RAFAEL CONINCK TEIGÃO

# CONTROLE DE USO DE RECURSOS EM SISTEMAS OPERACIONAIS UTILIZANDO O MODELO $\text{UCON}_{\text{ABC}}$

Curitiba PR
Setembro de 2007

RAFAEL CONINCK TEIGÃO

# CONTROLE DE USO DE RECURSOS EM SISTEMAS OPERACIONAIS UTILIZANDO O MODELO $\text{UCON}_{\text{ABC}}$

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

Área de concentração: *Ciência da Computação*

Orientador: Prof. Dr. Carlos Maziero

Curitiba PR
Setembro de 2007

*Esta folha deve ser substituída pela ata de defesa devidamente assinada, que será fornecida pela secretaria do programa após a defesa.*

*Com muito amor, à minha esposa Roberta e ao meu avô Wilson.*

# Agradecimentos

Aos meu pais, que me proporcionaram sempre a melhor educação e um lar amoroso. Aos meus avós, sempre presentes. A minha esposa Roberta, quem corajosamente me acompanhou nesta aventura do conhecimento. Ao meu orientador, por sua dedicação ao ensino. A Deus, por ter me oferecido uma família maravilhosa, educação excelente e as melhores oportunidades.

*Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less.*

Marie Curie

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Resumo

Esta dissertação discorre sobre as necessidades modernas de controle de utilização de recursos em sistemas operacionais multi-usuários, e apresenta uma proposta de projeto para a implementação do modelo de controle de uso $\mathrm{UCON}_{\mathrm{ABC}}$ em um sistema operacional real, através de uma gramática específica para a definição de políticas e um mecanismo de interceptação de chamadas de sistema. A expressividade da gramática é evidenciada através de exemplos de políticas, e o desempenho e funcionamento do mecanismos são avaliados através de um protótipo. Os resultados desta avaliação demonstram o correto funcionamento do mecanismo, e, também, que o desempenho final do sistema é satisfatório, o que torna o projeto proposto uma solução viável para o controle de utilização de recursos em sistemas operacionais.

**Palavras-chave:** controle de acesso, controle de uso, gestão de recursos.

# Abstract

This work addresses the modern needs of resource utilization control in multi-user operating systems, and presents a project proposal for implementing the $\text{UCON}_{\text{ABC}}$ usage control model in a real operating system, by using a grammar specifically created for defining policies and by creating a mechanism for intercepting system calls. The grammar's expressiveness is evidenced by policy examples, and the mechanism's performance and correctness are verified using a prototype. The results confirm the mechanism's correctness and also show that the system's general performance is satisfactory, which makes this project a viable solution for resource utilization control in operating systems.

**Keywords:** access control, usage control, resource management.

# Capítulo 1

# Introdução

## 1.1 Contexto

Sistemas operacionais multi-usuários estão amplamente disponíveis em ambientes corporativos e acadêmicos. Serviços de terminal (*Terminal Services*) acessados remotamente, como aqueles encontrados em sistemas UNIX, permitem que seus usuários utilizem todos os recursos da máquina, sem impor limites ao consumo desses recursos.

É desejável que o sistema operacional (SO) possa fornecer o máximo em processamento, entrada/saída (I/O) e memória para todos os usuários, porém é muito importante que um usuário não consuma totalmente um recurso em detrimento dos demais usuários, e é ainda mais importante que usuários com tarefas de alta prioridade possam executá-las sem serem atrapalhados por usuários com tarefas mais banais, mas que consumam muitos recursos.

Além do problema da competição por recursos entre os usuários, políticas de controle de recursos são úteis para garantir a segurança e a integridade dos sistemas e a privacidade de seus usuários. Arquivos também são recursos, e a proteção dos dados que neles estão contidos pode ser de grande relevância para essa segurança e privacidade.

O controle de acesso clássico, o modelo atualmente aplicado para proteger arquivos, está se tornando ineficiente para cobrir as necessidades dos sistemas modernos. Atualmente, o preço da conectividade é suficientemente baixo para permitir que um usuário mantenha-se permanentemente conectado em seu serviço de terminal; essa conectividade constante oferece ao usuário a possibilidade de, após adquirir um direito sobre um objeto, não precisar liberá-lo enquanto a informação nele contida for interessante. Como os modelos clássicos controlam o direito de acesso apenas quando ele é requisitado, enquanto o usuário não liberar o objeto, este não poderá ser tomado, mesmo que a política de segurança que controla seu acesso seja radicalmente alterada.

Mecanismos como *Resource Control Lists* (RCL) [28] podem retirar um direito sobre um objeto e terminar o acesso do usuário mesmo que o objeto esteja em uso, porém RCLs não oferecem a possibilidade de limitar o acesso, por exemplo, de acordo com o horário do dia ou o volume de memória consumida em todo o sistema, ou ainda vincular tarefas como pré-requisitos para o acesso.

## 1.2 Motivação

O modelo $\text{UCON}_{\text{ABC}}$, apresentado por Park e Sandhu [35], introduz três conceitos de fundamental importância para as novas necessidades em controle de acesso:

1. mutabilidade de atributos e verificação constante;
2. tarefas ou obrigações como pré-requisitos para se obter ou manter o acesso; e
3. variáveis externas e de sistema que influenciam o controle de acesso.

A **mutabilidade de atributos** e sua verificação constante permitem que a decisão sobre um acesso[1] seja influenciada por outros acessos que ocorreram no passado e/ou estão ocorrendo paralelamente ao acesso corrente. Não apenas o histórico de acesso do usuário é relevante para a avaliação, mas também o histórico dos demais usuários que acessam o mesmo objeto (*e.g.* pode-se querer limitar o número de usuários simultâneos acessando o mesmo objeto). Como a avaliação é constante, o modelo $\text{UCON}_{\text{ABC}}$ faz com que uma modificação em um atributo tenha efeito imediato sobre o acesso atual, podendo, inclusive, retirar o direito do usuário de acessar o objeto.

O conceito de **obrigações**, ou tarefas que o usuário deve executar para que lhe seja permitido o uso do recurso, oferece uma maior liberdade para o produtor do conteúdo. Pode-se criar a obrigatoriedade, por exemplo, de um usuário folhear as instruções de uso de algum dado, manter uma janela com propaganda aberta enquanto assiste a um filme, responder algum tipo de questionário antes de acessar o objeto ou ainda aceitar um contrato de uso.

Já a avaliação das variáveis externas e de sistema, as **condições**, ajudam o administrador a controlar como o uso dos objetos sob sua responsabilidade impacta o sistema. Um exemplo clássico da necessidade desse tipo de controle é aumentar a prioridade de tarefas mais importantes e diminuir a prioridade de tarefas sem relevância, ou mesmo impedindo que tarefas de menor importância, mas que consomem muitos recursos (*e.g.* jogos), executem enquanto tarefas mais relevantes precisam ser executadas; nesse exemplo, a variável externa poderia ser a quantidade de CPU sendo consumida por todos os processos no sistema: uma política poderia definir que, caso a percentagem de utilização da CPU chegasse a 70%, todos os processos de baixa prioridade seriam terminados ou suspensos. Outro exemplo de utilização de condições é impedir que objetos com dados sigilosos não sejam acessados fora do horário de expediente, como fazem os bancos para controlar grandes movimentações financeiras; a variável externa poderia ser a hora do dia.

Essa flexibilidade e expressividade do modelo $\text{UCON}_{\text{ABC}}$ o torna um modelo muito interessante para controle de recursos em sistemas operacionais, porém os seus autores fizeram apenas uma descrição matemática [35] e lógica [56] do modelo, sem se preocupar com detalhes de implementação, o que o deixou muito distante de sistemas reais. Algumas implementações em sistemas específicos [16, 7] existem, mas nenhuma é utilizada para controlar recursos em um sistema operacional.

Para trazer o poder do modelo $\text{UCON}_{\text{ABC}}$ para o controle de recursos em sistemas operacionais, este trabalho propõe a implementação do modelo dentro do *kernel* de um sistema operacional COTS (*Commercial Off-The-Shelf*), e se dispõe a resolver os problemas relacionados à representação de atributos, obrigações e condições e à vinculação de políticas a objetos.

[1] Neste documento usamos sem distinção os termos "acesso" e "uso".

## 1.3 Objetivos

Este trabalho tem como objetivo a tradução da descrição matemática e lógica do modelo $\text{UCON}_{\text{ABC}}$ para uma linguagem de programação, e sua implementação no núcleo de um sistema operacional, para que seja possível controlar o acesso e o uso de recursos.

### 1.3.1 Objetivos Específicos

Os 5 objetivos específicos deste trabalho são:

1. criar uma gramática que defina uma linguagem para a descrição de políticas dentro do sistema;
2. desenvolver funções de suporte para executar o avaliador da gramática;
3. implantar a gramática e as funções de suporte dentro do núcleo de um sistema operacional (como prova de conceito, será implementado o controle de uso de arquivos);
4. definir políticas que exemplificam a funcionalidade e expressividade da gramática; e
5. avaliar o desempenho do sistema operacional após a implantação do projeto.

## 1.4 Escopo

Este projeto é uma prova-de-conceito, demonstrando uma implementação do modelo $\text{UCON}_{\text{ABC}}$ em um sistema operacional, para permitir que políticas sejam criadas para controlar recursos nesse sistema.

A tradução do modelo $\text{UCON}_{\text{ABC}}$ para uma linguagem de programação tentará ser fiel à descrição matemática e lógica do modelo apresentado por Park, Sandhu, *et al* [35, 56], porém será limitada conforme descrito na seção 5.2.

Por se tratar de uma prova-de-conceito, a implementação e avaliação do protótipo tratará, como recurso do sistema operacional, apenas arquivos comuns, isto é, arquivos que não são utilizados para controlar dispositivos ou ponteiros simbólicos para arquivos; as únicas operações sobre esses arquivos que serão controladas são: abertura, leitura, escrita e fechamento.

## 1.5 Organização do Documento

Este documento está dividido entre os seguintes capítulos:

**Cap 1:** Esta introdução, o contexto, a motivação, os objetivos e o escopo deste trabalho.

**Cap 2:** Apresentação do estado-da-arte em controle de acesso e definição dos conceitos utilizados nos modelos discricionários, mandatórios e baseados em papéis.

**Cap 3:** Introdução ao modelo $\text{UCON}_{\text{ABC}}$ e descrição de algumas implementações desse modelo.

**Cap 4:** Apresentação do estado-da-arte em controle de uso de recursos em sistemas operacionais, detalhando as linguagens Ponder e XACML e os projetos Systrace e CKRM.

**Cap 5:** Descrição do projeto de implementação do modelo $\text{UCON}_{\text{ABC}}$ no núcleo de um sistema operacional, incluindo a proposta deste trabalho.

**Cap 6:** Avaliação da gramática e do mecanismo implementado, incluindo análise de impacto no tempo de execução das aplicações.

**Cap 7:** Conclusão deste trabalho, com discussão sobre trabalhos futuros.

**Anexo A:** Apresentação da máquina de estados do interpretador da gramática.

# Capítulo 2

# Controle de Acesso

No interesse de se gerenciar direitos que sujeitos podem exercer sobre objetos, foram criados vários modelos de controle de acesso. O objetivo desses modelos é possibilitar a criação de políticas que definem explicitamente que direitos um sujeito ou grupos de sujeitos podem exercer sobre objetos ou grupos de objetos, e permitir que esses direitos sejam pontualmente revisados, ou seja, que se possa levantar, dado um sujeito e um objeto, quais direitos estão disponíveis.

Este capítulo apresenta os principais modelos de controle de acesso clássicos e os mecanismos que normalmente são empregados em suas implementações. Os conceitos fundamentais, como sujeitos, objetos e direitos são introduzidos, assim como o relacionamento entre políticas, modelos e mecanismos. Ao final do capítulo, várias limitações desses modelos são levantadas.

## 2.1 Sujeitos, Objetos e Direitos

Para compreender os conceitos associados ao controle de acesso, é preciso, primeiramente, entender as partes envolvidas nos seus modelos. O sujeito representa a parte que deseja manipular outro elemento do sistema. Esse sujeito pode ser um usuário, um *software* ou qualquer outra entidade que possa iniciar uma ação sobre outro elemento. Algumas vezes os sujeitos são chamados de *principals*.

Os objetos são os elementos que podem ser manipulados como um recurso. Exemplos de objetos são arquivos, dispositivos e terminais. As ações que os sujeitos exercem sobre os objetos são os direitos. Assim, um usuário (sujeito) pode efetuar a leitura (direito) de um arquivo (objeto), por exemplo.

Sujeitos e objetos normalmente podem ser agrupados entre si, criando-se classes/grupos de sujeitos e objetos. Esses grupos facilitam a gerência dos direitos, pois estes podem ser vinculados às classes de sujeitos e objetos, ao invés de individualmente.

## 2.2 Políticas, Modelos e Mecanismos

Um modelo é uma representação lógica ou matemática de como deve ser tratada a relação entre os objetos, os sujeitos e os direitos para que se possa prover um controle de acesso. Modelos são descrições de alto-nível (*i.e.* mais genéricas) do funcionamento do controle de

acesso, e são independentes de implementação ou do sistema, desde que todos os seus requisitos de funcionamento estejam presentes. Exemplos desses requisitos são informações que podem ser utilizadas para a criação das políticas, como a disponibilidade de uma identificação do sujeito e do objeto. Neste capítulo serão apresentadas três abordagens diferentes para modelos de controle de acesso: Discricionário, Mandatório e Baseado em Papéis.

As políticas definem que sujeitos, ou grupos de sujeitos, podem exercer uma lista de zero ou mais direitos sobre um objeto, ou grupos de objetos. É importante que se possa vincular de forma clara objetos, sujeitos e direitos. Deve ser possível criar um mapeamento Objetos $\times$ Sujeitos $\rightarrow$ Direitos para definir políticas de acesso sobre um objeto.

O que pode ser definido nessas políticas é dependente do modelo sendo empregado, enquanto seu formato depende do mecanismo utilizado para aplicar o modelo. Ou seja, políticas descrevem *o que* deve ser feito em um sistema para se atingir o grau de segurança desejado. Ao contrário dos mecanismos, que descrevem *como* isto é alcançado.

Esta separação entre políticas e mecanismos é importante porque as políticas são mais efêmeras, e tendem a variar com o tempo. Se esta separação for corretamente realizada, uma mudança na política pode ser feita sem que se tenha que alterar o mecanismo, que normalmente está implementado no *software*.

## 2.3 Monitor de Referência

O Monitor de Referência (*Reference Monitor*) [1] pode ser visto como um filtro de acesso, por onde todas as requisições de acesso devem passar e que não pode ser desviado ou evitado, que decide, através de avaliações, se uma requisição deve ser autorizada ou não. É um modelo conceitual de um mediador de acesso, em que "todas as referências (a programas, dados, periféricos, *etc.*) feitas por programas em execução são avaliadas em contraste àquelas autorizadas ao sujeito" [1]. A implementação do conceito do monitor de referência é conhecida como Mecanismo de Validação de Referência [32] (*Reference Validation Mechanism* – RVM), e este mecanismo deve ser:

**Inviolável** ou íntegro durante toda o ciclo de vida do sistema, ou seja, não pode ser alterado por programas ou usuários;

**Sempre invocado** e aplicado em qualquer requisição, seja ela do sistema ou dos programas em execução; e

**Auditável** e fiel ao modelo que ele implementa. Sua implementação deve ser suficientemente pequena para que ele possa ser analisado em sua totalidade a procura de erros, e para que a segurança da implementação possa ser demonstrada logicamente.

A combinação de *hardware* e *software* que implementa o monitor de referência é o Núcleo de Segurança [1].

## 2.4 Controle de Acesso Discricionário

O controle de acesso discricionário (*Discricionary Access Control* – DAC) [24] propicia um controle que é administrado pelo usuário proprietário do recurso. A grande vantagem deste

mecanismo é sua grande flexibilidade, mas esta vantagem pode ser também seu maior problema. Não é incomum existirem casos em que o usuário esquece ou falha ao proteger corretamente um objeto.

Tradicionalmente, o modelo que descreve o DAC é a Matriz de Acesso. Dois mecanismos que implementam esse modelo são as Listas de Controle de Acesso (*Access Control Lists* – **ACL**) e Capacidades (***Capabilities***).

### 2.4.1 Modelo de Matriz de Acesso

Normalmente, quando se fala em controle de acesso, fala-se em alguma modificação na forma de tratar o que é representado através do modelo conhecido como Matriz de Acesso. Esse modelo, existente há mais de 30 anos [24], é a base para praticamente todos os modelos discricionários de controle de acesso atuais.

A Matriz de Acesso é uma forma de relacionar sujeitos, objetos e direitos em um formato bi-dimensional, para explicitamente conceder, a sujeitos, direitos sobre objetos. Como índice vertical da matriz, tem-se os sujeitos e como índice horizontal, os objetos e no cruzamento dos índices encontram-se os direitos que o sujeito possui sobre o objeto.

Formalmente, a Matriz de Acesso é definida como um conjunto de sujeitos $S$ e um conjunto de objetos $O$, mapeados sobre um conjunto de direitos $r \in R$ através da função $r()$, tais que:

$$\exists\, s \in S \wedge \exists\, o \in O / r(s, o) \in R$$

Conceitualmente, a matriz pode ser visualizada como a representação da figura 2.1.


Figura 2.1: Matriz de Acesso

Porém, a Matriz de Acesso é raramente usada diretamente; recorre-se normalmente a uma simplificação, utilizando-se apenas as linhas — Capacidades (***Capabilities***) — ou as colunas — Listas de Controle de Acesso (*Access Control Lists* - **ACL**).

### 2.4.2 Mecanismo ACLs

As Listas de Controle de Acesso (ACLs) descrevem, para cada objeto, quais sujeitos tem quais direitos sobre este objeto. Tomando-se a Matriz de Acesso como modelo, as ACLs são representadas como a coluna de cada objeto. Um modelo simples de ACL pode ser visto nos sistemas UNIX, em que os arquivos possuem listas de permissões de acesso (direitos) para o proprietário, o grupo e os demais usuários.

A figura 2.2 mostra um exemplo de ACL contendo os direitos que os sujeitos `s1, s2` e `s3` podem exercer sobre o objeto `o2`.

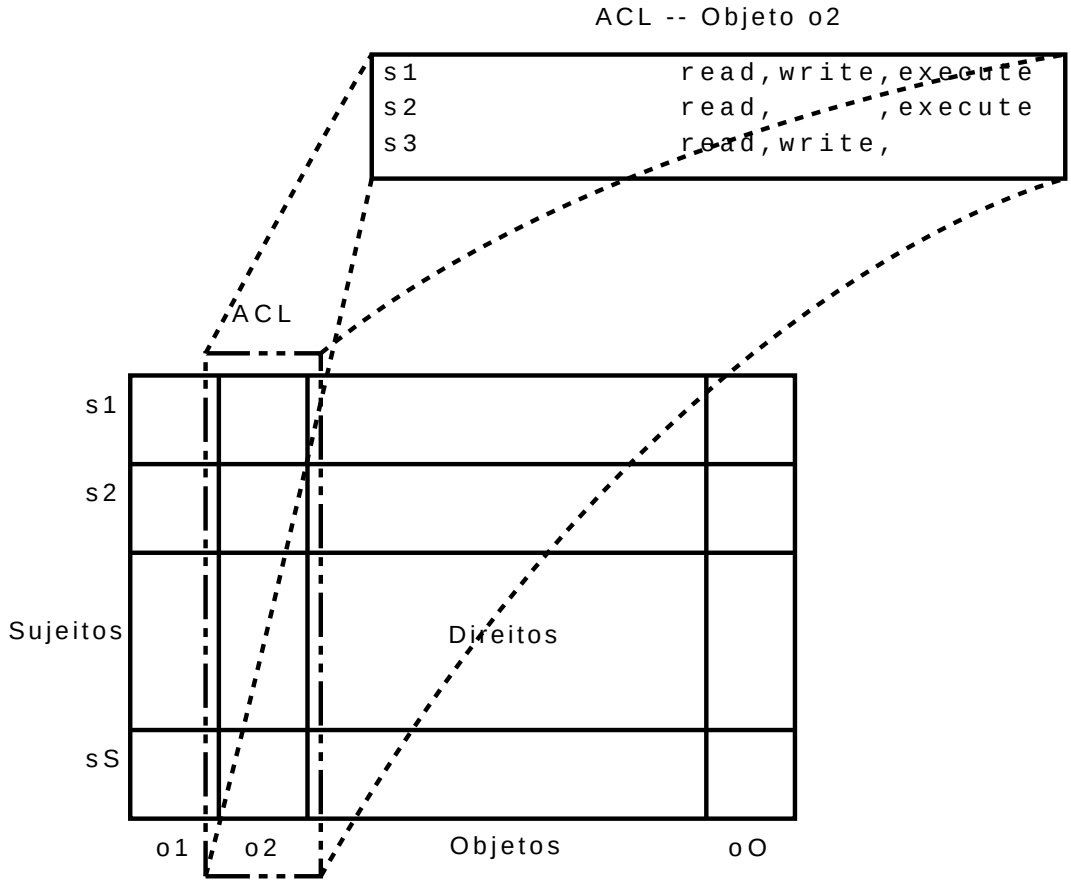


Figura 2.2: Exemplo de ACL

### 2.4.3 Mecanismo *Capabilities*

Similarmente às ACLs, as Capacidades (*Capabilities*) são representadas como as linhas da matriz de acesso. Elas descrevem quais direitos um sujeito possui sobre os objetos. *Capabilities* podem ser vistas como o inverso das ACLs.

A figura 2.3 na próxima página traz um exemplo de *capabilities* para o sujeito `s1`. Nela podem ser vistos os direitos que este sujeito pode exercer sobre os objetos `o1, o2, o3` e `o4`.

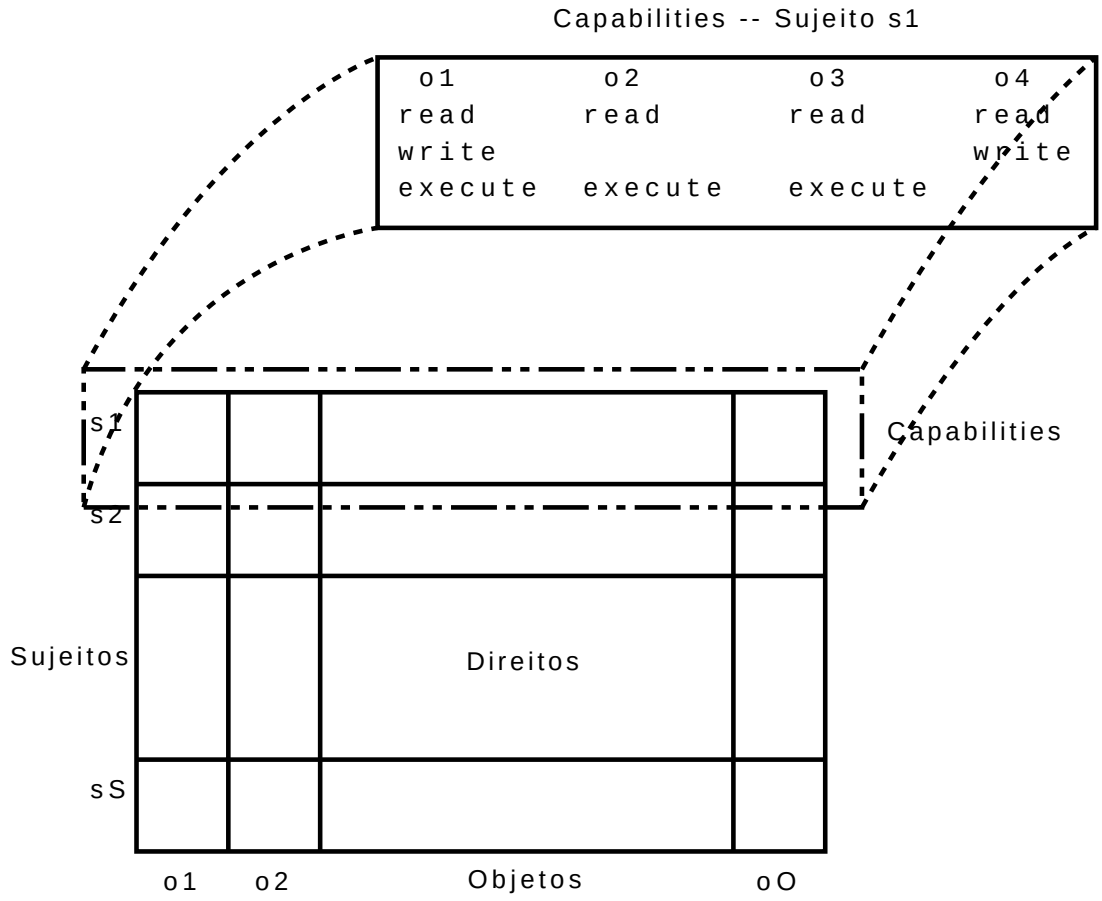


Figura 2.3: Exemplo de *Capabilities*

## 2.5 Controle de Acesso Mandatório

O controle de acesso mandatório (*Mandatory Access Control* – MAC) [3] é definido como modelos que não são controlados pelas identificações dos sujeitos ou objetos; ao invés disso, regras gerais e atributos dos sujeitos e objetos, gerenciados à discrição de um administrador central, fornecem as políticas de controle. Assim, elimina-se o problema de um usuário inadvertidamente reduzir a segurança de acesso a um objeto, e consegue-se uma maior garantia que as políticas estabelecidas serão seguidas, porém ao custo de uma flexibilidade grandemente diminuída.

Os principais modelos MAC apresentados neste capítulo focam na preservação da privacidade (**Bell-LaPadula**) e integridade (**Biba** e **Clark-Wilson**) de dados e na resolução de conflitos de acesso (***Chinese Wall***), quando os provedores de informação competem entre si, e o acesso às informações de dois provedores concorrentes pode levar a conflitos de interesse (como transferência de informação privilegiada entre empresas, no meio comercial).

### 2.5.1 Modelo Bell-LaPadula

O modelo Bell-LaPadula (BLP) foca na preservação da confidencialidade dos dados que ele protege. Cada sujeito possui um nível de permissão, como ultra-secreto ou secreto, e cada objeto possui uma classificação equivalente. Quando um sujeito requisita um acesso, seu nível

de permissão é confrontado com a classificação do objeto. O BLP define três propriedades de segurança que devem ser respeitadas:

1. A *Propriedade Simples de Segurança* garante que um sujeito não pode ler uma informação classificada acima do seu nível de permissão (***no read-up***).

2. A *Propriedade* * (estrela) impede que um sujeito escreva informações em um objeto classificado abaixo de seu nível (***no write-down***), ou move a classificação do objeto para o novo nível.

3. A *Propriedade de Segurança Discricionária* utiliza uma matriz de acesso para especificar um **DAC**.

Um administrador pode, quando necessário, mover informações de uma classificação maior para outra menor (violando a propriedade *). Essa intervenção é necessária porque quando o objeto é acessado por um sujeito com um nível de permissão maior que sua classificação, esta pode ser alterada para refletir o nível de permissão utilizado (evitando que o fluxo de informação ocorra de uma classificação maior para uma menor); com a utilização do sistema, muitos objetos, que ordinariamente teriam uma classificação menor, podem estar inacessíveis para os sujeitos que deveriam acessá-los, e surge, então, a necessidade de uma intervenção do administrador do sistema para desclassificar ou subclassificar esses objetos.

A figura 2.4 mostra um exemplo do relacionamento entre níveis de permissão e classificações.

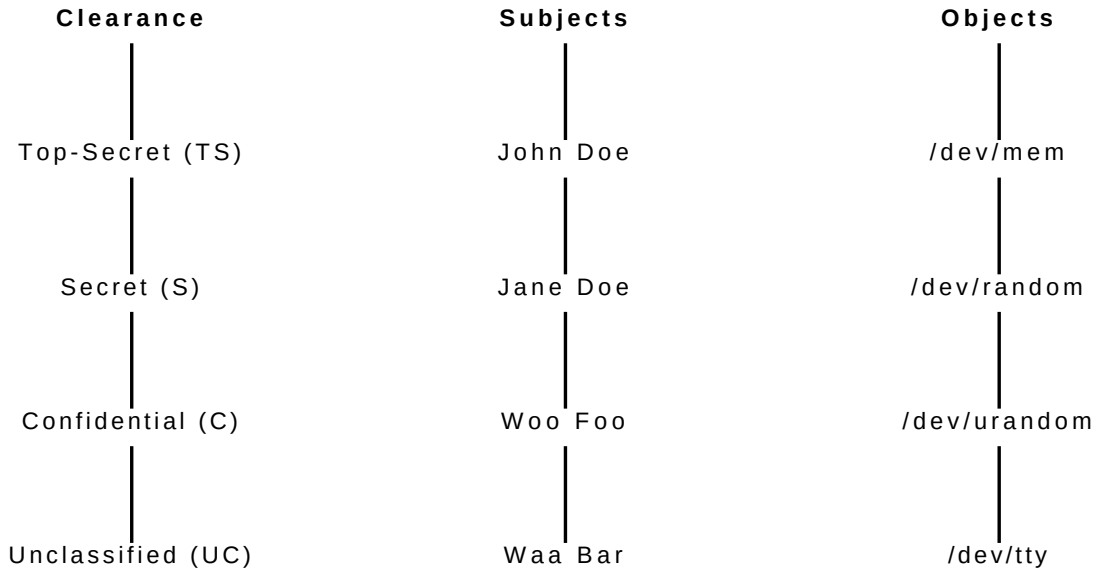


Figura 2.4: Níveis de Segurança e Classificação

Além do conceito de classificação, o modelo Bell-LaPadula apresenta o conceito de categorias. As categorias definem um tipo de informação a que se pode ter acesso (implementando o conceito de *need-to-know*), e objetos podem ser dispostos em várias categorias.

O conjunto de categorias a que um sujeito pode ter acesso é o conjunto potência das categorias. Esse conjunto forma uma treliça (*lattice*), assim como aquela apresentada na figura 2.5 na próxima página, utilizando as categorias ISS, EUA e RUS como exemplo.

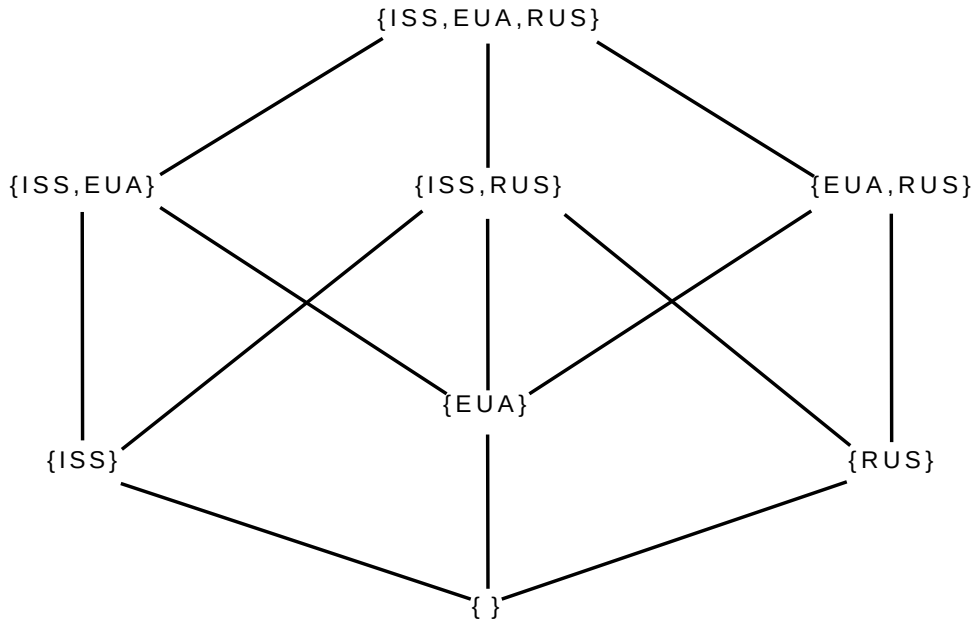


Figura 2.5: Exemplo de Treliça (*lattice*)

O par que relaciona o nível de permissão com uma categoria da treliça forma o nível de segurança do sujeito e a classificação do objeto:

- `Sujeito A` tem nível de permissão (`SECRET,{EUA}`);
- `Objeto Y` é classificado como (`CONFIDENTIAL,{ISS,RUS}`).

O `Sujeito A` não pode ter acesso ao `Objeto Y`, porque violaria o princípio de que apenas quem precisa ter acesso à informação o terá (*need-to-know*), mesmo sendo a permissão `SECRET` dominante sobre `CONFIDENTIAL`.

### 2.5.2 Modelo Biba

O modelo Biba pode ser visto como o oposto do modelo Bell-LaPadula: sua preocupação é com a integridade das informações. Um sujeito pode visualizar informação classificada no seu nível de acesso, ou em um nível superior (***no read-down***), e escrever apenas em um objeto classificado em seu nível, ou inferior (***no write-up***).

Esse modelo é composto por um conjunto de sujeitos, um conjunto de objetos e um conjunto de níveis de integridade. Há operações de relação que retornam a dominância entre dois níveis de integridade, e funções que retornam o nível de um sujeito ou objeto.

### 2.5.3 Modelo Clark-Wilson

Enquanto os modelos Bell-LaPadula e Biba são mais adaptados aos sistema militares, o modelo proposto por David Clark e David Wilson [8] preocupa-se mais com a sua aplicação em ambientes comerciais.

Esse modelo procura manter a integridade da informação durante sua modificação. Inicialmente, a informação encontra-se em um estado garantidamente íntegro, o seu estado inicial.

Quando ela é modificada, ocorre uma transição entre estados. Para garantir que a informação contida no estado final é íntegra, o modelo preocupa-se em proteger essa transição.

Um estado em que a informação tem sua integridade certificada é chamado de estado consistente. Uma transição entre dois estados consistentes é uma Transição Bem-Formada (do inglês *Well-Formed Transition*, ou WFT).

Dados que devem ter sua integridade garantida são chamados de itens de dados restritos (*Constrained Data Items*, ou CDI). Os CDIs devem respeitar as restrições de integridade (*Integrity Constraints*, ou IC). Estas podem ser políticas que definem quais valores são válidos para uma informação em um estado. Dados podem, também, não ter qualquer restrição (UDIs).

Quando o dado passa por uma mudança de estado, através de um procedimento de transformação (*Transformation Procedure*, ou TP), um procedimento de verificação de integridade (*Integrity Verification Procedure*, ou IVP) é executado para verificar se um CDI está em conformidade com seu IC. Um TP é uma transição bem-formada. Para garantir a separação de tarefa, o sujeito responsável pela IVP deve ser diferente daquele que executou a TP.

O modelo define ainda dois conjuntos de regras, um de certificação (CR) e outro de efetivação (ER):

**CR1:** garante que após um IVP, o estado é consistente.

**CR2:** um TP deve levar um conjunto de CDIs de um estado consistente para outro estado também consistente.

**ER1:** o sistema deve manter uma lista de TPs que são certificados para executar sobre um conjunto de CDIs, e apenas esses TPs devem ser autorizados a executar sobre esses CDIs.

**ER2:** o sistema deve manter uma associação entre sujeitos, TPs e CDIs, e apenas os sujeitos autorizados devem poder executar um TP sobre um CDI.

**CR3:** as relações definidas pela triplas $\mathrm{Sujeitos} \times \mathrm{TPs} \times \mathrm{CDIs}$ devem respeitar o princípio de separação de tarefas.

**ER3:** o sistema deve autenticar todo sujeito tentando executar um TP.

**CR4:** todo TP deve manter os históricos de alteração, ou dados de auditoria (*logs*), para a transição atual e todas as transições passadas.

**CR5:** TPs que recebem UDIs como entrada devem executar apenas operações válidas sobre todos os valores possíveis para o UDI, ou não efetuar qualquer modificação. A transformação deve conseguir converter o UDI em CDI, ou rejeitá-lo caso a conversão não seja possível.

**ER4:** Apenas entidades certificadoras podem alterar um TP. Estas entidade não podem, nunca, ter permissão de execução sobre este TP.

### 2.5.4 Modelo *Chinese Wall*

Assim como o modelo Clark-Wilson, o modelo *Chinese Wall* apresentado por Brewer e Nash [6] afasta-se dos modelos militares tradicionalmente utilizados até então. Esse modelo

é fortemente vinculado às necessidades de integridade e privacidade de dados em sistemas comerciais, e combina "discrição comercial com controles mandatórios impostos legalmente".

O conceito mercadológico de onde foi extraído o modelo *Chinese Wall* é aquele em que um consultor externo trabalhando para uma empresa não pode utilizar as informações confidenciais, obtidas durante sua consultoria, para aconselhar empresas concorrentes.

Os dados a que um sujeito tem acesso são divididos em classes de conflito de interesse. Cada classe agrega conjuntos de dados que pertencem a empresas concorrentes; uma política mandatória garante que cada sujeito tem acesso a apenas um único conjunto de dados. Essa política mandatória pode ser efetivada através de sistemas automáticos (*e.g.* computacionais) ou manuais (*e.g.* leis ou regras).

A figura 2.6 mostra um exemplo de classes de conflito. As classes definem indústrias diferentes; dentro de cada classe estão os conjuntos de dados; cada dado é um objeto que se deseja acessar.

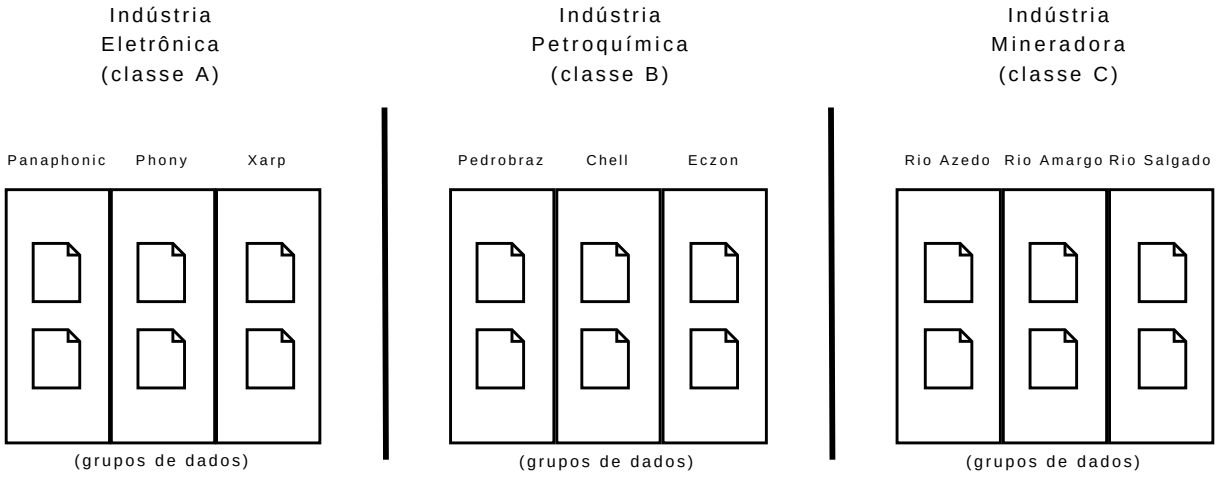


Figura 2.6: Exemplo de *Chinese Wall*

Neste exemplo, um consultor começa um trabalho na empresa "Phony" e acessa seus dados. Se, no futuro, ele resolver acessar os dados da empresa "Xarp", membro da mesma classe de conflito, o mecanismo que implementa esse modelo deverá negar o acesso. No entanto, caso ele requisite acesso aos dados da empresa "Chell", esta requisição deve ser permitida, uma vez que esta empresa está em uma classe diferente de conflito.

A "parede" a que o modelo se refere é criada quando o sujeito faz o seu primeiro acesso a um objeto de um conjunto de dados dentro de uma classe de conflito. Todos os demais conjuntos de dados estarão "do lado errado da parede".

Nota-se que não há qualquer restrição inicial quanto ao acesso do sujeito, e todos os dados dentro das classes estão disponíveis. Porém, quando o sujeito faz uma escolha sobre quais dados acessar, as paredes dentro da classe são formadas.

## 2.6 Controle de Acesso Baseado em Papéis

*Role-Based Access Control*, ou RBAC [41, 11], é um modelo recente que muito se adequa às necessidades das empresas, em que o controle de acesso está fortemente ligado à função, ou papel, do sujeito dentro da empresa. Por exemplo, entende-se que um diretor financeiro e

um gerente de vendas têm papéis diferentes e, logo, precisam acessar apenas as informações que são adequadas às suas funções. Da mesma forma, todos os diretores financeiros podem ter acesso aos mesmos dados, visto que desempenham o mesmo papel[1].

Matematicamente, tem-se para cada sujeito um mapeamento $SxP$, de $S$ sujeitos para $P$ papéis e para cada objeto, um mapeamento $PxD$, de $P$ papéis para $D$ direitos sobre o objeto. Assim, atribui-se papéis aos sujeitos e vincula-se estes papéis aos direitos de cada objeto.

Alguns dos principais conceitos no RBAC são:

**Revisão dos relacionamentos:** é um processo que permite entrar com um usuário e obter todos os papéis a ele relacionados, ou entrar com um usuário ou papel e obter todos os objetos e permissões relacionados. Esse processo deve ser escalável para um número grande de atributos (chegando à ordem de milhares) e suportar, quando necessário, sistemas distribuídos.

**Hierarquias:** são ordenações parciais entre papéis, que permitem, por exemplo, herança de privilégios entre níveis, em que um nível mais alto herda as permissões dos níveis mais baixos (*e.g.* um gerente recebe os privilégios de `gerente` e `contador` se o contador estiver abaixo dele). As hierarquias podem ser gerais, em que qualquer ordenação parcial pode ser criada, ou limitadas, em que as ordenações parciais criadas devem seguir uma estrutura simples, como uma árvore.

**Separação de Tarefas:** permite restringir as permissões do usuário ao menor conjunto necessário para a execução de uma tarefa. Assim, evita-se que papéis conflitantes sejam ativados simultaneamente (*e.g* um gerente pode ter os papéis `gerente` e `caixa`, mas pode ativar apenas um em cada sessão).

O RBAC é usualmente dividido em 3 modelos ou níveis, sendo que o primeiro deles, chamado RBAC Central (*Core*) é obrigatório na implementação de qualquer outro modelo RBAC. Estes modelos são descritos abaixo e resumidos na tabela 2.1 na próxima página.

**Central (*Core*):** Também conhecido como RBAC Plano [41], este modelo é obrigatório em qualquer implementação do RBAC. Consiste na atribuição muitos-para-muitos de papéis para sujeitos, em que um sujeito pode ter vários papéis e um papel pode ser atribuído a vários sujeitos; sujeitos devem poder habilitar vários papéis simultaneamente; e deve ser possível, através de uma ação administrativa, verificar quais papéis estão disponíveis para um dado sujeito.

**Hierárquico:** Neste modelo [11], pode-se atribuir a um papel as mesmas características de papéis que estão abaixo dele. Por exemplo, a um diretor seriam atribuídas as permissões de seu papel somadas às permissões dos papéis que vêm abaixo dele, como gerente e atendente; por sua vez às permissões do presidente seriam adicionadas as permissões do diretor e todas as permissões abaixo dele.

**Restrito:** O RBAC Restrito [11] permite a implementação de *separação de tarefas*, ou SoD (do inglês *Separation of Duty*). Na separação de tarefas, deve-se evitar que dois papéis conflitantes sejam ativados simultaneamente. O modelo prevê dois tipos de SoD:

[1] Obviamente podem existir casos em que diretores do mesmo "nível" possuem acessos diferentes, e o RBAC pode tratar isso.

1. Separação de Tarefas Estática (SSD – *Static Separation of Duty*): as restrições são colocadas no universo total de permissões disponíveis para o usuário.
2. Separação de Tarefas Dinâmica (DSD – *Dynamic Separation of Duty*): os conflitos de interesse são analisados dentro ou entre as sessões do usuário, e as limitações são dinamicamente impostas.

Pode-se supor, por exemplo, que para lançar um foguete seja necessária a autorização de um engenheiro e de um matemático. Ambos possuem papéis diferentes que, simultaneamente devem autorizar a decolagem. Pode-se cair no caso em que um engenheiro é também um matemático, ou vice-versa. Sem a separação de tarefas, este sujeito poderia ativar ambos os papéis e lançar, sozinho, o foguete. Se estes papéis forem marcados como conflitantes, ao menos uma pessoa com cada papel é necessária.

Tabela 2.1: Níveis do RBAC

| **Nível** | **Nome** | **Funcionalidades** |
|---|---|---|
| 0 | Central (*Core*) | • usuários recebem permissões através de papéis;<br>• deve suportar mapeamentos muitos-para-muitos entre usuários e papéis;<br>• deve suportar mapeamentos muitos-para-muitos entre permissões e papéis;<br>• deve suportar revisão de atribuição de papéis a usuários;<br>• sessões devem gerenciar os mapeamentos entre os usuários e um subconjunto dos papéis ativos; e<br>• usuários devem poder utilizar permissões de vários papéis simultaneamente. |
| 1 | Hierárquico (*Hierarchical*) | RBAC Central mais:<br>• deve suportar hierarquias de papéis (ordenação parcial);<br>– suporte a hierarquias arbitrárias;<br>– suporte a hierarquias limitadas (árvore, árvore invertida, *etc.*) |
| 2 | Restrito (*Constrained*) | RBAC Central mais:<br>• deve obrigar separação de tarefas (SOD – *Separation of Duty*);<br>– Separação de tarefas estática (SSD – *Static Separation of Duty*);<br>∗ com ou sem suporte a hierarquias.<br>– Separação de tarefas dinâmica (DSD – *Dynamic Separation of Duty*); |

## 2.7 Limitações dos Modelos de Controle de Acesso Tradicionais

Os modelos de controle de acesso tradicionais assumem que os sujeitos são localmente identificados e autenticados. Por isso, esses modelos não estão preparados para um controle de acesso distribuído, e o monitor de referência é único e esta posicionado junto com o objeto e o sujeito.

Em *Modular Authorization and Administration* [51], os autores propõem uma linguagem modular de autorização para suportar autorização distribuída entre grupos administrativos cooperantes, baseado principalmente no RBAC. Woo e Lam [52] também tentam resolver este problema de autorização distribuída ao introduzir uma linguagem para codificar requisitos de autenticação, que eles chamam de "bases de políticas". O artigo *Designing a Distributed Authorization Service* [53] avança nesse conceito, ao apresentar a sintaxe formal e a semântica para uma linguagem chamada *Generalized Access Control List* (GACL) para representar políticas de autorização baseadas em lista de controle de acesso (*Access Control List* - ACL).

Como a GACL está limitada aos mecanismos baseados em ACL, Ryutov e Neuman [40] apresentam uma linguagem de política que permite representar diversos modelos de controle (como ACL, *capabilities*, baseado em *lattice* e RBAC) e uma linguagem genérica de escrita de políticas de autorização e controle de acesso (GAA API) para facilitar a integração de autenticação e autorização.

Além da necessidade de se tratar os dados de forma distribuída, a avaliação dos direitos deve ser mais dinâmica. Nesse modelos, os direitos são validados apenas na requisição de acesso, e não mais verificados até a próxima requisição. Assim, a partir do momento que o sujeito inicia o acesso a um objeto, este acesso não pode ser revogado, mesmo que a política mude, até que o acesso seja finalizado normalmente.

## 2.8 Conclusão do Capítulo

Neste capítulo foram apresentados os principais modelos de controle de acesso discricionário (DAC), mandatório (MAC) e baseado em papéis (RBAC), seus conceitos fundamentais, e suas limitações quanto a verificação constante das políticas e controle de acesso a dados distribuídos.

A avaliação das políticas apenas no momento da requisição do acesso é a característica desses modelos que mais motivou o desenvolvimento deste trabalho. Essa característica diminui drasticamente a eficácia desses modelos em ambientes em que as políticas mudam rapidamente, ou em que o estado atual do sistema pode ter um impacto sobre a utilização de um objeto.

O próximo capítulo irá abordar o modelo de controle de uso $\mathrm{UCON}_{\mathrm{ABC}}$. Esse modelo, recentemente desenvolvido, oferece as ferramentas necessárias para agregar o estado do sistema na avaliação das políticas, e permite que essa avaliação seja constante e permanente durante o exercício de um direito sobre um objeto.

# Capítulo 3

# Controle de Uso

O termo "controle de acesso" implica que a avaliação das políticas é executada antes que o sujeito possa iniciar o exercício de um direito sobre um objeto. Essa avaliação normalmente é única para cada acesso; ou seja, quando o acesso é concedido, o sujeito pode exercer esse direito enquanto não for realizado um novo acesso e o objeto permanecer em seu poder. Por outro lado, "uso" refere-se ao período total em que o sujeito está em posse do objeto: as avaliações são múltiplas e contínuas durante toda a *utilização* do objeto. O modelo de controle de uso apresentado neste capítulo preocupa-se, então, com essa validação constante das políticas, e agrega, ainda, a possibilidade de incluir nelas dados externos e de ambiente.

Park e Sandhu, *et al* apresentam [34, 35, 56] o modelo de controle de uso $\text{UCON}_{\text{ABC}}$ como modelos matemáticos e lógicos, e não se preocupam com problemas de implementação, o que permite a apresentação de um modelo capaz de suportar muitas funcionalidades, mas cujo funcionamento depende de fatores que não estão facilmente disponíveis em sistemas reais (eles assumem, por exemplo, que um computador rodando um *software* cliente é inviolável). Apesar de existirem algumas implementações desse modelo conhecidas [16, 7], elas não o contemplam completamente, e deixam de utilizar os recursos de avaliação de políticas em tempo real ou de atualização de atributos.

## 3.1 Introdução ao $\text{UCON}_{\text{ABC}}$

O conceito de controle de uso (*Usage Control* – UCON) proposto por Park e Sandhu [34] engloba os modelos clássicos de controle de acesso tradicional, gerência de confiança (*trust management*) e gerência de direitos digitais (*Digital Rights Management* - DRM). O $\text{UCON}_{\text{ABC}}$ é um modelo matemático que introduz a utilização de predicados de Autorização, Obrigação e Condição[1] para tomar a decisão de negar, permitir ou controlar o uso de um recurso ou objeto digital. Assim, a utilização de um recurso não é mais controlada apenas por listas de controle de acesso (ACL) ou papéis (RBAC), por exemplo, mas também por fatores externos como a aceitação de um contrato através de um clique do *mouse* (obrigação) ou o horário do dia (condição).

Os predicados do $\text{UCON}_{\text{ABC}}$ também podem ser verificados durante a utilização, e não apenas na requisição inicial de acesso, possibilitando que um usuário tenha sua permissão de uso revogada enquanto detém o controle do recurso. Outra facilidade que é introduzida nesse

[1] As letras "ABC" que dão nome ao modelo referem-se às letras "A" de Autorização, "B" de oBrigação e "C" de Condição.

modelo é a mutabilidade dos atributos do recurso e do usuário, possibilitando políticas que garantam, por exemplo, a dedução do custo de uso de um recurso do crédito disponível a um usuário.

Usuários e objetos possuem atributos que são utilizados para a tomada de decisão pelo monitor de referência e esses atributos podem ser atualizados como resultado da avaliação de uma requisição de uso. Assim, um atributo de um objeto, atualizado durante uma requisição de uso de um usuário, pode influenciar a avaliação do uso de outro usuário.

A avaliação considera, além dos atributos, predicados para autorização, obrigação e condição. Estes predicados podem ser compostos separadamente, ou utilizados em conjunto, dependendo do controle de uso que se deseja implementar.

**Autorização** é um conjunto de predicados funcionais que são avaliados para decidir se um usuário pode exercer um dado direito sobre um objeto. Estes predicados englobam os modelos clássicos de controle de acesso, como o Controle de Acesso Discricionário (DAC), Mandatório (MAC) e Baseado em Papéis (RBAC).

**Obrigações** são predicados funcionais que tratam ações que devem ser cumpridas antes ou durante o uso para que um usuário possa exercer seus direitos sobre o objeto. Por exemplo, para um usuário conseguir executar um *software* novo, ele tem que enviar um *email* para o administrador informando que um novo executável foi instalado.

**Condições** são fatores de decisão ambientais ou externos ao objeto e ao sujeito. Esses fatores podem ser indicados em atributos, mas nenhum atributo deve ser utilizado diretamente no predicado. Por exemplo, o predicado "qualquer usuário não administrativo só pode executar um *software* se a carga total do processamento do sistema for menor que 80%" é uma condição, mas "um usuário só pode executar um *software* se sua utilização do sistema for menor que 80%", não o é. O fato de a utilização do processador por um usuário ser um atributo específico do usuário faz com que a avaliação não seja do sistema, mas de um atributo do usuário apenas, não se enquadrando na definição de condição dada por Park e Sandhu [35].

### 3.1.1 Momentos de Avaliação e Atualização

Os predicados do $\text{UCON}_{\text{ABC}}$ podem ser avaliados para se obter direito de uso de um objeto, ou para manter este direito. Quando o usuário ainda não detém direito de uso, e um conjunto de predicados será avaliado, a avaliação acontece **antes** do uso, então chama-se esta avaliação de `pre`. Assim, tem-se `preA`, `preB` e `preC`, para avaliações de Autorização, Obrigações e Condições. Da mesma forma, quando avalia-se o direito de permanecer utilizando um recurso, a avaliação acontece **durante** o uso, e ela é chamada de `on`: `onA`, `onB` e `onC`. A figura 3.1 na página oposta mostra esta continuidade na tomada de decisão.

Como pode ser visto na figura 3.1 na próxima página, assim como existem momentos diferentes para a verificação, existem, também, momentos diferentes para a atualização de atributos. Atributos do recurso e do sujeito podem ser imutáveis (`0`), ou modificados antes (`1`), durante (`2`) ou após (`3`) o uso (*e.g.* $\texttt{onA}_0$, $\texttt{onA}_1$, $\texttt{onA}_2$ e $\texttt{onA}_3$). A combinação dos momentos de avaliação com os de atualização gera os 16 modelos centrais do $\text{UCON}_{\text{ABC}}$, como mostra a tabela 3.1 na página oposta: cada "S" (sim) indica a existência de uma combinação no modelo.

Para avaliações `pre`, não existe atualização durante o uso. Isso ocorre porque, como não há avaliações neste momento (`on`), qualquer atributo alterado somente seria utilizado na

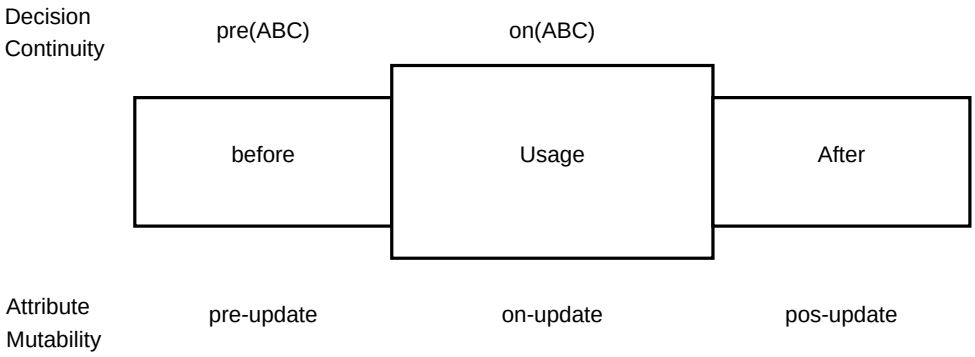


Figura 3.1: $\text{UCON}_{\text{ABC}}$: Momentos de Avaliação dos Predicados (extraído de [35])

Tabela 3.1: Os 16 Modelos Centrais do $\text{UCON}_{\text{ABC}}$ (extraído de [35])

| | 0 (imutável) | 1 (*pre-update*) | 2 (*on-update*) | 3 (*pos-update*) |
|---|---|---|---|---|
| `preA` | S | S | N | S |
| `onA` | S | S | S | S |
| `preB` | S | S | N | S |
| `onB` | S | S | S | S |
| `preC` | S | N | N | N |
| `onC` | S | N | N | N |

próxima avaliação, antes do próximo uso, fazendo com que uma atualização `2` tenha o mesmo efeito de uma atualização `3` (após o uso).

Nota-se, também, que os predicados de condição não atualizam atributos, existindo apenas $\texttt{preC}_0$ e $\texttt{onC}_0$. Isso acontece porque as condições são avaliadas apenas em função do sistema, e não dos atributos (mas um atributo pode informar qual condição deve ser avaliada).

## 3.2 Privacidade, *Trust Management* e DRM

Dois tópicos de grande interesse comercial são a autorização de usuários desconhecidos e a gerência de propriedade intelectual. Uma vez que os dados estão distribuídos e existem várias formas de acessá-los, pode ser desejável que um novo usuário, desconhecido do sistema, tenha acesso a esses dados.

O fornecimento de credencial para esse usuário chama-se *trust management* [5], livremente traduzido como "gerência de confiança". Porém, deseja-se controlar como esses dados estão sendo utilizados. Algumas vezes, pode se tratar de propriedade intelectual, e, por isso, deve-se garantir os direitos de propriedade da fonte dos dados. Isso é conhecido como gerência de direitos digitais, ou DRM (siga em inglês para *Digital Rights Management*). Muitas vezes também é desejável que os dados sejam mantidos em sigilo, e que apenas as pessoas que precisam acessá-los (*e.g.* médicos, consultores, *etc.*) tenham acesso a eles, o que é chamado de privacidade.

Até então, os esforços para garantir *Trust Management* e DRM têm se focado nos modelos tradicionais para atingir uma proteção quando os objetos estão centralizados no servidor, mas devido à inexistência de um monitor de referência que seja posicionado no lado do cliente,

esses esforços têm obtido resultados insuficientes, pois não se consegue garantir uma validação persistente do acesso [42].

Esforços como o *Palladium* da Microsoft [10] e o *Trusted Computing Platform Alliance* [22] tentam criar um ambiente confiável no lado do cliente, porém ainda não contam com grande adoção na indústria.

Para tentar unificar as diversas formas de se especificar direitos sobre conteúdos protegidos por DRM, foi criada a *Open Digital Rights Language* (ODRL) [18]. O intuito da ODRL é "prover mecanismos flexíveis e interoperáveis para suportar o uso transparente e inovador de recursos digitais em edição, distribuição e consumo de publicações digitais, imagens digitais, áudio e vídeo, objetos de aprendizado, *software* de computador e outras criações no formato digital" [18].

O *Enterprise Privacy Authorization Language* (EPAL) [26], por exemplo, tenta unificar as regras que controlam como informações sigilosas são tratadas em diferentes sistemas, através da criação de um mecanismo universal para a descrição das políticas de privacidade.

Enquanto o modelo UCON faz provisões [34] para a implementação de políticas que tratem de DRM, privacidade e *Trust Management*, neste trabalho apenas o lado do servidor será tratado e, apesar de conseguir-se uma validação persistente do acesso, não se pode estendê-la até o lado do cliente, pois este sistema trata apenas do sistema operacional em que ele está sendo aplicado. Para uma discussão sobre a utilização do modelo $\mathrm{UCON}_{\mathrm{ABC}}$ para privacidade garantida no lado do cliente, sugere-se o artigo *Usage Control Model and Architecture for Data Confidentiality in a Database Service Provider* [44].

## 3.3 Implementações do Modelo $\mathbf{UCON}_{\mathrm{ABC}}$

Devido ao modelo proposto por Park e Sandhu ter apenas um comprometimento matemático, muitas questões ficaram para serem solucionadas durante a implementação. Apesar de existir um modelo lógico [56], o $\mathrm{UCON}_{\mathrm{ABC}}$ encontra-se ainda descrito de forma muito distante das linguagens de programação. Além disso, apesar de o modelo matemático ter um ótimo funcionamento (*i.e.* contempla a problemática que ele pretende solucionar), a tradução para um sistema computacional não pode facilmente ser conseguida sem que a abstração seja perdida, isto é, sem que o modelo seja convertido para um formato mais específico, limitado pelas características das tecnologias empregadas na implementação, como será visto nas subsessões 5.2.1 e 5.2.2.

Provavelmente este é o motivo que impediu os autores de implementações anteriores de cobrirem alguns dos pontos apresentados no modelo, em especial a continuidade da avaliação das políticas. Mas, apesar da falta de implementações completas do modelo, alguns trabalhos já apresentam o $\mathrm{UCON}_{\mathrm{ABC}}$ como modelo de segurança para suas arquiteturas.

Em *Usage Control Model and Architecture for Data Confidentiality in a Database Service Provider* [44], Syalim *et al* utilizam o modelo $\mathrm{UCON}_{\mathrm{ABC}}$ para descrever uma arquitetura de controle de acesso para Provedores de Serviço de Banco-de-Dados (*Database Service Providers* – DSP). Um DSP funciona como um banco de dados *on-line*, acessível através de uma rede não privada. Este serviço requer controles especiais para *Trust Management* e *Digital Rights Management*, e permite que o administrador do DSP controle como e quais usuários tem acesso ao serviço, e que os usuários donos de bancos de dados controlem como outros usuários podem utilizar e acessar as informações neles armazenadas.

Os modelos RBAC e UCON são analisados como possíveis candidatos para fornecer controle de acesso em sistemas de tempo-real que utilizam o modelo TMO (*Time-triggered Message-triggered Object*) para garantia de serviço em *Dependable and Secure TMO Scheme* [23]. O modelo TMO suporta um ambiente de computação distribuída e necessita de um modelo de autenticação e controle de acesso para garantir sua integridade e confidencialidade, porém os autores argumentam que os modelos de controle de acesso tradicionais são inadequados para ambientes distribuídos, e apresentam exemplos para demonstrar que tanto RBAC como UCON podem ser aplicados para garantir a segurança que o TMO requer.

Os conceitos de privacidade de dados e proteção de direitos sobre propriedade intelectual (DRM) em um sistema em que os provedores de informação (bancos de dados) estão distribuídos com relação aos clientes (consumidores da informação) são tratados em *Distributed Usage Control* [36]. Os autores dão especial atenção aos predicados de obrigação para criar políticas que definem como esses dados devem ser acessados e que requisitos os consumidores devem cumprir para poder utilizá-los.

### 3.3.1 JaCoWeb-ABC

O projeto JaCoWeb-ABC [15] integra o modelo $\text{UCON}_{\text{ABC}}$ à especificação de segurança do CORBA, o CORBASec. O autor faz um mapeamento das características do modelo $\text{UCON}_{\text{ABC}}$ sobre as definições do CORBASec e levanta os principais pontos que devem ser modificados para que a integração possa ser feita. São eles:

1. Criação de um objeto chamado de `AttributesManager` e a modificação dos objetos `AccessPolicy` e `RequiredRights` para permitir a criação de atributos genéricos, já que o CORBASec permite apenas atributos pré-determinados;

2. Remodelagem do objeto `AccessDecision` para aceitar políticas de Autorização, Obrigação e Condição;

3. Criação de um avaliador de políticas ABC, que considere os atributos do objeto e do sujeito, além das obrigações e condições;

4. Criação de um processo de avaliação em dois níveis: ORB e IDL (o primeiro é transparente para a aplicação e o segundo funciona em conjunto com a aplicação).

Para a definição das políticas e dos atributos do sujeito e do objeto, foi criada a linguagem XEBACML (*eXtensible Expression Based Access Control Model Language*), baseada em XML. Abaixo pode-se visualizar exemplos, extraídos de "JaCoWeb-ABC: Integração do Modelo de Controle de Acesso $\text{UCON}_{\text{ABC}}$ no CORBASec" [15], utilizando XEBACML.

A figura 3.2 na próxima página mostra como podem ser definidos os atributos do sujeito, e a figura 3.3 na página seguinte mostra como são definidas as políticas e os atributos do objeto.

O JaCoWeb-ABC é provavelmente a primeira implementação do modelo $\text{UCON}_{\text{ABC}}$, e o desempenho do sistema final é próximo daquele sem a avaliação das políticas ABC. Porém, JaCoWeb-ABC não trata da continuidade de avaliação (*i.e.* políticas *on*), um ponto forte do modelo que não foi implementado devido à característica transacional do próprio CORBASec, em que a decisão de acesso é avaliada após a invocação de um método remoto.

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

Figura 3.2: JaCoWeb-ABC: Atributos do Sujeito (extraído de [15])

```
<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 3.3: JaCoWeb-ABC: Atributos e Políticas do Objeto (extraído de [15])

### 3.3.2 $\text{UCON}_{\text{ABC}}$ e B2B

Em "Aplicação do Modelo $\text{UCON}_{\text{ABC}}$ em Sistemas de Comércio Eletrônico B2B" [7] foi proposto um projeto de integração do modelo para controlar sistemas de comunicação entre empresas parceiras (*business-to-business* ou B2B), porém não foi feita uma implementação real.

Segundo os autores, os sistemas B2B modernos precisam de uma forma de controle de acesso além dos modelos tradicionais, porque em muitas situações tem-se um usuário externo

acessando dados críticos sobre uma empresa, utilizando uma série de sistemas diferentes. O termo "entidade composta" é utilizado para descrever o agregado dos diversos usuários externos, com seus atributos, e o sistema por eles utilizado para interagir com a empresa, que por sua vez também possui atributos específicos.

Os sistemas de comércio eletrônico B2B são divididos em duas partes: o Sistema da Empresa Provedora (SEP) e o Sistema da Empresa Consumidora (SEC). É no SEP que se concentra o módulo de controle de acesso, e é ele o responsável por avaliar as políticas ABC.

O funcionamento do sistema se dá segundo os passos abaixo:

1. O funcionário F1, da empresa consumidora, autentica-se no SEC de sua empresa;
2. O SEC autentica-se para o SEP (*e.g.* utilizando certificados digitais);
3. F1 autentica-se no SEP;
4. SEP procura as permissões deste usuário e as armazena em memória.

Após autenticado, o cliente pode solicitar um comando no sistema. Esse comando é passado do SEC para o módulo de interação do SEP, que é encarregado de traduzir a mensagem para um formato interno do sistema, e a mensagem traduzida é em seguida passada para o módulo de controle de acesso. Nesse módulo ocorre a verificação das políticas de Autorização, Obrigação e Condição. Caso o uso seja autorizado, o objeto é acessado e retornado, através do módulo de interação, para o SEC, que irá se encarregar de mostrar a informação para o cliente.

As políticas são verificadas por filtros de acesso. O primeiro filtro verificado é o de condições, pois contém as informações mais gerais sobre o acesso, como a hora do dia. Em seguida, o filtro de autorização é empregado; este é o filtro que utiliza os dados de permissões levantados durante o processo de autenticação do usuário. Por último, o filtro de obrigações verifica se há alguma tarefa que o usuário deve executar antes de obter acesso aos dados. Para que o acesso seja conseguido, todos os filtros devem retornar positivamente, e caso qualquer um deles falhe, a negação de acesso é imediatamente retornada, e os demais filtros não são analisados.

Os autores não tratam explicitamente a continuidade na verificação das políticas, e dizem apenas que ela é satisfeita porque mesmo depois do usuário estar autenticado, os filtros sempre são empregados para todos os acessos, mas isso não é realmente o que o modelo $\mathrm{UCON}_{\mathrm{ABC}}$ trata como continuidade, porque um objeto ao qual o usuário já obteve acesso não pode ser preemptado.

Para a gerência de permissões, os autores sugerem uma extensão do RBAC denominada Agrupamento Implícito (AI). Os agrupamentos implícitos ocorrem segundo os atributos do sujeito ou objeto que se está analisando. Atributos como localização geográfica podem configurar automaticamente usuários em um Agrupamento Implícito de Sujeitos (AIS) específico, e atributos do objeto, como seu tipo, podem agrupá-los em um Agrupamento Implícito de Objetos (AIO). Quando um atributo é alterado (*e.g.* o usuário muda de país), o sujeito ou objeto é automaticamente removido de seu AI atual e colocado no AI correspondente às suas novas características. Esse mecanismo facilita a gerência e reduz as probabilidades de ocorrerem erros nas atribuições das permissões.

Como pode ser necessário que vários objetos sejam acessados por AIS diferentes, seria preciso criar um AIO para cada AIS, e isso iria causar problemas gerenciais que poderiam

ser evitados se os objetos não estivessem agrupados. Então os autores sugerem utilizar um Agrupamento Implícito Parcial (AIP), em que apenas os sujeitos são agrupados.

Apesar desse trabalho não apresentar uma implementação completa, ele é interessante pois traz uma proposta de implementação, com sugestões para resolver alguns problemas na tradução do modelo para um sistema real.

## 3.4 Conclusão do Capítulo

Esse capítulo apresentou o modelo de controle de uso $\mathrm{UCON}_{\mathrm{ABC}}$, como proposto por Park e Sandhu, e detalhou duas implementações desse modelo. Os conceitos de privacidade, gerência de confiança e proteção de direitos digitais foram apenas rapidamente tratados, por não serem o foco principal deste trabalho.

As duas implementações descritas, apesar de incompletas, servem como uma interessante base de apoio para o desenvolvimento de novos sistemas utilizando o $\mathrm{UCON}_{\mathrm{ABC}}$, pois nelas podem ser vistos alguns detalhes quanto à tradução do modelo matemático para linguagens de programação.

No próximo capítulo serão apresentados diversos mecanismos e linguagens utilizados para controlar acesso e uso de recursos em sistemas operacionais, e como eles se comparam com o modelo $\mathrm{UCON}_{\mathrm{ABC}}$.

# Capítulo 4

# Controle de Acesso e de Uso de Recursos em Sistemas Operacionais

## 4.1 Introdução

O controle de acesso trata da autorização de um sujeito para exercer um direito sobre um objeto. Por outro lado, o controle de uso trata da forma como um direito é exercido sobre um objeto, da duração deste acesso e da quantidade do objeto que está disponível para um sujeito. Neste contexto, os objetos são recursos disponibilizados pelo sistema operacional, como memória, arquivos, diretórios, interfaces de rede, processador, entre outros.

Neste capítulo, diversos mecanismos e linguagens de controle de acesso e de uso de recursos serão apresentados. Destaque especial será dado aos mecanismos e às linguagens encontrados na literatura que são mais relevantes para esta pesquisa e, sempre que possível, será feita uma comparação com o modelo $\text{UCON}_{\text{ABC}}$.

## 4.2 Recursos em Sistemas Operacionais

O *hardware* em que um sistema operacional é executado é composto de uma série de recursos disponibilizados para as aplicações de usuário. Periféricos como disco rígido, memória e placa de rede, entre outros, são recursos que podem ser compartilhados entre os usuários do sistema. O SO oferece, também, recursos lógicos, como o sistema de arquivos.

A gerência desse recursos é feita pelo SO, que deve dividi-los entre as aplicações dos diversos usuários. Idealmente, todos os usuários deveriam receber recursos suficientes para executar seus programas sem perda de desempenho, mas como os recursos são limitados e compartilhados, eventualmente esses recursos acabam ficando sobrecarregados, e o desempenho do sistema diminui.

Assim, sobre esses recursos devem ser aplicados controles de acesso, para definir que usuários podem acessá-los, mas também controle de uso, para que nenhum usuário os monopolize em detrimento dos demais.

Sem um controle cuidadoso da distribuição dos recursos, um usuário poderia executar uma aplicação que os consuma em um ritmo superior àquelas dos demais usuários. Se os usuários não devem ter privilégios uns sobre os outros, isto é, se todas as aplicações devem ser executadas com a mesma prioridade, não é justo que todos sofram uma queda no desempenho

enquanto um usuário está excessivamente utilizando algum recurso. Além disso, podem existir casos em que alguns usuários são realmente privilegiados (como o administrador do sistema), mas não conseguem executar suas tarefas porque alguém consumiu todo um recurso antes que a aplicação de maior prioridade pudesse requisitá-lo.

Existem atualmente mecanismos para uma utilização justa[1] de alguns recursos, mas esses mecanismos são ainda muito específicos e largamente insuficientes. Os poucos controles disponíveis estão distribuídos por diversas ferramentas, como os quatro exemplos de recursos abaixo que, em alguns sistemas UNIX, utilizam quatro mecanismos diferentes em sintaxe e funcionamento:

**CPU:** pode-se afetar a distribuição do processador entre os processos através do uso de prioridades;

**Memória:** está vinculada a cada sessão do usuário, e pode ser limitada apenas durante o início da sessão;

**Disco:** pode ser controlado por *quota* (`edquota`);

**Rede:** um filtro de pacotes (*firewall*) pode negar ou reduzir um fluxo de pacotes, através de controles de qualidade de serviço (QoS).

## 4.3 Mecanismos de Controle de Acesso

Os mecanismos de controle de acesso tradicionalmente encontrados em sistemas operacionais modernos são baseados no modelo discricionário, normalmente implementados por listas de controle de acesso. Exemplos desses mecanismos são os sistemas de permissões disponíveis no UNIX [12] e MS-Windows [39].

Para atender as diversas necessidades de controle de acesso dos sistemas atuais, diferentes modelos e mecanismos de controle de acesso avançados foram criados. Aqueles mais relevantes encontrados na literatura serão descritos na seqüência.

### 4.3.1 Ponder

Ponder [9] é uma linguagem orientada a objeto que pode ser utilizada para definir políticas de controle de acesso para diversas finalidades, entre elas, recursos de sistemas operacionais, porém seu foco está em sistemas de objetos distribuídos. A linguagem suporta dois tipos de políticas: de Controle de Acesso Discricionário e de Obrigações. As políticas de controle de acesso são divididas em:

**Políticas de Autorização** definem que direitos um grupo de sujeitos tem sobre um grupo de objetos. Essas políticas podem ser positivas (indicando o que pode ser feito) ou negativas (explicitamente negando algum direito). A figura 4.1 na próxima página mostra a sintaxe das políticas de autorização.

A figura 4.2 na página oposta mostra um exemplo de política positiva em que os membros do domínio `NetworkAdmin` podem executar as ações `load(), remove(), enable()` e `disable()` em *switches* de `Nregion`.

[1] *I.e.* para evitar que alguns poucos usuários consumam constantemente muitos recursos do sistema.

```
inst ( auth+ | auth- ) policyName "{"
   subject [<type>] domain-Scope-Expression ;
   target [<type>] domain-Scope-Expression ;
   action action-list ;
   [ when constraint-Expression ; ]
"}"
```

Figura 4.1: Ponder: Sintaxe de Política de Autorização (extraído de [9])

```
inst auth+ switchPolicyOps {
   subject /NetworkAdmin ;
   action load(), remove(), enable(), disable() ;
   target <PolicyT> /Nregion/switches ;
}
```

Figura 4.2: Ponder: Política de Autorização Positiva (extraído de [9])

A figura 4.3 mostra um exemplo de política de autorização negativa que proíbe Engenheiros de Teste *Trainee* de executar testes de performance em roteadores.

```
inst auth- /negativeAuth/testRouters {
   subject /testEngineers/trainee ;
   action performance_test() ;
   target <routerT> /routers ;
}
```

Figura 4.3: Ponder: Política de Autorização Negativa (extraído de [9])

**Políticas de Filtragem de Informação** permitem criar ações que transformam um dado para apresentá-lo de forma higienizada, ou seja, sem informações que o usuário não possa acessar. Um exemplo apresentado pelos autores é o caso de um funcionário de folha de pagamento que tem que saber quanto deve ser pago, mas não pode ter acesso às informações dos diretores. Então, pode-se criar um filtro que limpa as informações dos diretores e apresenta apenas o valor dos pagamentos, mas mantém as informações dos outros funcionários.

Quando um sujeito requisita as informações, o sistema deve decidir se ele está autorizado a recebê-la integralmente, ou se deve recebê-la em um formato higienizado. Se este for o caso, ele consulta a política para identificar a transformação necessária e gera o resultado que será apresentado.

A figura 4.4 na página seguinte mostra a sintaxe das políticas de filtragem de informações.

**Políticas de Delegação** possibilitam que um sujeito delegue alguns de seus direitos (adquiridos através de autorização) para outro sujeito. Porém, essa delegação deve ser fortemente controlada, pois trata-se de um ponto em que falhas humanas podem impactar na segurança do sistema (*e.g.* caso um sujeito delegue inadvertidamente um direito que deveria

```
actionName { filter }

filter = [ if condition ] "{"
     {     (   in parameterName = expression ;  |
               out parameterName = expression ; |
               result = expression ;
           )
     }
"}"
```

Figura 4.4: Ponder: Política de Filtragem de Informações (extraído de [9])

ser restrito ao seu usuário apenas). Políticas negativas podem indicar quais direitos nunca podem ser delegados.

A figura 4.5 mostra a sintaxe das políticas de delegação.

```
inst deleg+ "(" associated-auth-policy ")" policyName "{"
    grantee     [<type>]   domain-Scope-Expression ;
    [ subject   [<type>]   domain-Scope-Expression ; ]
    [ target    [<type>]   domain-Scope-Expression ; ]
    [ action               action-list ; ]
    [ when                 constraint-Expression ; ]
    [ valid                constraint-Expression ; ]
"}"
```

Figura 4.5: Ponder: Política de Delegação (extraído de [9])

**Políticas de Privação** definem as ações que o sujeito não deve executar sobre um objeto, mesmo que lhe seja permitido. O sujeito é o responsável por garantir a efetivação das políticas de privação. A sintaxe das políticas de privação é a mesma das políticas de autorização negativa, apenas substituindo `auth-` por `refrain`.

Todas as políticas referem-se a objetos, cujo acesso acontece através de interfaces. Logo, a avaliação das políticas no Ponder acontece quando a interface para um objeto é invocada. Os objetos e suas políticas e sujeitos são agrupados em domínios, que podem ser controlados através de um serviço de diretório.

A definição da linguagem Ponder também traz o conceito de Políticas de Obrigação, mas, diferente do modelo $\text{UCON}_{\text{ABC}}$, essas políticas não são ações que os usuários devem executar e, sim, tarefas que o administrador deve cumprir, em resposta a eventos. Essas políticas são consideradas Políticas de Gerência, e não efetivamente políticas de controle de acesso.

Algumas políticas de obrigação podem definir quando e quem deve executar auditorias, o que deve ser feito após uma violação de segurança, entre outras tarefas administrativas. Essas políticas são ativadas por eventos, como um alarme de tempo ou um evento de monitoramento. A sintaxe de uma política de obrigação pode ser vista na figura 4.6 na página oposta.

A linguagem oferece a possibilidade de descrever em quais condições uma política é válida. A validade de uma política pode ser analisada em função de condições ambientais ou em função de outras políticas sendo consideradas no mesmo instante no sistema.

```
inst oblig policyName "{"
    on                  event-specification ;
    subject  [<type>]   domain-Scope-Expression ;
    [ target [<type>]   domain-Scope-Expression ; ]
    do                  obligation-action-list ;
    [ catch             exception-specification ; ]
    [ when              constraint-Expression ; ]
"}"
```

Figura 4.6: Ponder: Política de Obrigação (extraído de [9])

Papéis, semelhantes a RBAC, podem ser definidos, e hierarquias entre esses papéis podem ser criadas, para que direitos possam ser herdados de forma ordenada.

A linguagem Ponder possui algumas características que não estão presentes no modelo $\text{UCON}_{\text{ABC}}$, como delegações, porém as políticas que podem ser definidas são fortemente voltadas para autorização, e não são tão genéricas como no modelo $\text{UCON}_{\text{ABC}}$.

### 4.3.2 XACML

A *eXtensible Access Control Markup Language* (XACML) [45] é uma linguagem baseada em XML para definições de políticas de controle de acesso sobre um recurso, através da descrição dos requisitos necessários para que um sujeito exerça um direito sobre esse recurso. A figura 4.7 na próxima página mostra o fluxo de dados de um sistema utilizando XACML.

Os seguintes conceitos definidos na especificação do XACML, e vistos na figura 4.7 são relevantes para este trabalho:

***Policy Enforcement Point* (PEP)** é a entidade que realiza o controle de acesso, requisitando decisões e efetivando as decisões de autorização.

***Policy Decision Point* (PDP)** é a entidade que avalia a política aplicável e executa decisões de autorização.

***Policy Information Point* (PIP)** é a entidade que fornece os valores dos atributos.

***Policy Administration Point* (PAP)** é a entidade que cria as políticas ou os conjuntos de políticas.

***Context Handler*** é a entidade responsável por traduzir pedidos de decisões para o formato canônico do XACML, e converter as decisões para o formato nativo de resposta.

***Environment*** é semelhante às condições no modelo $\text{UCON}_{\text{ABC}}$, refere-se a condições ambientais, independentes do sujeito, recurso ou ação.

***Obligation*** é uma operação definida em uma política que deve ser executada pelo PEP junto com a efetivação de uma decisão de autorização.

O PEP cria uma requisição utilizando os dados do requisitante, do recurso e da ação passados, e enviará essa requisição para o PDP para a avaliação. O PDP procura pelas políticas

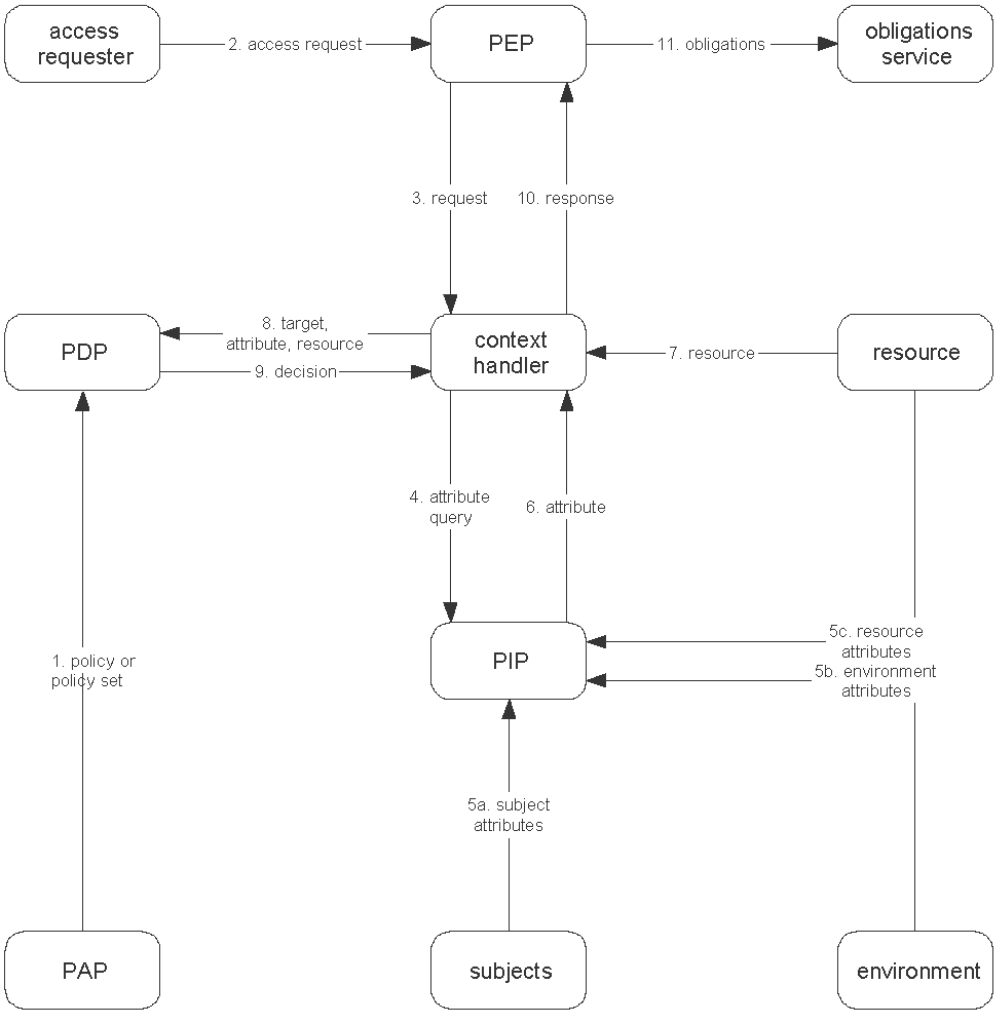


Figura 4.7: XACML: Fluxo de Dados (extraído de [45])

aplicáveis a esta requisição, toma uma decisão, e envia o resultado para o PEP, que irá se encarregar de efetivar a decisão tomada (permitir ou negar o acesso).

São quatro os valores que podem ser retornados pelo PDP, referentes à decisão quanto ao acesso: `Permitido`, `Negado`, `Indeterminado` ou `Não Aplicável`. O valor `Indeterminado` é retornado quando não há informações suficientes, há mais de uma política ou um grupo de políticas aplicáveis ou quando ocorreu um erro na tomada de decisão. O valor `Não Aplicável` é retornado quando não foi possível encontrar uma política ou um grupo de políticas compatíveis com a requisição.

Abaixo é apresentado um exemplo utilizando XACML. O cliente está tentando acessar um recurso no servidor para executar a ação `deploy`. O PEP converte a requisição do formato HTTP para XACML Request (figura 4.8 na página oposta). Em seguida, o PDP localiza a política aplicável, neste caso `DeployerPolicy` (figura 4.9 na página 32), utilizando as informações contida no atributo `Target`, baseado nos elementos `Subject`, `Resource` e `Action`.

Essa política contém apenas uma regra, `Rule`, que deve ter os elementos `Target` e `Condition` analisados. Como todas as condições e os elementos de `Target` são satisfeitos, o PDP irá retornar `Permitido` para o PEP, que deverá permitir o acesso.

```
<Request>
 <Subject>
  <Attribute AttributeId="group"
      DataType="http://www.w3.org/2001/XMLSchema#string"
      Issuer="admin@bea.com">
    <AttributeValue>Deployer
  </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://bea.com/deploy/control.html
  </Attribute>
 </Resource>
 <Action>
  <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:\
                         action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string">
    <AttributeValue>redeploy
  </Attribute>
 </Action>
</Request>
```

Figura 4.8: XACML: Requisição (extraído de [14])

A principal vantagem da XACML é sua provisão para expansões, que permite facilmente adicionar novas funções de controle de acesso, porém esta vantagem é obscurecida pela demasiada complexidade e extensão das políticas. A XACML também não trata mutabilidade de atributos, apesar de poder ser adaptada para que as políticas sejam continuamente avaliadas.

Devido a essas desvantagens, a XACML torna-se uma opção inferior ao modelo $\text{UCON}_{\text{ABC}}$ para controle de recursos em sistemas operacionais, e destina-se mais para ambientes atendidos por *Web Services*.

### 4.3.3 Systrace

O mecanismo Systrace, apresentado por Niels Provos [37] e implementado nos sistema operacionais OpenBSD, NetBSD e OpenDarwin[2], intercepta, ou interpõe, todas as chamadas de sistemas executadas por uma aplicação e verifica se cada chamada e seus argumentos são autorizados para esta aplicação, de acordo com sua política de execução.

As políticas podem ser geradas automaticamente ou interativamente. Para a geração automática, o usuário executa uma instância normal da aplicação, e todas as chamadas de sistema (*syscalls*) são registradas, juntamente de seus argumentos, criando uma lista de chamadas autorizadas. Em uma nova execução da aplicação, quando uma chamada é invocada mas não

[2]Existem, ainda, versões que foram portadas para os sistemas FreeBSD, MacOS X e GNU/Linux.

```
<Policy PolicyId="DeployerPolicy"
 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://bea.com/deploy/control.html</AttributeValue>
        <ResourceAttributeDesignator DataType=
                      "http://www.w3.org/2001/XMLSchema#anyURI"
                         AtributeId="urn:oasis:names:tc:xacml:\
                          1.0:resource:resource-id"/>
  </ResourceMatch></Resource></Resources>
  <Actions><AnyAction/></Actions>
 </Target>
 <Rule RuleId="RedeployRule" Effect="Permit">
  <Target><Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions><Action><ActionMatch MatchId="urn:oasis:names:\
                         tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType=
                  "http://www.w3.org/2001/XMLSchema#string">\
                   redeploy</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">Deployer
  </Condition>
 </Rule>
</Policy>
```

Figura 4.9: XACML: Política `DeployerPolicy` (extraído de [14])

está na lista daquelas autorizadas, o usuário recebe uma janela de confirmação, em que ele pode permitir a *syscall* ou terminar o programa. Alternativamente, o administrador do sistema tem a opção de sempre negar chamadas não autorizadas.

No modo interativo de criação de políticas, cada chamada de sistema deve ser autorizada pelo usuário, e, assim como no modo automático, suas informações são armazenadas para execuções futuras. Esse é o modo recomendado para a execução de binários que não se conhece o código-fonte.

A limitação da aplicação às chamadas de sistema configuradas na política é chamada de caixa-de-areia (*sandbox*[3]). Na figura 4.10 é apresentada a arquitetura de um sistema utilizando o mecanismo Systrace. A cada chamada de sistema efetuada pela aplicação na caixa-de-areia, o módulo do Systrace residente no *kernel* é invocado e, quando a política é simples, responde sua decisão diretamente para o *kernel*; se a política for complexa, um módulo é invocado em modo usuário, e a resposta é dada para o módulo no *kernel*, que repassa a decisão como se fosse sua.

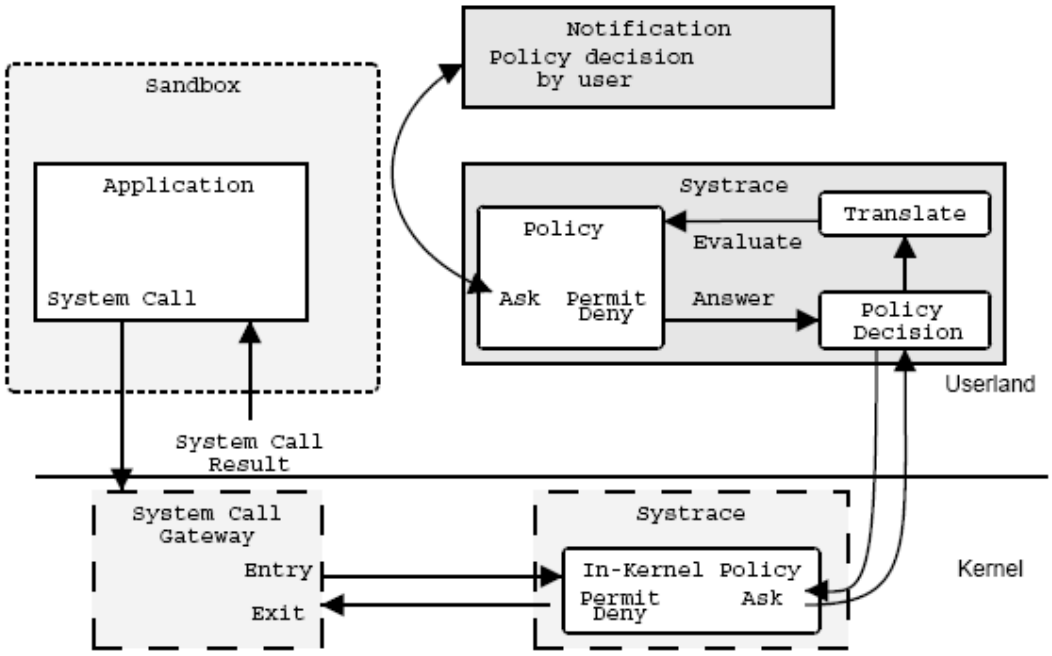


Figura 4.10: Systrace: Avaliação de uma Chamada de Sistema (extraído de [37])

A gramática que define a linguagem utilizada para criar as políticas para o Systrace pode ser vista na figura 4.11 na próxima página. Detalhes sobres os símbolos terminais encontram-se no OpenBSD Manual Pages para o Systrace [33].

A figura 4.12 na página seguinte mostra um trecho de um exemplo de política para a aplicação `ls`, que lista arquivos e diretórios em sistemas UNIX. Essa política diz que o usuário executando esta aplicação pode visualizar listagens contendo o seu diretório e os arquivos dentro dele, o diretório `/tmp` e o arquivo `/etc/pwd.db`; o diretório `/etc` não pode ser visto/listado por qualquer usuário que não pertença ao grupo `wheel`.

[3]Este termo tem um significado maior em segurança de sistemas: colocar uma aplicação em uma caixa-de-areia significa limitar as partes do sistema a que ela tem acesso.

```
filter = expression "then" action errorcode logcode
expression = symbol | "not" expression | "(" expression ")" |
  expression "and" expression | expression "or" expression
symbol = string typeoff "match" cmdstring |
  string typeoff "eq" cmdstring | string typeoff "neq" cmdstring |
  string typeoff "sub" cmdstring | string typeoff "nsub" cmdstring |
  string typeoff "inpath" cmdstring | string typeoff "re" cmdstring |
  "true"
typeoff = /* empty */ | "[" number "]"
action = "permit" | "deny" | "ask"
errorcode = /* empty */ | "[" string "]"
logcode = /* empty */ | "log"
```

Figura 4.11: Systrace: Gramática de Especificação de Políticas (extraído de [33])

```
Policy: /bin/ls, Emulation: native
[...]
native-fsread: filename eq "$HOME" then permit
native-fchdir: permit
[...]
native-fsread: filename eq "/tmp" then permit
native-stat: permit
native-fsread: filename match "$HOME/*" then permit
native-fsread: filename eq "/etc/pwd.db" then permit
[...]
native-fsread: filename eq "/etc" then deny[eperm], if group !=wheel
```

Figura 4.12: Systrace: Exemplo de Política para a Aplicação `ls` (extraído de [33])

A função fundamental do Systrace é prevenção de intrusão, mas pode-se ainda obter detecção de intrusão, através de monitoramentos remotos de vários sistemas que suportam esse mecanismo e enviam os avisos de violação de política para um gerenciador centralizado, e elevação segura de privilégios, através de políticas que indicam quais são, exatamente, as chamadas de sistemas que devem ser executadas com privilégios maiores (*e.g.* `root` ou `kmem`), diminuindo a necessidade de executar as aplicações inteiras com privilégios elevados.

Apesar do Systrace ser muito efetivo na prevenção e detecção de intrusão, sua finalidade é diferente daquela do modelo $\text{UCON}_{\text{ABC}}$. Suas políticas analisam apenas o estado atual da aplicação (se uma chamada de sistema específica é autorizada quando chamada com um conjunto de parâmetros), e não conseguem representar situações mais complexas, em que informações além do contexto atual são necessárias para uma tomada de decisão (*e.g.* quando precisa-se levar em conta o histórico de uso do sujeito). A caixa-de-areia criada pelo Systrace, apesar de muito interessante quando considera-se prevenção de intrusão, não é suficiente para efetuar controle de uso de recursos.

### 4.3.4 SELinux

O *Security-Enhanced Linux*, ou SELinux [55, 31], é uma série de melhoramentos de segurança, desenvolvidos pela Agência Nacional de Segurança dos EUA (NSA), para o *kernel*

do Linux. Estes melhoramentos oferecem ao administrador ferramentas para criar e efetivar políticas de controle de acesso mandatório (MAC) no sistema operacional, possibilitando o confinamento de usuários e serviços para evitar que, quando comprometidos, estes sujeitos não possam causar muitos danos no sistema.

Quatro sub-modelos de segurança formam o modelo do SELinux: *Type Enforcement* (TE), *Role Based Access Control* (RBAC), *User Identity* (UI) e *Multi-Level Security* (MLS). Os sujeitos e os objetos recebem rótulos (*labels*), utilizados para definir contextos de segurança; estes contextos possuem três atributos: um **tipo** ligado a um objeto ou **domínio** ligado a um sujeito[4], um **papel** que define uma função ou responsabilidade de um sujeito e uma **identificação de usuário**.

A políticas são descritas em uma linguagem de alto-nível, porém a complexidade do sistema pode facilmente levar a erros na definição da política [55]. Todo acesso que não for especificamente autorizado por uma política será negado ("*closed world policy*"). Existem dois tipos de políticas que podem ser criadas: decisões "sim/não", que resolve se um sujeito com seu *label* tem acesso a um objeto, também com seu *label*, e decisões de *labeling*, que definem qual contexto de segurança deve ser atribuído a um objeto, em relação aos contextos de segurança dos sujeitos e objetos a ele relacionados.

Objetos são divididos em classes (*i.e.* arquivo, diretório, *socket*, processo, entre outros), e a cada classe é vinculado um vetor de acesso, que define os direitos que podem ser concedidos sobre os objetos desta classe.

A seguir serão apresentadas algumas regras da política para o serviço `dhcpd`. As regras encontradas na figura 4.13 mostram exemplos que autorizam o serviço a enviar e receber dados através de *sockets*.

```
allow dhcpd_t netif_type : netif { tcp_send udp_send \
  rawip_send };
allow dhcpd_t node_type : node { tcp_recv udp_recv \
  rawip_recv };
```

Figura 4.13: Systrace: Exemplo de Regras para Enviar e Receber Dados Através de *Sockets* (extraído de [38])

A figura 4.14 apresenta regras que permitem ao próprio processo do serviço a tratar sinais recebidos.

```
allow dhcpd_t self : process { sigchld sigkill sigstop \
  signull signal fork };
```

Figura 4.14: Systrace: Exemplo de Regras para Tratar Sinais Recebidos (extraído de [38])

As regras descritas na figura 4.15 na próxima página autorizam o serviço a realizar diversas operações sobre o arquivo `/var/lib/dhcp/dhcpd.leases`.

[4]Apesar de "tipo" e "domínio" serem semanticamente distintos, o SELinux trata ambos pelo nome "tipo", permitindo, assim, a criação de apenas uma matriz de acesso.

```
allow dhcpd_t dhcpd_state_t : file { create ioctl read \
  getattr lock write setattr append link unlink rename };
type_transition dhcpd_t dhcp_state_t : file dhcpd_state_t;
```

Figura 4.15: Systrace: Exemplo de Regras para Realizar Operações em Arquivo (extraído de [38])

## 4.4 Controle de Uso

Controles que permitem limitar, por usuário, a forma ou a quantidade de utilização de um recurso, oferecem ao administrador um mecanismo para dividir estes recursos de forma mais adequada entre os usuários. Através de controles de uso, o administrador pode evitar abusos na utilização de um recurso, além de poder criar graus de privilégios entre os usuários, para que parcelas maiores de recursos importantes sejam destinadas àqueles mais privilegiados.

A seguir serão discutidos alguns mecanismos de controle de uso de recursos já disponíveis em diversos sistemas operacionais. É importante, porém, notar que a maioria desses mecanismos ou tratam apenas um recurso específico ou não fornecem controle sobre o acesso, criando, portanto, interfaces diferentes para a gerência do controle de uso e de acesso.

### 4.4.1 Alguns Mecanismos de Controle de Uso de Recursos em SO

Para Linux, o mecanismo de escalonador justo de processos, proposto por Rik van Riel [49], tenta dividir o processador de forma justa entre os usuários, mas está longe de ser uma solução para os demais recursos. Esse mecanismo concentra-se no escalonador de processos, não permite configuração diferenciada para os usuários e não tem a intenção de se estender aos demais recursos.

O sistema de tempo-real RT-Linux [54] oferece prioridade de execução e de alocação de memória para os processos marcados como tempo-real, e trata também as interrupções de sistema antes que o *kernel* do Linux as veja, conseguindo, assim, um ganho de desempenho para esses processos, porém não há muita granularidade no controle da alocação desses recursos entre os processos.

Pode-se, também, controlar a quantidade de memória que um usuário utiliza por sessão, mas não existe controle de memória que acumule a quantidade gasta em todas as sessões concorrentes do mesmo usuário. Outros recursos, como largura de banda de rede, arquivos que são utilizados concorrentemente e *software* com licença limitada são exemplos de pontos interessantes de controle, para os quais não há mecanismos de controle de uso prontamente disponíveis. O modelo $\text{UCON}_{\text{ABC}}$ oferece flexibilidade para que controles para esses recursos sejam criados.

O sistema operacional JX [13] é um sistema desenvolvido na linguagem Java que divide os recursos por domínios, ou divisões de processamento, e cada domínio é responsável pela gerência de seus recursos. Caso um recurso seja insuficiente em um domínio que esteja executando uma tarefa para outro domínio, o domínio que requisitou a tarefa pode emprestar partes de seus recursos para o que a está executando.

Semelhante ao conceito de domínios, os autores de *Resource Control of Distributed Applications in Heterogeneous Environments* [48] criaram um *framework* em que os recursos são agrupados por Unidades de Controle (UC). As UCs oferecem, através de uma interface

CORBA, mecanismos de controle de recursos que são traduzidos para os controles específicos de vários sistemas operacionais diferentes. Através das UCs, as aplicações podem alterar prioridades e reservar recursos, utilizando um única interface, em SOs como Linux, MS-Windows e MacOS.

O sistema operacional Solaris, da Sun Microsystems, possui um mecanismo [27] de definição de políticas que dizem quanto de CPU e memória um usuário pode consumir, quantos processos podem ser abertos, quantas sessões concorrentes ele pode abrir e qual a duração dessas sessões. Similarmente, a IBM oferece para o SO AIX o AIX 5L Workload Manager (WLM) [19], um sistema que permite controlar CPU, memória e I/O para processos individuais ou usuários. Também da IBM, o SO experimental K42 [2] torna cada recurso um objeto de sistema, com sua gerência executada dentro da instância; como cada recurso sabe sobre a necessidade das aplicações que o utilizam, ele pode adaptar-se ou fazer reservas de acordo com a aplicação corrente.

Uma técnica para garantir que um processo irá obter os recursos necessários para a sua execução é conhecida como *reserva de recursos*. Essa técnica, implementada por exemplo no sistema Linux/RK, pode ser utilizada de forma maliciosa, caso um processo consiga reservar 100% de um recurso e nunca mais o libere. Para proteger esses recursos e evitar ataques como negação-de-serviço, foi desenvolvido um mecanismo de controle de acesso semelhante a ACL, mas para recursos: *Resource Control Lists* (RCL) [28]. Com RCL, o sistema deve garantir que a alocação de recursos siga uma política pré-definida, ao invés de simplesmente concedê-los na ordem que foram solicitados.

As políticas definidas com RCL indicam quem pode acessar o recurso e quanto (em porcentagem e tempo) dele pode ser reservado para o usuário autorizado. Como o ponto de entrada da RCL e efetivação das políticas está nas rotinas utilizadas para invocar o recurso (como chamadas de sistema), sempre que a RCL for alterada, o direito de acesso pode ser retirado do usuário.

### 4.4.2 CKRM

O projeto *Class-based Kernel Resource Management* (CKRM) [30] tem como objeto criar um mecanismo que permita administradores do sistema operacional Linux criarem classes associadas a parcelas de cada recurso, para oferecer aos usuários quantidades diferenciadas de recursos. Os autores afirmam que há uma nova tendência em controle de recursos que pede uma gerência ligada à importância de uma tarefa, ou gerência orientada a objetivos, ao invés de ao simples consumo de recursos.

As classes são agrupamentos dinâmicos de objetos de um sistema operacional, organizados por seu tipo. À cada classe é vinculada uma parcela de cada recurso. Por exemplo, a classe de tarefas oferece controle sobre CPU, páginas físicas de memória, I/O de disco e largura de banda.

Um gerenciador de carga de sistema (*workload manager* ou WLM) é responsável por garantir a correta distribuição dos recursos, de acordo com políticas, e de redistribuí-los dinamicamente a fim de atingir metas pré-estabelecidas de qualidade de serviço (*i.e.* quando uma classe de maior prioridade começa a executar, o acesso a recursos por outras classes deve ser diminuído, para permitir que aquela de maior prioridade execute normalmente).

A arquitetura do CKRM pode ser vista na figura 4.16 na página seguinte.

Os componentes da arquitetura são:

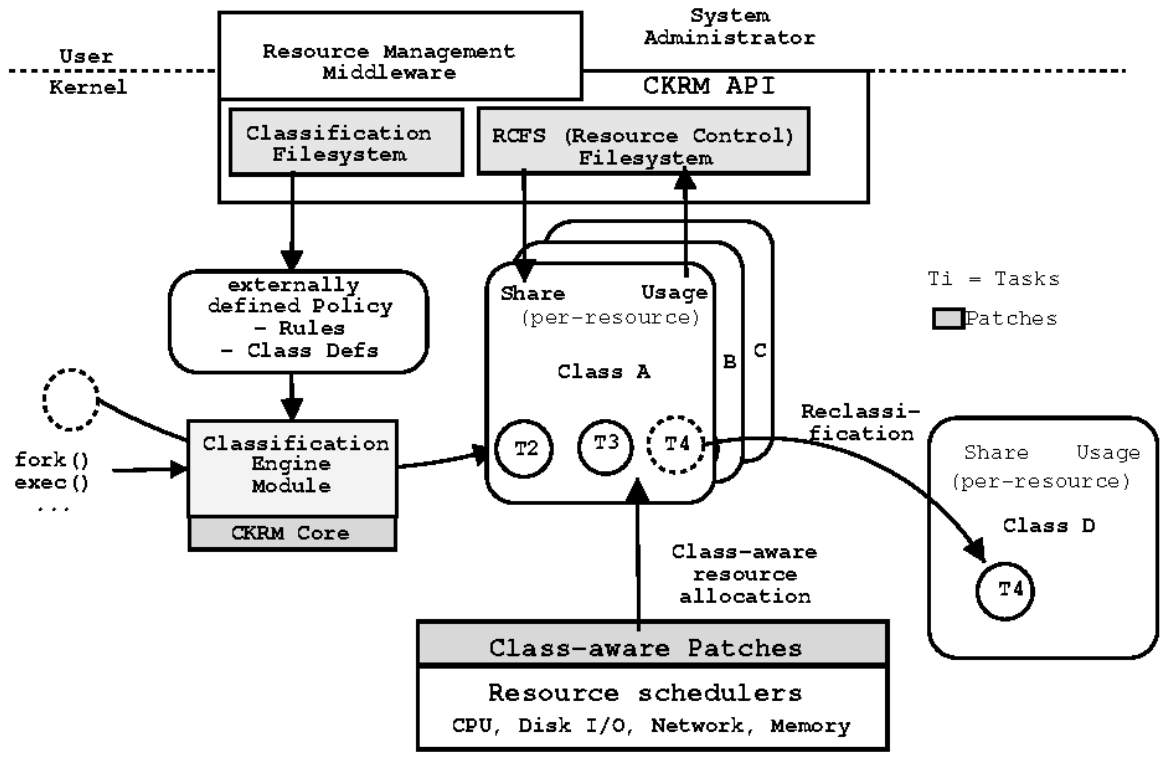


Figura 4.16: CKRM: Arquitetura do Mecanismo (extraído de [30])

***Core*** é o núcleo do CKRM, responsável pela definição das entidades básicas e pela conexão dos outros componentes. Entre as definições básicas estão as classes e os objetos do *kernel* a que elas se relacionam.

***Classification Engine* (CE)** é um componente opcional que auxilia na divisão dos objetos do *kernel* em classes, de acordo com o tipo do objeto.

***Resource Controllers*** controlam os recursos associados com um tipo de classe. Normalmente tem-se um *resource controller* para cada recurso disponível para a classe. Como cada controlador é independente, pode-se desativar aqueles que não são interessantes para a classe em questão (*e.g.* pode-se desativar o controlador de I/O se apenas os *ticks* de CPU forem necessários para uma classe).

***Resource Control File System* (RCFS)** é a principal interface entre os níveis de usuário e *kernel*. Pode ser montado e operado com as funções padrões de sistemas de arquivos (como `open`, `read` e `write`). As classes são mapeadas em diretórios, e escritas e leituras em arquivos virtuais dentro do RCFS operam funcionalidades dentro do *Core*.

O CKRM oferece controle apenas sobre o consumo de recursos, no sentido de evitar "exageros" no consumo. Diferente do modelo $\text{UCON}_{\text{ABC}}$, é um mecanismo baseado em políticas de qualidade-de-serviço, eficiente para evitar negações-de-serviço, mas insuficiente para efetivar políticas de segurança.

## 4.5 Conclusão do Capítulo

Este capítulo trouxe diversos mecanismos e linguagens disponíveis atualmente para controle de utilização de recursos em sistemas operacionais, e os comparou com o modelo $\text{UCON}_{\text{ABC}}$. A maioria dos mecanismos apresentados são específicos para o controle de apenas um recurso, e não podem ser aplicados de forma homogênea sobre os demais recursos.

O mecanismo implementado pelas RCLs atinge um resultado próximo do que estamos querendo conseguir em nossa implementação, até mesmo conseguindo retirar um direito sobre um objeto enquanto ele está sendo utilizado, de forma dinâmica. Porém, assim como as Listas de Controle de Acesso (ACLs), RCLs não contemplam a mutabilidade de atributos, obrigações e os diversos fatores externos que podem relevantes para o sistema.

Os diversos mecanismos apresentados neste capítulo mostram como são diversificadas as formas para se controlar recursos. Para cada necessidade específica foi criado um mecanismo específico, com sua interface própria e gerência especializada. Este trabalho tenta mostrar que, além da necessidade de se criar uma interface única para a gerência de recursos, os elementos do modelo $\text{UCON}_{\text{ABC}}$, como obrigações e mutabilidade de atributos, são interessantes para se tratar as novas formas como os recursos são disponibilizados e distribuídos entre os usuários.

Mutabilidade de atributos é importante para controle de recursos porque permite a criação de um histórico de utilização que pode ser atualizado por cada uso, em tempo real. Pode-se, por exemplo, consultar o histórico e verificar quanto tempo um usuário manteve o domínio sobre um recurso durante todo o tempo de existência do usuário no sistema; isso é diferente de limitar o tempo de uso dentro de uma sessão, como se consegue com RCL.

Obrigações introduzem o conceito de tarefas que o usuário deve executar para poder ter direito sobre um recurso. Com RCL, não se tem esse conceito de tarefa, e não há como aplicá-lo às políticas.

Informações sobre o estado atual do sistema também podem ser relevantes para tomadas de decisão. Condições, que apresentam dados como a carga total do sistema e a hora do dia, podem ser utilizadas em políticas para torná-las mais flexíveis com relação as necessidades dinâmicas do sistema. RCL fornece apenas um limite quanto ao tempo que um recurso pode ficar alocado para um usuário, mas isso não é uma condição, pois não é uma informação do sistema, e sim do usuário.

As RCLs cobrem grandemente as necessidades de controle de recurso, mas seu mecanismo base, as ACLs, não possui a expressividade e flexibilidade encontradas no modelo $\text{UCON}_{\text{ABC}}$. Considerando-se os demais trabalhos relacionados apresentados, vê-se que, apesar de a união dos diversos mecanismos provavelmente cobrir os principais recursos que se deseja controlar, o administrador iria ter que utilizar diversas interfaces, de forma descentralizada, para controlar o sistema; essa falta de uma interface única e centralizada é extremante indesejável, pois eleva a complexidade da tarefa de gerência e as chances de que um erro seja cometido na administração das políticas.

É importante perceber que todos esses recursos têm uma característica muito importante em comum: todos são acessados através de chamadas de sistema (*system calls* ou *syscalls*). Essas *syscalls* podem ser alteradas para executarem uma chamada a um mecanismo que implemente o modelo $\text{UCON}_{\text{ABC}}$. No próximo capítulo serão apresentadas a implementação e integração do modelo $\text{UCON}_{\text{ABC}}$ em um sistema operacional real, utilizando as chamadas de sistema como ponto de entrada do mecanismo, e a proposta deste trabalho será detalhada.

# Capítulo 5

# Aplicação do Modelo UCON$_{\mathrm{ABC}}$ ao Controle de Uso de Recursos de um Sistema Operacional

O controle de uso vai além do controle de acesso tradicional e trata os aspectos relacionados à mutabilidade dos atributos e validação contínua das restrições de uso. O modelo UCON$_{\mathrm{ABC}}$ estabelece um ferramental formal básico para tratar as novas necessidades de segurança e sistemas de controle de uso.

Como esse modelo foi apenas descrito por uma especificação formal, este trabalho propõe implementá-lo na forma de uma gramática LALR(1) e utilizá-la em um mecanismo de controle de uso de recursos, implantado no núcleo de um sistema operacional aberto.

## 5.1 Contexto e Motivação

Sistemas operacionais descendentes do UNIX, como o Linux e o OpenBSD, tratam seus recursos (*i.e.* dispositivos, memória, canais de comunicação, *et al*) como arquivos, e é natural que o controle de acesso utilizado para os arquivos seja também aplicado sobre os recursos.

O controle de acesso mais comumente encontrado em SO é o modelo discricionário (*Discretionary Access Control* - DAC), realizado por listas de controle de acesso (*Access Control Lists* - ACL). Extensões podem introduzir outras possibilidades de controle de acesso, como o SELinux [25], que fornece um controle de acesso mandatório (*Mandatory Access Control* - MAC), e existem alguns módulos que implementam alguma forma de controle baseado em papéis (*Role-Based Access Control* - RBAC) [41, 11].

Todavia, é importante destacar que em todos esses casos, o controle é estritamente de acesso. Ou seja, o mecanismo genérico não apresenta opções para controlar o tempo de uso de um recurso, por exemplo, nem verifica o horário em que esse recurso pode ser utilizado. Apesar de algumas dessas formas de controle adicionais poderem ser implementadas por mecanismos específicos para cada aplicação ou recurso, é desejável de um ponto de vista administrativo, poder efetuar o controle de todos os recursos do sistema através de uma política genérica e homogênea, que possa ser traduzida automaticamente nos mecanismos adequados ao controle de cada recurso específico.

Nota-se, também, que recursos como processador, interfaces de rede e outros dispositivos são fracamente controlados, pois os modelos aplicados em arquivos não se aplicam perfeita-

mente em vários casos de utilização de recursos. É fácil perceber a insuficiência do controle de acesso, quando se considera, por exemplo, uma placa de rede compartilhada por dois usuários, em que um deles copia constantemente grandes arquivos da Internet, enquanto que o outro apenas recebe alguns poucos *emails*: este último tem seu acesso prejudicado enquanto o primeiro usuário consome irrestritamente os recursos de rede. Um mecanismo de controle de uso deve balancear os recursos para que estes sejam consumidos de forma justa entre os usuários.

O mecanismo proposto neste trabalho deve permitir a utilização dos controles de acesso clássicos, como DAC, MAC e RBAC, e permitir a especificação de políticas mais abrangentes e elaboradas, voltadas para controle de uso, que considerem para a tomada de decisão, além da autorização, fatores de obrigações e condições.

Vários problemas de má utilização de recursos em sistemas operacionais, como um usuário infligindo um consumo de processador ou memória que impeça ou dificulte a utilização desses recursos por outros usuários, estão relacionados às deficiências dos mecanismos de controle de acesso. Com a expansão do controle de acesso para controle de uso espera-se solucionar estes problemas; e o modelo matemático do $\text{UCON}_{\text{ABC}}$ [35] demonstra seu potencial como solução adequada.

Alguns recursos são facilmente identificáveis como objetos de estudo interessantes para serem controlados. Acesso à rede, por exemplo, é um recurso de precisa de um controle melhor para evitar que algum usuário utilize constantemente toda a largura de banda, enquanto outros usuários com necessidades menores de banda não conseguem aproveitar o recurso; ou processamento de tarefas que comprometam o desempenho do sistema; ou mesmo para evitar que um usuário utilize mais memória do que seu limite, aproveitando-se da falta de controle deste recursos em sessões simultâneas.

## 5.2 Proposta do Trabalho

A proposta deste trabalho é criar um mecanismo que implemente o modelo $\text{UCON}_{\text{ABC}}$ em um sistema operacional (SO) comercialmente disponível e demonstrar sua efetividade no controle de recursos típicos de sistemas operacionais.

Para tanto, foi definida uma linguagem, através de uma gramática, para especificar as políticas de controle, um *parser* vinculado ao mecanismo de tomada de decisão de controle de acesso do SO, e um *enforcer* que garante a efetivação das decisões tomadas. Esta gramática, chamada *GUCON* (*Grammar for Usage Control* – Gramática para Controle de Uso), será baseada em um modelo lógico apresentado para o $\text{UCON}_{\text{ABC}}$ [56], e codificada utilizando o compilador de compiladores GNU Bison [43]. O código final, em linguagem C, será então inserido em um analisador léxico, que fará parte do mecanismo de avaliação de políticas (*engine*).

Como prova de conceito, as políticas serão vinculadas aos recursos disponíveis como arquivos simples, como arquivos regulares e diretórios, excluindo-se aqueles acessíveis através do diretório `/dev` e vínculos simbólicos. Para cada recurso/arquivo que se deseje controlar deverá ser criada uma política específica.

Para ativar o mecanismo, as chamadas de sistema (*syscalls*) serão interceptadas e tratadas. Se a chamada for de interesse para o controle de uso, a política associada ao recurso será interpretada linha a linha. Se a análise da política retornar um valor negativo, a chamada de sistema deverá ser encerrada e seu código de tratamento de erro será executado; se o valor de retorno da análise for positivo, a execução da *syscall* é autorizada.

Às chamadas de sistema caberá também a função de garantir a continuidade do modelo: algumas *syscalls* são executadas durante a utilização de um recurso, como `read` e `write` durante o uso de um arquivo, e estas *syscalls* chamarão o mecanismo. Como o sujeito somente poderá utilizar o objeto através de chamadas de sistema, sempre que o objeto for ser acessado, o mecanismo irá reavaliar as políticas `on`. Essa forma de execução implica que mudanças em atributos ou políticas somente terão efeito sobre um acesso quando uma *syscall* for executada.

Esse mecanismo será executado em paralelo com o mecanismo de controle de acesso existente no sistema operacional. Logo, caso não exista uma política para um recurso, o controle de acesso regular será utilizado para tomar a decisão de permitir ou negar a execução da chamada de sistema.

### 5.2.1 Limitações e Escopo

Espera-se um impacto negativo no desempenho do sistema operacional devido à natureza do mecanismo, que terá que passar as políticas por uma máquina de estados que irá reconhecê-las e efetuar a execução das regras. Esse impacto somente será medido após a implementação.

Além do impacto no desempenho, a funcionalidade das obrigações e condições será também limitada, pois não será permitida a execução de código fornecido livremente pelo usuário para representá-las. As obrigações serão codificadas em áreas de memória ou arquivos (chamados *slots*), que poderão ser acessados por algumas aplicações externas que irão preenchê-los com valores inteiros; caberá ao usuário criar políticas que interpretem corretamente esses valores.

As condições também serão limitadas àquelas codificadas internamente no mecanismo, que irão informar, com valores inteiros, dados como quantidade de processador ou memória livre e dia/hora atual. Assim, apenas algumas condições e obrigações estarão disponíveis.

É importante ressaltar ainda que o foco desta pesquisa limita-se ao controle de recursos centralizados em apenas um sistema operacional. A escalabilidade e a sua utilização em sistemas distribuídos deverá ser tema de pesquisa complementar, e poderá considerar a abordagem tomada por Nabhen *et al* para a criação de um mecanismo de controle de acesso distribuído baseado em políticas [29].

### 5.2.2 Problemas a Resolver

Diversas questões a serem solucionadas foram identificadas através da criação do protótipo, e são listadas abaixo. Esses problemas estão relacionados com a tradução de partes do modelo matemático para uma linguagem de programação:

**Representação de Políticas:** Para que as políticas possam ser especificadas, alguma forma de representação deverá ser criada para definir a interação entre atributos, autorização, obrigações e condições. Visando um mecanismo flexível, deve-se especificar, através de uma gramática, uma linguagem que aceite formulações aplicáveis a diversas situações – uma linguagem em que se possa representar as diversas políticas suportadas pelo $\text{UCON}_{\text{ABC}}$.

**Vínculo Recurso $\Leftrightarrow$ Política:** Devem existir políticas mandatórias, que colocam limites em todo o sistema, mas também políticas específicas para cada recurso, uma vez que recursos

diferentes têm necessidades distintas com relação ao seu uso. Então, deve-se permitir vincular políticas e recursos.

**Representação de Obrigações:** Obrigações são fortemente ligadas a dados de sistemas externos ao mecanismo, e muito dependentes de interação com esses sistemas. Como não se pode permitir que o mecanismo execute um código qualquer determinado pelo usuário, quando as políticas são definidas, por exemplo, deve-se definir um método para que os dados externos sejam inseridos no mecanismo.

**Representação de Condições:** Similarmente ao problema da coleta de dados para as obrigações, os dados referentes às condições também são externos ao mecanismo, e necessitam de um método para serem nele introduzidos.

O método mais adequado para lidar com dados externos é objeto dessa pesquisa, bem como a segurança do mecanismo como um todo, pois este será executado com os mesmos privilégios do *kernel*.

## 5.3 A Gramática Proposta

O objetivo da gramática proposta é aproximar-se ao máximo da descrição formal do UCON, sem perder a habilidade de implementá-la em um sistema real. Além disso, a gramática deve ser simples, para permitir uma implementação eficiente (rápida). Gramáticas podem ser descritas usando a notação EBNF (*Extended Backus-Naur Form*), compostas por símbolos terminais e não-terminais, que podem ser vistos respectivamente nas subseções 5.3.3 e 5.3.4.

### 5.3.1 Critérios Adotados

Para que a construção das políticas de controle seja simples e eficiente, a linguagem proposta deve seguir alguns critérios, descritos a seguir:

- A linguagem definida pela gramática não pode ser ambígua e deve expressar claramente os predicados funcionais para autorização, obrigação e condição.
- O acesso deve apenas ser permitido após a avaliação de todas as regras retornar verdadeiro. Cada regra terá a habilidade de completamente negar uma requisição de uso se seu retorno for falso, mas se não houver pelo menos uma regra que explicitamente negue o uso, o acesso deve ser permitido. Assim, um conjunto vazio de regras não tem o poder de negar um uso (a regra padrão é permiti-lo). Este posicionamento também foi utilizado em *Authorization in Distributed Systems: a Formal Approach* [52].
- Deve ser possível adicionar novas regras de controle sem mudar os atributos de usuários, quando os atributos necessários já estiverem presentes.
- O processo de avaliação deve ser eficiente, pois o tempo utilizado para tomar uma decisão não deve prejudicar a interação do sistema com o usuário (*i.e.* o tempo de tomada de decisão tem que ser semelhante ao tempo dos mecanismos de controle de acesso atuais).
- O analisador léxico deve ser implementável e ter uma fácil integração com as aplicações existentes (como sistemas operacionais).

### 5.3.2 Representação de Atributos, Obrigações e Condições

É importante notar que o avaliador das regras da gramática deve ser incorporado em um sistema real, então deve ser eficiente. Em um sistema real, alguns requisitos, como controles para garantir a integridade das bases de políticas, por exemplo, e limitações, inerentes à tradução do modelo formal para uma implementação real, são introduzidos. A forma como atributos, obrigações e condições são representados reflete estas limitações: na linguagem, atributos são implementados como variáveis inteiras ou *strings*:

**Variáveis inteiras** são utilizadas para guardar valores numéricos, como hora do dia, utilização de disco, etc.

**Variáveis *string*** são utilizadas para representar conjuntos de palavras, como os grupos ou papéis relacionados a um usuário.

As variáveis são representadas pelo símbolo `$` seguido de um nome (*e.g.* `$nome`). Quando uma variável é criada, um valor deve ser-lhe atribuído, e o tipo deste valor define o tipo da variável. Por exemplo, se uma variável recebe um valor inteiro, ela será sempre tratada como inteiro. A linguagem permite associar operadores em sequência. Por exemplo, é possível testar se o valor de uma variável é maior que a diferença de duas variáveis: `$credit > $cost - $discount`.

Obrigações e condições têm necessidades especiais para os dados que serão armazenados nas variáveis que as representam; como obrigações e condições tratam informações externas ao sistema, há uma questão de segurança a ser considerada. Existem algumas formas para se conseguir obter os dados externos para processar as obrigações e condições. A solução mais simples seria permitir que o usuário, ao declarar as políticas, indicasse algum código ou programa externo que seria executado sempre que fosse necessário buscar informações para obrigações ou condições (os dados seriam coletados de forma síncrona). Porém essa permissividade, apesar da enorme vantagem trazida por sua flexibilidade, implicaria em um código não controlado pelo mecanismo sendo executado com o mesmo nível de acesso do *kernel*.

Uma segunda possibilidade para obtenção desses dados é muito mais restrita, mas definitivamente mais segura. Essa abordagem envolve a coleta assíncrona dos dados necessários, e exige que o usuário preencha de alguma forma um espaço de memória (*buffer*) ou arquivo comum, disponibilizado pelo mecanismo e vinculado ao objeto controlado.

Esse espaço, chamado de *slot*, é disponibilizado quando um usuário requisita o acesso a um objeto, e pode ser preenchido por algumas aplicações definidas pelo usuário. A utilização de *slots* apresenta algumas desvantagens: os dados passados para o mecanismo serão limitados ao mesmo tipo de dado do *buffer*, a comunicação é assíncrona e o usuário deve criar sua própria ferramenta e garantir que esta seja executada corretamente para preencher o *slot*.

A figura 5.1 na próxima página mostra a relação entre o mecanismo de avaliação de políticas (*engine*) e a atualização do *slot*. O *engine* encontra o *slot* correto através de informações do usuário e do objeto e lê o valor armazenado. Um aplicativo lê o dado externo e o grava no *slot* periodicamente. Neste exemplo, o dado externo é a hora.

Obrigações são apresentadas como *slots*, que são acessados utilizando a palavra-chave `o$slot` seguida de um número (*e.g.* `o$slot 37`). Similarmente, uma palavra-chave é utilizada para acessar informações sobre condições. Como condições são vinculadas a dados do sistema, optou-se por codificar no mecanismo algumas funcionalidades para a coleta

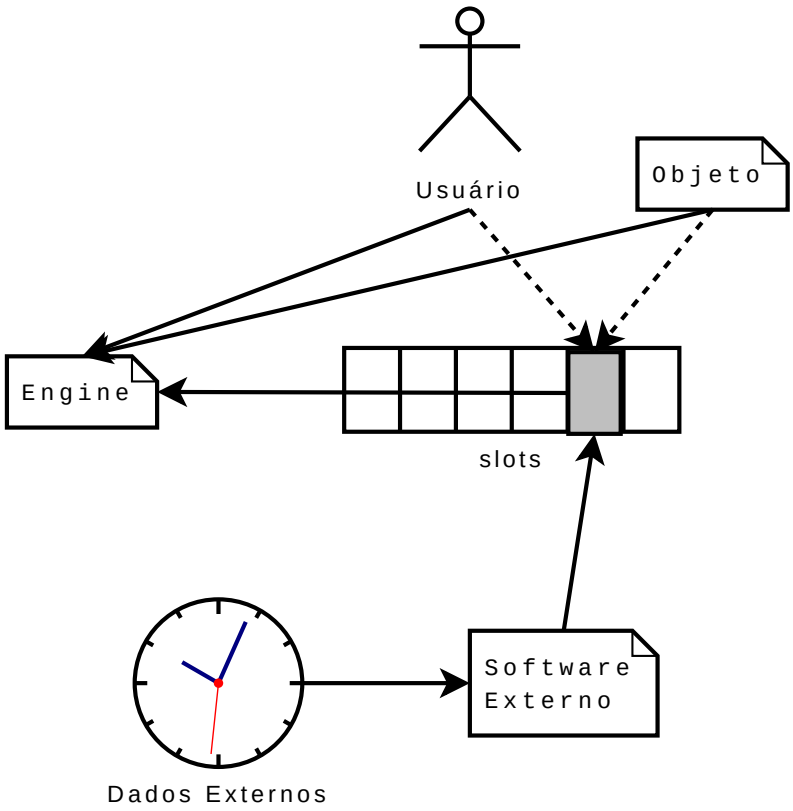


Figura 5.1: Exemplo de Utilização de *slot*

de alguns dados, como total de memória ou processador utilizado. Apenas condições pré-programadas estão disponíveis, como hora do dia (`c$time`), quantidade de CPU utilizada (`c$cpu_used`), quantidade de memória livre (`c$free_mem`) e espaço livre em uma partição do disco (`c$free_disk`). Estas condições específicas foram selecionadas para serem implementadas porque se relacionam fortemente com este trabalho, a implementação do *enforcer* em um sistema operacional para o controle de uso dos recursos locais.

### 5.3.3 Símbolos Terminais

Os símbolos terminais da gramática consistem em nomes de variáveis, palavras-chave, valores de constantes e operadores. Como foi mostrado no início desta seção, a representação de alguns aspectos do modelo foi tratada com limitações. As seguintes tabelas apresentam os símbolos terminais: a tabela 5.1 na página oposta mostra os símbolos relacionados às variáveis e aos valores, enquanto a tabela 5.2 na próxima página introduz os operadores e suas funcionalidades.

Tabela 5.1: Símbolos de Variáveis e Valores

| Símbolo | Tipo | Descrição |
|---|---|---|
| $*nome* | inteiro<br>*string* | Representa uma variável. |
| *dígitos* | inteiro | Um valor inteiro constante. |
| *string_1* $\cdots$ *string_n* | *string* | Um conjunto constante de *strings*. |
| o$slot | inteiro | Palavra-chave para acessar o valor de um *slot* (obrigação). |
| c$*nome* | inteiro | Representa uma condição (informação externa). |

Tabela 5.2: Símbolos dos Operadores

| Operador | Tipo | Funcionalidade |
|---|---|---|
| `=` | Atribuição | Atribui o valor à direita para a variável à esquerda. |
| `== != < > <= >=` | Comparação | Realiza comparações entre operandos. |
| `& \|` | Lógico | Operadores lógicos AND e OR. |
| `size` | Conjunto | Retorna o número de elementos em um conjunto. |
| `+ - * /` | Aritmético | Operações aritméticas básicas entre operandos numéricos. |
| `+ *` | Conjuntos | União e interseção entre operandos conjuntos. |
| `(`*expressão*`)` | Precedência | A expressão interna deve ser avaliada antes da parte externa. |
| `#` | - | Inicia um comentário (até o final da linha). |

### 5.3.4 Regras de Produção

A listagem na página seguinte apresenta as regras de produção da gramática definida utilizando a notação EBNF (*Extended Backus-Naur Form*) [21], e compilada utilizando o compilador de compiladores Bison [43]. A máquina de estados completa pode ser vista no anexo A. A descrição da gramática está dividida em 4 grupos de definições:

1. a descrição dos *tokens*: identificadores que serão utilizados para marcar qual o tipo do dado a que um símbolo terminal pertence (número, conjunto, atributo/variável, variável a ser declarada ou comentário);

2. os símbolos não-terminais que referem-se a grupos de regras, utilizados na composição de regras;

3. a definição da precedência dos operadores e sua associação (*i.e.* em que ordem os operadores serão avaliados); e

4. as regras de produção da gramática.

```
/* NOTE: not all tokens are defined here. Some are defined inline! Please see
 * the generated graph for a collection of all tokens. */
%token <value> NUM      /* numbers */
%token <set> SET        /* sets are words (strings separated by ' ') */
%token <att> ATT        /* attributes with name and value */
%token <att> NATT       /* name of attributes to be configured: Not ATT yet */
%token COMMENT          /* General comments identifier */

/* define the non-terminal types */
%type <value> values attmath
%type <value> condition obligation
%type <i_value> exp line

/* operations between values */
%nonassoc "==" '<' '>' "!=" "<=" ">="
%left '&' '|'
%left '-' '+'
%left '*' '/'
%left "size"

%%

/* input is our starting symbol: there can be as many empty lines as we want
 * followed by a line symbol */
input:                  /* empty */
        | input line
;

line:       exp '\n'
        | COMMENT
        | error {YYABORT;}
;

/* exp represents the operations over attributes */
exp:        NATT '=' values
        | ATT  '=' values
        | exp  "==" exp
        | exp  '<' exp
        | exp  '>' exp
        | exp  "!=" exp
        | exp  "<=" exp
        | exp  ">=" exp
        | exp  '&' exp
        | exp  '|' exp
        | '(' exp ')'
        | "size" exp
        | values
;

/* values represents all the values and attribute can have:
 * attmath -> math between attributes
 * condition -> one of the predefined condition symbols
 * obligation -> the value stored in a obligation slot
 */
```

```
values:     attmath
          | condition
          | obligation
;

/* The attmath symbol represents the operations that can be done between
 * valued and string attributes.
 * SET -> a space separated set of chars (words)
 */
attmath:    NUM
          | SET
          | ATT
          | attmath '+' attmath
          | attmath '-' attmath
          | attmath '/' attmath
          | attmath '*' attmath
;

condition:    "c$cpu_used"
            | "c$free_mem"
            | "c$free_disk"
            | "c$time"
;

/* It could be possible to have many slots, but for now lets leave one
 * per user */
obligation: "o$slot"
;

%%
```

## 5.4 Representação das Políticas

Políticas são expressas por combinações de atributos, constantes de inteiros e *strings*, obrigações, condições e os operadores que os relacionam. Cada usuário possui apenas um arquivo contendo seus atributos, não tendo políticas associadas a si. Por outro lado, cada objeto é associado a um arquivo de atributos e três arquivos de políticas, conforme indicado na figura 5.2 na página seguinte.

- O arquivo de atributos contém a inicialização dos atributos e deve ser processado antes dos outros arquivos.
- O primeiro arquivo de políticas contém as políticas *pre* e as atualizações *pre*.
- O segundo arquivo contém as políticas e atualizações *on*.
- Similarmente, o terceiro arquivo contém as atualizações *pos*.

Uma política de autorização trivial seria: `$usr_id == $owner_id`. Essa declaração deve ser colocada dentro do primeiro arquivo do objeto (*pre*), e define que o acesso será permitido se a identificação do usuário for igual ao proprietário do objeto.

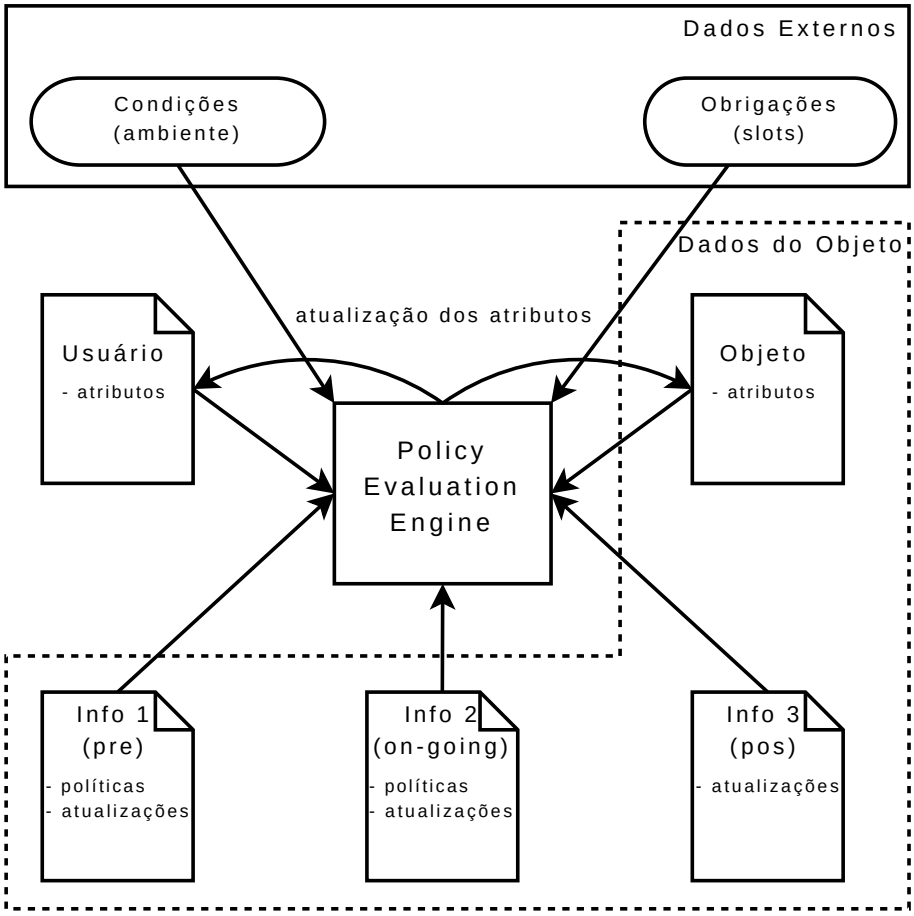


Figura 5.2: Arquivos Utilizados pelo *Engine*

Cada um dos três arquivos deve ser lido e suas regras devem ser avaliadas seqüencialmente. Se uma regra retornar `falso`, então o usuário terá seu pedido de uso negado e o restante do arquivo não será avaliado. Se o arquivo utilizado for de uma política *pre* ou *on*, e esta política negar o acesso, pelo menos o arquivo contendo as atualizações *pos* é avaliado, garantindo que os atributos serão corretamente atualizados.

## 5.5 Arquitetura do Protótipo

A implementação foi realizada como uma modificação no *kernel* do sistema operacional OpenBSD [50]. Este SO foi escolhido devido ao seu bom histórico com relação a falhas de segurança (“Apenas duas vulnerabilidades remotas na instalação *default* em mais de 10 anos” [50]) e por seu *kernel* bem documentado e relativamente menor que o de outros sistemas operacionais de código livre.

Como linguagem de programação foi escolhida a linguagem C, por ser esta a utilizada no *kernel*, e como ferramenta para a definição da gramática e geração do seu compilador, o compilador de compiladores Bison [43]. Assim, a implementação ficou dividida em duas partes principais: a gramática que é utilizada para criar um *Policy Evaluation Engine*, que irá validar e avaliar as políticas e atributos, e a parte funcional, chamada de *enforcer*, que faz interface com o *kernel*, executa o *engine* com os parâmetros corretos e que efetiva a decisão por este tomada.

O ponto de entrada no sistema é a função `trap`, responsável pelo tratamento de chamadas de sistemas (*syscalls*). Quando o *kernel* recebe uma chamada de sistema, o *enforcer* é ativado, antes que o código original da chamada de sistema seja executado, e são passadas as informações sobre o usuário que a invocou, o nome da chamada e o objeto envolvido no acesso. O *engine* irá avaliar cada regra e o direito requisitado para decidir quanto ao pedido de uso. Quando o *enforcer* receber a decisão do *engine*, será tomada a ação necessária para permitir ou negar o uso. A figura 5.3 ilustra este processo. Caso qualquer política falhe, o tratamento de erro da chamada de sistema é executado, retornando o erro `EACCES` (Erro de Acesso).

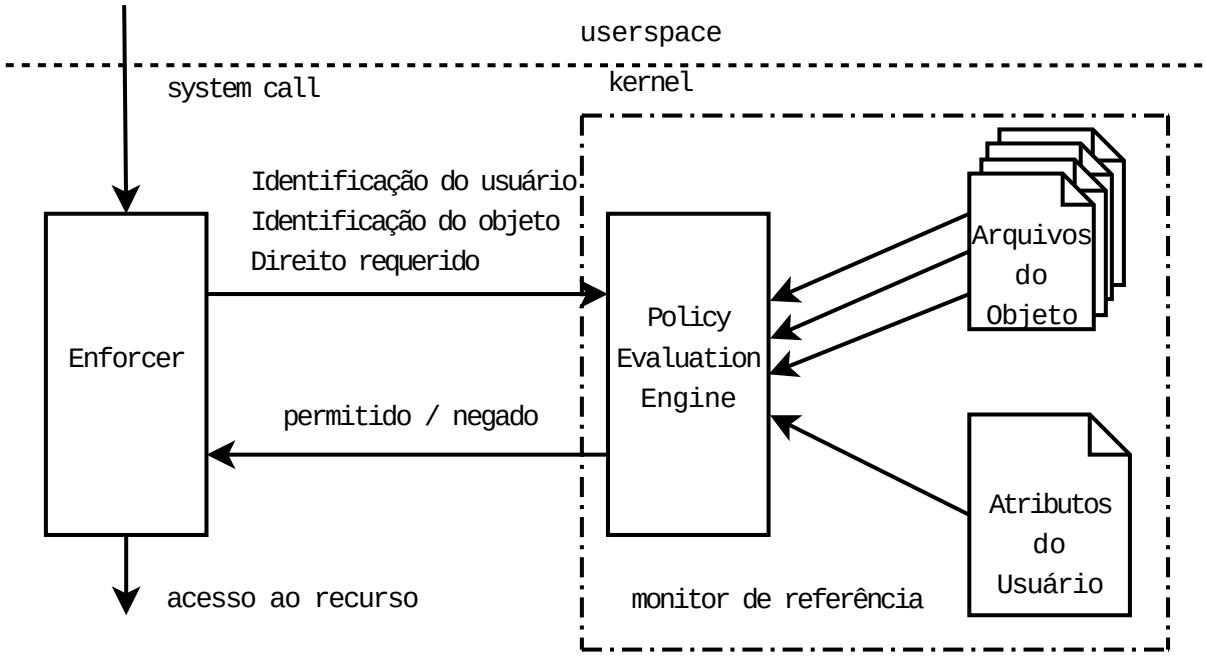


Figura 5.3: Interação *Enforcer/Engine*

A função `gucon_evaluate`, chamada pela função `syscall`, recebe como parâmetro a estrutura da chamada de sistema e o usuário que está executando o processo. Esta função encontra o *inode* e o *device number* vinculados ao arquivo em que a chamada deverá ser aplicada e procura pela política adequada no diretório do objeto: `/var/gucon/obj/`*devnumber*`/`*inode*.

O arquivo de política é escolhido segundo a chamada de sistema sendo invocada. A tabela 5.3 na página seguinte mostra todas as chamadas de sistema tratadas e o tipo de política a elas relacionada. Apenas as chamadas `SYS_open`, `SYS_close`, `SYS_read` e `SYS_write` foram realmente testadas; para as demais não foram criadas políticas, pois vão além do escopo deste trabalho.

Os atributos do objeto se encontram no mesmo diretório que as políticas; os atributos do usuário podem ser encontrados em `/var/gucon/usr/`*usuario*. O diretório do objeto contém também os arquivos que servem como *slot* de usuários, utilizados nas obrigações.

Caso o sistema não encontre políticas associadas a um dado arquivo, o sistema implementado recorre aos mecanismos nativos do sistema de arquivos, ao invés de simplesmente retornar um erro. Assim, quando não é possível avaliar uma política UCON, o núcleo utiliza o mecanismo DAC existente. Essa estratégia evita ter de associar políticas UCON a todos os arquivos do sistema.

Tabela 5.3: Chamadas de Sistema Tratadas

| Nome | Descrição | Política |
|---|---|---|
| `SYS_open` | Abre um arquivo para leitura ou gravação | pre |
| `SYS_close` | Fecha um arquivo referenciado por um descritor | pos |
| `SYS_read` | Lê de um arquivo | on |
| `SYS_write` | Escreve em um arquivo | on |
| `SYS_recvmesg` | Recebe uma mensagem de um *socket* | on |
| `SYS_recvfrom` | Recebe uma mensagem de um *socket* | on |
| `SYS_sendmsg` | Envia uma mensagem para um *socket* | on |
| `SYS_sendto` | Envia uma mensagem para um *socket* | on |
| `SYS_accept` | Aceita uma conexão em um *socket* | pre |
| `SYS_socket` | Cria um ponto de comunicação | pre |
| `SYS_connect` | Abre uma conexão em um *socket* | pre |
| `SYS_listen` | Espera por uma conexão em um *socket* | pre |
| `SYS_execve` | Executa um arquivo | pre |

### 5.5.1 Comparação com Outras Abordagens

Como será demonstrado na sessão 6.1, a linguagem definida pela gramática apresentada neste trabalho permite que políticas com um poder de expressão semelhante àquele encontrado nas linguagens descritas no capítulo 4 possam ser escritas, porém sem a grande complexidade encontrada, por exemplo, na XACML [17]. Essa simplicidade é favorável ao administrador do sistema, que pode ter uma visão mais clara das políticas.

Enquanto a linguagem Ponder [9] é voltada para políticas de autorização, e o CKRM [30], para políticas de qualidade-de-serviço, nossa abordagem trata, além desses dois aspectos, controle de acesso, controle de uso e mutabilidade de atributos. O Systrace [37], por sua vez, possibilita a criação de caixas-de-areia (*sandboxes*), que limitam o ambiente de execução da aplicação, mas não trata, efetivamente, o controle de uso de recursos.

## 5.6 Conclusão do Capítulo

Neste capítulo foram apresentados a proposta detalhada deste trabalho, com seu escopo, e os problemas que se espera resolver na tradução do modelo matemático do $\mathrm{UCON}_{\mathrm{ABC}}$ para uma linguagem de programação. Foram também descritos a gramática que define a linguagem utilizada para a criação das políticas de controle de uso e o projeto de integração com o sistema operacional.

Ao final do capítulo, encontra-se uma comparação da abordagem definida neste trabalho com aquelas apresentadas no capítulo 4. A implementação do modelo $\mathrm{UCON}_{\mathrm{ABC}}$, como apresentada aqui, traz um mecanismo de controle de uso mais completo que a linguagem Ponder [9] e o projeto CKRM [30], e com políticas mais legíveis que aquelas encontradas na XACML [17].

Essas vantagens podem ser visualizadas mais claramente ao analisar os exemplos de políticas apresentados no próximo capítulo (na sessão 6.1), em que a gramática e a implementação são avaliadas.

# Capítulo 6

# Avaliação do Sistema

Neste capítulo serão apresentados os resultados obtidos com os testes efetuados sobre a gramática e a implementação do protótipo. A avaliação da gramática isoladamente foi um fator importante para a validação do protótipo e a análise de seu resultado possibilitou a publicação do artigo "*A Grammar for Specifying Usage Control Policies*" [46].

## 6.1 Avaliação da Gramática

Essa avaliação tem como objetivo verificar o poder de expressividade da gramática, através de exemplos com diferentes graus de complexidade de expressão, e sua funcionalidade, ou seja, se o retorno da avaliação corretamente reflete aquilo que foi definido nas políticas e se a atualização dos atributos foi efetivada nos arquivos de atributos de usuários e objetos.

Avaliar a gramática antes de implantá-la no sistema operacional evita que erros de implantação sejam tratados como erros de implementação da gramática. Além disso, é mais fácil depurar apenas a gramática quando as complexidades do *kernel* não estão presentes.

Para exercitar o mecanismo de interpretação da linguagem, um programa que executa o *parser* da gramática, desenvolvido na linguagem C e compilado para o ambiente Linux, recebe como entrada o caminho completo para o arquivo contendo os atributos do usuário e um diretório contendo os atributos e as políticas do objeto, além do caminho para o próprio arquivo utilizado como objeto de acesso. Todos esses arquivos contêm texto simples.

Quando inicia a execução, o programa analisa as políticas *pre* e abre o arquivo com a chamada de sistema `open`. Então, até que o final do arquivo seja atingido, a chamada `read` é rotineiramente executada, sempre precedida da análise das políticas *on*. Quando o final do arquivo é atingido, executa-se as atualizações *pos* e a chamada de sistema `close` fecha o arquivo.

Após a análise de todas as políticas e atualizações, pode-se visualizar os arquivos de atributos, em formato de texto simples, para verificar se as atualizações foram executadas com sucesso. Além disso, pode-se, através da execução das chamadas de sistema, validar se a política está realmente sendo efetivada (*i.e.* se o acesso está sendo permitido ou negado como esperado).

Para a avaliação da gramática, vários exemplos extraídos dos trabalhos de Park e Sandhu foram convertidos de sua representação matemática original para a linguagem definida pela gramática GUCON, criando-se políticas que foram avaliadas pelo programa, e este deveria responder como "`Acesso Permitido`" ou "`Acesso Negado`" dependendo do resultado de cada avaliação.

A seguir serão descritos alguns exemplos de políticas definidas utilizando a linguagem proposta na gramática[1]. Ao comparar estes exemplos com aqueles encontrados em [35], percebe-se que a gramática proposta consegue exprimir as políticas definidas formalmente na proposta original do modelo UCON.

### 6.1.1 DAC com ACL Usando UCON$_{\mathrm{preA}_0}$

O controle de acesso Discricionário [24] com Listas de Controle de Acesso é um mecanismo de controle de acesso simples de implementar usando esta linguagem. Os atributos do usuário contêm apenas sua identificação: `$usr_id = 5456`. O *enforcer* irá passar o direito requisitado (`$right`) como um inteiro, 0 para leitura e 1 para escrita. O arquivo de atributos do objeto deve ser:

```
$obj_perm_read  = 1549 4334 5456    # permissão de leitura
$obj_perm_write = 4456 5456 7896    # permissão de escrita
```

O arquivo com as políticas *pre* contém apenas as linhas a seguir. Elas definem que, se um usuário está requisitando uma permissão de leitura ou escrita, sua identificação deve estar na lista `$obj_perm_*` respectiva.

```
( $right == 0 & size ($usr_id * $obj_perm_read)  != 0 ) |
( $right == 1 & size ($usr_id * $obj_perm_write) != 0 )
```

### 6.1.2 Políticas MAC Usando UCON$_{\mathrm{preA}_0}$

Um controle de acesso mandatório [3] pode ser implementado tratando-se o nível de acesso como um atributo do usuário e a classificação como um atributo do objeto. O arquivo do usuário contém apenas seu nível de acesso (`$clearance`). O arquivo de atributos do objeto contém sua classificação (`$classif`). No acesso de leitura, o nível de acesso deve ser maior ou igual à classificação; no acesso em escrita, o nível de acesso deve ser menor ou igual à classificação. O conteúdo do arquivo *pre* define essa política:

```
( $right == 0 & ($clearance >= $classif) ) |
( $right == 1 & ($clearance <= $classif) )
```

### 6.1.3 Controle de Uso com Obrigação Usando UCON$_{\mathrm{onB}_0}$

O usuário deve manter uma janela com publicidade aberta enquanto algum direito sobre um dado objeto é exercido. Um programa externo, possivelmente aquele que controla a janela, irá atualizar um *slot* de obrigações indexado pela identificação do usuário. O arquivo contendo os atributos do usuário possui apenas sua identificação (`$usr_id`). O objeto possui apenas um arquivo, contendo as políticas *on*, que será analisado sempre que uma chamada de sistema é invocada para exercer algum direito sobre o objeto:

[1] Vários exemplos apresentam regras divididas em várias linhas, mas esse formato é utilizado apenas para facilitar suas visualizações. As políticas utilizadas no sistema devem conter uma regra por linha, e cada regra deve ser terminada por uma quebra de linha.

```
o$slot $usr_id == 1
```

O *slot* indexado pela identificação do usuário é criado pelo sistema quando o usuário requisita um direito sobre o objeto. O programa externo que controla a janela de publicidade escreve 1 no *slot* logo que a janela for aberta, e escreve 0 quando for fechada.

### 6.1.4 Limitação de Acessos Usando $\text{UCON}_{\text{preA}_{13}\text{preC}_0}$

Um dado objeto pode ser acessado por 10 usuários simultaneamente entre 8 e 18 horas e por 20 usuários após às 18 horas. Usuários que já estão acessando o objeto não terão seus acessos revogados quando o horário mudar para 8 horas, mas nenhum novo usuário é admitido até haver disponibilidade. O arquivo de atributos do objeto contém o número de acessos simultâneos aceito e o horário de início e fim do período de redução da quantidade de usuários:

```
$users     = 0
$max_day   = 10
$max_night = 20
$day_start = 8
$day_end   = 18
```

O arquivo com as políticas *pre* controla o número de usuários simultâneos e atualiza a variável controlando este número:

```
(
  (
    (c$time > $day_start) &
    (c$time < $day_end)
  ) &
  ($users < $max_day)
) |
(
  (
    (c$time < $day_start) |
    (c$time > $day_end)
  ) &
  ($users < $max_night)
)
$users = $users + 1
```

Nota-se que na última linha, se o usuário tem seu acesso permitido, então o número de usuários é atualizado. Se a primeira regra falha, a análise do arquivo é interrompida e nenhuma atualização é executada. Deve existir também um arquivo com as políticas *pos*, para decrementar o número de usuários quando algum deles pára de acessar o objeto:

```
$users = $users - 1
```

### 6.1.5 Limitação de Usuários Simultâneos

Quantidade de usuários simultâneos limitada, revogação das permissões por tempo de uso, utilizando $\text{UCON}_{\text{onA}_{123}}$. O arquivo de atributos do usuário terá uma variável para salvar o tempo total de utilização e outra variável para guardar o horário da última ação:

```
$total_usage = 0
$last_action = 0
```

O objeto deve conter uma variável para configurar o número máximo de usuários, outra para especificar o tempo limite de uso (6 horas neste exemplo) e uma terceira variável para manter o número atual de usuários:

```
$max_users = 10
$max_usage = 6
$users     = 0
```

As políticas serão divididas em três arquivos. O arquivo *pre* conterá:

```
$users < $max_users
$users = $users + 1
$last_action = c$time
```

O arquivo *on* será:

```
$total_usage = $total_usage + (c$time - $last_action)
$max_usage < $total_usage
$last_action = c$time
```

Finalmente, o arquivo *pos* conterá:

```
$total_usage = 0
$last_action = 0
$users = $users - 1
```

Ao zerar os atributos do usuário, é possível liberar o objeto e requisitá-lo novamente em seguida, dando chance de utilização aos outros usuários, sem que o usuário atual tenha que perder sua permissão de uso quando não existe outro usuário esperando pelo objeto. Esta política poderia ser modificada para deixar salvo o tempo total de uso, requisitando uma ação administrativa para permitir que o usuário utilize o objeto novamente.

### 6.1.6 Controle Baseado em Papéis

Escolheu-se apresentar um exemplo de implementação de Controle de Acesso Baseado em Papéis [41, 11], porque acredita-se que RBAC não apenas contém muito do que é esperado de controles de Autorização, mas também por sua crescente importância como tecnologia de controle de acesso.

Neste exemplo, serão gradualmente adicionados os três níveis existentes no RBAC. Não trataremos do suporte à revisão da relação permissões-papéis pois, apesar de fornecermos dados suficientes para outras aplicações levantarem a informação requerida, nós acreditamos que uma revisão eficiente pode apenas ser executada fora do nosso sistema. Para os requisitos de cada nível, uma política ou atributo correspondente será detalhado na linguagem GUCON.

Três dos requisitos do RBAC Central são parte dos atributos do usuário:

```
$roles = director manager teller
$active_roles = manager teller
```

Estas duas variáveis fornecem uma atribuição usuário-papel muitos-para-muitos (`$roles` é um grupo de papéis, e cada papel pode ser atribuído a vários usuários), o suporte à revisão da relação usuário-papel (obtido por consulta ao conteúdo de `$role`) e um usuário pode ativar vários papéis simultaneamente (`$active_roles` representa um grupo de papéis). O controle de acesso é implementado como uma combinação dos atributos do objeto e das políticas. Um único atributo do objeto é necessário:

```
$required_roles = teller manager
```

As políticas *pre* devem cuidar para que pelo menos um dos papéis necessários esteja presente na lista de papéis do usuário, e atualizar os papéis ativos para adicionar este papel:

```
size ($required_roles * $roles) != 0
$active_role = $active_role + ($required_roles * $roles)
```

### 6.1.7 RBAC Hierárquico

Esta gramática dá suporte para uma hierarquia ou ordenação parcial limitada (RBAC Hierárquico). Como a gramática não suporta estruturas de dados complexas, como árvores ou listas encadeadas, as hierarquias devem ser construídas como variáveis de conjunto. Para ilustrar, considere a hierarquia apresentada na figura 6.1.

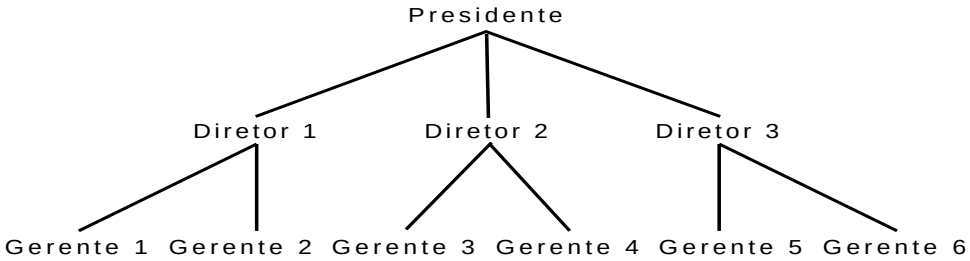


Figura 6.1: Exemplo de uma Hierarquia

Para descrever esta hierarquia, um quarto arquivo que contém valores válidos para todo o sistema pode ser utilizado para guardar as relações entre papéis. Nós iremos assumir que este arquivo existe, mas para manter a simplicidade do exemplo, em nossos testes a hierarquia foi implementada como parte dos atributos do objeto. Para um sistema real, esta abordagem é particularmente ruim, pois a mesma informação seria replicada diversas vezes, em todos os atributos dos diversos objetos que a utilizam.

As regras para a construção da hierarquia valem-se da capacidade de se expandir variáveis para os seus conteúdos quando atribui-se valores para uma nova variável. Assim, os papéis não seriam apenas baseados em elementos *string*, mas também em outras variáveis:

```
$Diretor_1 = Gerente_1 Gerente_2 Diretor_1
$Diretor_2 = Gerente_3 Gerente_4 Diretor_2
```

```
$Diretor_3 = Gerente_5 Gerente_6 Diretor_3
$Presidente  = $Diretor_1 $Diretor_2 $Diretor_3 Presidente
```

Cada diretor tem seu próprio papel e os papéis dos gerentes abaixo dele. O presidente tem seus papéis criados pela expansão de cada papel de todos os diretores e por seu próprio papel.

### 6.1.8 RBAC Restrito

Para se obter uma política RBAC restrita é necessário adicionar suporte à separação de tarefas (SoD). Esta modificação consiste em uma modificação da política *pre* e a adição de políticas *on*. As políticas *pre* para SoD Dinâmica (DSD) devem garantir que dois papéis conflitantes não sejam ativados ao mesmo tempo:

```
size ($active_role * $required_roles) == 1
```

Estamos assumindo que os papéis listados em `$required_roles` são mutuamente exclusivos. Se este não for o caso, nós poderíamos simplesmente adicionar uma nova variável aos atributos do objeto, listando os papéis conflitantes. A seguinte política *on* garante que se um papel conflitante for ativado pelo acesso a outro objeto (em que esses papéis não são conflitantes) ou se o papel necessário for desativado, então o acesso será revogado:

```
size ($active_role * $required_roles) == 1
```

Esta regra é a mesma adicionada ao arquivo *pre*, mas como ela é verificada durante o acesso, ela deve ser adicionada ao arquivo devido.

## 6.2 Avaliação do Protótipo

A avaliação do protótipo visa verificar o comportamento do sistema operacional e das aplicações quando o acesso/uso é negado através de uma política $\text{UCON}_{\text{ABC}}$. É especialmente interessante observar como as aplicações irão lidar com a retirada de um recurso durante o uso, através de uma violação de política *on*, pois normalmente, quando se trata de um mecanismo de controle de acesso clássico, o acesso é negado apenas antes da obtenção do direito sobre o recurso.

Também será avaliado o impacto do mecanismo sobre o tempo de execução de aplicações que fazem grande uso de chamadas de sistemas. A cada chamada de sistema o mecanismo irá abrir os arquivos envolvidos (atributos do sujeito e do objeto, e a política adequada à chamada) e interpretar as políticas que estão armazenadas em formato de texto; espera-se que esse processo tenha um grande impacto negativo no desempenho da aplicação.

### 6.2.1 Utilizando a Máquina Virtual QEMU

#### Ambiente de Avaliação

Para acelerar o desenvolvimento do mecanismo, a implementação foi frequentemente testada utilizando a máquina virtual QEMU[2] [4], versão 0.9.0, executando no ambiente operacional Linux. Por esta razão, optamos por apresentar aqui os resultados da avaliação preliminar do protótipo, efetuada neste ambiente. A subseção seguinte (6.2.2) traz os resultados da avaliação do sistema em uma plataforma real.

O computador real utilizado para executar o QEMU possui dois processadores AMD Opteron 64-bits 2GHz, 1024MB de *cache* L2 e 4GB de RAM. O ambiente emulado limita a memória disponível em 128MB e utiliza todo o processador disponível na máquina real.

A versão do sistema operacional instalado no QEMU, e que recebeu a implementação do mecanismo de controle de uso, é o OpenBSD 4.0-current Oct 04 2006. O *kernel* modificado com a implementação é compilado em outro ambiente com OpenBSD e copiado para o ambiente de teste.

#### Metodologia

Para testar o funcionamento do sistema, foi definido um cenário de acesso/uso simultâneo sobre um arquivo de música no formato MP3, com tamanho de 5 MBytes, a ser reproduzido pelo *mpg123*, um aplicativo simples de reprodução de áudio com interface em modo texto. Os atributos definidos para o arquivo MP3 são:

```
$obj_maxusers  = 10     # usuários simultâneos (max)
$obj_currusers = 0      # usuários simultâneos (atual)
$obj_maxcpu    = 30     # % máxima de CPU (Condition)
$obj_slotvalue = 5      # valor arbitrário para Obligation
$obj_groups    = USERS ADMINS # usuários autorizados (SET)
```

Por outro lado, o atributo do usuário é:

```
$usr_group = USERS     # grupo deste usuário
```

As política *pre*, *on* e *pos* foram assim definidas:

```
# pre policy
size ($obj_groups * $usr_group) >= 1 # grupo é autorizado?
$obj_currusers <= $obj_maxusers # máximo não foi atingido?
$obj_currusers = ($obj_currusers + 1) # usuários++
```

```
# on policy
$obj_maxcpu <= c$cpu_used # utilização de CPU é aceitável?
$obj_slotvalue >= o$slot # valor do slot é válido?
```

```
# pos policy
$obj_currusers = $obj_currusers - 1 # usuários--
```

[2] O QEMU é um emulador completo de um PC, e oferece um equivalente virtual para o processador, o disco rígido, as placas de rede e vídeo, entre outros periféricos.

O valor semântico da variável `$obj_slotvalue` não é definido aqui, pois este valor é irrelevante para o sistema. Apenas o administrador do arquivo pode definir o objetivo deste valor, que poderia ser, por exemplo, o número de minutos que o usuário já gastou em uma janela de outro *software* que atualiza o valor do *slot* (representando uma obrigação).

Inicialmente foram criadas situações especiais para verificar se as políticas estão realmente sendo respeitadas. A primeira modificação foi a do valor do *slot*, envolvendo a política *on*. Sempre que o *slot* era modificado para ser maior que o valor armazenado em `$obj_slotvalue`, esta política falhava e o usuário perdia o seu acesso ao arquivo. O *software* que acessava o arquivo MP3 (*mpg123*) tratou normalmente o erro `EACCES`.

Em seguida, foi gerada uma série de acessos concorrentes, variando entre 1 e 15 usuários simultâneos. A cada novo usuário que abria ou fechava o arquivo, as políticas *pre* e *pos* corretamente modificavam os atributos do objeto e, no caso da política *pre*, a permissão de uso do objeto era bloqueada quando o número máximo de usuários (`$obj_maxusers`) era atingido.

Para analisar o desempenho do sistema, verificou-se quantas vezes as chamadas de sistema relacionadas a arquivos foram invocadas a cada reprodução do arquivo MP3. Enquanto `SYS_open` e `SYS_close` foram invocadas 4 vezes cada, a chamada `SYS_read` foi invocada 18922 vezes. Assim, neste cenário, apenas a avaliação da política *on* tem impacto efetivo no desempenho do sistema, pois apenas esta é continuamente avaliada.

A figura 6.2 na página oposta apresenta o tempo gasto pelo núcleo (*systime*) para uma reprodução do arquivo, em função do número de regras na política *on*. Pode-se observar que, a cada nova regra adicionada a essa política, o tempo gasto pelo núcleo aumenta em cerca de 4 segundos. Em outras palavras, cada nova regra na política *on* aumenta em cerca de $200\mu s$ o tempo de cada execução da chamada de sistema `SYS_read`. Os tempos apresentados são valores médios obtidos em 10 execuções.

Com o *kernel* original (sem o GUCON), o tempo médio gasto no sistema é de 680 $ms$. O tempo gasto no contexto de usuário não é apresentado porque não varia com o número de regras, pois a avaliação é feita inteiramente pelo *kernel*.

### 6.2.2 Utilizando uma Máquina Real

#### Ambiente de Avaliação

Uma máquina real, exclusiva para a execução do protótipo, foi utilizada para medir com maior precisão o impacto do mecanismo no tempo de execução das aplicações. O computador está equipado com um processador AMD Athlon 64 Processor 3200+ (2.0GHz), 512MB de RAM e disco rígido Maxtor 6Y080L0 7200 RPM.

A versão do sistema operacional instalado na máquina real, e que recebeu a implementação do mecanismo de controle de uso, é o OpenBSD 4.1. O *kernel* modificado com a implementação é compilado em outro ambiente com OpenBSD e copiado para o ambiente de teste.

#### Metodologia

Novamente foram utilizados um arquivo de música no formato MP3, com tamanho de 5 MBytes, e o aplicativo para reprodução de áudio *mpg123*, com interface em modo texto. Como a chamada de sistema mais invocada é aquela que faz a leitura do arquivo, `SYS_read`, foi

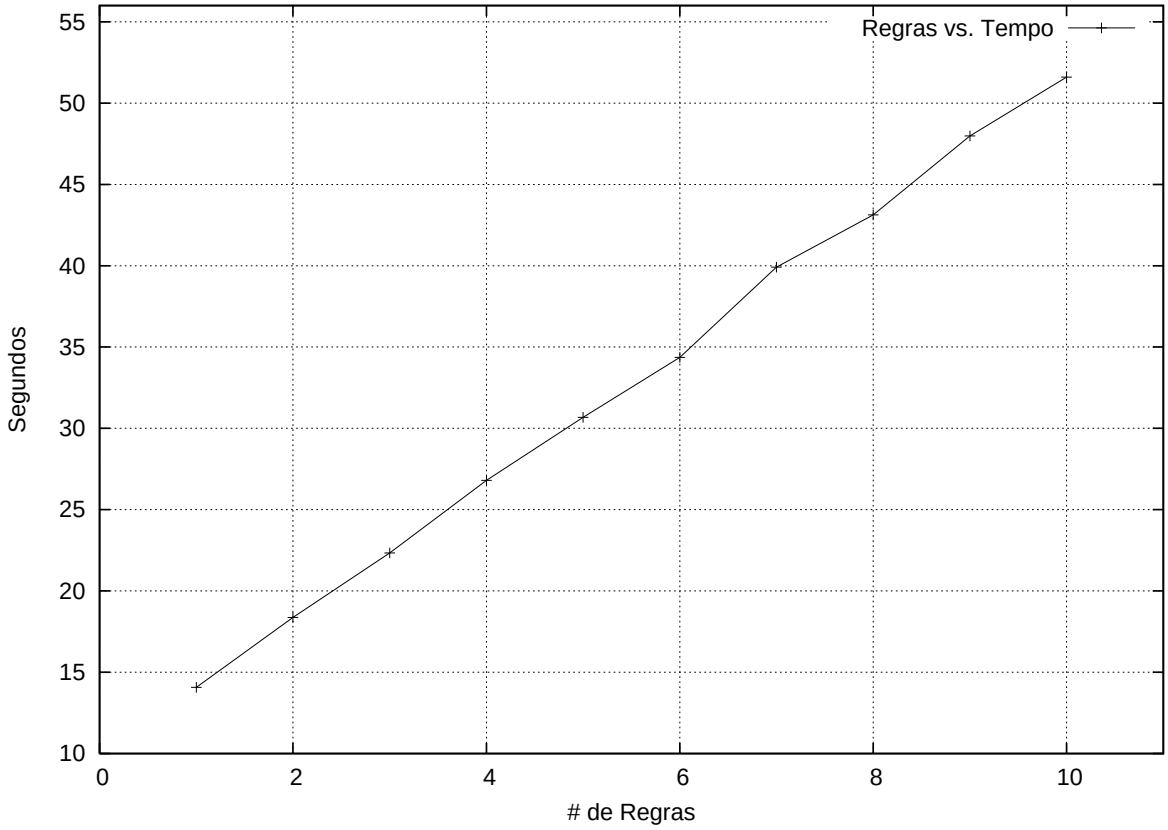


Figura 6.2: Relação Regras vs. Tempo para 18922 Chamadas (emulador)

utilizada apenas a política *on* para medir o desempenho do sistema com a variação no número de regras.

O arquivo de atributos do objeto contém a seguinte linha, que inicializa uma variável qualquer $x$:

```
$x = 1
```

E a regra que será repetida, no arquivo de políticas *on*, para avaliar o desempenho do sistema é:

```
$x == 1
```

A chamada de sistema `SYS_read` é invocada 18922 vezes em cada execução e, em cada uma delas, o mecanismo de avaliação das políticas $\text{UCON}_{\text{ABC}}$ é chamado para avaliar e efetivar a política *on* para o arquivo MP3. A tabela 6.1 na próxima página traz a média ($\overline{x}$) dos tempos gastos dentro do *kernel* (*systime*) durante a execução do aplicativo, além do desvio padrão da amostra ($\sigma$) e a diferença de tempo encontrada com a variação no número das regras ($\Delta$). A amostra, para cada variação no número de regras, é composta por 10 tomadas do *systime*, com desvio padrão igual ou inferior a 5% (cinco porcento). Na tabela 6.1, o $\Delta$ é medido entre a amostra atual e aquela imediatamente antecessora; na amostra com 20 regras, o $\Delta$ refere-se à diferença entre a média atual e a média da amostra com 10 regras.

A figura 6.3 na página seguinte apresenta graficamente os tempos médios de execução, extraídos da tabela 6.1.

Tabela 6.1: Tempo de Execução $\times$ Número de Regras

| **# Regras** | $\overline{x}$ | $\sigma$ | $\Delta$ |
|---|---|---|---|
| 0 | 0.442 | 0.048 | — |
| 1 | 0.475 | 0.034 | 0.033 |
| 2 | 0.502 | 0.043 | 0.027 |
| 3 | 0.528 | 0.030 | 0.026 |
| 4 | 0.588 | 0.044 | 0.060 |
| 5 | 0.604 | 0.050 | 0.016 |
| 6 | 0.617 | 0.040 | 0.013 |
| 7 | 0.641 | 0.049 | 0.024 |
| 8 | 0.728 | 0.049 | 0.087 |
| 9 | 0.738 | 0.046 | 0.010 |
| 10 | 0.781 | 0.050 | 0.043 |
| 20 | 1.067 | 0.045 | 0.286 |

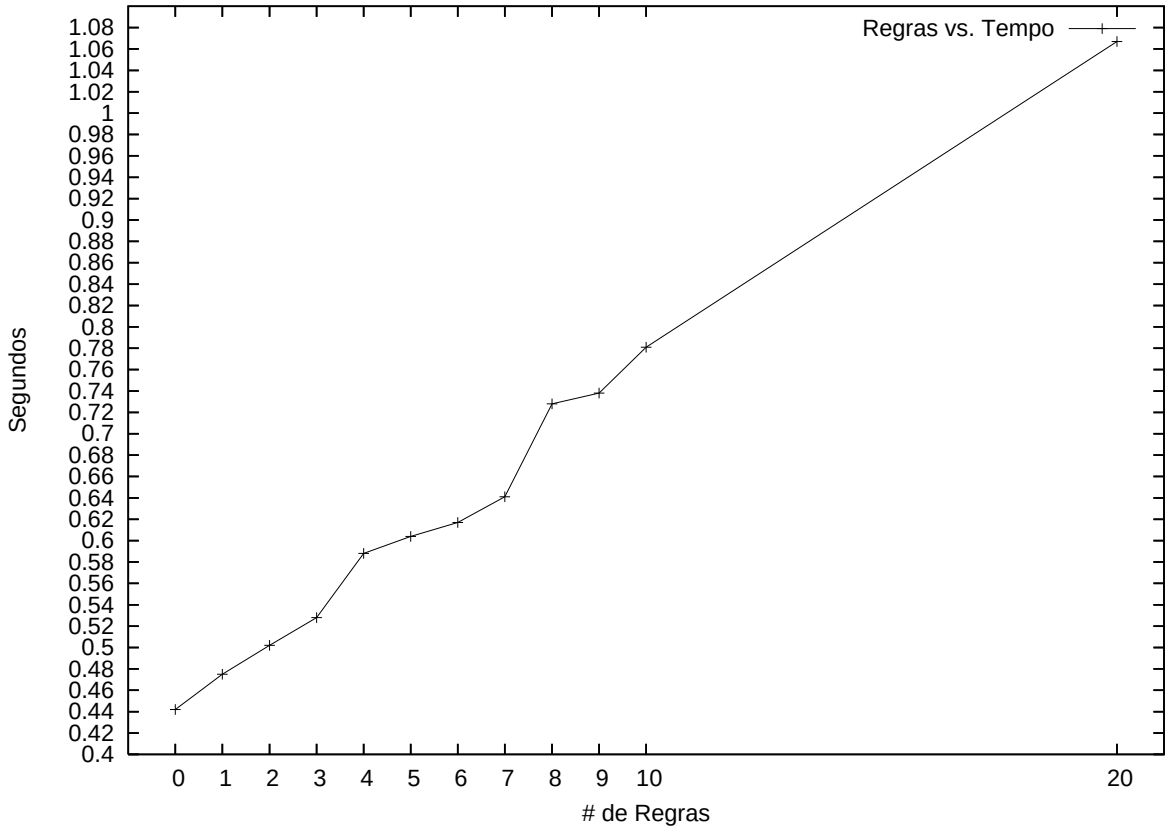


Figura 6.3: Relação Regras vs. Tempo para 18922 Chamadas (máquina real)

A cada nova regra adicionada, o impacto na execução da aplicação é, em média[3], de $33.9ms$ adicionados ao *systime*, o que implica, quando divide-se pelo número de chamadas (18922), em um impacto de $1.792\mu s$ por regra e por chamada. Regras mais complexas, como aquelas que utilizam diversos operadores matemáticos ou lógicos, ou que são mais compridas,

[3]A variação do tempo entre 10 e 20 regras não foi utilizada para o cálculo da média de impacto, porque os valores intermediários não foram levantados.

implicam em um aumento linear no tempo de processamento, pois demandam mais operações ou simplesmente levam mais tempo para serem lidas.

## 6.3 Discussão dos Resultados

Os testes realizados mostram que a gramática proposta e sua implementação podem ser utilizadas no contexto de um sistema operacional real para prover os recursos oferecidos pelo $\text{UCON}_{\text{ABC}}$, mesmo com um impacto no desempenho do sistema. Este impacto poderia ser reduzido caso fossem implementadas técnicas de *cache* de regras, para evitar ter que lê-las constantemente do disco, ou, em escala menor, se fosse criado um formato binário para armazenar as políticas, para reduzir o tamanho do *parser* e aumentar sua eficiência. Todavia, essas otimizações estão além do escopo atual deste trabalho.

No entanto, fica claro que os conceitos utilizados para a representação de Atributos, Obrigações e Condições satisfazem a descrição do modelo $\text{UCON}_{\text{ABC}}$, em especial, o conceito de *slots* para representar Obrigações.

A diferença entre o efeito no desempenho constatado no emulador e na máquina real é, provavelmente, causado pela lentidão do acesso ao disco virtual do emulador, durante a leitura das regras e das políticas.

Outra constatação que consideramos significativa foi perceber, nos diversos testes realizados com outros aplicativos, como editores de texto, e mesmo com o decodificador de MP3 utilizado nos experimentos, que a grande maioria dos programas trata de forma adequada o erro `EACCES` retornado pela função `syscall` quando de uma negação de acesso/uso por não-cumprimento de alguma política. Esse comportamento facilita a implantação de modelos e políticas mais complexos (como os apresentados neste artigo).

## 6.4 Conclusão do Capítulo

Este capítulo apresentou os testes funcionais e a avaliação de desempenho do mecanismo que implementa o modelo $\text{UCON}_{\text{ABC}}$. A gramática foi testada através de uma série de políticas, algumas extraídas de exemplos encontrados em alguns trabalhos relacionados [35, 56, 41].

O teste efetuado no emulador QEMU mostrou que o mecanismo responde como esperado pelo sistema operacional e pelas aplicações, isto é, que a resposta `EACCES` é adequada, e que a preempção do recurso (*i.e.* efetivação de uma política *on* violada) é possível.

O impacto do mecanismo no tempo de execução do *kernel* foi medido em uma máquina real, e foi determinado que, em média, cada regra utilizando apenas um operador, uma variável e um escalar acresce $1.792\mu s$ por chamada de sistema.

Estes resultados foram discutidos na seção 6.3, e foram apresentadas sugestões para reduzir o tempo de sistema utilizado para processar cada regra, como utilizar *caches* de regras para objetos acessados frequentemente.

O próximo capítulo trará uma discussão final do projeto e a conclusão deste trabalho.

# Capítulo 7

# Considerações Finais

Este trabalho apresentou uma prova-de-conceito de um mecanismo para controle de recursos em um sistema operacional, que implementa o modelo $\text{UCON}_{\text{ABC}}$. Esta é, provavelmente, a mais completa implementação deste modelo atualmente, ao menos no contexto de sistemas operacionais.

Entre os problemas que foram resolvidos durante a tradução da descrição matemática e lógica do modelo para uma linguagem de programação, estão a representação das políticas e sua vinculação aos objetos, e a representação das condições e das obrigações.

A gramática criada para descrever as políticas é uma das principais contribuições desta pesquisa, pois é nela que o modelo $\text{UCON}_{\text{ABC}}$ é efetivamente implementado, assim como o tratamento das obrigações e das condições. A apresentação desta gramática, sozinha, possibilitou a publicação de um artigo completo no evento *IEEE International Conference on Communications – ICC'07* [46]. Um novo artigo completo, descrevendo os experimentos realizados dentro do emulador QEMU, para validar o mecanismo e medir, superficialmente, o tempo gasto na avaliação das políticas, foi aceito e será apresentado no evento *VII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais – SBSEG 2007* da *Sociedade Brasileira de Computação* [47].

Apesar de o impacto do mecanismo no tempo de execução da aplicação, quando executado no QEMU, ter sido razoavelmente alto, os resultados detalhados deste impacto, levantados na máquina real, são muito satisfatórios. Especialmente considerando-se que técnicas como *cache* e compilação de políticas não foram utilizadas, e poderiam, se implementadas em trabalhos futuros, reduzir ainda mais o tempo de execução das aplicações.

A forma de acesso das condições e obrigações é uma das limitações deste trabalho. Como é muito importante evitar que executáveis fornecidos pelo usuário sejam carregados no espaço de memória do *kernel*, não foi possível seguir integralmente a descrição do modelo, em que condições e obrigações podem receber, através de programas auxiliares, qualquer informação externa. Ao invés disso, as condições foram reduzidas àquelas codificadas diretamente no mecanismo, como hora do dia e quantidade de processamento total sendo utilizada no sistema, entre outras, e as obrigações foram limitadas a um único valor do tipo inteiro, por usuário, para cada objeto, que deve ser preenchido em um arquivo por um programa auxiliar, executado no espaço do usuário.

Os objetivos deste trabalho foram: a criação de uma gramática para especificação de políticas $\text{UCON}_{\text{ABC}}$, a implementação do *parser* desta gramática em um mecanismo de controle de uso, e a implantação deste mecanismo em um sistema operacional, para a realização

de controle de acesso/uso de recursos. Considerando-se os resultados obtidos na avaliação do desempenho do mecanismo, a expressividade da gramática, e o funcionamento geral do protótipo, pode-se afirmar que estes objetivos foram atingidos. A próxima seção irá apresentar os objetivos traçados para trabalhos futuros.

## 7.1 Trabalhos Futuros

O grupo de pesquisa em sistemas autonômicos da IBM define [20] 8 elementos que caracterizam esses sistemas. Entre eles, destacamos os seguintes:

- um sistema autonômico deve ter conhecimento sobre seu estado atual;
- esse sistema deve configurar-se e reconfigurar-se automaticamente;
- ele deve monitorar seus recursos, e controlá-los a fim de evitar problemas (como monopólio na utilização de um recurso);
- deve proteger-se de ataques e manter a segurança e integridade do sistema; e
- um sistema autonômico deve conhecer seu ambiente e seus sistemas vizinhos.

Essas características podem ser tratadas pelo modelo $\text{UCON}_{\text{ABC}}$ e, com algumas adaptações, pelo mecanismo apresentado neste trabalho. O campo de pesquisa em sistemas autonômicos ainda é recente, e os mecanismos para implantá-los ainda não estão bem definidos. Assim, a integração do modelo $\text{UCON}_{\text{ABC}}$, e de nosso mecanismo, com o modelo de sistemas autonômicos é o objeto de estudo de nosso grupo de pesquisa.

# Referências Bibliográficas


[1] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Volume II, Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[2] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42 overview. Technical report, IBM, August 2002.

[3] D.E. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, MITRE Corporation, Massachusetts, USA, March 1976.

[4] Fabrice Bellard. QEMU: Open source processor emulator. webpage `http://fabrice.bellard.free.fr/qemu/`, 2007. acesso em 25 de junho de 2007.

[5] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[6] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[7] Alexandre Camy, Carla Westphall, and Rafael Righi. Aplicação do modelo $UCON_{ABC}$ em sistemas de comércio eletrônico B2B. In $5^{th}$ *Brazilian Symposium on Information Security and Computing Systems (SBSEG)*, 2005.

[8] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195, 1987.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[10] Electronic Privacy Information Center – EPIC. Microsoft palladium: Next generation secure computing base. webpage `http://www.epic.org/privacy/consumer/microsoft/palladium.html`, 2002. acesso em 25 de junho de 2007.

[11] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[12] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical Unix and Internet security ($3^{rd}$ ed.)*, chapter 6. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[13] M. Golm and J. Kleinoeder. Ubiquitous computing and the need for a new operating system architecture, 2001. acesso em 22 de junho de 2007.

[14] Phil Griffin. Introduction to XACML. webpage `http://dev2dev.bea.com/pub/a/2004/02/xacml.html`, 2004. acesso em 28 de junho de 2007.

[15] Marcelo Shinji Higashiyama. JaCoWeb-ABC: Integração do modelo de controle de acesso $\text{UCON}_{\text{ABC}}$ no CORBASec. Master's thesis, Pontifícia Universidade Católica do Paraná, 2005.

[16] Marcelo Shinji Higashiyama, Lau Lung, Rafael Obelheiro, and Joni Fraga. JaCoWeb-ABC: Integração do modelo de controle de acesso $UCON_{ABC}$ no CORBASec. In $5^{th}$ *Brazilian Symposium on Information Security and Computing Systems (SBSeg)*, 2005.

[17] Vincent C. Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in XACML. In *Proc. 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007)*, July 2007.

[18] Renato Iannella. Open Digital Rights Language (ODRL) version 1.1. Technical report, IPR Systems, webpage `http://www.w3.org/TR/odrl/`, September 2002. acesso em 22 de junho de 2007.

[19] IBM. AIX 5L workload manager (WLM). webpage `http://www.redbooks.ibm.com/redbooks/SG245977.html`. acesso em 22 de junho de 2007.

[20] IBM Research. Autonomic computing. webpage `http://www.research.ibm.com/autonomic/`, 2007. acesso em 20 de julho de 2007.

[21] ISO. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Genève, Switzerland, 1996.

[22] Roger L. Kay. How to implement trusted computing: A guide to tighter enterprise security. White paper, Endpoint Technologies Associates, webpage `https://www.trustedcomputinggroup.org/news/Industry_Data/Implementing_Trusted_Computing_RK.pdf`, 2006. acesso em 22 de junho de 2007.

[23] Jungin Kim and Bhavani Thuraisingham. Dependable and secure TMO scheme. In *IEEE Intl Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 133–140, 2006.

[24] Butler W. Lampson. Protection. *SIGOPS Operating System Review*, 8(1):18–24, 1974.

[25] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In Clem Cole, editor, *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42. USENIX, 2001.

[26] Michael J. May. Privacy system encoded using EPAL 1.2. Technical report, University of Pennsylvania, webpage http://www.seas.upenn.edu/\~mjmay/pubs/epal/Privacy-System-as-EPAL.pdf, August 2004. acesso em 11 de agosto de 2006.

[27] Richard McDougall. Solaris resource manager. Sun BluePrints 806-3839-10, Sun Microsystems, webpage http://www.sun.com/blueprints/0499/solaris1.pdf, April 1999. acesso em 10 de julho de 2007.

[28] Akihiko Miyoshi and Ragunathan (Raj) Rajkumar. Protecting resources with resource control lists. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 85, Washington, DC, USA, 2001. IEEE Computer Society.

[29] Ricardo Nabhen, Edgard Jamhour, and Carlos Maziero. A policy based framework for access control. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *ICICS*, volume 2836 of *Lecture Notes in Computer Science*, pages 47–59. Springer, 2003.

[30] Shailabh Nagar, Rik van Riel, Hubertus Franke, Chandra Seetharaman, Vivek Kashyap, and Haoqiang Zheng. Improving Linux resource control using CKRM. In *Proceedings of the Linux Symposium*, volume 2, pages 511–524, Ottawa, Canada, 2004.

[31] National Security Agency (NSA). Security-enhanced linux (SELinux). webpage http://www.nsa.gov/selinux, 2007. acesso em 15 de novembro de 2007.

[32] Office of Standards and Products, editors. *Trusted Computer System Evaluation Criteria*, chapter 6. The Department of Defense – DoD, National Computer Security Center, Fort Meade, MD 20755-6000, 1985.

[33] OpenBSD. *OpenBSD reference manual: SYSTRACE*, May 2007. webpage http://www.openbsd.org/cgi-bin/man.cgi?query=systrace. acesso em 17 de julho de 2007.

[34] Jaehong Park and Ravi Sandhu. Usage control: A vision for next generation access control. In Vladimir Gorodetsky, Leonard J. Popyack, and Victor A. Skormin, editors, *Intl Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, volume 2776 of *LNCS*, pages 17–31. Springer, 2003.

[35] Jaehong Park and Ravi Sandhu. The $UCON_{ABC}$ usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[36] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Communications of the ACM*, 49(9):39–44, 2006.

[37] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[38] Red Hat. *Red Hat SELinux GuidePolicy Types – Example Policy Reference – dhcpd*, 2007. webpage http://www.linuxtopia.org/online_books/redhat_

selinux_guide/selg-section-0021.html. acesso em 15 de novembro de 2007.

[39] Rob Reichel. Inside windows nt security. *Windows/DOS Developer's Journal*, 4(4):6–19, 1993.

[40] Tatyana Ryutov and Clifford Neuman. Representation and evaluation of security policies for distributed system services. In *DARPA Information Survivability Conference Exposition*, Healton Head, South Carolina, January 2000.

[41] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *ACM workshop on Role-based access control*, pages 47–63, 2000.

[42] P. Schneck. Persistent access control to prevent piracy of digital information. *Proceedings of the IEEE*, 87(7):1239–1250, July 1999.

[43] Richard Stallman and Charles Donnelly. *Bison – The Yacc-compatible Parser Generator*. Free Software Foundation, Inc., 2005. ISBN 1-882114-44-2.

[44] Amril Syalim, Toshihiro Tabata, and Kouichi Sakurai. Usage control model and architecture for data confidentiality in a database service provider. *IPSJ Digital Courier*, 2:621–626, 2006.

[45] OASIS Access Control TC. eXtensible Access Control Markup Language (XACML) version 2.0. OASIS standard, OASIS, webpage http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, February 2005. acesso em 30 de outubro de 2006.

[46] Rafael Teigão, Carlos Maziero, and Altair Santin. A grammar for specifying usage control policies. In *IEEE Intl Conference on Communications (ICC)*, 2007.

[47] Rafael Teigão, Carlos Maziero, and Altair Santin. Implementação de políticas $\text{UCON}_{\text{ABC}}$ em um núcleo de sistema operacional. In $7^{th}$ *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG)*, 2007.

[48] V. Tsaoussidis and A. Deka. Resource control of distributed applications in heterogeneous environments. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pages 380–383, 1998.

[49] Rik van Riel. Fair scheduler patch. webpage http://www.surriel.com/patches/2.4/2.4.19-fairsched, 2002. acesso em 25 de março de 2006.

[50] Alexander von Gernler. OpenBSD. webpage http://openbsd.org, 2007. acesso em 25 de junho de 2007.

[51] Horst F. Wedde and Mario Lischka. Modular authorization and administration. *ACM Transactions on Information and System Security*, 7(3):363–391, 2004.

[52] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: a formal approach. In *IEEE Symposium on Research in Security and Privacy*, pages 33–51, 1992.

[53] Thomas Y. C. Woo and Simon S. Lam. Designing a distributed authorization service. In *IEEE INFOCOM*, pages 419–429, 1998.

[54] Victor Yodaiken. Resource control in RT-Linux. In Kevin Jeffay, editor, *IEEE Real-Time Systems Symposium: Workshop on Resource Allocation Problems in Multimedia Systems*, December 1996.

[55] Giorgio Zanin and Luigi Vincenzo Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145, New York, NY, USA, 2004. ACM.

[56] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *ACM symposium on Access control models and technologies*, pages 1–10, 2004.

# Apêndice A

# Máquina de Estados do *Policy Evaluation Engine*

Neste anexo será apresentada a máquina de estados utilizada no reconhecimento e avaliação das regras expressas em uma política. Essa máquina de estados foi gerada pelo aplicativo *Bison* [43], utilizando as regras de produção descritas na subseção 5.3.4

```
Gramática
    0 $accept: input $end
    1 input: /* vazio */
    2      | input line
    3 line: exp '\n'
    4     | COMMENT
    5     | error '\n'
    6 exp: NATT '=' values
    7    | ATT '=' values
    8    | exp "==" exp
    9    | exp '<' exp
   10    | exp '>' exp
   11    | exp "!=" exp
   12    | exp "<=" exp
   13    | exp ">=" exp
   14    | exp '&' exp
   15    | exp '|' exp
   16    | '(' exp ')'
   17    | "size" exp
   18    | values
   19 values: attmath
   20       | condition
   21       | obligation
   22 attmath: NUM
   23        | SET
   24        | ATT
   25        | attmath '+' attmath
   26        | attmath '-' attmath
   27        | attmath '/' attmath
   28        | attmath '*' attmath
   29 condition: "c$cpu_used"
   30          | "c$free_mem"
   31          | "c$free_disk"
   32 obligation: "o$slot"
Terminais, com as regras onde eles aparecem
$end (0) 0
```

```
'\n' (10) 3 5
'&' (38) 14
'(' (40) 16
')' (41) 16
'*' (42) 28
'+' (43) 25
'-' (45) 26
'/' (47) 27
'<' (60) 9
'=' (61) 6 7
'>' (62) 10
'|' (124) 15
error (256) 5
NUM (258) 22
SET (259) 23
ATT (260) 7 24
NATT (261) 6
COMMENT (262) 4
"==" (263) 8
"!=" (264) 11
"<=" (265) 12
">=" (266) 13
"size" (267) 17
"c$cpu_used" (268) 29
"c$free_mem" (269) 30
"c$free_disk" (270) 31
"o$slot" (271) 32
Não-terminais com as regras onde eles aparecem
$accept (29)
    à esquerda: 0
input (30)
    à esquerda: 1 2, à direita: 0 2
line (31)
    à esquerda: 3 4 5, à direita: 2
exp (32)
    à esquerda: 6 7 8 9 10 11 12 13 14 15 16 17 18,
    à direita: 3 8 9 10 11 12 13 14 15 16 17
values (33)
    à esquerda: 19 20 21, à direita: 6 7 18
attmath (34)
    à esquerda: 22 23 24 25 26 27 28, à direita: 19 25 26 27 28
condition (35)
    à esquerda: 29 30 31, à direita: 20
obligation (36)
    à esquerda: 32, à direita: 21
estado 0
    0 $accept: . input $end
    $padrão  reduzir usando a regra 1 (input)
    input  ir ao estado 1
estado 1
    0 $accept: input . $end
    2 input: input . line
    $end           deslocar, e ir ao estado 2
    error          deslocar, e ir ao estado 3
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    COMMENT        deslocar, e ir ao estado 8
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    line        ir ao estado 15
    exp         ir ao estado 16
    values      ir ao estado 17
    attmath     ir ao estado 18
```

```
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 2
    0 $accept: input $end .
    $padrão  aceitar
estado 3
    5 line: error . '\n'
    '\n'  deslocar, e ir ao estado 21
estado 4
   22 attmath: NUM .
    $padrão  reduzir usando a regra 22 (attmath)
estado 5
   23 attmath: SET .
    $padrão  reduzir usando a regra 23 (attmath)
estado 6
    7 exp: ATT . '=' values
   24 attmath: ATT .
    '='  deslocar, e ir ao estado 22
    $padrão  reduzir usando a regra 24 (attmath)
estado 7
    6 exp: NATT . '=' values
    '='  deslocar, e ir ao estado 23
estado 8
    4 line: COMMENT .
    $padrão  reduzir usando a regra 4 (line)
estado 9
   17 exp: "size" . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 24
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 10
   16 exp: '(' . exp ')'
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 25
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 11
   29 condition: "c$cpu_used" .
    $padrão  reduzir usando a regra 29 (condition)
estado 12
   30 condition: "c$free_mem" .
    $padrão  reduzir usando a regra 30 (condition)
estado 13
   31 condition: "c$free_disk" .
    $padrão  reduzir usando a regra 31 (condition)
estado 14
   32 obligation: "o$slot" .
```

```
    $padrão  reduzir usando a regra 32 (obligation)
estado 15
    2 input: input line .
    $padrão  reduzir usando a regra 2 (input)
estado 16
    3 line: exp . '\n'
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    "=="  deslocar, e ir ao estado 26
    '<'   deslocar, e ir ao estado 27
    '>'   deslocar, e ir ao estado 28
    "!="  deslocar, e ir ao estado 29
    "<="  deslocar, e ir ao estado 30
    ">="  deslocar, e ir ao estado 31
    '&'   deslocar, e ir ao estado 32
    '|'   deslocar, e ir ao estado 33
    '\n'  deslocar, e ir ao estado 34
estado 17
   18 exp: values .
    $padrão  reduzir usando a regra 18 (exp)
estado 18
   19 values: attmath .
   25 attmath: attmath . '+' attmath
   26        | attmath . '-' attmath
   27        | attmath . '/' attmath
   28        | attmath . '*' attmath
    '-'  deslocar, e ir ao estado 35
    '+'  deslocar, e ir ao estado 36
    '*'  deslocar, e ir ao estado 37
    '/'  deslocar, e ir ao estado 38
    $padrão  reduzir usando a regra 19 (values)
estado 19
   20 values: condition .
    $padrão  reduzir usando a regra 20 (values)
estado 20
   21 values: obligation .
    $padrão  reduzir usando a regra 21 (values)
estado 21
    5 line: error '\n' .
    $padrão  reduzir usando a regra 5 (line)
estado 22
    7 exp: ATT '=' . values
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 39
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    values      ir ao estado 40
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 23
    6 exp: NATT '=' . values
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 39
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
```

```
    values      ir ao estado 41
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 24
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
   17    | "size" exp .
    $padrão  reduzir usando a regra 17 (exp)
estado 25
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
   16    | '(' exp . ')'
    "=="  deslocar, e ir ao estado 26
    '<'   deslocar, e ir ao estado 27
    '>'   deslocar, e ir ao estado 28
    "!="  deslocar, e ir ao estado 29
    "<="  deslocar, e ir ao estado 30
    ">="  deslocar, e ir ao estado 31
    '&'   deslocar, e ir ao estado 32
    '|'   deslocar, e ir ao estado 33
    ')'   deslocar, e ir ao estado 42
estado 26
    8 exp: exp "==" . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 43
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 27
    9 exp: exp '<' . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 44
    values      ir ao estado 17
```

```
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 28
   10 exp: exp '>' . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 45
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 29
   11 exp: exp "!=" . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 46
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 30
   12 exp: exp "<=" . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 47
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 31
   13 exp: exp ">=" . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 48
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
```

```
    obligation  ir ao estado 20
estado 32
   14 exp: exp '&' . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 49
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 33
   15 exp: exp '|' . exp
    NUM            deslocar, e ir ao estado 4
    SET            deslocar, e ir ao estado 5
    ATT            deslocar, e ir ao estado 6
    NATT           deslocar, e ir ao estado 7
    "size"         deslocar, e ir ao estado 9
    '('            deslocar, e ir ao estado 10
    "c$cpu_used"   deslocar, e ir ao estado 11
    "c$free_mem"   deslocar, e ir ao estado 12
    "c$free_disk"  deslocar, e ir ao estado 13
    "o$slot"       deslocar, e ir ao estado 14
    exp         ir ao estado 50
    values      ir ao estado 17
    attmath     ir ao estado 18
    condition   ir ao estado 19
    obligation  ir ao estado 20
estado 34
    3 line: exp '\n' .
    $padrão  reduzir usando a regra 3 (line)
estado 35
   26 attmath: attmath '-' . attmath
    NUM  deslocar, e ir ao estado 4
    SET  deslocar, e ir ao estado 5
    ATT  deslocar, e ir ao estado 39
    attmath  ir ao estado 51
estado 36
   25 attmath: attmath '+' . attmath
    NUM  deslocar, e ir ao estado 4
    SET  deslocar, e ir ao estado 5
    ATT  deslocar, e ir ao estado 39
    attmath  ir ao estado 52
estado 37
   28 attmath: attmath '*' . attmath
    NUM  deslocar, e ir ao estado 4
    SET  deslocar, e ir ao estado 5
    ATT  deslocar, e ir ao estado 39
    attmath  ir ao estado 53
estado 38
   27 attmath: attmath '/' . attmath
    NUM  deslocar, e ir ao estado 4
    SET  deslocar, e ir ao estado 5
    ATT  deslocar, e ir ao estado 39
    attmath  ir ao estado 54
estado 39
   24 attmath: ATT .
    $padrão  reduzir usando a regra 24 (attmath)
estado 40
    7 exp: ATT '=' values .
    $padrão  reduzir usando a regra 7 (exp)
estado 41
    6 exp: NATT '=' values .
    $padrão  reduzir usando a regra 6 (exp)
```

```
estado 42
   16 exp: '(' exp ')' .
    $padrão  reduzir usando a regra 16 (exp)
estado 43
    8 exp: exp . "==" exp
    8    | exp "==" exp .
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 8 (exp)
estado 44
    8 exp: exp . "==" exp
    9    | exp . '<' exp
    9    | exp '<' exp .
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 9 (exp)
estado 45
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   10    | exp '>' exp .
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 10 (exp)
estado 46
    8 exp: exp . "==" exp
    9    | exp . '<' exp
```

```
   10    | exp . '>' exp
   11    | exp . "!=" exp
   11    | exp "!=" exp .
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 11 (exp)
estado 47
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   12    | exp "<=" exp .
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 12 (exp)
estado 48
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   13    | exp ">=" exp .
   14    | exp . '&' exp
   15    | exp . '|' exp
    '&'  deslocar, e ir ao estado 32
    '|'  deslocar, e ir ao estado 33
    "=="  erro (não associativo)
    '<'   erro (não associativo)
    '>'   erro (não associativo)
    "!="  erro (não associativo)
    "<="  erro (não associativo)
    ">="  erro (não associativo)
    $padrão  reduzir usando a regra 13 (exp)
estado 49
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
```

```
   14    | exp '&' exp .
   15    | exp . '|' exp
    $padrão  reduzir usando a regra 14 (exp)
estado 50
    8 exp: exp . "==" exp
    9    | exp . '<' exp
   10    | exp . '>' exp
   11    | exp . "!=" exp
   12    | exp . "<=" exp
   13    | exp . ">=" exp
   14    | exp . '&' exp
   15    | exp . '|' exp
   15    | exp '|' exp .
    $padrão  reduzir usando a regra 15 (exp)
estado 51
   25 attmath: attmath . '+' attmath
   26        | attmath . '-' attmath
   26        | attmath '-' attmath .
   27        | attmath . '/' attmath
   28        | attmath . '*' attmath
    '*'  deslocar, e ir ao estado 37
    '/'  deslocar, e ir ao estado 38
    $padrão  reduzir usando a regra 26 (attmath)
estado 52
   25 attmath: attmath . '+' attmath
   25        | attmath '+' attmath .
   26        | attmath . '-' attmath
   27        | attmath . '/' attmath
   28        | attmath . '*' attmath
    '*'  deslocar, e ir ao estado 37
    '/'  deslocar, e ir ao estado 38
    $padrão  reduzir usando a regra 25 (attmath)
estado 53
   25 attmath: attmath . '+' attmath
   26        | attmath . '-' attmath
   27        | attmath . '/' attmath
   28        | attmath . '*' attmath
   28        | attmath '*' attmath .
    $padrão  reduzir usando a regra 28 (attmath)
estado 54
   25 attmath: attmath . '+' attmath
   26        | attmath . '-' attmath
   27        | attmath . '/' attmath
   27        | attmath '/' attmath .
   28        | attmath . '*' attmath
    $padrão  reduzir usando a regra 27 (attmath)
```