a primeira totalmente transparente para a aplicação, nos casos em que o JaCoWeb-ABC possua todas as informações necessárias para o processo de decisão do acesso, e a segunda que possibilita trabalhar em conjunto com a aplicação, quando o controle de segurança depende de informações externas que devem ser passadas pela aplicação. A combinação destas duas funcionalidades irá definir um modelo de controle de acesso muito mais preciso e rigoroso sobre as ações dos usuários em um sistema e possibilitar o seu bloqueio em casos de identificação de práticas indevidas.

Palavras-Chave: $UCON_{ABC}$, CORBASec, XACML, Controle de Acesso

Abstract

The JaCoWeb-ABC architecture is an extension of the CORBASec application that applies the $UCON_{ABC}$ access control model to its security layer. JaCoWeb-ABC defines configurable access controls that deploy authorization, obligation and condition policies. It is capable adapting traditional models such as MAC, DAC and RBAC, as well as others existing models, based on the policies defined by the system administrator. These security policies can be defined in 2 different manners. The first one is totally transparent to the application, for cases where the JaCoWeb-ABC has all the necessary information for the access decision process, and the second one makes it possible to work together with the application, in cases where security controls depend on external information that must be submitted by the application. The combination of these two functionalities will define a much more accurate and strict access control model for the actions of users within a system, making it possible to block it in case undue practices are identified.

Keywords: $UCON_{ABC}$, CORBASec, XACML, Access Control

Capítulo 1

Introdução

1.1 Motivação

A popularização da Internet possibilitou que empresas de pequeno, médio e grande porte passassem a utilizar esta rede como um novo canal para seus negócios, oferecendo aos seus clientes serviços virtuais como e-commerce, Internet Banking, leilões, etc. Desta forma, as empresas passaram a direcionar o desenvolvimento de sistemas para aplicações do tipo cliente-servidor e buscar por soluções baseados em sistemas distribuídos, como CORBA, RMI, DCOM, WebServices, que trabalham a partir de transações iniciadas pelo cliente e executadas pelo servidor. Estes serviços cada vez mais tem sido alvo de práticas indevidas por usuários da Internet. Inicialmente alguns destes usuários, mal intencionados, tinham como objetivo apenas indisponibilizar estes serviços, mas nestes últimos tempos, esta prática tem sido direcionada para a realização de crimes digitais, levando empresas a terem enormes prejuízos e passar a investir fortemente na segurança de seus sistemas.

Muitos conceitos de segurança já foram estudados e aplicados em sistemas, como os modelos controle de acesso obrigatório (MAC) [28], discricionário (DAC) [9], baseado em papéis (RBAC) [29], assim como outros modelos de controle de acesso. Conceitos como SSL e certificados digitais também passaram a ser muito utilizados principalmente para garantir a confidencialidade, integridade e autenticidade das informações. Mesmo assim, a aplicação destes conceitos nem sempre impede que usuários do sistema efetuem procedimentos abusivos ou até impróprios, caso não existam políticas de segurança específicas para este fim.

Focando esta necessidade, Ravi Sandhu e Jae Hong Park apresentaram em seus

trabalhos [24], [25], [26] e [39] um novo modelo de controle de Acesso, nomeado $UCON_{ABC}$ (Usage Control), que define políticas de Autorizações, oBrigações e Condições, e trabalha com entidades como o Sujeito e Objeto, que possuem atributos de controle que podem ser alterados em razão de seus acessos (mutabilidade de atributos). Uma grande vantagem deste modelo é a capacidade de adaptar diversos modelos de controle de acesso existentes em suas políticas de segurança, além de possibilitar um controle mais preciso sobre as ações de um usuário.

1.2 Proposta

Este trabalho é uma continuação do projeto JaCoWeb Security [33] desenvolvido no Laboratório de Controle e Microinformática (LCMI) da Universidade Federal de Santa Catarina (UFSC)[1]. Maiores informações sobre a Família JaCoWeb Security estão explicados na seção 4.5.

O modelo CORBA de Segurança (CORBASec) [19] possui apenas uma especificação, padronizada pela OMG, que aplica o modelo de controle de acesso discricionário em sua camada de segurança. Este modelo é muito restrito e não garante um total controle sobre as ações de um usuário ao s recursos de um sistema. Desta forma, esta dissertação propõe a adaptação do modelo $UCON_{ABC}$ na especificação CORBASec, definindo um controle de acesso/uso mais rígido e rigoroso sobre as ações de um usuário, possibilitando até a sua revogação em casos identificados como impróprios no sistema.

Um grande desafio desta proposta é definir uma especificação que seja flexível o bastante para implementar a política de segurança definida no modelo $UCON_{ABC}$ na infra-estrutura do CORBASec, pois conforme descrito pelos próprios autores deste modelo [24], se trata de um modelo teórico que pode não ser implementável diretamente. A introdução do modelo $UCON_{ABC}$ permitirá à infra-estrutura CORBASec trabalhar com conceitos de autorização, obrigação, condição e também, a mutabilidade de atributos, permitindo um controle muito mais preciso e rigoroso sobre as ações de usuários em sistemas distribuídos desenvolvidos em CORBA. Para este novo modelo, também estamos propondo uma linguagem para definição de políticas de segurança, capaz de especificar as políticas do

[1] Laboratório de Controle e Microinformática (LCMI) da UFSC.
http://www.lcmi.ufsc.br

modelo $UCON_{ABC}$ de uma forma simplificada e de fácil interpretação, baseada em XML.

1.3 Organização do Trabalho

Esta dissertação está organizada da seguinte maneira: o capítulo 2 faz uma síntese dos modelos tradicionais de controle de acesso, para posteriormente apresentar os conceitos e definições modelo do $UCON_{ABC}$ [24] proposto por Jae Park e Ravi Sandhu. O capítulo 3 apresenta algumas especificações da OMG, como a arquitetura OMA, o CORBASec e o RAD, além de trabalhos relacionados. O capítulo 4 apresenta as linguagens para definições de políticas de controle de acesso e segurança, muito utilizados para representação de políticas por meio de uma linguagem, de forma a facilitar a sua compreensão e distribuição. O capítulo 5 apresenta a proposta de implementação e arquitetura do modelo ABC sobre o CORBASec. O capítulo 6 apresenta uma proposta para a criação de uma nova linguagem, nomeada como XEBACML, para a definição de políticas de controle de acesso baseada em expressões, bem como uma proposta de especificação dos tipos e funções suportadas por esta linguagem. O capítulo 7 apresenta os resultados obtidos por meio da implementação de um protótipo, com a aplicação de exemplos extraídos do $UCON_{ABC}$. O capítulo 8 apresenta as conclusões deste trabalho.

Capítulo 2

Modelos de Controle de Acesso

Este capítulo faz uma síntese dos modelos tradicionais de controle de acesso, que servirão de base para a compreensão de um novo modelo proposto por Jae Hong Park e Ravi Sandhu, chamado de Usage Control ($UCON_{ABC}$), que é um controle de acesso poderoso, capaz de adaptar todos os estes modelos, bem como definir políticas muito mais precisas sobre as ações de um usuário sobre um sistema.

2.1 Controle de Acesso Tradicional

Os modelos tradicionais de controle de acesso [24], como o modelo Discricionário (DAC) [9], Obrigatório (MAC) [28], baseado em papéis (RBAC) [29], se resumem em um Sujeito (S) com permissões de acesso a um determinado Objeto (O). Estas permissões são avaliadas a partir de atributos (ATTR) que identificam um sujeito (ATTR(S)) e um objeto (ATTR(O)), que definem os seus direitos (R) de utilização, conforme apresentado na figura 2.1.

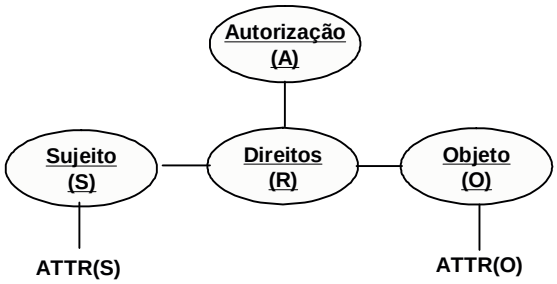

Figura 2.1 – Controle de Acesso Tradicional

Os modelos tradicionais de controle de acesso estão baseados em uma lista estática de autorizações, isto é, uma vez que um sujeito possui o acesso a um determinado objeto, não são levadas em consideração situações como a definição de uma ordem para acesso aos objetos, controles sobre um uso abusivo ou possibilitar acessos a estes objetos somente após o cumprimento de determinadas obrigações e condições no sistema. Estes controles, quando realmente necessários, geralmente são implementados e controlados dentro da própria aplicação, e que nem sempre estão codificadas de uma forma padronizada, coerente e livre de possíveis erros de implementação por parte do desenvolvedor do sistema.

2.1.1 Modelo Discricionário (DAC)

O controle de acesso discricionário (DAC) [9] é um modelo no qual os direitos que um sujeito possui sobre um determinado objeto são definidos de forma individualizada. O modelo de matriz de acesso proposto por Lampson [14] em 1971 é um modelo muito simples, em que os acessos são definidos a partir de uma matriz de acesso, onde cada linha representa um sujeito e cada coluna um objeto. O direito do acesso do sujeito ao objeto é representado pela interseção da linha com a coluna (tabela 2.1).

	O1	O2
S1	(read)	(read, write, execute)
S2	-	(write)
S3	(read, execute)	-

Tabela 2.1 – Matriz de Acesso

Uma das formas de se implementar este modelo é a partir de uma lista de controle de acesso (Access Control List – ACL) (tabela 2.2) , onde cada objeto é associado a uma ACL, que indica para cada sujeito do sistema, que direitos ele possui sobre o objeto. Outra forma é a partir de uma lista de competências (tabela 2.3), que é um modelo análogo ao ACL, só que associando para cada sujeito, os direitos que eles possuem sobre os objetos.

Objeto	Lista de Controle de Acesso (ACL)
O1	S1(read), S3(read, execute)
O2	S1(read, write, execute), S2(write)

Tabela 2.2 - Lista de Controle de Acesso (ACL)

Sujeito	Lista de Competências
S1	O1(read), O2(read, write, execute)
S2	O2(write)
S3	O1(read, execute)

Tabela 2.3 - Lista de Competências (Capability)

2.1.2 Modelo Obrigatório (MAC)

O Controle de Acesso Obrigatório (*Mandatory Access Control*—MAC) [28] é baseado na classificação de sujeitos e objetos. Cada sujeito no sistema possui um nível de habilitação (*clearance*) e cada objeto possui um nível de classificação. Esta classificação pode ser definida a partir de 4 níveis de sensibilidade, definidas como `NÃO-CLASSIFICADO`, `CONFIDENCIAL`, `SECRETO` e `ULTRA-SECRETO` (tabela 2.4).

ULTRA-SECRETO
SECRETO
CONFIDENCIAL
NÃO-CLASSIFICADO

Tabela 2.4 - Níveis de Segurança

O modelo obrigatório de Bell LaPadulla [3] visa preservar a confidencialidade das informações, definindo duas propriedades:

1) Propriedade de segurança simples (*no read up*): nenhum sujeito pode ler informações que possuam uma classificação superior ao seu nível de habilitação (*clearance*), definindo a seguinte classificação:

NÃO-CLASSIFICADO ≥ CONFIDENCIAL ≥ SECRETO ≥ ULTRA-SECRETO

Tabela 2.5 - Modelo MAC - Propriedade No Read Up

2) Propriedade estrela (*no write down*): nenhum sujeito pode escrever informações abaixo do seu nível de habilitação (*clearance*):

NÃO-CLASSIFICADO ≤ CONFIDENCIAL ≤ SECRETO ≤ ULTRA-SECRETO

Tabela 2.6 - Modelo MAC - Propriedade No Write Down

Outros modelos de controle de acesso obrigatórios também são propostos, como o modelo de integridade Biba [35] que é análogo ao Bell Lapadulla.

2.1.3 Modelo Baseado em Papéis (RBAC)

O Controle de Acesso Baseado em Papéis (Role-Based Access Control—RBAC) [10, 29] é um modelo que define os acessos aos objetos do sistema baseado nas atividades executadas pelo usuário. A definição destas atividades é dada um nome de “Papel”, e para cada papel são definidos os seus direitos sobre o sistema. Desta forma, os direitos não são mais definidos por usuário, mas sim, para cada usuário associa-se um papel em razão de suas responsabilidades. A partir deste modelo, surgiram várias propostas sobre a sua utilização e funcionalidades. Observando o potencial deste modelo, o NIST (*National Institute of Standards and Technology*) viu a necessidade de padronizá-lo. Em [10], é apresentada uma proposta para a padronização do RBAC (NIST-RBAC). O modelo Core RBAC, que é o modelo básico e central.

O Core RBAC define um modelo mínimo de componentes para a utilização do RBAC. Ele é composto por usuários (USERS), papéis (ROLES), objetos (OBS), operações (OPS) e permissões (PRMS). A figura 2.2 apresenta este modelo, onde um usuário é associado a um ou mais papéis (UA), e o papel é associado a uma ou mais permissões (PA), que define quais operações um usuário poderá efetuar sobre um objeto.

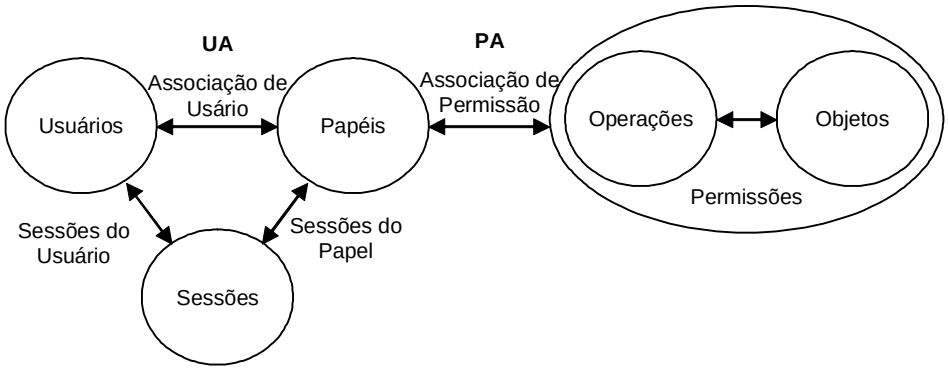

Figura 2.2 - Core RBAC

Os outros 3 modelos RBAC propostos pelo NIST são ([11]):

- RBAC Hierárquico: trabalha com uma hierarquia, definindo uma relação de herança entre os papéis.
- Separação Dinâmica de Responsabilidades: define relações de exclusividade de papéis, não permitindo que um usuário ative dois papéis em uma mesma sessão. Exemplo: nenhum usuário de um sistema pode ativar ao mesmo tempo os papéis de caixa e gerente, por mais que o usuário esteja habilitado aos dois papéis.
- Separação Estática de Responsabilidades: visa tratar situações de conflitos de interesses, não permitindo que um usuário possua dois papéis conflitantes em um mesmo sistema. Exemplo: nenhum usuário do sistema pode possuir os papéis de caixa e gerente em um sistema, forçando que o usuário possua um e apenas um papel no sistema.

2.2 O Modelo $UCON_{ABC}$ – Usage Control Model

No modelo Usage Control (UCON), chamado $UCON_{ABC}$, proposto por Jaehong Park e Ravi Sandhu em [24, 25, 26, 39], são definidas políticas de *autorização*, *obrigação* e *condição* (ABC), capazes de adaptar diversos modelos de controle de acesso em um único modelo, resultando em um controle muito mais preciso e rigoroso sobre as ações que um sujeito possui sobre um determinado objeto. Uma característica importante é que os direitos de acesso de um sujeito sobre um objeto são determinados em tempo de execução, e podem ser alterados conforme as ações do usuário dentro do sistema.

2.2.1 Componentes do modelo $UCON_{ABC}$

O modelo $UCON_{ABC}$ é composto de oito componentes: *Sujeitos* (S), *Atributos do Sujeito* (ATT(S)), *Objetos* (O), *Atributos do Objeto* (ATT(O)), *Direitos* (R), *Autorizações* (A), *oBrigações* (B), *Condições* (C), que estão representados na figura 2.3.

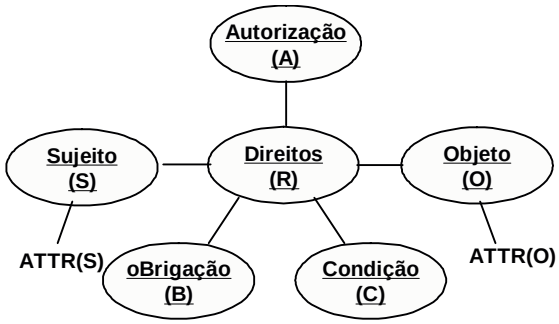

Figura 2.3- Modelo de Controle de Acesso ABC

Os conceitos de Sujeitos e Objetos são os mesmos dos modelos de controle de acesso tradicionais. Direitos habilitam o acesso de um sujeito a um objeto. As principais características do modelo $UCON_{ABC}$ estão relacionadas à mutabilidade dos atributos, em que o direito não é avaliado em uma matriz estática de acesso. A decisão é efetuada em tempo de execução no momento do acesso do sujeito ao objeto. A seguir, estarão sendo apresentados mais detalhes de cada um destes componentes.

2.2.2 Sujeitos (S) e Atributos do Sujeito (ATT(S))

O Sujeito é uma entidade associada com atributos, e que exercem certos direitos sobre objetos. No modelo UCON, um sujeito pode ser observado como sendo um indivíduo humano ou uma entidade que realiza uma requisição de acesso (usuário, processo, máquina, etc). Este sujeito é definido e representado por seus atributos. Os Atributos do Sujeito (ATT(S)) são propriedades que podem ser usados para o processo de decisão de acesso.

Na prática, um dos mais importantes atributos do sujeito é a sua identificação, mas ela não é exigida pelo modelo $UCON_{ABC}$. A identificação do sujeito é muito importante para determinar um acesso em controles tradicionais. Entretanto, exigir a identificação do sujeito como um atributo obrigatório impede serviços anônimos. O conceito geral de controle de acesso baseado em atributos é comum na literatura, conforme os modelos de controle de acesso tradicionais, e este aspecto no ABC é construído sobre este modelo.

Se um atributo é imutável, ele não pode ser alterado pelas atividades de um usuário. Somente ações administrativas podem mudar este atributo. Exemplos de atributos imutáveis podem ser as permissões de acesso a arquivos de um sistema Unix, Linux ou Windows. Um atributo mutável pode ser modificado no acesso de um sujeito aos objetos do sistema. Um exemplo deste tipo de atributo pode ser a quantidade de erros de senha que um usuário cometeu em transações de saque em ATMs pelo uso de um cartão de débito.

2.2.3 Objetos (O) e Atributos do Objeto (ATT(O))

Objetos são entidades os quais um sujeito possui direitos determinados de acesso. Assim como os Sujeitos, os objetos também possuem atributos associados que podem ser utilizados em um processo de decisão de acesso. Os seus atributos também podem ser imutáveis ou mutáveis. Exemplos de atributos imutáveis pode ser, o “nível de classificação” definido para um Objeto no modelo MAC, ou a sua “Lista de Controle de Acesso (ACL”) definida pelo modelo DAC. Um exemplo de um atributo mutável pode ser um “contador” que é incrementado a cada usuário que efetua um acesso a uma determinada sala de Bate-Papo na Internet, controlada por um atributo imutável que define a quantidade máxima de usuários simultâneos a este serviço.

2.2.4 Direitos (R)

Direitos (R) são privilégios que um sujeito pode exercer sobre um objeto. Os Direitos consistem na definição de funções que habilitam um sujeito a acessar um objeto. Do ponto de vista do controle de acesso tradicional, direitos habilitam o acesso de um sujeito a um objeto, em um modo particular, como leitura ou gravação. Uma característica importante do modelo $UCON_{ABC}$ é que neste os atributos são mutáveis, dependendo da atividade dos sujeitos no sistema. Isso significa que a decisão de autorização é efetuada em tempo de execução, no momento do acesso do sujeito ao objeto, caracterizando desta forma, o controle de uso. As funções de decisão de acesso são determinadas a partir de avaliações dos atributos do sujeito, atributos do objeto, autorizações, obrigações e condições, que serão mais bem detalhadas no decorrer deste capítulo.

2.2.5 Autorização (A)

Autorizações (A) são exigências que devem ser satisfeitas antes (pre-authorization ou preA) ou durante (ongoing-authorization ou onA) a decisão do acesso de um Sujeito sobre um

Objeto. Certas autorizações podem exigir atualizações de atributos do sujeito ou objeto antes (pre), durante (ongoing) ou depois (post) do acesso ao Objeto. Exemplos de autorização antes (preA) do acesso podem ser os próprios modelos de controle de acesso tradicionais, que avaliam se o sujeito pode ou não acessar o objeto. Exemplos de autorizações durante o acesso (onA) podem ser avaliações periódicas feitas durante o acesso do sujeito a um serviço cobrado por minuto na Internet, em que um sujeito possui um atributo de créditos que são decrementados a cada minuto e após estes créditos serem totalmente consumidos, o acesso do sujeito poderá ser cancelado, caso não haja uma recarga de créditos.

2.2.6 oBrigação (B)

Obrigações (B) são exigências que um sujeito deve exercer antes (preB) ou durante (onB) o acesso a um determinado objeto. A medida que o sujeito cumpre suas obrigações no sistema, o acesso a novos objetos poderão ser concedidos. Geralmente as obrigações tratam-se de um histórico dos acessos de um usuário sobre um sistema. Desta forma, as obrigações que um usuário cumprir poderão determinar os seus acessos futuros dentro do sistema. Um exemplo de obrigação antes do acesso (preB) pode ser a obrigação do usuário informar um e-mail antes de acessar um artigo de uma empresa. Um exemplo de obrigação durante o acesso (onB) seria obrigar um usuário a permanecer com uma janela de propaganda aberta durante o seu acesso a um determinado site. Caso o usuário feche esta janela de propaganda, o acesso ao site também poderá ser cancelado.

2.2.7 Condição (C)

Condições (C) são fatores externos relacionados ao ambiente do sistema que podem influenciar sobre o acesso do usuário sobre um objeto, que não necessitam estar relacionados com os atributos do sujeito e objeto. Estes fatores de decisão podem ser antes (preC) ou durante (onC) o acesso do sujeito sobre o objeto. Na definição do modelo $UCON_{ABC}$ entretanto, Condições não podem alterar os atributos do sujeito e do objeto. Elas apenas avaliam se as exigências foram satisfeitas. Um exemplo de condição antes do acesso (preC) seria definir que um sujeito somente poderia acessar um determinado site na Internet durante o horário comercial, de segunda a sexta-feira, das 9:00 às 18:00. Um exemplo de condição durante o acesso (onC) seria avaliar se os acessos que o usuário está efetuando após sua identificação no sistema estão sendo feitos apenas no horário comercial.

2.3 Os modelos centrais da família ABC

Durante o processo de decisão do acesso, feito antes (pre) ou durante (ongoing) o acesso do sujeito ao objeto, o modelo $UCON_{ABC}$ permite definir políticas de atualização de atributos do sujeito e atributos do objeto, que pode ser antes (pre-Update), durante (ongoing-Update) ou depois (pos-Update) da sua utilização, conforme apresentado na figura 2.5.

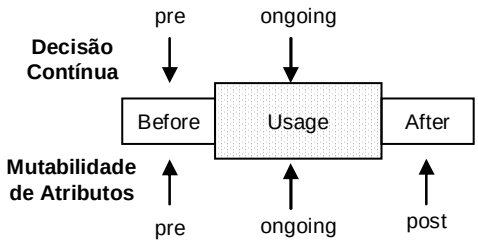

Figura 2.4 - Continuity and Mutability Properties

As diferentes combinações de momento de decisão (antes do acesso, durante o acesso) e de atualização de atributos (antes, durante ou após o acesso) geram uma família de 16 modelos possíveis (tabela 2.7). Essa variedade de modelos possibilita a definição de políticas bastante flexíveis, que vão de modelos clássicos de controle de acesso (DAC, MAC, RBAC) a DRM (Digital Rights Management) e pay-per-use [24].

	(0)	**(1)**	**(2)**	**(3)**
preA	**Y**	**Y**	N	**Y**
onA	**Y**	**Y**	**Y**	**Y**
preB	**Y**	**Y**	N	**Y**
onB	**Y**	**Y**	**Y**	**Y**
preC	**Y**	N	N	N
onC	**Y**	N	N	N

(0) - Imutable; (1)-pré-Update; (2)-on-Update; (3)-pos-Update

Tabela 2.7 - 16 modelos básicos do ABC

Em [24], são apresentados as 16 definições da família ABC e 21 exemplos que apresentam as possibilidades de adaptar o modelo $UCON_{ABC}$ a diversos modelos de controle de acesso existentes, como os modelos MAC, DAC, RBAC. Outros controles também são apresentados, como o DRM, pay-per-use, aceitação de licenças, controles médicos, etc. A família ABC permite a combinação de um ou mais modelos, o que possibilita a definição de

políticas de segurança muito mais rigorosas, flexíveis e objetivas. O resultado é um maior controle sobre as ações de um usuário sobre um sistema.

2.4 Definições utilizadas pelo $UCON_{ABC}$

As definições apresentadas pelo $UCON_{ABC}$ estão representadas abaixo:

- $preA_0$, $preA_1$, $preA_3$, onA_0, onA_1, onA_2, onA_3, $preB_0$, $preB_1$, $preB_3$, onB_0, onB_1, onB_2, onB_3, $preC_0$, onC_0: Família ABC, composta por os 16 modelos básicos.
- S, O, R, ATT(S), ATT(O): Sujeito, Objeto, Direitos, Atributos do Sujeito e Atributos do Objeto respectivamente.
- preUpdate, onUpdate, posUpdate: procedimentos para atualização de atributos do sujeito e objeto, antes, durante ou depois do acesso do sujeito ao objeto.
- allowed, stopped: avaliam expressões lógicas sobre os atributos do sujeito e objeto, retornando os valores true ou false, que irão definir a permissão do acesso ao objeto.
- OBS, OBO, OB: obrigação do sujeito, obrigação do objeto e ação da obrigação, respectivamente.
- preOBL, onOBL: elementos de obrigação.
- getPreOBL, getOnOBL: função que seleciona a obrigação a ser realizada.
- getFulFilled: função que avalia se a obrigação requisitada foi realizada (retorna true ou false).
- preCON, onCON: elementos da condição.
- getPreCON, getOnCON: seleciona os elementos da condição.
- preConChecked, onConChecked: função que avalia se a condição foi realizada (retorna true ou false).

No *Capítulo 7*, estaremos apresentando alguns exemplos de aplicação do modelo $UCON_{ABC}$ apresentados em [24], associando estes exemplos com a definição das políticas definidas em nossa proposta, para uma melhor compreensão deste trabalho. Nem todas as definições das políticas de autorização, obrigação e condição serão apresentadas, mas elas darão uma idéia clara de suas funcionalidades. As definições dos modelos básicos do $UCON_{ABC}$, bem como exemplo de aplicação de cada um destes modelos estão melhor detalhados no *Apêndice A*.

2.5 Conclusões sobre o Capítulo

O modelo $UCON_{ABC}$ é uma nova geração de controle de acesso, que permite um controle mais preciso e rigoroso sobre a utilização de usuários aos objetos de um sistema. Ele permite adaptar diversos modelos de controle de acesso existentes, a partir de suas definições e possibilitando combinar os seus 16 modelos básicos para atender as necessidades de segurança do sistema. O $UCON_{ABC}$ é um modelo teórico, que conforme os seus autores, é um modelo que não pode ser implementado diretamente, pois ele possui muitas definições baseadas em expressões lógicas, matemáticas e de atribuição, que necessitam ser abstraídas para possibilitar sua implementação em sistemas reais.

Devido ao $UCON_{ABC}$ ser um modelo recente, em nossas pesquisas ainda não encontramos implementações deste modelo em sistemas reais. Em [12], é avaliado que o $UCON_{ABC}$ [26] por se tratar simplesmente de um modelo teórico que ainda não possui protótipos implementados, seria significativamente caro e complexo de ser implementado em sistemas reais.

Capítulo 3

Linguagens para Definição de Políticas de Segurança

As linguagens para definição de políticas de segurança possuem uma ampla área de pesquisa, que tem como objetivo representar políticas que muitas vezes possuem conceitos abstratos (exemplo: controles de acesso baseados em expressões, matrizes, tabelas, definições específicas, etc) em uma linguagem capaz de ser compreendida, interpretada e implementada por ferramentas. Estaremos apresentando algumas destas linguagens utilizadas para definição de políticas de segurança, como Ponder [8] e XACML [15, 30, 31], que são linguagens que definem políticas de controle de acesso e que são aplicadas em ferramentas, produtos ou soluções. Um dos objetivos deste capítulo é apresentar os conceitos utilizados por estas linguagens, que muitas vezes podem definir uma mesma política de controle de acesso de maneiras diferentes, mas que de certo modo possuem certa semelhança com relação ao objetivo final.

3.1 Sobre as Linguagens para Definição de Políticas de Segurança

Antes de apresentar as linguagens utilizadas para definição de políticas de segurança, é importante avaliarmos algumas semelhanças entre estas linguagens, bem como definições de segurança suportadas por estas linguagens. De uma maneira geral, podemos fazer uma analogia com o modelo de controle de acesso $UCON_{ABC}$[24], pois estas linguagens também trabalham com Sujeitos que possuem Direitos de acesso sobre determinados Objetos (figura 3.1).

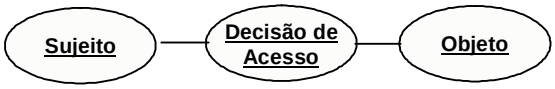

Figura 3.1 - Objetivo de uma linguagem para definição de políticas de segurança

Estas linguagens definem políticas para proteção do objeto ou recurso de um sistema, definindo quais sujeitos estão autorizados a acessar determinado objeto, baseando-se em usuários, grupos de usuários, papéis, etc. Estas políticas também possibilitam definir condições de acesso, como por exemplo, em quais horários o acesso pode ser permitido ou até restringir o acesso por localidade, geralmente utilizando-se o IP ou rede de origem do acesso. Estas linguagens também podem possibilitar a definição de políticas para controle de auditoria, logs de segurança, emissão de alertas, etc.

Um ponto importante a ser relatado é que nem todas as políticas de segurança definidas por estas linguagens tem como objetivo avaliar uma decisão de acesso para permitir o acesso a um determinado recurso, mas muitas das definições estão relacionadas a políticas de segurança, que se forem válidas, podem bloquear ou revogar o acesso de um determinado usuário no sistema. Exemplo: o modelo Ponder [8] possibilita definir uma política de segurança, que avalia se um usuário errar 3 vezes consecutivas a sua senha de login, efetuar o bloqueio deste usuário do sistema.

Estas linguagens têm como uma importante característica definir políticas de segurança independentes de plataforma ou aplicação, possibilitando que uma mesma política de segurança seja utilizada em ambientes heterogêneos, e facilitando também a distribuição destas políticas entre diversos recursos, (ex.: objetos, equipamentos, sistemas, etc) existentes em uma rede.

3.2 Linguagens existentes

3.2.1 Ponder

O Ponder [8] é uma linguagem declarativa, orientada a objetos utilizada para especificar políticas de segurança, que podem ser gerenciadas e distribuídas em um ambiente de rede. As políticas de autorização do Ponder podem ser implementadas utilizando vários mecanismos de controle de acesso como firewalls, roteadores, sistemas operacionais, banco

de dados, aplicações Java, etc. O Ponder trabalha com políticas de autorização positivas e negativas, ou seja, políticas positivas são avaliações de decisão que se forem válidas, irão permitir o acesso, e políticas negativas são avaliações de decisão que se forem válidas, irão negar o acesso a um determinado recurso. O Ponder também suporta políticas de obrigação, que são eventos que definem regras de uma condição, que se for válida, é seguida por uma ação. Pode ser utilizado para gerenciar atividades de segurança, como registros de usuários ou eventos, logs de auditoria para relacionar com acessos a recursos críticos ou violações de segurança. Ele provê um framework comum e unificado capaz de especificar políticas de segurança em plataformas heterogêneas. Os conceitos chaves desta linguagem incluem: *domínios* para grupos de objetos os quais as políticas podem ser aplicadas, *regras* que definem as políticas de segurança relacionadas a um determinado grupo, e *relacionamentos* que definem quais regras devem ser associadas a um determinado grupo de objetos.

Os domínios definem grupos de objetos para os quais políticas podem ser aplicadas e podem ser utilizados determinados tipos de objetos em diversos ambientes, de acordo com seus limites geográficos, tipos de objeto, responsabilidade e autoridade. As Políticas nativas do Ponder definem que atividades um membro do domínio pode exercer sobre um conjunto de objetos. Estes são essencialmente políticas de controle de acesso, que protegem recursos e serviços de acesso não autorizados. Uma política de autorização positiva define quais são as ações permitidas que um sujeito possui sobre objetos e uma política de autorização negativa especifica as ações que serão negadas caso o sujeito tente exercer sobre estes objetos.

A linguagem para definição de políticas de autorização define o nome da política, e por meio de parâmetros, indica quais são os sujeitos (ou grupos de sujeitos), objetos (ou grupos de objetos) e ações sobre o objeto serão permitidas por meio de uma autorização positiva, suportando múltiplas definições. A figura 3.2, são definidas duas políticas de autorização positivas que autorizam os Sujeitos do grupo *NetworkAdmins* e *QoSAdmins* a executar as operações *load, remove, enable, disable* sobre objetos do tipo PolicyT, definido como Nregion/Switches e Nregion/Routers.

```
type auth+ PolicyOpsT (subject s, target <PolicyT> t) {
      action load(), remove(), enable(), disable() ; }
      inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, Nregion/switches);
```

```
inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /Nregion/routers);
```

Figura 3.2 – Política de autorização positiva do Ponder

A figura 3.3 define uma política de autorização negativa, que proíbe que as ações *performance_test* sejam executadas pelos sujeitos pertencentes ao grupo *testEnginners ou trainee*, sobre objetos do tipo <routerT>/routers (roteadores) entre 9:00 e 17:00.

```
inst auth– /negativeAuth/testRouters {
     subject /testEngineers/trainee ;
     action performance_test() ;
     target <routerT> /routers ;
     when time.between ("0900", "1700")
}
```

Figura 3.3 - Política de autorização negativa do Ponder

O Ponder suporta também uma definição para especificação de políticas de segurança de obrigação positivas ou negativas, que definem ações que caso sejam válidas, irão executar uma determinada ação. As políticas de obrigação permitem uma filtragem de informação que poderão ser usadas para transformar parâmetros de entrada ou saída em uma interação. Esta característica permite que ações sejam tomadas pelo sistema, de forma a identificar anormalidades, possibilitando ativar logs de auditoria, desativar usuários de um sistema, etc. O exemplo abaixo (figura 3.4) mostra uma política de obrigação que define que, caso um usuário tente efetuar 3 logins inválidos, a sua conta seja automaticamente bloqueada.

```
inst oblig loginFailure {
    on 3*loginfail(userid) ;
    subject s = /NRegion/SecAdmin ;
    target <userT> t = /NRegion/users ^ {userid} ;
    do t.disable() -> s.log(userid) ;
}
```

Figura 3.4 - Políticas de obrigação do Ponder

A linguagem de definição de políticas do Ponder permite a definição de papéis e também uma combinação de políticas de autorização e obrigação, conforme mostrada na figura 3.5.

```
type role ServiceEngineer (CallsDB callsDb) {
     inst oblig serviceComplaint {
           on customerComplaint(mobileNo) ;
           do t.checkSubscriberInfo(mobileNo, userid) ->
              t.checkPhoneCallList(mobileNo) ->
              investigate_complaint(userId);
```

```
            target t = callsDb ; // calls register }
        inst oblig deactivateAccount { . . . }
        inst auth+ serviceActionsAuth { . . . }
        // other policies
}
```

Figura 3.5 - Combinação de Políticas do Ponder

Com a utilização de uma linguagem como o Ponder, é possível distribuir determinado grupo de políticas à recursos de uma rede (firewalls, roteadores, aplicações, etc), de forma a refletir a sua estrutura organizacional, facilitando a sua administração com a padronização de políticas que possuem uma linguagem de definição comum.

3.2.2 XACML

O XACML (*eXtensible Access Control Markup Language*) [15, 30, 31] é uma linguagem, open-souce, padronizada pela OASIS (*Organization for the Advancement of Structured Information Standards*) [16] para definição de políticas de controle de acesso baseada em XML [32]. O XACML descreve uma linguagem de políticas de controle de acesso baseada em requisições e respostas (ambas codificadas sobre XML). A linguagem para definição de políticas do XACML é usada para descrever requisitos de controle de acesso, e por se tratar de um padrão aberto, possuem pontos para extensão, possibilitando que um desenvolvedor defina novas funções, tipos de dado, combinações lógicas, etc. A linguagem de requisição/resposta facilita uma avaliação se as ações requisitadas pelo cliente podem ou não ser permitidas, cabendo à aplicação interpretar que ação deve ser tomada baseado no resultado da resposta. A resposta sempre inclui um retorno sobre se a requisição foi permitida, usando quatro valores: Permitido, Negado, Indeterminado (ocorreu um erro na interpretação ou faltou informar algum valor requerido, e desta forma a decisão não pode ser concluída) ou Não Aplicável (a ação solicitada não é válida).

Uma configuração típica para implementação e utilização do XACML é quando se quer controlar o acesso de usuários a recurso de um sistema (ex.: um arquivo, transação, serviço web, etc). Nesta situação será utilizado o PEP (*Policy Enforcement Point)*. O PEP formará uma requisição baseada nos atributos do requisitante, o recurso em questão, a ação, e outras informações pertinentes à requisição. O PEP irá então enviar a requisição para o PDP (*Policy Decision Point)*, que irá avaliar a requisição, identificar quais são as políticas que se

aplicam à requisição, retornando o resultado desta avaliação. A resposta é retornada para o PEP, que poderá então permitir ou negar o acesso ao requisitante.

Abaixo, é apresentado um exemplo uma política em XACML sobre uma Requisição (Request.xml), uma Política (Policy.xml) e uma Resposta (Response.xml), que está representado na figura 3.6 (extraído de [15]). Neste exemplo, um usuário tenta acessar uma página na Web. A requisição (Response.xml) inclui a identidade do sujeito, o grupo ao qual ele pertence, o recurso que ele deseja acessar e a ação sobre este recurso. A política (Policy.xml) define que qualquer um pode executar qualquer ação sobre o recurso, e contém uma regra que se refere a uma ação específica. A condição (Condition) define uma regra que apenas certos membros de um grupo podem acessar o recurso. A resposta (Response.xml) retorna o resultado desta decisão do acesso.

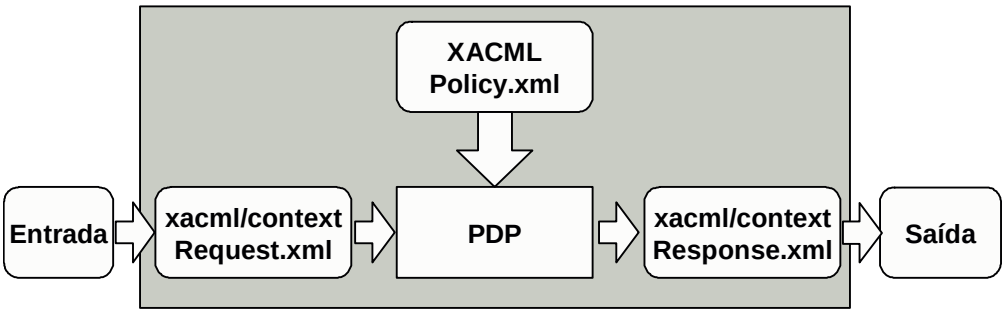

Figura 3.6 – XACML

Requisição (Request.xml):

```
<Request>
  <Subject>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
               DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name">
        <AttributeValue>seth@users.example.com</AttributeValue>
    </Attribute>
    <Attribute AttributeId="group"
               DataType="http://www.w3.org/2001/XMLSchema#string"
               Issuer="admin@users.example.com">
      <AttributeValue>developers</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
               DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      <AttributeValue>
           http://server.example.com/code/docs/developer-guide.html
      </AttributeValue>
```

```
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
               DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
</Request>
```

Figura 3.7 - Exemplo de Requisição em XACML

A Política (Policy.xml)

```
<Policy PolicyId="ExamplePolicy" RuleCombiningAlgId=
          "urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
                  http://server.example.com/code/docs/developer-guide.html
            </AttributeValue>
            <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
                  AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <Rule RuleId="ReadRule" Effect="Permit">
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValueDataType="http://www.w3.org/2001/XMLSchema#string">read
            </AttributeValue>
            <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
```

```
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
        <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="group"/>
      </Apply>
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">developers
      </AttributeValue>
    </Condition>
  </Rule>
</Policy>
```

Figura 3.8 - Exemplo de uma Política em XACML

Resposta (Response.xml)

```
<Response>
  <Result>
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

Figura 3.9 - Exemplo de uma resposta em XACML

Existem muitas linguagens para definição de políticas de segurança proprietárias ou específicas para determinadas aplicações, mas o XACML possui muitos pontos a seu favor [30]:

- É padronizado. Utilizando uma linguagem padrão, está se utilizando de um conceito que está sendo revisado continuamente por uma grande comunidade de usuários e estudiosos. Como XACML está sendo cada vez mais utilizado, ele será facilmente incorporado por outras aplicações que utilizam os mesmos padrões de linguagem.
- É genérico. Isso significa que ele pode prover um controle de acesso para um ambiente particular ou um recurso específico, podendo ser utilizado em qualquer ambiente. Uma política pode ser definida uma única vez, e ser utilizada por diferentes tipos de

aplicações. E quando uma linguagem comum e padronizada é utilizada, o gerenciamento destas políticas é facilitado.

- Ele é distribuído. Isso significa que uma política pode ser definida em um único ponto, tornando-se referência em localizações arbitrárias. O resultado é que preferencialmente também permite controlar uma política única, diferentes pessoas ou grupos podem gerenciar separadamente sub-políticas como apropriadas, e o XACML trabalha combinando corretamente o resultado destas diferentes políticas em uma única decisão.
- É poderoso. A base da linguagem permite ser extendida. A padronização da linguagem já suporta uma abrangente variedade de tipos de dados, funções e regras sobre a combinação de resultados de diferentes políticas. Além disso, já existem grupos de padronização sobre as extensões que definem a utilização do XACML para trabalhar em conjunto com outros padrões como o SAML e LDAP.

Existem outras especificações baseadas no XACML divulgada pela OASIS, como em [1], que define a implementação do modelo de controle de acesso RBAC para o XACML.

3.2.3 Outras linguagens

Rei: A linguagem Rei [13] define políticas para permissões, revogações, obrigações e dispensas. A linguagem consiste de algumas definições que são extremamente flexíveis e permitem especificar diferentes tipos de políticas de segurança (segurança, privacidade, gerenciamento, etc). A linguagem das políticas não é amarrada a uma aplicação específica, possibilitando que estas políticas sejam definidas para ambientes heterogêneos. Esta linguagem também procura tratar conflitos de políticas de segurança, como por exemplo, um determinado objeto possui duas políticas de controle de acesso, sendo a primeira permitindo que o sujeito A acesse o objeto B, e a segunda que nega o acesso do sujeito A ao objeto B. Para isso, a linguagem define duas idéias para especificação de políticas que são utilizadas para resolver estes conflitos: a modalidade preferencial (negativa sobre positiva ou vice-versa)

ou determinar a prioridade entre políticas (ex.: é possível definir que em casos de conflitos, uma política Federal sempre sobreponha uma política Estadual).

Firmato: O Firmato [2] é uma linguagem utilizada para a definição de políticas de segurança para especificação de regras de Firewalls. Esta linguagem foi criada com objetivo de separar a especificação da política de segurança, de uma forma padronizada para equipamentos de Firewall e Roteadores de diferentes fornecedores. Isso permite definir as políticas de segurança independentes da topologia de rede, gerar arquivos de configuração de firewalls automaticamente a partir de políticas de segurança. Desta forma, esta solução permite uma distribuição de uma única mesma política sobre equipamentos de rede de diferentes fornecedores (exemplo: gateways). Ele permite a especificação de objetos e serviços utilizados especificamente para equipamentos de rede, como suas terminologias como protocolos, serviços, regras, interfaces, IPs, etc. E a partir desta linguagem, definiu-se um processo para a compilação destas políticas para possibilitar a geração de arquivos de configuração.

3.3 Conclusão

As linguagens para definição de políticas de segurança possibilitam definir políticas de controle de acesso de uma forma padronizada, independente de equipamento, sistema operacional e aplicação. Desta forma, uma mesma política de segurança pode ser distribuída e utilizada em ambientes heterogêneos, indiferente de fornecedor.

A linguagem Ponder [8] possui um conceito muito interessante, muito voltado à distribuição de políticas de controle de acesso sobre um ambiente amplo e heterogêneo. No Ponder, os conceitos de Sujeito e Objeto possuem muita similaridade com o modelo $UCON_{ABC}$, trabalhando com políticas de pré-autorização e pré-condição. Uma diferença entre estes modelos está na obrigação, pois uma obrigação no modelo Ponder se trata de uma condição que se for válida, é seguida por uma ação que pode ser o registro de um log de eventos de auditoria ou a emissão de um alerta de segurança, diferente do $UCON_{ABC}$ [24], em que uma obrigação se trata de uma exigência anterior baseada geralmente em um histórico de acessos.

A linguagem XACML [15] define políticas de segurança de uma forma padronizada, no formato XML [32], e por se tratar de um padrão aberto, caso alguma de suas especificações não atenda às suas necessidades, ela permite que o próprio desenvolvedor defina e implemente novas funções de controle de acesso que sejam interpretadas pela sua camada de decisão PDP (*Policy Decision Point*). Ele é um padrão muito utilizado, de fácil utilização. Uma de suas desvantagem fica por conta da definição de suas políticas, que muitas vezes necessita efetua0r um controle simples, como controlar o acesso a recursos em horários pré-determinados (ex.: acessos negados entre 9:00 e 17:00), e que resultam em uma política extensa, contendo muitas linhas codificadas em XML. Também possui algumas similaridades com o modelo $UCON_{ABC}$, trabalhando com sujeitos e objetos. Ainda não possui um controle de obrigações e também não define políticas para as questões relacionadas à mutabilidade de atributos.

Não identificamos em nossas pesquisas, linguagens para definição de políticas de controle de acesso baseadas no $UCON_{ABC}$, talvez porque o principal foco de todas estas linguagens sempre foi abstrair as políticas de controle de acesso baseadas muitas vezes em expressões, vetores e matrizes, para uma linguagem de mais fácil implementação. E o $UCON_{ABC}$ define políticas de controle de acesso totalmente baseadas em expressões lógicas, matemáticas, trabalhando com conjuntos, vetores e matrizes, aumentando uma complexidade sobre a abstração de suas funcionalidades em uma linguagem capaz de ser interpretada por um processo de decisão (conforme apresentado no Capítulo 2 e Apêndice A). Para isso, seria necessário criar um compilador e interpretador de expressões lógicas, matemáticas e de atribuição, sendo uma grande dificuldade sobre a camada de controle de acesso.

Capítulo 4

Modelo CORBA de Segurança

O padrão CORBA, definido pela OMG (*Object Management Group*), especifica uma arquitetura de software que suporta aplicações distribuídas e garante a interoperabilidade entre diferentes plataformas de hardware, sistemas operacionais [22]. A especificação CORBA 1.1 definiu a linguagem de definição de interfaces IDL (*Interface Definition Language*) e as Interfaces Programáveis de Aplicações - API (*Application Programming Interfaces*) que possibilitam a interação cliente-servidor via objetos com uma implementação específica de um ORB (*Object Request Broker*). A IDL possibilita definir a interface do método de um objeto implementado em um servidor, que será invocado a partir de uma aplicação cliente de uma forma transparente para a aplicação. A especificação CORBA 2.0 [20] passou por um grande processo de revisão, onde foram adicionadas diversas novas características, entre elas, a especificação do serviço de segurança CORBA (CORBASec).

4.1 A arquitetura OMA

Para compreender o funcionamento do CORBA, é necessário ter conhecimento sobre a arquitetura OMA [21, 22], arquitetura de referência utilizada em todas as especificações padronizadas pela OMG. O OMA é uma especificação para diversos serviços utilizados na implementação de sistemas distribuídos. Ele é composto por 5 componentes [34, 35], conforme apresentado na figura 4.1:

- Object Request Broker (ORB): é o componente central da arquitetura OMA, sendo equivalente à arquitetura CORBA. É um componente básico responsável pela

comunicação dos objetos, para possibilitar um processo de requisição e resposta, por meio de uma invocação remota do método (requisição) de um objeto, que não necessariamente precisa estar localizado no mesmo servidor, de uma forma totalmente transparente para a aplicação.

- Serviços de Objeto (COSS): são serviços (interfaces e objetos) independentes do domínio da aplicação, fundamentais para o desenvolvimento de aplicações constituídas de objetos distribuídos ou também possibilitando uma interoperabilidade das aplicações. Estes serviços trabalham em conjunto com o ORB, fornecendo uma infra-estrutura necessária às aplicações.
- Facilidades Comuns: assim como o COSS, são objetos que podem ser utilizados pelas aplicações, por meio de interfaces, que podem ser compartilhadas por várias aplicações, mas que ao contrário do COSS, trabalham em um nível mais alto de implementação, mais próxima do usuário.
- Interfaces de domínio: possuem a mesma função dos serviços de objetos e facilidades comuns, porém são interfaces orientadas a domínios de aplicação específicos, como por exemplo telecomunicações e finanças.
- Objetos de aplicação: os objetos de aplicação são implementadas especificamente para uma determinada aplicação. Como estes objetos representam a aplicação propriamente dita, as interfaces destes objetos não são padronizadas pela OMG.

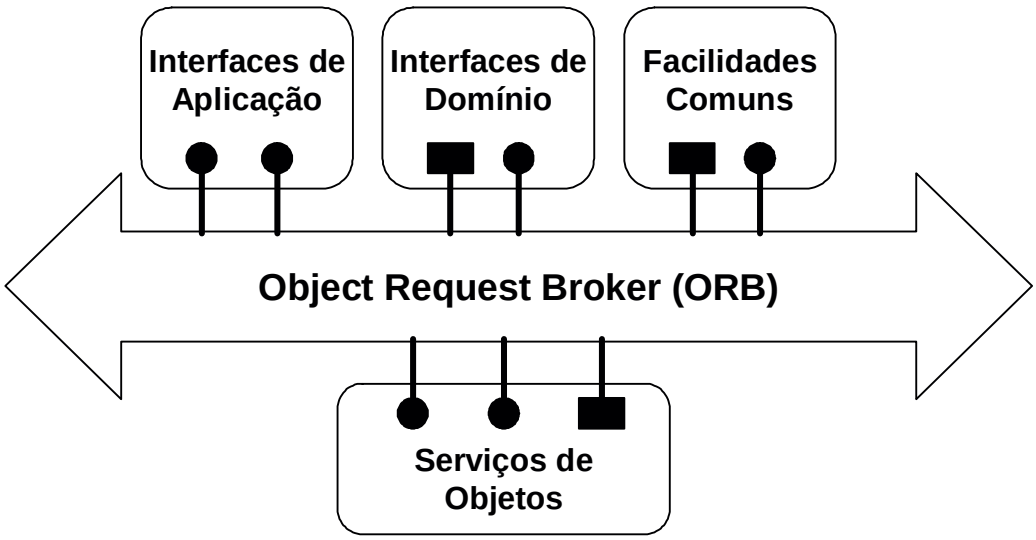

Figura 4.1 - Arquitetura OMA

4.2 A arquitetura CORBA

A figura 4.2 apresenta os componentes da arquitetura CORBA, que descreve a comunicação de objetos em sistemas distribuídos, com o objetivo de abstrair todo o meio de comunicação no processo de invocação de um método remoto de um objeto, possibilitando que todo este processo seja independente de plataforma ou linguagem de programação. O elemento fundamental para a compreensão do CORBA é compreender o seu conceito sobre a IDL (*Interface Definition Language*), responsável pela padronização de interfaces de objetos CORBA.

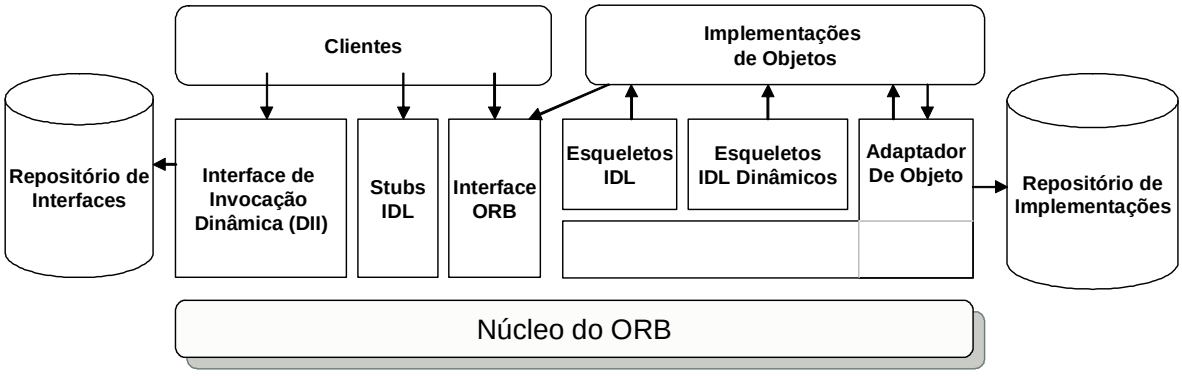

Figura 4.2 - Arquitetura CORBA

- Implementações de Objetos: as implementações de objetos representam as operações definidas pela IDL do CORBA. Esta IDL será implementada em uma linguagem de programação específica (Java, C/C++, Cobol, etc).
- Clientes: são aplicações simples, que por meio das IDLs, irão invocar os métodos dos objetos que se encontram no servidor.
- Núcleo ORB: O núcleo ORB provê mecanismos de comunicação, responsável pelo processo de invocação de um método por um cliente em um objeto do servidor, implementada a partir de uma IDL, de forma totalmente transparente para a aplicação.
- Interface ORB: para separar as aplicações dos detalhes de implementação do ORB, a especificação CORBA definiu uma interface abstrata para o ORB. Esta interface provê várias funções de auxílio, como funções para converter referências de objetos para strings e vice-versa.

- Stubs IDL ou Interfaces de Invocação Estática: O stub IDL é o mecanismo que efetivamente cria e envia as requisições em nome de um cliente até o núcleo do ORB, que é o encarregado de localizar a implementação do objeto adequado e de ativá-la. Os Stubs descrevem os serviços fornecidos pelas implementações de objetos.
- Esqueletos IDL Estáticos: O Esqueleto IDL é um mecanismo que entrega as requisições dos clientes à implementação do objeto CORBA. Em conjunto, os stubs e os esqueletos permitem, através do núcleo do ORB, a comunicação entre os clientes e as implementações de objeto.
- Interface de Invocação Dinâmica (*DII—Dynamic Invocation Interface*): usando a DII um cliente pode invocar requisições diretamente ao objeto destino sem ter conhecimento, em tempo de compilação, das interfaces dos objetos. Aplicações usam uma interface de invocação dinâmica para emitir pedidos aos objetos sem requerer um stub IDL específico.
- Esqueletos IDL Dinâmicos (*DSI—Dynamic Skeleton Interface*): análogos à DII, os esqueletos IDL dinâmicos permitem ao núcleo ORB entregar requisições a uma implementação de objeto que não tem conhecimento, em tempo de compilação, do tipo de objeto que está implementando. O cliente que faz uma requisição não tem idéia se a implementação está utilizando esqueletos IDL estáticos ou dinâmicos.
- Adaptador de Objeto: o adaptador de objeto auxilia o núcleo do ORB na entrega de requisições e na ativação de objetos. Ele associa uma implementação de objeto ao ORB de modo a permitir que esta possa acessar serviços do ORB.
- Repositórios de Interfaces: as interfaces de objetos (IDLs) podem ser adicionadas a um repositório de interfaces. Para usar as interfaces de invocação dinâmica, os clientes necessitam criar uma requisição que deve incluir a referência do objeto, a operação e a lista de parâmetros. As identificações do objeto e da operação podem ser obtidas em um repositório de interfaces, que é um banco de dados que fornece armazenamento persistente de definições de interfaces de objetos.
- Repositório de Implementações: o repositório de implementações contém informações que são usadas pelo ORB e pelo adaptador de objeto para localizar e ativar implementações de objetos em tempo de execução.

4.3 CORBA Security Services Specification

O modelo CORBA de segurança (CORBASec) é uma especificação aberta de segurança em sistemas distribuídos [18, 19, 36]. O CORBAsec relaciona objetos e componentes em quatro níveis numa estratificação de sistema: *o nível de aplicação*, o *nível de middleware* formado por objetos de serviço (COSS), serviços ORB e o núcleo do ORB; o *nível de tecnologia de segurança* formado pelos serviços de segurança subjacentes; e por fim, o *nível de proteção básica* composto por uma combinação de funcionalidades de sistemas operacionais e do hardware,

O CORBASec trabalha com serviços de segurança ao nível de interceptadores, possibilitando proteger os objetos de serviço de forma transparente para a aplicação. Ele é composto, conforme apresentado na figura 4.3 por objetos de serviço (COSS), como o PrincipalAuthenticator, Credential, AccessPolicy, RequiredRights, AccessDecision, SecurityManager, PolicyCurrent, Vault e SecurityContext. O objeto PrincipalAuthenticator é responsável por garantir o processo de autenticação de principais no CORBA, tendo como objetivo, a aquisição de credenciais que serão utilizadas para a sua identificação no sistema. No CORBASec, as políticas são descritas na forma de *atributos de segurança* dos recursos do sistema (atributos de controle) e dos principais (atributos de privilégio). O CORBASec fornece apenas a família CORBA que contém quatro tipos de direitos: *g (get)*, *s (set)*, *m (manage)* e *u (use)*, embora permita a livre definição de outras famílias de direitos.

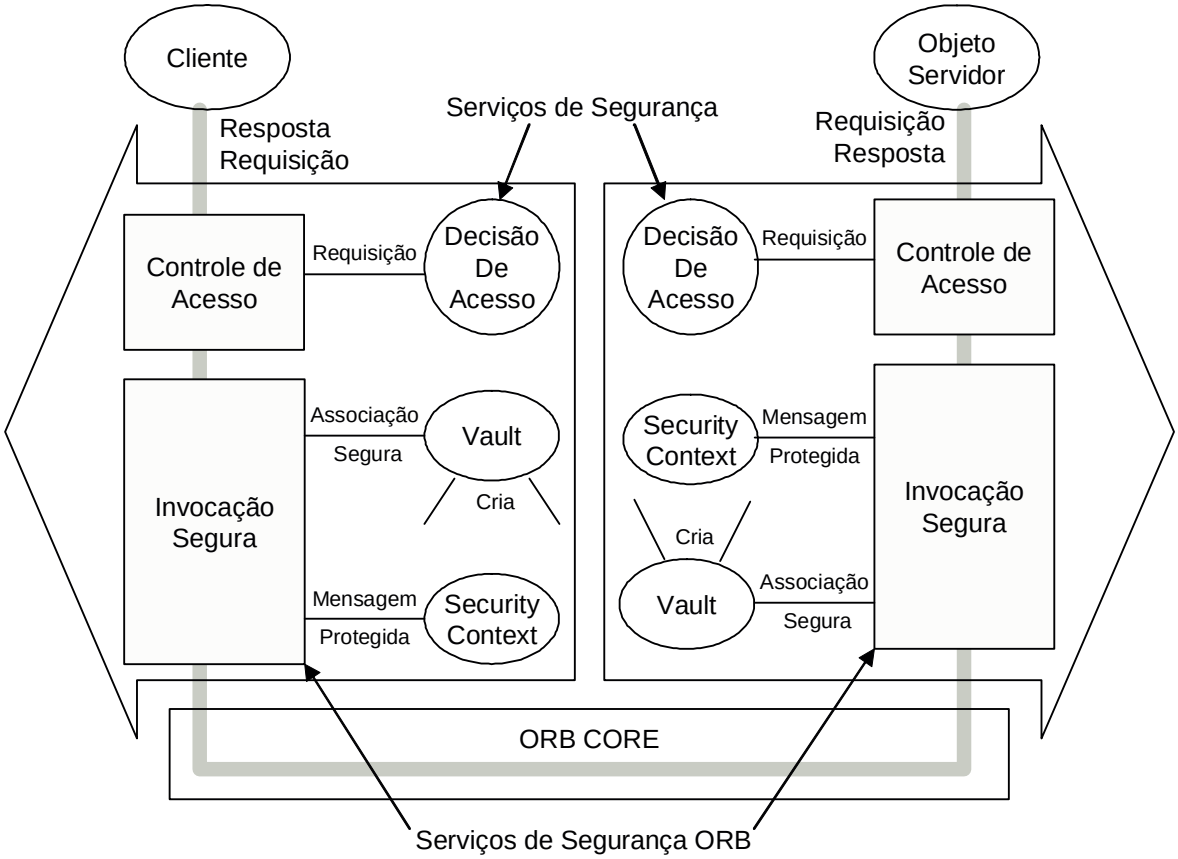

Figura 4.3 - Arquitetura CORBASec

O único modelo de controle de acesso definido e especificado pela OMG sobre o CORBASec se trata de um modelo discricionário que implementa o objeto DomainAccessPolicy, que é uma extensão do Objeto AccessPolicy, que contém um conjunto de principais identificados pelos seus atributos de privilégio, e para cada um deles, uma lista de direitos de acesso para execução dos métodos sobre os objetos (tabela 4.1)

Atributos de Privilégio	Direitos
Ana	`gs--`
Bob	`g-m-`

Tabela 4.1 - DomainAccessPolicy

O objeto RequiredRights define para a operação de invocação de um objeto, alguns direitos necessários ou requeridos (atributos de controle). (tabela 4.2)

Direitos Requeridos	Combinação	Operação	Interface
`Corba: g—m`	All	Get_balance	Bank

Corba: gs--	Any	Deposit	Bank
Corba: gsu-	All	Open	Bank

Tabela 4.2 - RequiredRights

O objeto AccessDecision é responsável pelo processo de autorização de acesso ao método de um objeto, a partir da sua invocação pelo cliente de uma aplicação. A sua intervenção é feita a partir do uso de Interceptadores, que são objetos de serviço colocados no caminho da invocação entre o cliente e o servidor. Neste processo, um cliente efetua a invocação do método remoto que se encontra no servidor, passando por um processo iniciado pelo PrincipalAuthenticator, que gera as credenciais de identificação que serão utilizadas no processo de decisão do acesso, avaliando os atributos de privilégio e de controle encontrados nos objetos DomainAccessPolicy e RequiredRights (figura 4.4).

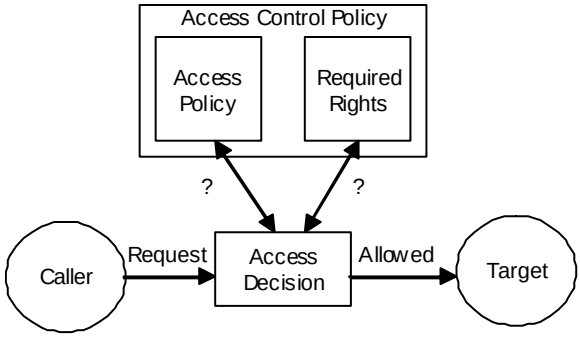

Figura 4.4 - Controle de Acesso no CORBASec

Neste exemplo, o cliente Bob possui apenas os direitos “gs”. Se ele tentar executar a invocação da operação “Get_balance”, os direitos requeridos são “gm” e a combinação deve ser All. Neste caso, Bob terá o seu direito de acesso negado pelo objeto AccessDecision. Caso ele tente executar a operação *Deposit*, como os direitos requeridos são “gs” mas o combinator é apenas Any, o objeto AccessDecision permitirá o seu acesso, pois Bob possui apenas um dos direitos (“g”).

4.4 RAD – Resouce Access Decision Facility

O RAD [4, 5, 23] tem como objetivo separar os controles efetuados por uma camada de controle de acesso da lógica de negócio, mas que diferente do modelo CORBASec, a aplicação possui ciência e também participa do processo de controle de acesso. A arquitetura RAD não tem a intenção de substituir as camadas de middleware ou outros ambientes de

segurança, tal como o CORBASec. A sua idéia básica é complementar a existência do controle de acesso efetuado em nível transparente para a aplicação fornecido pelo CORBASec, com a possibilidade de efetuar autorizações mais sofisticadas, que são independentes da lógica da aplicação. Este controle complementar estaria baseado em informações externas à esta camada transparente, muitas vezes baseados em informações passadas pelo próprio usuário da aplicação (por exemplo via parâmetros do método invocado) e que não são interceptadas por este nível transparente de controle de acesso.

O RAD [4, 5] provê mecanismos para decisão de acesso sobre objetos de Aplicação de Sistema (AS – *Application System*) e está apresentado na figura 4.5. Nesse processo, uma aplicação cliente envia uma requisição para o AS (passo 1). Se esta requisição necessita ser autorizada, o AS envia uma ou mais requisições para o RAD (passo 2). Para cada requisição de acesso, são passadas as credenciais de segurança do cliente, nome do recurso a ser acessado e o nome da operação a ser executada sobre o recurso. O retorno desta requisição se trata de um valor lógico (verdadeiro e falso ou permitido e não permitido) sobre este acesso (passo 3). O AS deve utilizar o retorno desta requisição de acesso e retornar esta resposta para o cliente da aplicação (passo 4). Conforme apresentado, a interação entre um cliente, um objeto de aplicação e o RAD é muito importante para que este processo de controle de acesso baseado nestes serviços esteja em perfeito sincronismo, de forma que a segurança imposta nesta solução funcione adequadamente.

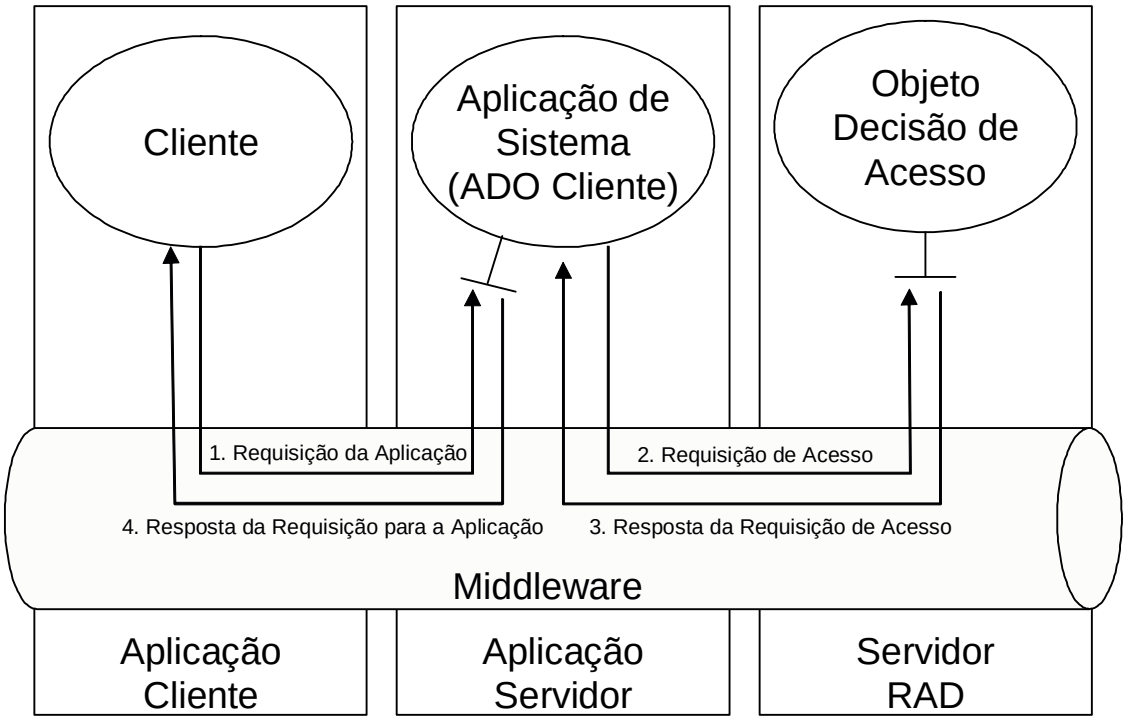

Figura 4.5 - Interação entre as aplicações Cliente, Servidor e o servidor RAD

A arquitetura RAD permite a implementação de seus componentes por vários fornecedores, focando uma definição para políticas de controle de acesso de forma padronizada, com performance, escalabilidade e outras propriedades de sistema. Um servidor RAD é composto dos seguintes componentes:

1. Objeto Decisão de Acesso (ADO);
2. Objeto Avaliador de Políticas (PolicyEvaluator - PE);
3. Localizador do Avaliador de Políticas (PolicyEvaluatorLocator - PEL);
4. Serviço de Atributos Dinâmico (DynamicAttributeService - DAS)
5. Objeto Combinação de Decisão (DecisionCombinator - DC).

A figura 4.6 apresenta o processo de decisão efetuado pelo RAD. A aplicação (AS) envia uma requisição para o ADO, via sua interface de chamada (*acess_allowed*). O ADO obtém as políticas associadas com o objeto da requisição (PEL) e os atributos dinâmicos do principal no contexto do objeto da requisição e sua operação (DAS). O ADO efetua o processo para a combinação das políticas de controle de acesso a partir do objeto DecisionCombinator (DC), que recebe como parâmetros as informações obtidas pelo PEL e

DAS, efetuando o processo de avaliação (PE). O ADO recebe o resultado desta decisão, que finalmente é passada para a aplicação (AS).

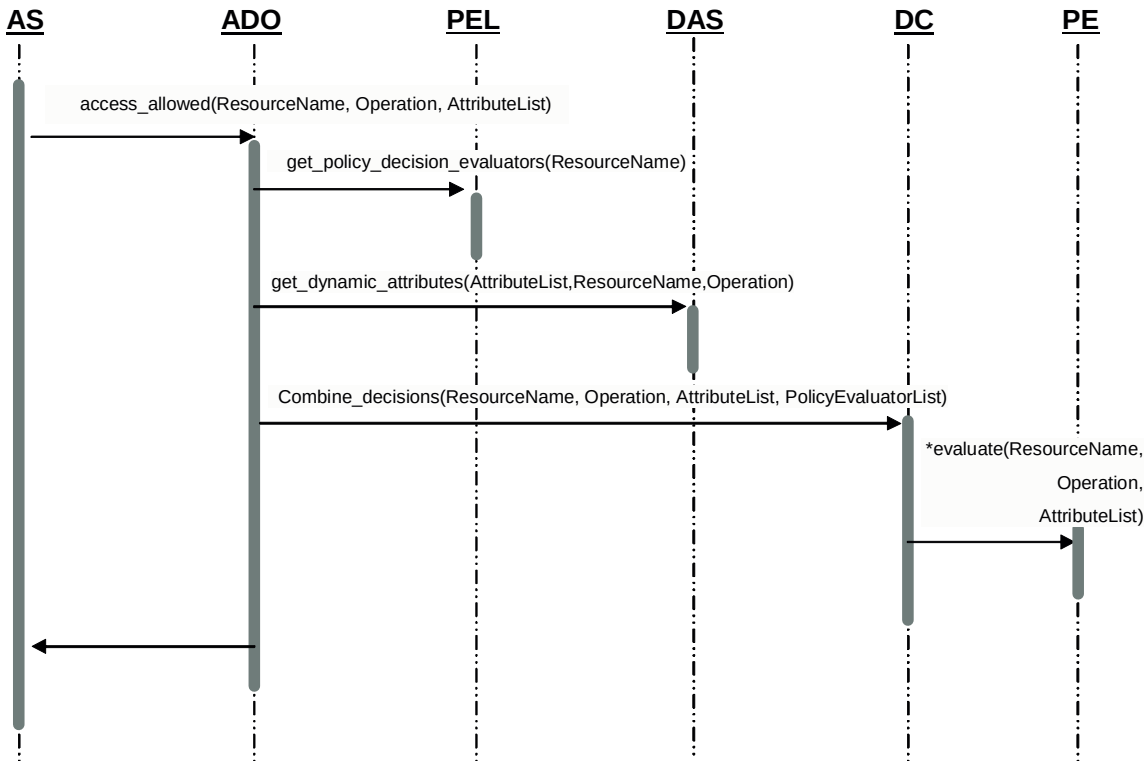

Figura 4.6 - Diagrama de seqüência do processo do RAD

4.5 A Família JaCoWeb Security

O JaCoWeb Security [33] é um projeto desenvolvido no Laboratório de Controle e Microinformática (LCMI) da Universidade Federal de Santa Catarina (UFSC)[2]. Ele foi criado com o objetivo de implementar um esquema de segurança para aplicações distribuídas em redes de longa distância, como a Internet, envolvendo os modelos de segurança Java/CORBA/Web. Sua implementação está baseada no JacORB [7], que é um ORB free, possuindo seu código aberto e distribuído, desenvolvido em Java pela Freie Universitaet Berlin. O trabalho desenvolvido por este projeto tem como base seguir as especificações padronizadas pela OMG relacionadas ao padrão de segurança CORBA (CORBASec) [19], implementando um esquema de autorização (utilizando conceitos de controle de acesso,

[2] Laboratório de Controle e Microinformática (LCMI) da UFSC.
http://www.lcmi.ufsc.br

autenticação e criptografia) ao nível ORB, e também propor a aplicação de políticas de controle de acesso ainda não especificados pela OMG.

O trabalho definido em [33], aplica a criptografia sobre as informações, utilizando o conceito de chaves públicas e privadas para o fechamento de um canal cifrado por meio de SSL. Este conceito também foi utilizado para o processo de autenticação de principais, que identifica um principal por meio de sua chave privada.

Em [37], o serviço Policap foi desenvolvido a partir de documentos drafts (propostas iniciais) liberados na época pela OMG. É um serviço que oferece operações tanto para funções administrativas quanto operacionais sobre objetos de políticas, cumprindo o papel de gerente de domínio para objetos de politica (DomainAccessPolicy) e de gerente de direitos para objetos de direitos requeridos (RequiredRights) do nosso domínio. A idéia é que na invocação de um método de um objeto, os objetos DomainAccessPolicy e RequiredRights sejam fornecidos ao cliente da invocação. O seu esquema de autorização é feito de forma transparente pelo ORB, que são validadas tanto no lado cliente quanto no lado servidor. A idéia de se efetuar autorizações em duas fases é que, caso o processo de autorização seja negado pelo ORB do lado cliente, a invocação do método nem chegará a ser invocado no servidor, evitando um tráfego desnecessário na rede.

Em [38], é proposto o modelo “JaCoWe - Um Esquema de Autorização para Redes de Larga Escala Integrando os Modelos de Segurança Java, CORBA, Web”, implementando a camada de autorização sobre um Applet assinado.

Em [35], é definido o JaCoWeb-Obrigatório, em que foram efetuadas extensões das políticas de autorização do JaCoWeb sobre o Policap, aplicando o controle de acesso obrigatório de Bell e Lapadula, trabalhando em conjunto com as políticas discricionárias já implementadas no projeto JaCoWeb. Outra continuação deste trabalho é o RBAC-JaCoWeb [18], que aplica o controle de acesso baseado em papéis, seguindo o modelo RBAC simétrico do padrão RBAC-NIST.

A família JaCoWeb sempre focou em aplicar o esquema de autorização sobre o ORB, de forma que todo o controle de acesso fosse efetuado de forma totalmente transparente para a aplicação. Outro ponto é que as suas implementações também eram focadas em efetuar processos de autorização relacionados do lado Cliente da aplicação, que nem sempre pode estar dentro de uma rede considerada confiável. Por este motivo, este controle de acesso também teve que ser efetuado no lado servidor, devido à IDL ser uma interface de

conhecimento público, que poderia ser facilmente implementado e utilizado sem os devidos controles de segurança.

4.6 Conclusão

O CORBASec define uma especificação de segurança capaz de efetuar um controle de acesso transparente para a aplicação ao nível do ORB. Esta camada transparente permite que um desenvolvedor implemente uma aplicação sem se preocupar com a segurança sobre os acessos aos métodos remotos dos objetos do sistema. Isso possibilita que as políticas de controle de acesso sejam definidas por um administrador de segurança de um sistema, sem a necessidade de serem efetuadas alterações diretamente na aplicação. Uma desvantagem sobre esta solução é que nem todos os controles de segurança definidos no CORBASec podem ser controlados apenas no nível transparente, uma vez que muitas decisões sobre os acessos a recursos do sistema dependem de ações que um usuário venha a tomar. Estas decisões, por exemplo, podem ser baseadas em parâmetros passados aos métodos remotos do objeto, e que não são interceptados e utilizados pela camada de controle de acesso do ORB.

Visando este problema, a OMG divulgou a especificação RAD (*Resource Access Decision*) [23], que permite que sejam definidas políticas de controle de acesso ao nível da aplicação, possibilitando que o desenvolvedor implemente os controles de segurança em sua aplicação, trabalhando em conjunto com o RAD para o processo de decisão de acesso. Uma vantagem sobre este processo é que a aplicação pode possuir um maior domínio sobre as ações de um usuário sobre um sistema. Sua desvantagem é que a implementação da aplicação irá possuir controles de segurança em conjunto com a lógica de negócio do sistema, que como possuem uma dependência do desenvolvedor, faz com que ele tenha uma preocupação a mais sobre o seu desenvolvimento.

Uma boa possibilidade é efetuar a combinação destas duas funcionalidades, juntando o nível transparente de controle de acesso juntamente com o nível de aplicação. Mas uma desvantagem sobre esta combinação é que o administrador de segurança do sistema acabará tendo que administrar dois pontos distintos de definição das políticas de controle de acesso, e que dificultaria o seu trabalho, uma vez que as definições destas políticas, além de não trabalharem em conjunto, também poderiam ter conceitos totalmente distintos, aumentando a complexidade sobre a definição das políticas de controle de acesso.

Capítulo 5

A proposta JaCoWeb-ABC

O $UCON_{ABC}$ define um modelo muito completo que possibilita adaptar diversos conceitos de controle de acesso à sua funcionalidade. A especificação CORBASec é muito conhecida e difundida no meio acadêmico, integrando diversos conceitos de segurança em sua camada, como processos de autenticação, autorização, auditoria, confidencialidade, integridade. O fato do CORBASec definir uma camada de segurança totalmente transparente para a aplicação, que é controlada pelo ORB, pode ser vantajoso em diversos casos, pois as aplicações não precisam se preocupar em implementações sobre o que vai ser protegido em seu sistema. Infelizmente, este controle transparente pode muitas vezes não ser suficiente, pois muitas das tomadas de decisão a serem definidas nas aplicações estão baseadas em ações que o usuário venha a tomar posteriormente, após receber o acesso.

Neste caso, estas ações estariam baseadas a partir de parâmetros passados pela invocação do método remoto, que seriam interceptados pela própria aplicação e a partir deste ponto, qualquer responsabilidade sobre o controle de segurança teria que ser implementado pela própria aplicação. Para estes casos, entra o papel do responsável pelo sistema, que teria que buscar por ferramentas de mercado ou então partir para soluções proprietárias, que possibilitassem integrar este segundo nível de controle de acesso diretamente na aplicação. A utilização desta segunda solução nem sempre está livres de falhas de implementação ou conceituação, que consequentemente resultariam em acessos indevidos ao sistema, e também acabariam dificultando a codificação da aplicação, que nem sempre separa a lógica do negócio da lógica de segurança. Para estes casos, poderia ser utilizado o RAD [23], mas mesmo assim, ainda existiria uma complexidade na definição e administração das políticas de

segurança, pois elas teriam que ser definidas em dois pontos totalmente distintos (ORB e RAD), podendo possuir conceitos de políticas distintas e totalmente diferentes, o que muitas vezes acabaria inviabilizando a utilização do nível transparente de controle de acesso controlado pelo ORB, pois ele poderia muito bem ser suprido e implementado totalmente ao nível da aplicação.

Considerando este problema, a proposta desta dissertação é integrar o modelo $UCON_{ABC}$ à especificação CORBASec, centralizando estes 2 níveis de controle de acesso em sua camada de segurança. Com isso, estaremos adicionando estas funcionalidades em sua camada de decisão, atualmente implementada pelo objeto AccessDecision e também estaríamos fornecendo um segundo nível de controle de acesso que seria utilizada pela própria aplicação. Assim, será possível definir políticas de segurança utilizadas pelo controle transparente para a aplicação, nos casos em que esta camada de segurança tenha todas as informações necessárias para uma tomada de decisão de acesso, como também definir políticas a serem utilizadas pela aplicação, nos casos em que as decisões de acesso sejam baseadas em informações externas à esta camada, que seriam capturadas por parâmetros passados por um usuário à aplicação. A combinação destas duas funcionalidades irá definir um modelo de controle de acesso muito mais preciso e rigoroso sobre as ações de um usuário de um sistema. E esta solução irá facilitar tanto o papel do administrador de segurança, que terá condições de definir políticas de segurança de forma centralizada, como do implementador do sistema, que poderá segregar a codificação sobre o controle de segurança de forma mais padronizada e separada de sua lógica de negócio.

5.1 Mapeamento do modelo $UCON_{ABC}$ no CORBASec

Em termos de controle de acesso, um sujeito no modelo $UCON_{ABC}$ é equivalente a um principal no CORBASec. Cada principal possui atributos de privilégio, associados a ele através do objeto Credentials, e a esses atributos de privilégio são concedidos direitos, através do objeto AccessPolicy. Um objeto do $UCON_{ABC}$ corresponde a um método de uma interface CORBA; a cada método é associado um conjunto de direitos requeridos (juntamente com uma combinação), através do objeto RequiredRights. Uma autorização é implementada no CORBASec pelo objeto AccessDecision, que compara os direitos concedidos a um principal (AccessPolicy) e os direitos requeridos por um método (RequiredRights) para decidir se o

principal pode invocar esse método.

A descrição acima resume a equivalência entre o modelo $UCON_{ABC}$ e o modelo de controle de acesso do CORBASec. Com base nessa descrição, fica evidente que vários componentes do modelo $UCON_{ABC}$ não encontram correspondente no CORBASec. Para a aplicação dos conceitos existentes no modelo $UCON_{ABC}$ no CORBASec, foram identificadas e mapeadas as seguintes diferenças entre os modelos, que terão que ser adaptadas para possibilitar a sua implementação:

1) *atributos*: na especificação CORBASec, o Sujeito e Objeto possuem apenas atributos pré-determinados, conforme apresentado nas tabelas 4.1 e 4.2. Já o modelo $UCON_{ABC}$ considera que atributos de sujeitos e objetos são arbitrários, ao passo que a especificação CORBASec, embora permita a definição de novos atributos de controle, exige que esses atributos sejam especificados através de uma IDL, o que torna essa definição fixa. Portanto, torna-se necessário definir uma maneira para representar atributos de sujeitos e objetos de forma a aproveitar a flexibilidade oferecida pelo modelo $UCON_{ABC}$.

2) *mutabilidade dos atributos*: no modelo CORBASec, os atributos do sujeito e objeto somente podem ser alteradas a partir de ações administrativas de inclusão ou remoção de direitos. Já no modelo $UCON_{ABC}$, a alteração dos atributos pode ocorrer a partir do acesso de um sujeito sobre um objeto, baseado na definição de suas políticas de segurança.

3) *direito*: no modelo CORBASec, o direito de acesso está baseado somente em um modelo discricionário, onde são avaliados os atributos de direito do sujeito e objeto. No modelo $UCON_{ABC}$, também são avaliados os atributos do sujeito e objeto, mas com a possibilidade de adaptar diversos modelos de controle de acesso, incluindo o modelo discricionário existente no CORBASec.

4) *Autorização, Obrigação e Condição*: o modelo CORBASec trabalha apenas com um processo de autorização (modelo discricionário). Desta forma, além de uma remodelagem do processo de autorização, os controles de obrigação e condição do modelo $UCON_{ABC}$ precisam ser totalmente definidos e implementados para o CORBASec.

5) *Definição de políticas de controle de acesso*: A definição de políticas no CORBASec é muito simples e estão especificadas basicamente nos objetos DomainAccessPolicy e RequiredRights, sendo que o processo de decisão é feito apenas com um cruzamento dos atributos de direito do sujeito e objeto associados ao atributo

"combinator". Já no modelo $UCON_{ABC}$, estas políticas são representadas a partir de expressões lógicas e de atribuição. O resultado destas expressões irá definir o direito do acesso ao objeto. Para o cálculo destas expressões, são utilizados os atributos de um sujeito e objeto, além de variáveis externas como por exemplo, a data/hora utilizadas pelo processo de Condição.

6) *Níveis de avaliação de acesso*: Todo o processo de decisão de acesso no modelo CORBASec é feito apenas no nível do ORB. Desta forma, durante o acesso de um sujeito sobre o método de um objeto, este processo de decisão é feito de forma totalmente transparente para a aplicação. Caso a permissão do acesso seja negada pelo objeto AccessDecision, o método nem chega a ser executado. Já o modelo $UCON_{ABC}$ permite definir um controle de acesso transparente para a aplicação, nos casos em que os acessos podem ser decididos apenas com as informações dos atributos do sujeito e objeto. Além disso, o $UCON_{ABC}$ também permite trabalhar em conjunto com a aplicação, nos casos em que o controle de acesso poderá ser baseado em fatores externos à camada, como por exemplo, obrigar um usuário a informar um e-mail antes de continuar a acessar outras partes de um sistema. Nesta situação, o e-mail deverá ser analisado, para certificar-se de que se trata de um e-mail válido, cabendo à aplicação continuar o processo de decisão que deverá informar ao modelo $UCON_{ABC}$ que a obrigação foi cumprida pelo sujeito.

5.2 Alterações necessárias no CORBASec para adaptação do modelo $UCON_{ABC}$

A partir do mapeamento dos modelos, identificamos a necessidade das seguintes alterações sobre os componentes do CORBASec para adaptação do modelo $UCON_{ABC}$:

1) Os objetos AccessPolicy e RequiredRights deverão ser remodelados para que possibilitem uma definição de qualquer atributo. Desta forma, surgiu a necessidade de criarmos um objeto nomeado de "AttributesManager" (apresentado e detalhado na seção 5.6, figura 5.4) para gerenciar atributos que deve permitir a definição de atributos para um sujeito e objeto. Nesta implementação, permanecerão inalterados apenas os atributos de identificação do sujeito (atributos de privilégio) e objeto (interface e operação) definidos no CORBASec. O restante dos atributos (direitos e combinação) poderão passar a ser controlados pelo objeto AttributesManager.

2) O objeto AccessDecision deverá ser remodelado, de forma que possibilite efetuar o

processo de decisão baseado em políticas de Autorização, Obrigação e Condição (decisão ABC). A definição destas políticas deve ser feita para cada Objeto (método remoto). Neste processo de decisão, podem ser definidas políticas de atualização de atributos para o sujeito e objeto.

3) Para avaliação das políticas ABC, deve ser implementado um interpretador de expressões lógicas e de atribuição capaz de calcular as expressões utilizadas para decisão e atualização baseadas nos atributos do sujeito e objeto e também variáveis externas ao CORBASec, como a data/hora.

4) Devido ao CORBASec ser um modelo transacional, iniciada a partir da invocação de um método remoto, o processo de decisão ABC poderá ser efetuada somente antes (preA, preB, preC) do acesso de um sujeito sobre o objeto, pois em uma invocação de um método não existe possibilidade de se efetuar o processo de decisão durante (ongoing) um acesso (onA, onB, onC). Mas o conceito ongoing pode ser adaptado no CORBASec a partir da invocação de 2 métodos distintos, sendo a primeira responsável por liberar um acesso e a segunda para um acesso periódico a um objeto (ex.: em um serviço disponibilizado em um site, um método pode servir para que um sujeito responda uma pergunta a cada 15 minutos e outro método para possibilitar o acesso a serviços deste site, que somente poderá ser acessado caso o usuário tenha respondido as perguntas em períodos inferiores a 5 minutos). Pelo mesmo motivo, o processo de atualização de atributos também poderá ser efetuado somente antes (preUpdate) ou após (posUpdate) o acesso de um sujeito sobre um objeto, não existindo o conceito de atualização durante o acesso (onUpdate).

5) Necessidade de se criar um processo de avaliação em 2 níveis, sendo a primeira de forma totalmente transparente para a aplicação e a segunda trabalhando em conjunto com a aplicação. A combinação destas duas funcionalidades permitirá efetuar um controle mais preciso e eficiente sobre as ações de um usuário no sistema.

5.3 Níveis de decisão e verificação das políticas ABC

Para possibilitar o processo de decisão de acesso e permitir também o processo para atualização de atributos definidos nas políticas do modelo $UCON_{ABC}$ (definido na figura 2.4) ao objeto AccessDecision do CORBASec, criamos o objeto AccessDecisionABC, representado na figura 5.1, que possibilita efetuar os processos de avaliação ABC juntamente com os processos de pré-update e pós-update. Este objeto será utilizado na implementação do

objeto AccessDecision do CORBASec, a partir da invocação de um sujeito a um objeto remoto do CORBA. Neste processo, o objeto AccessDecision identifica o Sujeito (principal) e Objeto (método remoto) da transação, e repassa estas informações para o objeto AccessDecisionABC. A partir da identificação do Objeto, o AccessDecisionABC carrega as políticas de segurança ABC (policyABC) e as políticas de atualização de preUpdate e posUpdate. E conforme definido no modelo $UCON_{ABC}$, o processo de avaliação destas políticas será baseado nos atributos do Sujeito e Objeto, que serão utilizados nos cálculos de expressões definidas nas políticas de decisão de acesso ABC e no processo de atualização de atributos.

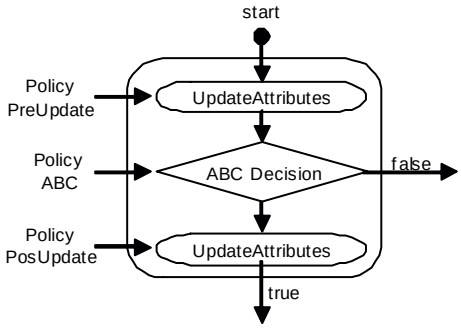

Figura 5.1 – Objeto AccessDecisionABC

Para possibilitar a divisão do processo de decisão em 2 níveis de acesso, o objeto AccessDecisionABC foi adaptado para que fosse chamado nestas 2 situações: a primeira no nível do ORB em que o processo de decisão é feito de forma totalmente transparente para a aplicação. Este nível de decisão já é suportado pelo CORBASec e para isso, criamos o objeto AccessDecisionABC_ORB que será utilizado pelo objeto AccessDecisionABC. Já o segundo nível, utilizado nos casos em que os controles de segurança devam ser baseados em dados ou processamentos informados pelo próprio cliente, em que não seriam possíveis de serem decididos caso a política de acesso fosse efetuada apenas no nível do ORB. Este processo de avaliação deve ser feito durante a implementação da IDL de um objeto CORBA, possibilitando que esta camada de segurança trabalhe em conjunto com a aplicação. Por exemplo, o processo de decisão pode estar baseado no parâmetro de um método remoto invocado pelo cliente. Desta forma, este processo de tomada de decisão deve ser feito em conjunto com a aplicação, que deve preparar o dado (ex.: quantidade), repassá-lo ao objeto AccessDecisionABC e efetuar o tratamento do retorno deste processo de decisão, que irá retornar os valores vardadeiro (*true*) caso o acesso seja permitido ou falso (*false*) caso o acesso seja negado. Para este segundo nível de acesso, foi criado o objeto de

AccessDecisionABC_IDL, que será utilizado pela aplicação durante a implementação da IDL e irá efetuar a avaliação das políticas de decisão e atualização carregados pelo objeto AccessDecisionABC. Abaixo estamos detalhando a especificação de cada um destes objetos, que também estão apresentadas na figura 5.2.

AccessDecisionABC_ORB: controle de acesso ABC no nível do Middleware, que funciona de forma transparente para a aplicação, sendo necessária a configuração das políticas de segurança que serão controladas pelo objeto. Nesta situação, o processo de decisão depende apenas das informações existentes nos atributos do sujeito e objeto. Todo este processo ocorre da seguinte maneira: a partir do objeto PrincipalAuthenticator do CORBASec, um Sujeito efetua o processo de autenticação criando assim as suas credenciais de identificação. Neste momento, é chamado o interceptador de controle de acesso que efetua a chamada ao método access_allowed() do objeto AccessDecisionABC, que recebe como parâmetros as suas credenciais, que possuem os atributos de identificação do Sujeito, e também o nome da Interface e Operação do objeto que está sendo invocado (passo 1). O objeto AccessDecision_ORB chama o objeto AccessDecisionABC passando a este, os parâmetros de identificação do Sujeito e Objeto, que em seguida invoca o objeto AccessDecisionABC (passo 2). A partir das credenciais, o objeto AccessPolicy associa Sujeito (S) aos seus atributos (ATT(S)), e pelo nome da Interface e Operação, o objeto RequiredRights associa o Objeto (O), seus atributos (ATT(O)) e carregando as políticas de decisão e atualização do $UCON_{ABC}$. A partir dos atributos e das políticas carregadas, o objeto AccessDecisionABC realiza o processo de decisão (ABC) e atualização. Após este processo (passo 3), caso o método access_allowed retorne o valor “true”, o método implementado a partir da IDL será invocado (passo 4a). Caso o retorno seja “false”, o ORB irá lançar uma exceção para o cliente da invocação (passo 4b).

AccessDecisionABC_IDL: controle de acesso ABC no nível da aplicação, que deve trabalhar em conjunto com a aplicação para o processo de decisão. Para isso ser possível, a arquitetura JaCoWeb-ABC disponibiliza uma API que deve ser utilizada durante a implementação da IDL do objeto no servidor, para que ela possa trabalhar em conjunto com sua camada de decisão. Esta API, nomeada AccessDecisionABC_IDL.access_allowed(), deve receber 4 parâmetros que estão disponíveis pela especificação CORBASec durante a implementação da IDL no servidor: CurrentObject, para aquisição das credenciais do sujeito,

ObjectImplemented, para identificação do nome da interface do objeto, nameMethod, que é o nome do método invocado (operação), e Vector. Vector é um objeto que possibilita adicionar uma lista de variáveis que serão utilizadas em políticas definidas pelo JaCoWeb-ABC. Nesta situação, a IDL implementada no servidor deve capturar informações externas a este objeto como, por exemplo, parâmetros recebidos pelo método invocado. Depois, deve preparar estes dados para adicioná-los em uma variável Vector (passo 5). Em seguida deve chamar a API *access_allowed* (passo 6), que irá efetuar o mesmo processo definido na chamada do objeto AccessDecisionABC_ORB, chamando o objeto AccessDecisionABC (passo 7). Este objeto será responsável por executar o processo de decisão e atualização, e retornar o resultado deste processo (passo 8) para a aplicação. E finalmente a aplicação deverá tratar a condição do acesso, que terá um retorno true ou false, continuando o processo de decisão (passos 9a e 9b).

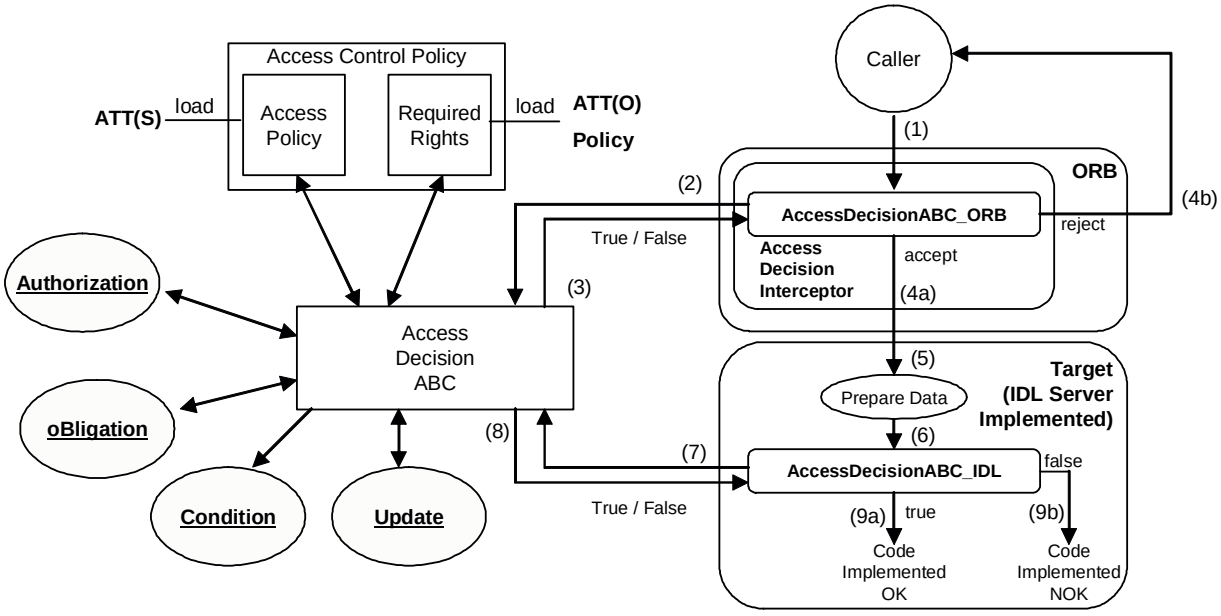

Figura 5.2- Processo de avaliação efetuado pelo objeto AccessDecisionABC

5.4 Processo de Atualização de Atributos

Assim como nos 16 modelos básicos da família ABC, também definimos a partir dos processos de avaliação o objeto AccessDecisionABC, uma tabela contendo 6 modelos do JaCoWeb-ABC (abcORB1, abcORB0, abcORB3, abcIDL1, abcIDL0, abcIDL3), como forma de adaptá-lo ao modelo (tabela 5.1).

	(0)	(1)	(2)	(3)
abcORB	**Y**	**Y**	N	**Y**
abcIDL	**Y**	**Y**	N	**Y**

(0) – Imutable, (1) – preUpdate,
(2) – onUpdate, (3) – posUpdate

Tabela 5.1 - Família JaCoWeb-ABC

Nesta tabela, unificamos todo o processo de avaliação de autorização, obrigação e condição em 2 pontos distintos definidos como abcORB e abcIDL (figura 5.2). Como o modelo CORBA é um modelo transacional, o conceito de avaliação ongoing (onA, onB, onC) não é aplicado, uma vez que a transação inicia e termina na invocação do método. Desta forma, caso exista necessidade, o conceito de ongoing ser aplicado sobre o modelo CORBA a partir da invocação de 2 métodos conforme já explicado anteriormente.

5.5 Processo de Cálculo de Expressões

Os processos de decisão definidos pelas políticas de segurança do $UCON_{ABC}$ são totalmente baseados em expressões lógicas e de atribuição, mas com representações muito abstratas de difícil implementação. Desta forma, tivemos que definir e implementar um interpretador capaz de calcular estas expressões, trabalhando com os atributos do sujeito e objeto. Para isso, criamos 2 objetos para o cálculo destas expressões:

Objeto LogicExpression: foi criado para calcular expressões lógicas utilizadas pelos processos de decisão ABC.

Objeto AttribExpression: foi criado para o processo de atualização de atributos, sendo capaz de calcular as expressões baseadas no tipo de dado dos atributos para o processo de atribuição. Ele é capaz de trabalhar expressões com tipos lógicos, matemáticos, strings, datas, vetores e matrizes. As operações de cada um destes tipos estão detalhadas no *Capítulo 6* .

A identificação dos atributos utilizada pelas expressões é definida da seguinte maneira: para acesso a atributos do sujeito, a expressão deve utilizar a string "S->" para referenciar um atributo do Sujeito. Já o atributo do objeto deve ser referenciado pela string "O->". A figura 5.3 apresenta um exemplo de passagem de expressões para estes objetos. No Objeto LogicExpression, uma expressão lógica é passada. Nesta expressão, é avaliado se os créditos de um sujeito são maiores que o valor de um objeto e se o perfil do sujeito é de um cliente,

retornando um valor lógico true ou false. Para o objeto AttributExpression, é passado a uma expressão de atribuição, onde o atributo do sujeito “creditos” irá receber o resultado da subtração dos créditos do sujeito pelo valor do objeto. No final, o atributo “creditos” é atualizado pelo resultado da expressão.

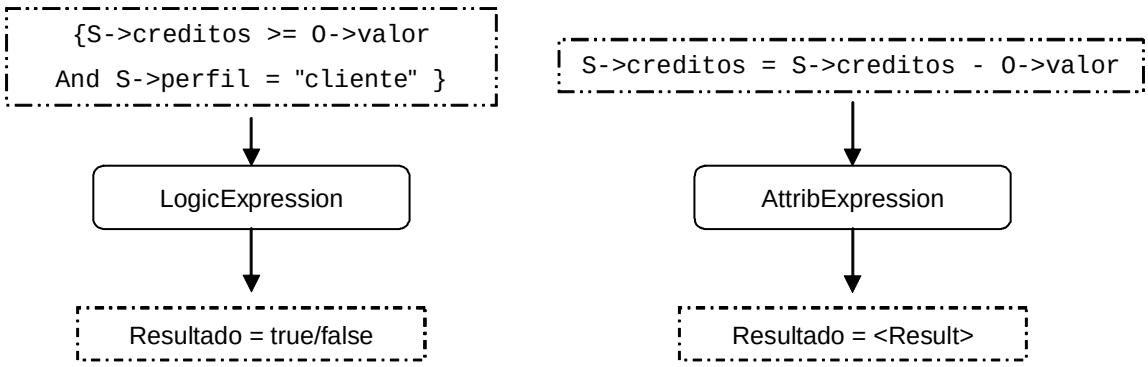

Figura 5.3 - Objetos LogicExpression e AttribExpression

Além da identificação de atributos do sujeito e objeto, os objetos LogicExpression e AttribuExpression também podem interpretar em suas expressões chamadas de funções específicas, como por exemplo, capturar informações como a data e hora do sistema. Para isso, a interpretação destas chamadas teve que ser especificada e implementada por estes objetos. Mais detalhes sobre as funções interpretadas por estes objetos estão explicados no *Capítulo 6*.

5.6 Visão geral do Modelo

Para adaptar as políticas do modelo $UCON_{ABC}$ na arquitetura JaCoWeb-ABC, representadas nas figuras 5.1 e 5.2, utilizamos a linguagem XML [32] capaz de especificar os atributos do sujeito e objeto e também as políticas de autorização, obrigação, condição e atualização. O modelo de políticas ABC utilizadas pelo modelo JaCoWeb-ABC, que estão melhor detalhadas no *Capítulo 6*, pode ser melhor compreendido a partir dos exemplos que serão apresentados no Capítulo 7. A figura 5.4 apresenta uma visão geral da arquitetura JaCoWeb-ABC. Todas as políticas de controle de acesso ABC estarão definidas para cada objeto remoto do CORBASec, uma vez que todos os controles são efetuados sobre estes objetos. Para esta definição, foram criados os objetos PolicyObject, que possui todas as

políticas de decisão e atualização do modelo $UCON_{ABC}$ baseado nas famílias abcORB e abcIDL. O objeto PolicyABC possui as políticas de Autorização (policyA), Obrigação (policyB) e Condição (policyC), e também as políticas de atualização de atributos (objeto PolicyUpdate) que podem ser feitas antes (preUpdate) ou depois (posUpdate) do processo de decisão. A política de decisão de acesso ABC (Authorization, oBligation, Condition) pode ser efetuada de forma transparente para a aplicação (PolicyABC_ORB) ou trabalhar em conjunto com a aplicação (PolicyABC_IDL), que neste caso necessita ser implementado durante a codificação da IDL, uma chamada à API do JaCoWeb-ABC.

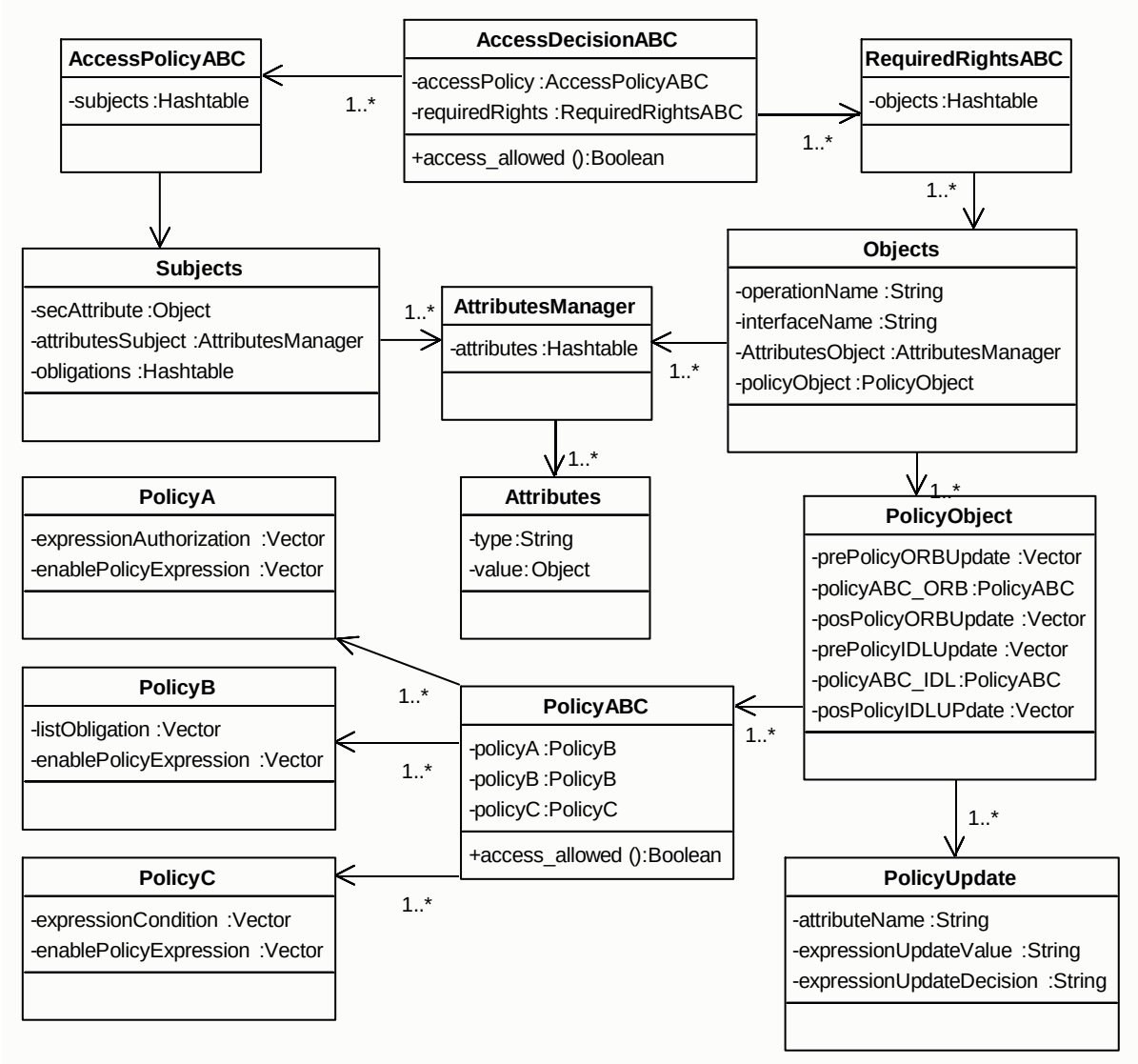

Figura 5.4 - Componentes da Arquitetura JaCoWeb-ABC

5.7 Conclusão

Este capítulo descreveu uma proposta para estender a especificação CORBASec, para integrar o modelo de controle de acesso $UCON_{ABC}$ à sua camada de segurança. Para isso, foi

realizado um trabalho para tentar compreender diversos conceitos e definições abstratas destes modelos e convertê-los em um processo possível de ser estruturado e implementado computacionalmente. Mais detalhes sobre o resultado desta implementação poderá ser analisado no *Capítulo 7*. Podemos citar as seguintes contribuições relacionadas à nossa proposta sobre este capítulo:

1) Extensão da especificação CORBASec, aplicando o modelo $UCON_{ABC}$ à sua camada de segurança.

2) Se trata de uma das primeiras implementações (se não a primeira) do modelo $UCON_{ABC}$ em sistemas computacionais, contendo suas principais características, como controles de autorização, obrigação, condição e também a mutabilidade de atributos.

3) Definição de um controle de acesso em 2 níveis: no nível do ORB, totalmente transparente para a aplicação, e durante a implementação da IDL, possibilitando trabalhar em conjunto com a aplicação. A combinação destas duas funcionalidades irá definir um controle de acesso muito mais preciso e rigoroso sobre as ações de um usuário sobre um sistema.

4) Centralização das políticas de controle de acesso, facilitando o processo de administração das políticas de segurança.

5) Definição e implementação de um controle de acesso baseado em Expressões lógicas e matemáticas, baseando-se em atributos do sujeito, objeto e variáveis de sistema. A maioria das implementações de controle de acesso estão baseadas em atributos fixos e com objetivos bem específicos. Nesta proposta, os atributos podem ser utilizados e avaliados conforme a necessidade imposta pelos administradores de segurança.

Capítulo 6

XEBACML - Linguagem para Definição de Políticas Baseadas em Expressões do Modelo JaCoWeb-ABC

O capítulo 5 apresentou toda a metodologia utilizada para adaptar e implementar o modelo $UCON_{ABC}$ à especificação CORBASec. Mas outro foco que também está sendo dado neste trabalho está em como definir estas políticas utilizadas pelo modelo JaCoWeb-ABC em um formato que seja de fácil compreensão e interpretação, como as linguagens Ponder e XACML.

Com relação a estas duas linguagens, nenhuma delas trata especificamente a utilização de políticas baseadas em expressões, não possuem compatibilidade com o modelo $UCON_{ABC}$ para utilização de atributos do sujeito e objeto. Também não focam a definição de políticas em 2 níveis de acesso, conforme o modelo proposto JaCoWeb-ABC, que utiliza um nível transparente, efetuado pelo ORB e também possibilita trabalhar em conjunto com a aplicação, que no caso seria utilizado durante a implementação da IDL.

Não utilizamos o XACML, pois por mais que ele permita ser extendida, seriam necessárias modificações significativas em sua estrutura, e de uma maneira geral, muitas vezes uma política simples, resultam em definições de políticas muito extensas. Outro fator é que o XACML não possui definições claras sobre a utilização de atributos, tanto do Sujeito como do Objeto, e desta forma, teriam que ser efetuadas modificações na base de sua linguagem. O Ponder também não é adaptável ao Modelo $UCON_{ABC}$, uma vez que o conceito de obrigação do Ponder é diferente do modelo $UCON_{ABC}$, além do Ponder ser mais voltado ao

gerenciamento, controle e distribuição de políticas sobre recursos de uma rede.

Assim, optamos em utilizar a linguagem base XML [32], também utilizada pelo XACML, mas para definir políticas baseadas em expressões, onde propomos uma nova linguagem nomeada como XEBACML (*eXtensible Expression Based Access Control Model Language*). O XEBACML, que estará sendo baseado para carregar os componentes definidos na figura 5.4 do JaCoWeb-ABC, permitirá definir políticas de controle de acesso muito mais próximas ao modelo $UCON_{ABC}$ em 2 níveis (abcORB e abcIDL), definindo também os atributos e valores do Sujeito e Objeto do Sistema, utilizados no processo de decisão. Como proposta, na seção 6.7 definimos como proposta uma especificação contendo uma definição dos tipos de variáveis, funções e operações suportados pela linguagem XEBACML e interpretadas pelo JaCoWeb-ABC. Para melhor compreensão sobre a utilização destas políticas do JaCoWeb-ABC, bem como a associação destas políticas com o modelo $UCON_{ABC}$, estão apresentados no Capítulo 7 (Prova de Conceito).

6.1 Especificação das Políticas do XEBACML

Para os componentes da arquitetura JaCoWeb-ABC (figura 5.4), utilizamos a linguagem XML [32] (Extensible Markup Language) para a definição dos atributos do sujeito e do objeto, e também para a definição das políticas de autorização, obrigação, condição e atualização de atributos. A figura 6.1 apresenta a definição dos atributos de um Sujeito, assim como uma lista das obrigações que um sujeito já cumpriu dentro do sistema.

```
<Subject ID="Bob">
    <Obligations>{informarEmail, efetuarLogin }</Obligations>
    <attribute name="perfil" type="string" value="cliente"/>
    <attribute name="creditos" type="Number" value="120.45" />
</Subject>
```

Figura 6.1 – Definição dos Atributos do Sujeito

Já a figura 6.2 apresenta a definição dos atributos de um Objeto e também a definição das políticas de decisão e atualização utilizadas pelo JaCoWeb-ABC. Estas definições poderão ser melhor representadas a partir dos exemplos apresentados no *Capítulo 7.*. Nesta definição, a política de decisão de acesso ABC pode ser efetuada de forma transparente para a

aplicação (PolicyABC_ORB) ou trabalhar em conjunto com a aplicação (PolicyABC_IDL). Para cada um destes processos de decisão, podem ser definidas políticas de atualização de atributos antes (preUpdate) ou depois (posUpdate) do acesso de um Sujeito sobre um Objeto. As especificações definidas em PolicyABC_ORB e PolicyABC_IDL são idênticas, e a única diferença entre elas é o nível onde serão avaliadas (no ORB ou na IDL).

```
<Object  interface="Loja"
         operation="Comprar">
    <attribute name="valor" type="Number" Value="20.40" />
    <attribute name="tipo" type="String" Value="A" />
    <PolicyABC_ORB / PolicyABC_IDL>
        <preUpdate>
            <Expression>
                 <attrib> AttribExpression</attrib>
                 <enable> LogicExpression </enable>
            </Expression>
        </preUpdate>;
        <Authorization>
           <Expression>
               <expr>LogicExpression</expr>
               <enable>LogicExpression</enable>
           </Expression>
        </Authorization>
        <Obligation>
             <expr>Lista Obrigações</expr>
             <enable>LogicExpression</enable>
        </Obligation >
        <Condition>
             <expr>LogicExpression</expr>
             <enable>LogicExpression</enable>
        </Condition>
        <posUpdate>
            <Expression>
                 <attrib> AttribExpression</attrib>
                 <enable> LogicExpression </enable>
            </Expression>
        </posUpdate>;
```

```
    </PolicyABC_ORB / PolicyABC_IDL>
</PolicyObject>
```

Figura 6.2 – Definição dos atributos e Políticas do Objeto

Estaremos detalhando na seqüência cada uma das definições das políticas de autorização, obrigação, condição e atualização de atributos. Nesta definição, também foram necessários especificar quais os tipos de dados que os atributos que o modelo JaCoWeb-ABC deve trabalhar, bem como o processo de avaliação de expressões lógicas e expressões de atribuição utilizadas pelas políticas.

6.2 Tag <enable> Expression

A tag "<*enable*>" definida em todas as políticas de decisão e atualização (figura 6.2) é opcional. Ela contém uma expressão lógica que avalia se a expressão definida pela política de decisão ou atualização deve ou não ser executada. No exemplo apresentado na figura 6.3, para o acesso de um sujeito a um objeto, o acesso é controlado para que no máximo 3 sujeitos acessem este objeto. Mas caso o cliente seja especial, não existirá restrições sobre este acesso.

```
<Authorization>
    <Expressions>
        <exprA> O->maximoAcessos < 3 </attrib>
        <enable> S->tipoCliente <> "especial" </enable>
    </Expressions>
</Authorization>
```

Figura 6.3 – Tag Enable Expression do JaCoWeb-ABC

6.3 Definição de Políticas de Atualização de atributos (Atributos mutáveis)

Definições: a política de atualização de atributos pode ser definida antes (preUpdate) ou depois (posUpdate) do processo de decisão. O processo de atualização utiliza expressões de atribuição que caracterizam a mutabilidade dos atributos. Estas expressões são definidas pela tag "attrib". No exemplo abaixo (figura 6.4), é efetuado uma atualização do atributo "creditos" do sujeito. Caso o cliente seja do tipo especial, é dado um desconto sobre o valor do serviço. Caso contrário, o valor total do serviço é cobrado do Sujeito.

```
<preUpdate ou posUpdate >
    <Expressions>
        <attrib> S->creditos = S->creditos – O->valor </attrib>
```

```
        <enable> S->tipoCliente <> "especial" </enable>
    </Expressions>
    <Expressions>
        <attrib>
           S->creditos = S->creditos –
                  (O->valor – O->valorDesconto)
        </attrib>
        <enable> S->tipoCliente = "especial" </enable>
    </Expressions>
</preUpdate ou posUpdate>
```

Figura 6.4 - Definição de Políticas para Atualização de Atributos do JaCoWeb-ABC

6.4 Definição de Políticas de Autorização (A)

A Autorização é baseada a partir da avaliação dos atributos do sujeito e objeto. A expressão lógica pela tag “<expr>” será utilizada no processo de autorização. Podemos exemplificar uma utilização neste caso (figura 6.5) quando uma determinada expressão de autorização limita a quantidade máxima de acessos simultâneos de clientes apenas quando o atributo do Sujeito “tipoCliente” seja diferente de “especial”. Neste caso, clientes especiais teriam acesso ao objeto sem este tipo de restrição.

```
<Authorization>
  <Expressions>
        <exprA>
           O->numeroAcessos < O->maximoAcessos
        </exprA>
        <enable>
            S->tipoCliente <> "especial"
        </enable>
  </Expression>
</Authorization>
```

Figura 6.5 - Definição de Políticas de Autorização do JaCoWeb-ABC

6.5 Definição de Políticas de Obrigação (B)

Em uma obrigação é avaliada a partir de um histórico de obrigações que já foram cumpridas por um sujeito. Cada sujeito possui o atributo “O*bligations*”que armazena um histórico de suas obrigações. Esta lista de obrigações é representada entre parênteses “{ ... , ..., ...}“, e separadas por “,”(vírgula).Durante o acesso de um sujeito sobre um objeto,

devem ser definidos na tag “*<listObligations>*” quais obrigações devem ter sido cumpridas pelo sujeito para que o acesso ao objeto seja permitido. No exemplo abaixo (figura 6.6), a obrigação “informarEmail” deve ser cumprida pelo sujeito, para que ele consiga acessar o objeto. Somente no caso do cliente ser do tipo especial, esta obrigação não será necessária.

```
<Obligation>
   <Expressions>
      <listObligation>
        { informarEmail }
      </listObligation>
      <enable>
          S->tipoCliente <> "especial"
      </enable>
   </Expression>
</Obligation>
```

Figura 6.6 - Definição de Políticas de Obrigação do JaCoWeb-ABC

Nesta situação, para o cumprimento desta obrigação por um sujeito, é possível definir em uma expressão de atribuição, a chamada de uma função para a inclusão desta obrigação (insertObligation) no seu atributo “Obligations” (figura 6.7). Logicamente, também foi criado uma função para a exclusão de uma obrigação (removeObligation) deste atributo. A chamada destas funções são interpretadas e executadas pelo objeto AttribExpression.

```
<posUpdate>
    <Expressions>
         <attrib> S.insertObligation("informarEmail")</attrib>
    </Expressions>
</posUpdate>
```

Figura 6.7 - Exemplo de Política para o Cuprimento de uma Obrigação

6.6 Definição de Políticas de Condição (C)

As condições são avaliadas baseadas em condições do sistema, podendo capturar valores como data e hora atuais do sistema ou a localização do Sujeito, baseada em seu IP. As condições podem utilizar também informações de atributos do sujeito e objeto. A figura 6.8 apresenta um exemplo, em que o acesso de um sujeito sobre um objeto somente será permitido durante o horário comercial, definido como sendo entre 8 até 18 horas e nos dias úteis (segunda-feira a sexta-feira). O acesso somente será permitido se o sujeito em questão

for o administrador do sistema. Note que a expressão de condição efetua uma chamada de funções, para captura da data e hora atuais do sistema. Desta forma, estas funções também são interpretadas pelo objeto LogicExpression.

```
<Condition>
   <Expressions>
      <exprC>
        SYSTEM.getCurrentDate.getHour() >= 8 and
        SYSTEM.getCurrentDate.getHour() <= 18 and
        SYSTEM.getCurrentDate.getDayWeek() >= 1 and
        SYSTEM.getCurrentDate.getDayWeek() <= 5
      </exprC>
      <enable>
          S->tipoCliente <> "administrador"
      </enable>
   </Expression>
</Condition>
```

Figura 6.8 - Políticas de Condição do JaCoWeb-ABC

6.7 Especificações da Linguagem XEBACML

As expressões utilizadas pelas políticas do JaCoWeb-ABC, definidas pela linguagem XEBACML, utilizam definições que estão apresentadas nesta seção. Aqui basicamente são apresentados os operadores lógicos, aritméticos, comparativos e de atribuição, bem como os tipos de atributos utilizados pelo JaCoWeb-ABC e as funções os quais ele suporta. Como a linguagem XEBACML, esta especificação não é fechada e possibilita que os usuários criem extensões das funções, tipos e funcionalidades definidas pela linguagem XEBACML.

6.7.1 Expressõe Lógicas

As expressões lógicas utilizam atributos do sujeito e objeto, bem como variáveis do sistema, para avaliar uma expressão lógica, que pode retornar verdadeiro ou falso. Os seguintes operadores são suportados pelo JaCoWeb-ABC:

Operadores Lógicos (and, or, not): utilizado para a ligação de duas ou mais expressões no caso do “and” e “or”. Ou para negação de uma expressão (no caso do “not”);

Ex.: <Expr1> and <Expr2> or (not <expr3>)

Operadores relacionais (=, >=, <=, >, <, <>): utilizado para comparação de 2 valores.

Ex.: O→value >= S→saldo

Funções: Os tipos Data, Vector e Matriz, além de funções do sistema (SYSTEM) podem ser utilizadas nas expressões. As funções de cada tipo serão mais bem detalhadas na seção B.2. Todas as funções definidas nesta especificação são suportadas pelos objetos LogicExpression e AttribExpression.

6.7.2 Tipos de Atributos e Operações suportados pelo JaCoWeb-ABC

Para as expressões das politicas suportadas pelas políticas do JaCoWeb-ABC, tivemos que definir inicialmente quais tipos de atributos este modelo iria suportar em suas expressões lógicas e de atribuição.

6.7.2.1 Tipo Inteiro (I)

Definição: valor numérico sem casas decimais;

```
Declaração:
   <attribute name="nomeAtributo" type="Integer" Value="integerValue" />
```

Operações suportadas: “+ , -, / , *” (operações de adição, subtração, divisão e multiplicação).

```
Ex.: S->maxErros + 1
```

6.7.2.2 Tipo Número (N)

Definição: valor numérico com casas decimais;

```
Declaração:
   <attribute name="nomeAtributo" type="Number" Value="numberValue" />
```

Operações suportadas: “+ , - , / , *”. (operações de adição, subtração, divisão e multiplicação).

```
Ex.: S->saldo – O->valor
```

6.7.2.3 Tipo String (S)

Definição: valor texto.

```
Declaração:
     <attribute name="nomeAtributo" type="String" Value="stringValue" />
```

Operações suportadas: “+”. Apenas possibilita uma concatenação de Strings.

```
Ex.: S->nome + S->sobrenome
```

6.7.2.4 Tipo Data/ Hora (D)

Definição: valor que contém data/hora.

```
Declaração:
  <attribute name="nomeAtributo" type="D" Value="dateValue" />
```

Funções: O tipo data não possibilita trabalhar com a data diretamente, mas somente a partir de funções. As funções suportadas por este tipo são:

- setDate(data): define uma data para o atributo (parâmetro String no formato “mm-dd-yyyy”). Pode ser utilizada a função definida pelo objeto SYSTEM. Ex.: attribDate.setDate(SYSTEM.getDate()).
- setTime(hora): define a hora para o atributo (parâmetro String no formato “hh:mm:ss”). Pode ser utilizada a função definida pelo objeto SYSTEM. Ex.: attribTime.setTime(SYSTEM.getTime()).
- getDay(): retorna o dia referente à data. O seu retorno é um número inteiro referente ao dia do mês.
- getMonth(): retorno o mês referente à data. O seu retorno é um número inteiro entre 1 e 12.
- getYear(): retorna o ano referente à data. O seu retorno é um número inteiro.
- getDayWeek(): retorna o dia da semana. O seu retorno é um número inteiro entre 1 e 7.
- getHour(): retorna a hora referente do atributo. O seu retorno é um número inteiro entre 0 e 23.
- getMinutes(): retorna os minutos do atributo. O seu retorno é um número inteiro entre 0 e 59.

- getSeconds(): retorna os segundos do atributo. O seu retorno é um número inteiro entre 0 e 59.
- getDifTime (AttribDate): retorna a quantidade de segundos entre 2 objetos do tipo Date. O seu retorno é do tipo inteiro.

6.7.2.5 Tipo Vector (V)

Definição: vetor de elementos de um determinado tipo de atributo. Suportados apenas os tipos Inteiro, Número, String e Data. Este tipo deve ser definido no campo typeData, incluindo o caracter do tipo correspondente. A definição da lista de atributos deve estar entre parênteses "{ ... }", e separadas por vírgula. Neste tipo de dado, não existem operações de atribuição.

```
Declaração:
    <attribute name="nomeAtributo" type="Vector" typeData="charType">
         {dado1, dado2, dado3}
    </attribute>
```

Funções:

- addElement(attribute): adiciona um elemento ao vetor.
- removeElement(index): elimina um elemento do vetor, a partir do índice passado por parâmetro.
- clear(): elimina todos os elementos do vetor.
- getElement(index): retorna um elemento do vetor, a partir do índice passado.
- containsAllValues(<vector>): verifica se todos os elementos do vetor passado por parâmetro estão contidos dentro vetor.
- containsAnyValues(<vector>): verifica pelo menos um elemento do vetor passado por parâmetro está contido dentro do vetor.
- contains(value): verifica se o atributo passado por parâmetro está contido dentro do vetor.

6.7.2.6 Tipo Matriz(M)

Definição: Matriz de elementos de um determinado tipo de atributo, contendo uma chave e o valor. Cada chave e valor é definido entre parênteses, no seguinte formato "{ {key1, value1}, {key2, value2} }". A chave é do tipo String, e o valor pode ser do tipo Inteiro,

Número, String e Data. Este tipo deve ser definido no campo typeData, incluindo o caracter do tipo corresponente..

```
Declaração:
  <attribute name="nomeAtributo" type="M" typeData="charType">
     { {key1, value1} ,
       {key2, value2} ,
       {key3, value3} }
  </attribute>
     Ex.: S→notas= {  {“Geografia”,   8.5},
                      {“Matemática”, 10.00},
                      {“História”,    6.5} };
```

Funções:

- addKeyValue(key, value): adiciona uma chave e um valor à matriz.
- removeKeyValue(key): remove um valor da matriz, baseado em sua chave.
- getValue(key): retorna um elemento da matriz, baseado em sua chave.
- setValue(key, value): altera um valor da matriz, baseado em sua chave.
- clear(): elimina todos os elementos da matriz.

6.7.2.7 Funções definidas em SYSTEM

Definição: Variáveis de sistema podem ser utilizadas por meio da palavra reservada SYSTEM, que possui um conjunto de funções, como a captura da data/hora do sistema, que podem ser utilizadas no processo de decisão.

Funções:

currentDate(): se trata de uma variável do tipo Date, mas que contém informações sobre a data/hora correntes. Pode ser utilizado pela função setDate.

6.8 Conclusão

Este capítulo descreveu a proposta de criação da linguagem XEBACML, que utiliza a linguagem XML para representar as políticas de controle de acesso do modelo $UCON_{ABC}$ utilizadas na proposta JaCoWeb-ABC. O resultado foi que a representação desta linguagem ficou muito mais compreensível e simplificado que o XACML, uma vez que cada elemento de decisão pode ser definido por meio de expressões, facilitando a construção de políticas mais complexas, que irão consumir muito menos linhas para a esta definição. Para este

capítulo, podemos citar as seguintes contribuições:

1) Uma linguagem que pode ser padronizada por meio do XML, e capaz de especificar políticas de controle de acesso baseadas no modelo $UCON_{ABC}$, possibilitando definir controles de Autorização, Obrigação e Condição, além de permitir o processo de mutabilidade de atributos antes (pre-Update) e depois (pos-Update) do processo de decisão.

2) A criação de uma nova linguagem para definição de políticas de controle de acesso (XEBACML), baseada em expressões lógicas e aritméticas.

3) Uma pré-especificação dos tipos e funções suportados pelo JaCoWeb-ABC e que permite uma extensão de novas funcionalidades.

Capítulo 7

Prova de Conceito e Análise de Desempenho

7.1 Implementação do Protótipo

Para a aplicação do modelo JaCoWeb-ABC, utilizamos o JacORB 2.2 [35], que é um ORB free desenvolvido em Java pela Freie Universitaet Berlin/XTRADYNE Technologies AG da Alemanha. Foi utilizada a versão do Java 1.5.0_01. Para o processo de autenticação de principais, foram utilizados os recursos disponíveis pelo JacORB, a partir da autenticação SSL, baseado em chaves públicas e privadas, garantindo também a confidencialidade das informações por um canal cifrado. Para este processo, foram necessários compreender, adaptar e implementar as funcionalidades de algumas classes que só possuíam a interface de chamada, como ServerInitializer, ServerAccessDecisionInterceptor, AccessDecisionImpl, RequiredRightsImpl, SunJssePrincipalAuthenticatorImpl, aplicando as funcionalidades básicas do modelo CORBASec tradicional, para posteriormente implementarmos o modelo JaCoWeb-ABC.

Para a implementação, os componentes do JaCoWeb-ABC, definidos na figura 5.4, tiveram que ser implementados. Para o processo de decisão e atualização, tivemos que criar os objetos LogicExpression e AttribExpression, capazes de interpretar e calcular expressões a partir dos atributos do sujeito e objeto, conforme definidos na figura 5.3. Estes objetos são os componentes mais complexos deste projeto, pois tivemos que desenvolvê-los, de forma que eles pudessem calcular as expressões lógicas, aritméticas, atribuição e funções definidas e suportadas pelo JaCoWeb-ABC. O carregamento das políticas ABC foi implementado no

construtor do objeto AccessDecisionABC, em que é chamado quando este objeto é instanciado pela primeira vez, que efetua a leitura das políticas que estão definidas dentro de um arquivo. O objeto AccessDecisionInterceptor foi configurado de forma que as invocações de métodos de objetos CORBA, o objeto AccessDecisionABC_ORB fosse utilizado. O objeto AccessDecisionABC_IDL também foi criado para disponibilizar o controle de acesso a ser implementado no nível da aplicação, durante a implementação de uma IDL.

7.2 Prova de conceito

Para validação do modelo, retiramos alguns exemplos de Authorization, oBligation e Condition definidos em [24]. Para cada um deles foi implementada uma pequena aplicação, analisando o seu contexto sobre o modelo CORBA e adaptando as políticas de segurança para a linguagem XEBACML. Para visualização das políticas de segurança definidas, as ações dos sujeitos sobre os objetos e também as mudanças de seus atributos, criamos uma interface gráfica, mostrada na figura 7.3.

Nesta interface, o responsável pela monitoração pode selecionar um Sujeito e um objeto. Clicando no botão *Refresh*, são carregados a identificação sujeito e objeto, os seus atributos, bem como as políticas do $UCON_{ABC}$ definidas para o acesso ao objeto.

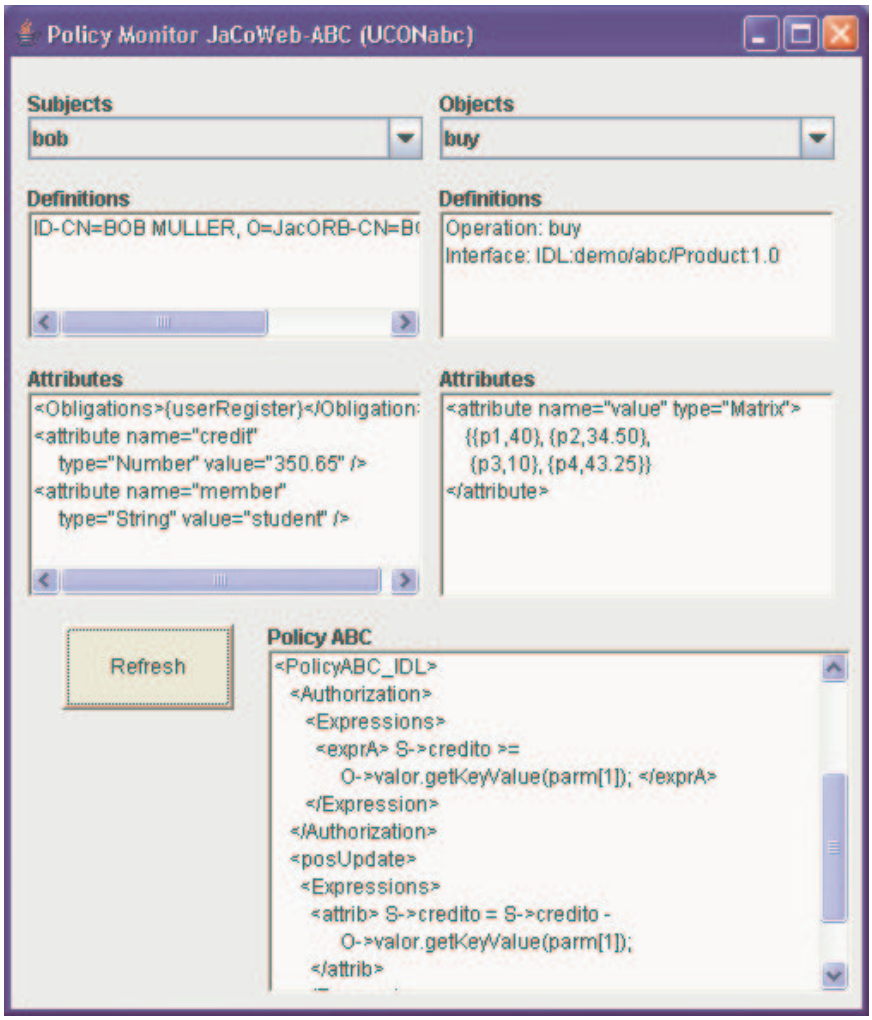

Figura 7.1 – Monitor de Políticas do JaCoWeb-ABC

7.2.1 Definição do Modelo MAC (Exemplo 1)

Definição: o modelo Obrigatório de Bell-LaPadula (BLP) é baseado em uma Security Label (relação de dominância), onde cada objeto possui uma classificação (*classification*) e um sujeito possui um nível de habilitação (*clearance*). Neste controle, um sujeito pode ler apenas objetos com uma classificação menor ou igual ao seu nível de habilitação, e pode escrever apenas em objetos com um nível superior ao seu.

Política $UCON_{preA0}$

L is a lattice of security labels with dominamce relation $\geq$.

clearance: S→L

classification: O→L

ATT(S) = {clearance}

ATT(O) = {classification}

Allowed(s, o, read) = clearance(s) ≥ classification(o)

Allowed(s, o, write) = clearance(s) ≤ classification(o)

Política JaCoWeb-ABC$_{abcORB0}$

No CORBA, devem ser construídos 2 métodos: uma para leitura (read) e outra para gravação (write).

```
<Subject ID="Bob">
  <attribute name="clearance" type="Integer" value="2" />
</Subject>

<Object interface="Object1" operation="read">
  <attribute name="classification" type="Integer" value="1" />
  <PolicyABC_ORB>
     <Authorization>
         (S->clearance >= O->classification)
      </Authorization>
  </PolicyABC_ORB>
</Object>

<Object interface="Object1" operation="write">
    <attribute name="classification" type="Integer" value="1" />
    <PolicyABC_ORB>
        <Authorization>
            (S->clearance <= O->classification)
        <Authorization>
    <PolicyABC_ORB>
</Object>
```

7.2.2 DRM pay-per-use with pre-credit. (Exemplo 22)

Definição: Se o crédito de um sujeito não é menor que o valor do objeto requisitado, o seu acesso é permitido. Uma vez que a requisição é permitida, os créditos do sujeito são reduzidos pelo valor definido pelo objeto. Nesta situação, o valor do objeto possui um valor único.

Política UCON$_{preA1}$

M é o crédito acumulado.

credit: S→M

value: OxR→M

ATT(s): {credit}

ATT(o,r): {value}

Allowed(s,o,r)➔credito(s)>=valor(o,r)

preUpdate(credito(s)): credito(s) = credito(s)-valor(o,r);

***Política* JaCoWeb-ABC$_{abcORB3}$**

```
<Subject ID="name">
  <attribute name="credit" type="Number" value="145.45" />
</Subject>

<Object interface="Product" operation="buy" />
    <attribute name="value" type="Number" value="34.50" />
    <PolicyABC_ORB>
        <Authorization>
            S->credit >= O->value
        </Authorization>
        <posUpdate>
            S->credit= S->credit – O->value
        </posUpdate>
    </PolicyABC_ORB>
</Object>
```

7.2.3 DRM pay-per-use, on credit, multiple values. (Exemplo 23)

Definição: Se o crédito de um sujeito não é menor o valor do objeto requisitado, o seu acesso é permitido. Uma vez que a requisição é permitida, os créditos do sujeito são reduzidos pelo valor definido pelo objeto. Nesta situação, o objeto possui uma lista de produtos com valores diferentes, que podem ser selecionados a partir de sua identificação.

Política UCON$_{preA1}$

M é o crédito acumulado.

credit: S→M

value: OxR → 2^M

ATT(s): {credit}

ATT(o,r): {value}

M = {m | m Є value(o, r), m <= credit (S)}, a set of available values for selection.

Allowed(s,o,r)➔∃m ∈ value(o, r), m <= credit (s)

preUpdate(credit(s)): credit(s) = credit(s) - λ(M), onde λ é uma função que seleciona o valor para atualização.

***Política* JaCoWeb-ABC$_{abcIDL3}$**

Para definição desta política no CORBA, a informação do valor do objeto a ser selecionada estará baseada em uma informação externa à camada JaCoWeb-ABC, pois o usuário irá selecionar qual é o objeto que ele deseja adquirir. Nesta situação, a política trabalhará em conjunto com a aplicação, e na implementação da IDL, deverá ser passada a identificação do Produto para a API "access_allowed" do objeto AccessDecisionABC_IDL. A implementação da IDL também está sendo apresentada, com sua codificação feita em Java.

```
<Subject ID="name">
  <attribute name="credit" type="Number" value="145.45" />
</Subject>

<Object interface="Product" operation="buy" />
    <attribute name="value" type="Matrix">
        {{p1,40}, {p2,34.50},
         {p3,10}, {p4,43.25}}
    </attribute>
    <PolicyABC_IDL>
        <Authorization>
            S->credit >= O->value.getKeyValue(parm[1])
        </Authorization>
        <posUpdate>
            S->credit= S->credit - value.getKeyValue(parm[1])
        </posUpdate>
    </PolicyABC_IDL>
</Object>
```

Implementação da IDL no servidor (Código Java)

```
Public String buy (String IDProduct)
{
   boolean ret;
   Vector vet = new Vector();
   // preparação do dado
   vet.add(IDProduct)
   ret = AccessDecisionABC_IDL.access_allowed (this, "buy", current, vet);
   if(ret == true)
      return "ok";
   else
      return "error";
}
```

7.2.4 Modelo DAC / CORBASec Tradicional (Exemplo 2)

Definição: Políticas discricionárias tradicionalmente utilizam uma matriz de acesso utilizada para definir uma lista de controle de acesso. Neste caso, uma identidade (ID) e uma lista de controle de acesso (ACL) são os atributos do sujeito e objeto respectivamente. ACL é uma função que mapeia o objeto a múltiplos IDs. Nesta situação, estaremos apresentando como estaríamos portando as políticas discricionárias da especificação CORBASec sobre o JaCoWeb-ABC.

Política UCON$_{preA0}$

N is a set of identiy names

Id: S → N, one to one mapping

ACL: O → 2^{NxR}

ATT(S) = {id}

ATT(O) = {ACL}

allowed(s,o,r) → (id(s), r) Є ACL(o)

***Política* JaCoWeb-ABC$_{abcIDL0}$**

Na política JaCoWeb-ABC, definimos o atributo “grantedRights” do tipo Vector para o Sujeito, que irá conter uma lista de direitos (ver tabela 2). Para o objeto, foi definida uma política genérica, que possui o atributo “requiredRights” que também é do tipo Vector, e contém os direitos requeridos para o seu acesso e também o atributo combinator que são atributos do próprio objeto RequiredRights (ver tabela 3). O objeto possui 2 políticas de autorização, que são selecionadas a partir do valor do atributo *combinator*, que utiliza ou a função *containsAnyValues* caso o *combinator* for “any” ou *containsAllValues* caso o *combinator* for “all”.

```
<Subject ID="name">
    <attribute name="grantedRights" type="Vector" typeData="C">
         {"r", "W", "X"}
    </attribute>
</Subject>
<Object interface="Object" operation="nameMethod">
```

```
 <attribute name="requiredRights" type="Vector" typeData="C">
      {"r", "x"}
 </attribute>
 <attribute name="combinator" type=”String" value="any" />
<PolicyABC_ORB>
   <Authorization>
       <Expressions>
          <exprA>
             O->requiredRights.containsAnyValues(S->grantedRights)
          </exprA>
          <enable>
              O→combinator="any"
          </enable>
       </Expression>
       <Expressions>
          <exprA>
             O→requiredRights.containsAllValues (S→grantedRights)
          </exprA>
          <enable>
              O→combinator="all"
          </enable>
       </Expression>
   </Authorization>
</PolicyABC_ORB>
</Object>
```

7.2.5 Licence Agree (Exemplo 8)

Define uma política de obrigação, onde o acesso a um determinado objeto somente será permitido caso o Sujeito efetue o aceite do acordo de licenciamento (Licence Agreement).

Política UCON$_{ABC}$ (*preA0*)

A licence agreement obligation, every time (without attribute).

OBS=S

OBO={licence_agreement)

OB={agree}

getPreOBL(s,o,r): {{s,licence_agreement,agree}}

allowed(s,o,r)➔ preFulFilled(s,licence_agreement, agree}

***Política* JaCoWeb-ABC$_{abcIDL0}$**

Para a política JaCoWeb-ABC, existirá a necessidade de efetuar 2 passos. O primeiro é a definição de um método para o cumprimento da obrigação para o aceite do acordo de

licenciamento. E o outro é verificação desta obrigação no objeto que o usuário deseja acessar.

```
<Subject ID="Bob">
   <Obligations>
      {}
   </Obligations>
</Subject>

<Object interface="Service" operation="agreeAccept">
   <PolicyABC_ORB>
      <posUpdate>
         <Expressions>
            <attrib> S.insertObligation("agreeAccept") </attrib>
         </Expressions>
      </posUpdate>
   </PolicyABC_ORB>
</Object>

<Object interface="Service" operation="getService">
   <PolicyABC_ORB>
       <Obligation>
          <Expressions>
             <listObligation>
                   { agreeAccept }
             </listObligation>
          </Expression>
       </Obligation>
   </PolicyABC_ORB>
</Object>
```

7.2.6 Time Limitation (exemplo 13)

Definição: O acesso a objetos é permitido somente em períodos determinados para um sujeito. Para isso, um sujeito possui o atributo “shift” que define qual o período do seu acesso (“day” ou “night”). Os horários definidos para o dia são das 8 às 16 horas e para a noite, das 16 às 24 horas.

Política $UCON_{ABC}$ (preC0onC0)

dayH (8am to 4pm) , nightH (4pm to 12pm)

currentT is current time.

preCON:{{currentT Є dayH},

{currentT Є nightH}}

getPreCON(s,o,r) = { {currentT Є dayH, if shift(s) = “day”,

currentT Є nightH, if shift(s) = “night”}};

getOnCON(s,o,r) ={ {currentT Є dayH, if shift(s) = “day”,
currentT Є nightH, if shift(s) = “night”}};
allowed(s,o,r)➔ preConChecked(getpreCON(s,o,r))
stopped(s,o,r)➔ ¬ preConChecked(getpreCON(s,o,r))

Política do abcIDL

Foi definido o atributo shift para o Sujeito e para o objeto foram utilizadas as funções de captura da data/hora correntes do sistema, a partir da função SYSTEM. A observação a ser identificada nesta política é que que a função getHour() retorna o valor da hora, entre 0 e 23.

```
<Subject ID="Bob">
    <attribute name="shift" type="S" value="day">
</Subject>

<Object interface="serviceWeb" operation="readText">
   <PolicyAC_ORB>
      <Condition>
         <Expressions>
            <exprC>
                 SYSTEM.getCurrentDate.getHour() >= 8 and
                 SYSTEM.getCurrentDate.getHour() < 16
            </exprC>
             <enable>
                 S->shift="day"
             </enable>
         </Expression>
      </Condition>
      <Condition>
         <Expressions>
            <exprC>
                 (SYSTEM.getCurrentDate.getHour() >= 16  and
                  SYSTEM.getCurrentDate.getHour() <= 23) or
                  SYSTEM.getCurrentDate.getHour() = 0
            </exprC>
             <enable>
                 S->shift="night"
             </enable>
         </Expression>
      </Condition>
   </PolicyAC_ORB>
</Object>
```

7.3 Análise de Desempenho

Para execução de testes de desempenho do modelo, utilizando o exemplo jaco.demo.benchmark, que vem junto com o pacote do JacORB, tendo como base 3 avaliações sobre a invocação de um objeto CORBA, apresentadas na figura 7.1. Estas medições foram executadas da seguinte forma:

1) Utilização normal: invocações de métodos sem políticas de segurança.
2) Com SSL: invocações de métodos com a utilização de criptografia baseada em SSL.
3) Políticas do modelo JaCoWeb-ABC (SSL+ABC): invocações de métodos com a utilização de criptografia e a intervenção das políticas de segurança definidas para o objeto AccessDecisionABC.

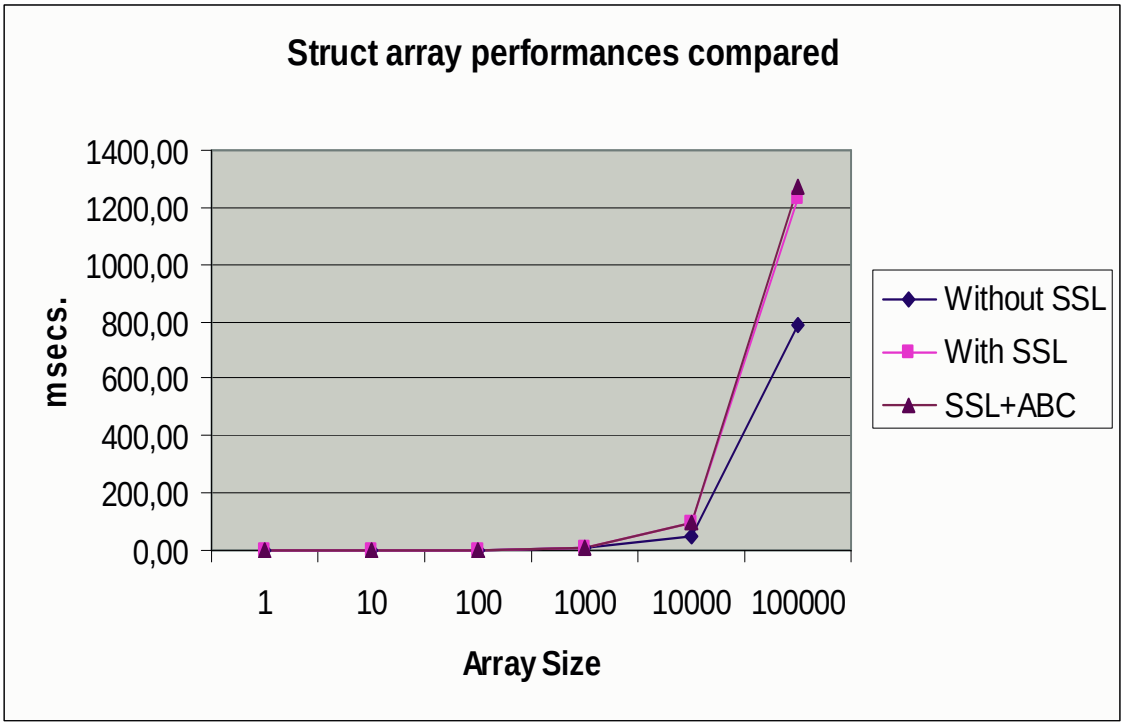

Figura 7.2 – Performance do Modelo JaCoWeb-ABC

Nesta avaliação foram definidas políticas uma política de autorização e uma política de atualização, para os métodos ping(), intTransfer(), octetTransfer(), structTransfer(), stringTransfer() apresentados abaixo (figura 7.2):

```
<Subject ID="Bob">
    <attribute name="contador" type="Integer" value="0" />
    <attribute name="grantedRights" type="Vector" typeData="C">
        {"g", "s", "u"}
    </attribute>
</Subject>
```

```
<Object interface="benchmark" operation="intTransfer">
    <attribute name="requiredRights" type="Vector" typeData="C">
         {"r", "x"}
    </attribute>
    <attribute name="combinator" type=”String" value="any" />

   <PolicyABC_ORB>
      <Authorization>
          <Expressions>
             <exprA>
                O->requiredRights.containsAnyValues(S->grantedRights)
             </exprA>
             <enable>
                 O→combinator="any"
             </enable>
          </Expression>
          <Expressions>
             <exprA>
                O→requiredRights.containsAllValues (S→grantedRights)
             </exprA>
             <enable>
                 O→combinator="all"
             </enable>
          </Expression>
      </Authorization>
      <posUpdate>
         <Expressions>
            <attrib> S->contador = S->contador + 1 </attrib>
         </Expressions>
      </posUpdate>
   </PolicyABC_ORB>
</Object>
```

Figura 7.3 – Exemplo de Política do JaCoWeb-ABC para Análise de Desempenho

7.4 Conclusão

A implementação do protótipo foi uma das fases mais complexas deste trabalho, uma vez que todo o conhecimento sobre o funcionamento do CORBA no JacORB 2.2 [35] teve que ser analisado. Mesmo o processo sobre as funcionalidades de segurança do JacORB não está muito bem especificado, além de muitas implementações ainda não estarem concluídas, e por este motivo, muitas funcionalidades tiveram que ser adaptadas para que o modelo se tornasse funcional.

A partir dos exemplos aplicados na prova de conceito extraídos de [24], é possível constatar que a proposta desta dissertação é compatível com o modelo de controle de acesso $UCON_{ABC.}$ A análise de desempenho mostra que o modelo pode possuir uma boa performance. Mas esta avaliação também depende muito da quantidade de políticas de decisão e atualização que deve ser executado em cada acesso aos Objetos, sendo fundamental que a implementação dos objetos LogicExpression e AttribExpression esteja codificada de forma bem otimizada.

Capítulo 8

Conclusão

O modelo $UCON_{ABC}$ é uma nova geração de controle de acesso, que permite um controle mais preciso e dinâmico sobre a utilização de usuários aos objetos de um sistema. O modelo JaCoWeb-ABC efetuou mudanças significativas com relação à última especificação do CORBASec divulgada pela OMG [19]. Não foram efetuadas mudanças na implementação dos objetos CORBA relacionada à parte Cliente e também questões sobre a administração das políticas de segurança englobada neste trabalho. Assim, novos trabalhos necessitarão ser continuados para poder englobar todas estas mudanças à especificação CORBASec da OMG.

O JaCoWeb-ABC definiu um modelo capaz de segregar as atividades efetuadas por uma camada de controle de acesso, de uma forma que determinados acessos pudessem ser controlados de forma transparente para a aplicação (abcORB). Em outros casos, a aplicação deverá trabalhar em conjunto com o JaCoWeb-ABC (abcIDL), devido à decisão sobre o acesso ao objeto depender de informações externas que devem ser passados pela aplicação à camada JaCoWeb-ABC. A combinação destes dois conceitos permite que um sistema tenha um controle mais preciso e rigoroso sobre as ações de um usuário, possibilitando segregar a camada de controle de acesso da lógica de negócio, que muitas vezes está implementada junto

com a aplicação, que nem sempre está adequada e padronizada, e livre de falhas por parte do desenvolvedor.

Com a utilização da linguagem XEBACML, foi possível especificar uma linguagem capaz de definir as políticas de segurança definidas pelo modelo $UCON_{ABC}$, de uma forma muito mais próxima à sua funcionalidade, que é baseada em expressões lógicas, aritméticas e de atribuição. A linguagem XACML é poderosa, mas uma de suas desvantagens está no resultado final de suas políticas, que muitas vezes acabam sendo muito extensas. Ela também não trabalha com o conceito de mutabilidade de atributos. Por estes motivos, resolvemos não extender a linguagem XACML e propusemos uma nova linguagem. Mas como trabalhos futuros, podemos propor a integração das funcionalidades do XEBACML ao XACML, unificando estes trabalhos, como a sua definição de atributos do Sujeito e Objeto, controles baseados em um nível transparente de controle de acesso e o seu nível de aplicação extraídos do JaCoWeb-ABC, assim como os controles de autorização, obrigação e condição, baseados em expressões do modelo $UCON_{ABC}$.

Devido ao CORBA trabalhar com invocações de métodos remotos, em um formato transacional, acreditamos que os mesmos princípios aplicados nesta dissertação poderão também ser utilizados em camadas de segurança de outras tecnologias similares, como RMI, DCOM e WebServices, além de WebServers, como o WebSphere da IBM ou IIS da Microsoft. Para esta implementação ser possível, estas tecnologias necessitam analisar a sua arquitetura, para identificar e associar os seus componentes ao modelo $UCON_{ABC}$. Feito isso, deve ser avaliado como serão efetuados a segregação das políticas de segurança que se encontram pela camada de invocação do método remoto (neste caso, o modelo abcORB) e disponibilizar APIs às aplicações para que elas possam atuar em conjunto com as políticas ABC definidas no servidor (modelo abcIDL).

Em [12], é avaliado que o $UCON_{ABC}$, por se tratar simplesmente de um modelo teórico que ainda não possui protótipos de implementados, seria significativamente caro e complexo de ser implementado em sistemas reais. Mas acreditamos que pelo potencial apresentado e os resultados obtidos a partir do protótipo implementado, muitos trabalhos passarão a utilizá-lo como referência para a implementação de um nível mais elevado de controle de acesso.

Referências Bibliográficas

[1] ANDERSON, A. “Core and hierarchical role based access control (RBAC) profile of XACML”. v2.0. OASIS Standard, 1 February 2005

[2] BARTAL, Y.; MAYER, A.; NISSIM, K.; Wool, A., “Firmato: A Novel Firewall Management Toolkit”, IEEE Symposium on Security and Privacy, Oakland, Califórnia, 1999.

[3] BELL, D. E., and LAPADULA, L. J., “Secure Computer Systems: Unified Exposition and Multics Interpretation”. MITRE Technical Report MTR-2997 Rev. 1, MITRE Corporation, Bedford, MA, March 1976.

[4] BEZNOSOV, K., DENG, Y., BLAKLEY, B., Burt, C., BARKLEY, J. “A Resource Access Decision Service for CORBA-based Distributed Systems”, Proceedings of the 15th Annual Computer Security Applications Conference, Phoenix, Arizona, USA. 1999.

[5] BEZNOSOV, K., ESPINAL, L., and DENG, Y., “Performance Considerations for CORBA-based Application Authorization Service,” PODC Middleware Symposium (pending acceptance), 2000

[6] BIBA, K. J. “Integrity Considerations for Secure Computer Systems”. MITRE Technical Report MTR-3153, MITRE Corporation, Bedford, MA, 1977.

[7] BROSE, G., “JacORB: Implementation and Design of a Java ORB”, Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems, Cottbus, Germany, Chapman &

Hall, 1997.

[8] DAMIANOU, N., DULAY, N., LUPU, E., and SLOMAN, M., “The Ponder Policy Specication Language”. Proceedings of the Workshop on Policies for Distributed Systems and Networks. 2001.

[9] DOWNS, D. D., RUB, J. R., KUNG, K. C., and JORDAN, C. S., “Issues in Discretionary Access Control,” Proceedings of the 1985 IEEE Symposium on Security and Privacy, pp. 208-218, April 1985.

[10] FERRAIOLO, D.F. and KUHN, D.R. "Role Based Access Control" 15th National Computer Security Conference - the original RBAC paper. 1992.

[11] FERRAIOLO, D., SANDHU, R., GAVRILA, S., KUHN, D.R. and chandramouli, R., "A Proposed Standard for Role Based Access Control, ACM Transactions on Information and System Security , vol. 4, no. 3 - draft of a consensus standard for RBAC. 2001

[12] HE, Q., “Privacy Enforcement with an Extended Role-Based Access Control Model”. North Carolina State University Computer Science Technical Report TR-2003-09. 2003.

[13] KAGAL, L., FININ, T., and JOHSHI, A., “A Policy Language for Pervasive Computing Environment”. Policy 2003: Workshop on Policies for Distributed Systems and Networks. Springer-Verlag. 2003

[14] LAMPSON, B. W. “Protection”. In Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, pages 437–443. Princeton University, March 1971.

[15] MOSES, T., “OASIS eXtensible Access Control Markup Language (XACML)” Version 2.0, OASIS Standard, 2005,

[16] OASIS. Organization for the Advancement of Structured Information

Standards. URL: http://www.oasis-open.org/

[17] OBELHEIRO, R. R., “Modelos de Segurança Baseados em Papéis para Sistemas de Larga Escala: A Proposta RBAC-JACOWEB”. Dissertação de mestrado, PGEEL–UFSC, Florianópolis, SC, Fevereiro de 2001.

[18] OBELHEIRO, R. R., FRAGA, J. S. “Role-Based Access Control for CORBA Distributed Object Systems” In: 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS´2002), San Diego, CA., 2002

[19] OMG – Object Management Group, "Security Service Specification, v1.8.", OMG Document, 2002.

[20] OMG – Object Management Group, "The Common Object Request Broker 2.0/IIOP Specification", OMG Document, 1996.

[21] OMG. Object Management Group. “A Discussion of the Object Management Architecture”. OMG Document 00-06-41, 1997.

[22] OMG. Object Management Group. “The Common Object Request Broker: Architecture and Specification”. OMG Document 00-11-07, 2000.

[23] OMG. Object Management Group. “Resource Access Decision Facility”, v1.0. OMG Doc. 99-03-02, 1999.

[24] PARK, J., and SANDHU, R., “The $UCON_{ABC}$ Usage Control Model” ACM Transactions on Information and System Security (TISSEC), Feb. 2004.

[25] PARK, J., and SANDHU, R., “Usage Control: A Vision for Next Generation Access Control”, The Second International Workshop

"Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS), St. Petersburg, Russia, 2003.

[26] PARK, J., and SANDHU, R., “Towards Usage Control Models: Beyond Traditional Access Control”, Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT’02), Monterey, CA, pp. 57-64, ACM, 2002.

[27] RYUTOV, T. V., NEUMAN, B. C., and KIM, D., “Dynamic Authorization and Intrusion Response in Distributed Systems”, In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, D.C. 2003.

[28] SANDHU, R., “Lattice-based access control models”. IEEE Computer, 26(11) 9-19, 1993.

[29] SANDHU,R., “Role-Based Access Control”, Advances in Computer Science, vol. 46, Academic Press, 1998.

[30] SUN’s XACML Guide.
URL: http://sunxacml.sourceforge.net/guide.html

[31] SUN’s XACML implementation,
URL: http://sunxacml.sourceforge.net/

[32] W3C. Extensible Markup Language.
URL: http://www.w3c.org/XML

[33] WANGHAM, M. S., LUNG, L. C., MERKLE, WESTPAHLL, C. M., FRAGA, J. S., PADILHA, R., SOUZA, L., “Projeto, Implementação e Avaliação do JaCoWeb Security”. Anais do XXVI Conferencia Latinoamericana de Informatica - CLEI'2000. Estado do México, México.

2000.

[34] WANGHAM, M. S.,. “Estudo e Implementação de um Esquema de Autorização Discricionária Baseado na Especificação CORBAsec”. Dissertação de mestrado, PGEEL–UFSC, Florianópolis, SC, 2000.

[35] WESTPHALL, C. M. ; FRAGA, Joni da Silva ; WESTPHALL, Carlos Becker ; BIANCHI, Silvia Cristina Sardela . “Políticas de Segurança Obrigatórias: Bell e Lapadula no CORBAsec”. v. 2. p. 846-861., In: SBRC 2002 - Búzios - RJ. SBRC 2002 - Rio de Janeiro : SBC-NCE-UFRJ, 2002.

[36] WESTPHALL, C. M. and Fraga, J. S., “A Large-Scale System Authorization Scheme Proposal Integrating Java, CORBA and Web Security Models and a Discretionary Prototype”. In Proc. IEEE LANOMS’99, pages 14–25, 1999.

[37] WESTPHALL, C. M., FRAGA, J. S. Fraga, and WANGHAM, M. S., “PoliCap—Um Serviço de Política para o Modelo CORBA de Segurança”. In Anais do 18o Simpósio Brasileiro de Redes de Computadores (SBRC 2000), pages 355–370, Belo Horizonte, MG, maio de 2000.

[38] WESTPHALL, C. M., FRAGA, J. S., and LUNG, L. C., “Jacowe - A Large - Scale System Authorization Scheme Integrating Java, CORBA and Web Security Models”. Anais da XXV Conferencia Latinoamericana de Informatica - CLEI'99, p.1025-1036. Assunção, Paraguai, 1999.

[39] ZHANG, X., PARK, J., PARISI-PRESICCE, F., and SANDHU, R., “A Logical Specification for Usage Control”, 9th ACM Symposium on Access Control Models and Technologies, 2004.

Apêndice A

Definições dos Modelos da Família $UCON_{ABC}$

A.1 - $UCON_{preA}$ – pre-Authorizations Models.

Autorizações podem ser consideradas como o principal processo de um controle de acesso, que é vastamente discutido em disciplinas de controle de acesso. Tradicionamente, a grande maioria das pesquisas relacionadas a controle de acesso focam sobre pre-autorizações, que são os acessos a serem decididos antes do direito ser exercido. O UCON possui 3 modelos baseados no $UCON_{preA}$. O $UCON_{preA_0}$, em que não são definidos políticas para atualizações de atributos (imutabilidade), e também os $UCON_{preA_1}$ e $UCON_{preA_3}$, que definem polítias de ataualização (mutabilidade) de atributos antes ou depois do acesso do sujeito sobre o objeto respectivamente. Para estes 3 modelos, foram criadas as seguintes definições:

Definição 1: O modelo $UCON_{preA_0}$ é composto pelos seguintes componentes:

- S, O, ATT(S), ATT(O) e preA: sujeitos, objetos, atributos do sujeito, atributos do objeto e pré-autorização respectivamente.

- allowed(s, o, r) $\Rightarrow$ preA(ATT(s), ATT(o), r).

Definição 2: O modelo $UCON_{preA_1}$ é idêntico ao $UCON_{preA_0}$, exceto que adiciona o seguinte processo de pré-atualização

- preUpdate(ATT(s)), preUpdate(ATT(o)): são procedimentos opcionais para efetuar operações de atualização sobre ATT(s) e ATT(o) respectivamente..

Definição 3: O modelo $UCON_{preA_1}$ é idêntico ao $UCON_{preA_0}$, exceto que adiciona o seguinte processo de pós-atualização

- postUpdate(ATT(s)), postUpdate(ATT(o)): um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Exemplo 1($UCON_{preA_0}$): Política obrigatória (Modelo BellLappadula)

L é a relação de dominância >=

Clearance: $S \rightarrow L$

Classification: $O \rightarrow L$

ATT(S) = {clearance}

ATT(O) = {classification}

Allowed(s, o, read) $\Rightarrow$ clearance(s) $\geq$ classification(o)

Allowed(s, o, write) $\Rightarrow$ clearance(S) $\leq$ classification(o)

Exemplo 2 ($UCON_{preA_0}$): DAC fecha políticas utilizando ACL com um ID individual

N é a definição de nome de identidade

id: $S \rightarrow N$, mapeamento um a um.

ACL: $O \rightarrow 2^{NxR}$

ATT(S) = {id}

ATT(O) = {ACL}

Exemplo 3 ($UCON_{preA_1}$): DRM pay-per-use com créditos pré-pagos.

M é um valor em dinheiro

credit: S → M

value: O x R → M

ATT(s) : {credit}

ATT(o): {value}

Allowed(s, o, r) ⇒ credit(s) ≥ value(o)

preUpdate(credit(s)): credit(s) = credit(s) – value(o, r).

Exemplo 4 ($UCON_{preA_3}$): DRM baseado em membros

M é um valor em dinheiro

ID é o número de identificação do membro

TIME é a utilização do acesso por minuto

member: S → ID

expense: S → M

usageT: S → TIME

value: O x R → M {custo por minuto de r sobre o)

ATT(s) : {member, expense, usageT}

ATT(o, r) : {valorPorMinuto}

allowed(s, o, r) $\Rightarrow$ member(s)

postUpdate(expense(s)) : expense(s) = expense(s) + (value(o, r) x usageT(s))

A.2 - $UCON_{onA}$ – Modelos de ongoing-Authorizations.

No modelo $UCON_{onA}$, o acesso ao objeto é permitido sem necessidade de nenhuma operação de pré-decisão. Entretanto, existe um processo de decisão durante o acesso, que são feitas continuamente ou repetidamente enquanto o direito do uso é exercido. Caso certas exigências não sejam satisfeitas, o direito sobre o uso é revogado e o seu exercício é interrompido. Autorizações durante o acesso são raramente discutidos na literatura. A sua utilização geralmente é feita a partir de uma monitoração de um acesso, enquanto o seu direito é exercido. Para o $UCON_{onA}$, foram definidos 4 modelos: o $UCON_{onA_0}$, que é um que define um acesso com imutabilidade de atributos. E os $UCON_{onA_1}$, $UCON_{onA_2}$ e $UCON_{onA_3}$, que definem políticas de atualizações de atributos (mutabilidade) antes, durante e após o acesso respectivamente. Para estes modelos, foram criadas as seguintes definições:

Definição 4: O modelo $UCON_{onA_0}$ é composto pelos seguintes componentes:

- S, O, ATT(S), ATT(O): os conceitos são os mesmos do $UCON_{preA_0}$
- onA(ongoing-authorizations);
- allowed(s, o, r) $\Rightarrow$ true
- stopped(s, o, r) $\Leftarrow$ ¬onA(ATT(s), ATT(o), r)

Definição 5: O modelo $UCON_{onA_1}$ é idêntico ao $UCON_{onA_0}$, exceto que ele adiciona o seguinte processo de pré-atualização:

- preUpdate(ATT(s)), preUpdate(ATT(o)), um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Definição 6: O modelo $UCON_{onA_2}$ é idêntico ao $UCON_{onA_0}$, exceto que ele adiciona o seguinte processo de ongoing-atualização:

- onUpdate(ATT(s)), preUpdate(ATT(o)), um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Definição 7: O modelo $UCON_{onA_3}$ é idêntico ao $UCON_{onA_0}$, exceto que ele adiciona o seguinte processo de pós-atualização:

- postUpdate(ATT(s)), preUpdate(ATT(o)), um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Exemplo 5 ($\mathbf{UCON_{onA_{13}}}$): Limitar o número de acessos simultâneos, revogando o acesso caso este limite seja alcançado.

T é o tempo.

UN define o número de acessos simultâneos

N é um conjunto de identificadores de nomes

Id: $S \rightarrow N$

usageNum : $O \rightarrow UM$

startT : $O \rightarrow 2^{NxT}$

ATT(s) : {id}

ATT(o) : {startT, usageNum}

allowed(s, o, r) $\Rightarrow$ true

stopped(s, o, r) $\Leftarrow$ (usageNum(o) > 10) ^

(id(s),t) $\in$ startT(o) where t = min{t' | $\exists$s', (id(s'), t') e start(T(o)}

preUpdate(startT(o) : startT(o) = startT(o) $\cup$ {(id(s), t)}, where s is currently requesting subject of usage

preUpdate(usageNum(o)) : usageNum(o) = usageNum(o) + 1

posUpdate(startT(o)) : startT(o) = startT(o) – {(id(s), t)}, where s is a subject os stopped usage

postUpdate(usageNum(o)) : usageNum(o) = usageNum(o) – 1

Exemplo 6 ($\mathbf{UCON_{onA_{123}}}$): limitar o número de acessos simultâneos, revogando os acessos com um tempo longo ociosidade.

T é a definição da hora da última atividade

UM define o número de acessos simultâneos.

N é a identifição de nomes

Id : S $\rightarrow$ N

usageNum : O $\rightarrow$ UN

lastActiveT : O $\rightarrow 2^{NxT}$

ATT(s) : {id}

ATT(o) : {lastActiveT, usageNum}

Allowed(s, o, r) $\Rightarrow$ true

Stopped(s, o, r) ⇐ (usageNum(o) > 10) ^

(id(s),t) ∈ lastActiveT(o) where t = min{t' | ∃s', (id(s'), t') ∈ lastActiveT(o)}

preUpdate(usageNum(o)) : usageNum(o) = usageNum(o) + 1

onUpdate(lastActiveT(o)), repeated updates on lastActiveT(o)

postUpdate(usageNum(o)) : usageNum(o) = usageNum(o) - 1

Exemplo 8 ($UCON_{preB_0}$): uma obrigação para acordo de licenciamento (sem atributos).

OBS = S

OBO = {licence_agreement}

OB = {agree}

getPreOBL(s, o, r) = {(s, licence_agreement, agree)}

allowed(s, o, r) ⇒ preFulfilled(s, licence_agreement, agree)}

A.3 - $UCON_{preB}$ – pre-oBligations Models

UCONpreB trata de processos de decisão relacionada às obrigações que um sujeito deve cumprir **antes** de permitir o acesso ao Objeto. As funções de pré-obrigações são avaliadas com base a um histórico dos acessos efetuados pelo Sujeito, que certificam se as obrigações estão sendo cumpridas, retornando um valor verdadeiro ou falso. Um exemplo de pré-obrigação seria forçar que um usuário a fornecer seu nome e e-mail para efetuar um download de um arquivo. Neste caso, o usuário tem que cumprir a obrigação de informar seu nome e e-mail, para que seu acesso seja permitido. O modelo $UCON_{preB}$ consistem de 2 passos: o primeiro passo é identificar qual a obrigação necessária para acesso do Sujeito ao Objeto. O segundo é avaliar se esta obrigação foi cumprida pelo Sujeito, sem nenhum erro

(ex.: o usuário informar um e-mail inválido). Para este modelo, foram definidas 3 definições: UCONpreB0, que trata um acesso com imutabilidade de atributos, e o $UCON_{preB_1}$ e $UCON_{preB_3}$, que trata de um processo decisão com atualização de atributos antes ou depois do acesso respectivamente. Nestas definições, foram criadas as seguintes funções para avaliação das obrigações. O predicado *preB* avalia se todos os elementos de pré-obrigação (*preOBL*) foram cumpridos pelo Sujeito, utilizando a função *preFulfilled* que avalia se as obrigações foram realizadas, retornando verdadeiro ou falso.

Definição 8: O modelo $UCON_{preB_0}$ é composto pelos seguintes componentes.

- S, O, ATT(S), ATT(O): possui os mesmos conceitos que o modelo $UCON_{preA}$
- OBS, OBO e OB, (Obrigação do sujeito, Obrigação do Objeto e Ações de Obrigação respectivametente).
- preB e preOBL (predicados de pré-obrigação e elementos de pré-obrigação respectivamente).
- preOBL $\subseteq$ OBS x OBO x OB
- preFulfilled: OBS x OBO x OB $\rightarrow$ {true, false};
- getPreOBL: S x O x R $\rightarrow$ 2^{preOBL}, uma função para selecionar os elementos de pré-obrigação para o processo de decisão de acesso.
- preB (s, o, r) = $\wedge_{(obs, oboi, obi) \in getPreOBL(s, o, r)}$ preFulfilled (obsi, oboi, obi);

  pre(s, o, r) = true, somente se getPreOBL(s, o, r) = $\varnothing$ (conjunto vazio)
- allowed(s, o, r) $\Rightarrow$ preB(s, o, r)

Definição 9: O modelo $UCON_{preB_1}$ é idêntico ao $UCON_{preB_0}$, exceto que ele adiciona o seguinte processo de pré-atualização:

- postUpdate(ATT(s)), preUpdate(ATT(o)), um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Definição 10: O modelo $UCON_{preB_3}$ é idêntico ao $UCON_{preB_0}$, exceto que adiciona o seguinte processo de post-atualização:

- postUpdate(ATT(s)), postUpdate(ATT(o)): um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Exemplo 9 ($UCON_{preB_0}$): Uma obrigação para acordo de licenciamento de forma seletiva (com atributos do objeto).

OBS = S

OBO = {high_licence_agreement, low_licence_agreement }

OB = {agree}

Level: O → {high, low}

ATT(o) = {level}

getPreOBL(s, o, r) = { (s, high_licence_agreement, agree), if level(o) = ‘high’;

(s, low_licence_agreement, agree), if level(o) = ‘low’; }

allowed(s, o, r) ⇒ preFulfilled(getPreOBL(s, o, r)

Exemplo 10 ($UCON_{preB_1}$): Uma obrigação para acordo de licenciamento de forma seletiva apenas no primeiro acesso do usuário.

OBS = S

OBO = {licence_agreement}

OB = {agree}

registered: S → {yes, no}

ATT(s) : {registered}

getPreOBL(s, o, r) = { (s, high_licence_agreement, agree), if level(o) = ‘high’;

∅ , if level(o) = ‘low’; }

allowed(s, o, r) ⇒ preFulfilled(getPreOBL(s, o, r)

preUpdate(registered(s) : registered(s) = ‘yes’

A.4 - $UCON_{onB}$ – ongoing-oBligations Models

O Modelo $UCON_{onB}$ é similar ao $UCON_{preB}$, exceto que as obrigações que um Sujeito devem ser cumpridas enquanto ele exerce o acesso ao Objeto, de uma forma periódica ou contínua. Para isso, para este modelo foi introduzido um parâmetro de tempo *T* como parte do elemento de *onOBL*. Aqui, *T* define um determinado intervalo de tempo ou evento. Por exemplo, um usário deve clicar sobre uma propaganda em intervalos de 30 minutos (tempo) ou a cada 20 páginas acessadas (evento). Outro exemplo de uma obrigação durante o acesso seria um usuário ser obrigado a manter uma janela de propaganda ativa em seu computador durante o acesso a um determinado site ou serviço. O $UCON_{onB}$ é composto por quatro modelos: O $UCON_{onB_0}$ que define as obrigações a serem cumpridas durante o acesso, sem nenhuma atualização de atributos (imutabilidade). E os modelos $UCON_{onB_1}$, $UCON_{onB_2}$ e $UCON_{onB_3}$ são os idênticos ao $UCON_{onB_0}$, exceto que eles incluem um processo de atualização de atributos antes, durante ou depois do acesso ao objeto respectivamente.

Definição 11: O modelo $UCON_{onB_0}$ é composto dos seguintes componentes:

- S, O, ATT(S), ATT(O), OBS, OBO, OB: são os mesmos conceitos do modelo $UCON_{preB_0}$;

- T define os elementos de tempo ou eventos.

- onB e onOBL (predicados de obrigação durante o acesso e elementos de obrigação durante o acesso respectivamente).

- onOBL $\subseteq$ OBS x OBO x OB x T

- getOnOBL: S x O x R $\rightarrow 2^{onOBL}$, função para selecionar os elementos de obrigação durante, que serão utilizados no processo de decisão de acesso.

- onFulfilled: OBS x OBO x OB x T $\rightarrow$ {true, false}

- onB(s, o, r) = $\wedge_{(obsi, oboi, obi, ti) \in getOnOBL(s, o, r)}$ onFulfilled(obsi, oboi, obi, ti);

- onB(s, o, r) = true, somente se getOnOBL(s, o, r) = $\varnothing$ (conjunto vazio)

- allowed(s, o, r) $\Rightarrow$ true

- stopped(s, o, r) $\Leftarrow \neg$onB(s, o, r).

Definição 12: O modelo $UCON_{onB_1}$ é idêntico ao $UCON_{onB_0}$, exceto que adiciona os seguintes processos de pre-update.

- preUpdate(ATT(s)), preUpdate(ATT(o)): um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Definição 13: O modelo $UCON_{onB_2}$ é idêntico ao $UCON_{onB_0}$, exceto que adiciona os seguintes processos de ongoing-update.

- onUpdate(ATT(s)), onUpdate(ATT(o)): um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Definição 14: O modelo $UCON_{onB_3}$ é idêntico ao $UCON_{onB_0}$, exceto que adiciona o seguinte processo de post-atualização:

- postUpdate(ATT(s)), postUpdate(ATT(o)): um procedimento opcional para exercer operações de atualização sobre ATT(s) e ATT(o) respectivamente.

Exemplo 11 ($UCON_{onB_0}$): janela contendo informação enquanto o acesso do sujeito sobre o objeto está sendo exercido.

OBS = S

OBO = {ad_window}

OB = {keep_active}

T = {always}

getOnOBL(s, o, r) = { (s, ad_window, keep_active, always) }

allowed(s, o, r) $\Rightarrow$ true

stopped(s, o, r) $\Leftarrow$ ¬onFulfilled(getOnOBL(s, o, r))

A.5 - $UCON_{preC}$ – pre-Conditions Models

As pré-condições definem certas exigências que devem ser satisfeitas antes do acesso de um Sujeito ao Objeto. Em geral, as avaliações de condição não estão diretamente relacionadas a atributos do sujeitos e objetos. Neste processo de decisão de acesso, as Condições podem levar em consideração o ambiente atual da aplicação, como um status do sistema, a localização do usuário ou estar baseado na data/hora do acesso. Desta forma, a principal

diferença deste modelo é que, diferente das autorizações e obrigações, os modelos de condição não podem efetuar atualização de atributos, pois não são utilizados atributos do sujeito e objeto, mas sim, atributos relacionados ao sistema (ex.: a data / hora corrente do sistema operacional). Embora atributos do sujeito e objeto não possam ser utilizados em avaliações de Condição, eles podem ser utilizados em um processo de seleção de quais elementos de condição (preCON) serão utilizados no processo de decisão de acesso. Exemplo: para um Aluno acessar a sua sala de aula, são definidas duas condições: a primeira que permite acessos apenas entre 8:00 e 12:00 e a segunda entre 13:00 e 17:00. A condição será selecionada baseada no atributo do aluno, que indica se ele é um aluno do turno da manhã ou da tarde. $UCON_{preC}$ introdus predicados de pre-condições (preC) que tem que ser avaliados antes da requisção de direitos serem exercidos.

Definição 15: O modelo $UCON_{preC_0}$ tem os seguitnes componentes:

- S, O, ATT(S), ATT(O) são os mesmos conceitos do modelo $UCON_{preA_0}$
- preCON (define elementos de pré-condição)
- getPreCON: $S \times O \times R \rightarrow 2^{preCON}$
- preConChecked: preCON → {true, false};
- $preC(s, o, r) = \wedge_{preCon_i \in getPreCON(s, o, r)} preConChecked(preCon_i)$
- $allowed(s, o, r) \Rightarrow preC(s, o, r)$

Exemplo 12 ($UCON_{preC_0}$): Limitar localização

studentAREA, facultyAREA (código da região de acesso estudantes de colégio ou faculdade)

curArea define o código da região

ATT(s) = {member}

preCON = { (curArea ∈ studentAREA), (curArea ∈ facultyAREA) }

getPreCON(s, o, r) = { (curArea ∈ studentAREA), if member(s) = ‘student’;

(curArea ∈ facultyAREA), if member(s) = ‘faculty’ }

allowed(s, o, r) ⇒ preConChecked(getPreCON(s, o, r))

A.6 - $UCON_{onC}$ – ongoing-Conditions Models

O modelo $UCON_{onC}$ é semelhante ao $UCON_{preC}$, exceto que as condições são avaliadas durante o acesso do Sujeito ao Objeto. Como no modelo $UCON_{preC}$, existe apenas a definição do modelo $UCON_{preC_0}$, que define o processo de decisão de acesso com a imutabilidade dos atributos.

Definição 16: O modelo $UCON_{onC_0}$ tem os seguitnes componentes:

- S, O, ATT(S), ATT(O) são os mesmos conceitos do modelo $UCON_{preA_0}$
- onCON (define elementos de condição durante o acesso)
- getOnCON: S x O x R $\rightarrow 2^{preCON}$
- onConChecked: onCON → {true, false};
- onC (s, o, r) = $\wedge_{onCon_i \in getOnCON(s, o, r)}$ onConChecked($onCon_i$)
- allowed(s, o, r) ⇒ onConChecked(getPreCON(s, o, r))
- stopped(s, o, r) ⇐ ¬onConChecked (getPreCON(s, o, r))

Exemplo 13 ($UCON_{preC_0\ onC_0}$): Limitação de horários

dayH, nighH (escala de um escritório durante o dia e noite)

currentT é a hora corrente.

preCON : { (currentT ∈ dayH, currentT ∈ nightH) }

onCon : { (currentT ∈ dayH), (currentT ∈ nightH) }

getPreCON(s, o, r) = {currentT ∈ dayH), if escala(s) = ‘day’,

currentT ∈ nightH, if escala(s) = ‘night’ }

getOnCON(s, o, r) = {currentT ∈ dayH), if escala(s) = ‘day’,

currentT ∈ nightH, if escala(s) = ‘night’ }

allowed(s, o, r) ⇒ preConChecked(getPreCON(s, o, r))

stopped(s, o, r) ⇐ ¬onConChecked (getPreCON(s, o, r))