Engenharia de Domínio aplicada ao Desenvolvimento Robusto e Eficiente de Sistemas Operacionais

Luciano Porto Barreto

Laboratório de Sistemas Distribuídos - LaSiD
Departamento de Ciência da Computação - DCC
Universidade Federal da Bahia - UFBA
Av. Adhemar de Barros, S/N, Prédio do CPD, Campus de Ondina
Salvador-BA, CEP 40.170-110

lportoba@ufba.br

Abstract. *Software engineering techniques have been modestly applied to simplify the construction of operating systems. As a result, OS development is still restricted to experts and considered as a complex and error-prone task. In this paper, we describe a domain engineering approach applied to the development of a framework dedicated to the implementation of process schedulers. This framework, named Bossa, has been implemented in the Linux kernel and extensively used to develop and test new scheduling policies. We present the steps of the domain engineering process and discuss the lessons learned during the development of the framework.*
Keywords: *Domain Analysis, Domain-specific Architectures, Operating Systems.*

Resumo. *As técnicas de engenharia de software têm sido modestamente aplicadas para simplificar a construção de sistemas operacionais. Como resultado, o desenvolvimento de sistemas operacionais permanece restrito a especialistas e é considerado como tarefa complexa e propensa a erros. Neste artigo, descrevemos uma abordagem de engenharia de domínio aplicada ao desenvolvimento de um framework dedicado à implementação de escalonadores de processos. Esse framework, chamado Bossa, foi implementado no núcleo do Linux e extensivamente utilizado no desenvolvimento e teste de políticas de escalonamento. São apresentados os passos realizados no processo de engenharia de domínio e discutidos aspectos relacionados ao aprendizado adquirido durante o desenvolvimento desse framework.*
Palavras-chave: *Análise de Domínio, Arquiteturas de Domínio Específico, Sistemas Operacionais.*

1 Introdução

O desenvolvimento de componentes de um sistema operacional, em geral, e do escalonador de processos, em particular, é uma tarefa laboriosa e restrita somente aos especialistas em sistemas operacionais. Em primeiro lugar, é necessário o conhecimento preciso da estrutura e do funcionamento do núcleo do sistema operacional. O desenvolvimento de um escalonador requer a escrita de código de baixo nível, geralmente em C ou Assembly, o que restringe as possibilidades de verificação e favorece a introdução de erros. Infelizmente, um erro de programação num escalonador induz quase sempre a uma falha geral do sistema, cuja causa é de difícil identificação. Por razões de desempenho, o código dos escalonadores é extremamente otimizado explorando,

inclusive, especificidades da arquitetura do computador. Esse alto nível de otimização torna difícil a compreensão, manutenção e teste de um escalonador. Por fim, o código relacionado ao escalonamento de processos não é modular, pois encontra-se disperso em diferentes partes do sistema operacional, a exemplo das chamadas de sistemas e dos tratadores de interrupção. Esses problemas constituem entraves importantes ao desenvolvimento de novas políticas de escalonamento de processos e sua experimentação em sistemas reais.

Apesar dessa constatação, raros esforços obtiveram sucesso em facilitar a construção de sistemas operacionais através de técnicas modernas de engenharia de software, tais como orientação a aspectos [8, 9], reflexão computacional [30], e linguagens de programação dedicadas [23]. Por isso, o código dos sistemas operacionais permanece complexo, difícil de estender, e pouco compreensível [6]. Considerando o exemplo do Linux, a distribuição Redhat 7.1 (núcleo 2.4) comporta aproximadamente 30 milhões de linhas de código, das quais o núcleo é o maior componente, representando aproximadamente 2.4 milhões de linhas [29]. Comparando essa distribuição com a Redhat 6.2, atestamos um acréscimo de 60% no número de linhas de código. Outro exemplo relevante refere-se ao sistema operacional FreeBSD cujo código fonte dobrou em tamanho entre as versões 2 e a versão 4 [7]. Tal nível de complexidade requer especialistas para a manutenção e desenvolvimento de novas funcionalidades. Além disso, a ausência de metodologias de desenvolvimento adequadas a esse domínio aumenta significativamente a propensão a erros. De fato, estudos recentes revelam a existência de erros em certos componentes do núcleo do Linux e mostram que esses erros persistiram à evolução do sistema em diferentes versões do núcleo [12]. Estes problemas corroboram a necessidade da concepção e aplicação efetiva de técnicas de engenharia de software para a melhoria da confiabilidade dos sistemas operacionais modernos.

Com o intuito de facilitar o desenvolvimento de escalonadores de processo, concebemos um framework, chamado Bossa [1, 3], que permite, através de abstrações específicas do domínio, facilitar e automatizar o desenvolvimento e a instalação de escalonadores de processos no núcleo do Linux. Este framework é composto de uma linguagem de programação dedicada e um suporte de execução que permite a instalação de políticas de escalonamento escritas nessa linguagem. Neste artigo, apresentamos as etapas do processo de engenharia de domínio realizado no desenvolvimento do framework Bossa e o aprendizado adquirido na concepção e validação desse framework.

O restante desse artigo está estruturado da seguinte forma. A seção 2 apresenta uma visão geral do framework Bossa e das etapas envolvidas na engenharia de domínio. As seções 3 e 4 descrevem a análise domínio realizada no contexto do escalonamento de processos com o intuito de identificar as abstrações fundamentais para a implementação de escalonadores. As seções 5 e 6 descrevem o projeto e a implementação do domínio, respectivamente. A seção 7 efetua uma avaliação qualitativa e quantitativa da utilização concreta do framework enquanto a seção 8 ressalta o aprendizado obtido no decorrer da sua concepção. A seção 9 realiza a comparação com trabalhos correlatos e a seção 10 conclui o artigo apresentando perspectivas de extensão desse trabalho.

2 O Framework de desenvolvimento Bossa

O framework Bossa articula-se sobre dois eixos principais: (i) um suporte de execução extensível para a integração e implementação de escalonadores e (ii) uma linguagem dedicada que facilita o desenvolvimento de políticas de escalonamento. O suporte de execução define uma máquina abstrata sobre a qual pode-se instalar e executar políticas de escalonamento no núcleo do SO. A linguagem dedicada fornece, por sua vez, abstrações específicas para a implementação de políticas de escalonamento facilitando o seu desenvolvimento e sua integração no suporte de execução.

A Figura 1 descreve o processo de desenvolvimento e os componentes da arquitetura

Bossa. O desenvolvedor escreve uma política de escalonamento utilizando as abstrações fornecidas pela linguagem dedicada. Após a análise sintática, essa política é submetida a um verificador que certifica sua correção quanto às propriedades específicas do domínio de escalonamento. Enfim, a política é traduzida em código C para ser em seguida integrada no núcleo do sistema operacional. Nas seções seguintes, são apresentados as etapas da engenharia de domínio aplicada ao longo da elaboração desse framework.

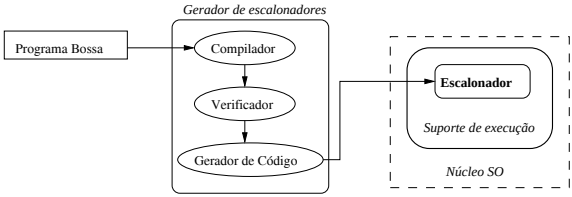

Figura 1: Desenvolvimento de escalonadores de processo em Bossa

2.1 Etapas da Engenharia de Domínio

Em primeiro lugar, é fundamental delimitar o domínio da aplicação através da definição precisa do objetivo e do escopo do framework (*i.e.*, o que desejamos programar). Assim, a escolha de um domínio muito restrito reduz o espectro de utilização da framework, pois tal procedimento pode inviabilizar a implementação de políticas importantes. Por outro lado, um domínio de aplicação muito abrangente aumenta a complexidade da análise de domínio e de desenvolvimento do framework correspondente, já que, nessa situação, o framework deve considerar um número volumoso de casos. Além disso, ao passo que aumentamos o domínio de aplicação, um framework específico tende a se aproximar de um framework de uso geral, desfavorecendo a concisão e a verificação de propriedades. Existe, portanto, um compromisso a ser equacionado no que concerne a amplitude do domínio e as funcionalidades a serem providas por um framework dedicado.

Uma vez que o domínio esteja adequadamente delimitado, o processo de engenharia de domínio se decompõe em três etapas principais: análise, projeto e implementação do domínio. A análise de domínio visa identificar os pontos comuns e as variações entre os programas do domínio. A fase de projeto define as abstrações de software (*e.g.*, tipos de dados, operadores, funções) para a implementação de políticas. A etapa de implementação tem por objetivo avaliar as diferentes possibilidades de desenvolvimento em função do sistema alvo e do ambiente de execução escolhido. Detalhamos essas etapas nas seções a seguir.

Análise de domínio

A análise de domínio é responsável pela caracterização precisa do problema, identificação dos pontos comuns e das variações entre as políticas [11], e definição das propriedades e restrições específicas ao domínio. Para tanto, deve-se inicialmente escolher entre as políticas existentes aquelas que são representativas do problema visado. Uma forma de adquirir tal conhecimento consiste em utilizar técnicas de análise de domínio tais como *Organization Domain Modeling* [24], *Feature Oriented Domain Analysis* [15] ou *Domain-Specific Software Architectures* [26]). Outras metodologias mais específicas foram propostas para a concepção e implementação de linguagens dedicadas, como GenVoca [4] e Sprint [10].

O ponto central da análise de domínio consiste na separação entre os pontos comuns (*commonalities*) e as variações (*variabilities*) existentes nas políticas. O objetivo é identificar o

máximo de pontos comuns a fim de reutilizá-los de maneira sistemática no desenvolvimento de novas políticas. Um ponto comum define uma característica intrínseca a todos os membros de uma *família de programas*. Segundo Parnas [21], uma família de programas é um conjunto de programas que compartilham uma quantidade suficiente de características comuns a ponto de ser interessante estudá-los como um todo. Por exemplo, entre a família de escalonadores de processos baseados em prioridade, o atributo *prioridade* é um ponto comum entre todos os membros. Uma variação determina, por sua vez, as diferenças entre os membros de uma mesma família de programas. Considerando ainda o exemplo de escalonadores baseados em prioridades, os valores das prioridades podem ser considerados como uma variação entre esses escalonadores. Assim, a parametrização de uma variação caracteriza um membro específico da família de programas. Por fim, é preciso definir os valores aceitáveis para as variações.

Por vezes, é difícil estabelecer a fronteira entre um ponto comum e uma variação, pois isso depende da granularidade/escopo do framework. Por exemplo, a medida em que alargamos o domínio estudado, os pontos comuns tendem a desaparecer. Assim, no exemplo de escalonadores baseados em prioridade, o atributo prioridade provavelmente deixaria de ser um ponto comum caso quiséssemos também programar escalonadores baseados em tempo compartilhado (*round robin*), por exemplo.

Projeto do domínio

A etapa de projeto do domínio refere-se à especificação da linguagem dedicada através da definição do conjunto de abstrações e operadores oferecidos pela linguagem através de uma sintaxe concreta e uma semântica. É bastante provável que os conceitos e objetos resultantes da análise de domínio dêem origem a novas construções tais como tipos de dados, declarações e operadores. É essencial que essas abstrações sejam intuitivas aos programadores do domínio, fáceis de serem utilizadas, coerentes e passíveis de verificação. A definição da semântica da linguagem é importante para formalizar as relações entre as restrições existentes entre diferentes construções da linguagem de modo a assegurar a correção dos programas gerados. As técnicas de verificação podem ser integradas no compilador/interpretador da linguagem ou realizadas por ferramentas externas como os verificadores de modelos e os geradores de cenários.

Implementação do domínio

A última fase do processo de desenvolvimento corresponde à implementação da linguagem dedicada e de seu sistema de execução. De fato, essa implementação estrutura-se sobre diversas escolhas de concepção normalmente associadas às restrições do ambiente. Por exemplo, o custo de execução associado à interpretação de programas desfavorece a utilização dessa abordagem para sistemas com exigências severas quanto ao desempenho. Em contrapartida, os sistemas baseados em compilação limitam as possibilidades de substituição dinâmica de políticas. Uma terceira via passa pela utilização de um esquema híbrido tal como a compilação em tempo de execução (*just-in-time compilation*).

3 Análise de domí nio aplicada ao escalonamento de processos

O framework Bossa é fruto de uma análise de requisitos conduzida a partir do estudo da implementação de diversos escalonadores descritos em livros, artigos e em sistemas operacionais reais tais como Linux, Windows NT e BSD. O objetivo desse estudo reside na identificação dos componentes essenciais de uma política de escalonamento visando a definição de abstrações adequadas ao domínio.

As tarefas fundamentais de um escalonador de processos consistem em criar, suspender, finalizar, escolher e despachar os diferentes processos sob sua supervisão. Para tanto,

a especificação de um escalonador deve definir: os atributos associados a cada processo (*e.g.*, prioridade, *deadline*), os critérios empregados na seleção de um processo (*e.g.*, maior prioridade, *deadline* mais próximo), as circunstâncias nas quais essa seleção ocorre (*i.e.*, pontos de escalonamento) e a interface disponível às aplicações e aos usuários. Pode-se resumir os principais aspectos relativos à especificação e à implementação de um escalonador de processos através das seguintes funcionalidades:

Gestão de atributos e de filas de espera de processos. Durante a vida útil de um processo, o escalonador mantém um conjunto de informações relativas a sua execução. Essas informações são armazenadas sob a forma de atributos tais como a prioridade e a periodicidade de execução de um processo. Em geral, esses atributos são atualizados por diferentes mecanismos do sistema operacional (*e.g.*, no bloqueio ou desbloqueio de um processo ou a cada interrupção de relógio) e influenciam diretamente a escolha do novo processo a ser executado. Um *estado* é também associado a um processo e representa seu status atual de execução (*e.g.*, em execução, pronto a executar, suspenso e finalizado). Tipicamente, um processo é armazenado em uma fila de espera contendo processos num mesmo estado ou tendo características similares (*e.g.*, processos prontos ou organizados em níveis de prioridade). Alguns escalonadores, a exemplo do utilizado no BSD, utilizam diversas filas de espera classificando os processos prontos por ordem de prioridade.

Especificação do critério de escalonamento. Os critérios utilizados por um escalonador para escolher um novo processo são geralmente relacionados aos atributos dos processos e às informações relativas ao estado global do sistema operacional. Esses atributos podem ser pré-definidos ou atualizados automaticamente pelo sistema operacional. Por exemplo, um escalonador que utiliza a noção de prioridade pode empregar prioridades fixas ou dinâmicas. Outros exemplos de atributos utilizados incluem o *deadline*, a latência de execução (*i.e.*, o tempo decorrido entre o desbloqueio de um processo e sua posterior reexecução), ou ainda parâmetros mais específicos como o consumo médio de energia. Pode-se também combinar diferentes atributos com o intuito de enriquecer a gama de escolhas de uma política.

Definição de pontos de escalonamento e do modelo de preempção. As ações que definem o comportamento de uma política de escalonamento são efetuadas em diferentes partes do núcleo. Esses pontos são denominados *pontos de escalonamento*. As ações típicas de um escalonador consistem na atualização de atributos de um ou diversos processos ou na adição ou remoção de um processo de uma fila de espera (*e.g.*, na criação de um novo processo). Conceitualmente, os pontos de escalonamento variam em função do algoritmo e do modelo de preempção adotados pela política. O modelo de preempção estabelece as circunstâncias nas quais o escalonador deve interromper o processo em execução para escolher um novo processo. Em um escalonador baseado em prioridades, a chegada de um processo de mais importante na fila de espera de processos prontos representa um exemplo típico de preempção. De fato, a chegada de tal processo pode ser provocada por diversos eventos do sistema (*e.g.*, criação de um processo ou desbloqueio de um processo em espera de um recurso do sistema). Em contrapartida, uma variante da política anterior poderia estabelecer que o processo em execução nunca pudesse ser interrompido por um processo de mais alta prioridade (neste caso, a chegada do processo seria irrelevante). Nos escalonadores de tempo compartilhado, a passagem de um tic de relógio é geralmente um ponto de escalonamento, pois a política deve verificar se o processo em execução esgotou o seu quantum de execução. Ainda que a especificação dos pontos de escalonamento dependa das características da política, alguns eventos requerem obrigatoriamente a intervenção do escalonador. Por exemplo, o fim do processo em execução exige que o escalonador escolha e despache um novo processo.

Definição de uma interface para as aplicações. Um escalonador deve ainda fornecer uma interface de comunicação para as aplicações. A interface de um escalonador é composta de um conjunto de funções que são utilizadas para inicializar, consultar ou modificar certos atributos de um processo (*e.g.*, prioridade) ou do sistema operacional. Essa interface pode ainda servir para ajustar o comportamento do escalonador em função de necessidades específicas de uma aplicação. Em suma, a interface define o nível de configurabilidade fornecido pelo escalonador. Geralmente, as funções dessa interface são implementadas através de chamadas de sistema tal como `setpriority` no caso de Linux.

4 Resultados da análise de domí nio

A partir da análise das políticas de escalonamento estudadas, identificamos os pontos comuns e as variações entre as diferentes políticas. Esses resultados permitiram a identificação dos conceitos de base relativos ao desenvolvimento de escalonadores e, em seguida, a elaboração de uma linguagem dedicada para facilitar a concepção de políticas de escalonamento.

4.1 Pontos comuns

Os pontos comuns de uma família de escalonadores agregam as características presentes em todos os escalonadores. No contexto dos escalonadores de processo estudados, foram identificados os seguintes pontos comuns:

Estado de um processo. Em relação à gestão de processos, a noção de estado é frequentemente empregada para se referir aos processos que possuem o mesmo status de execução. Identificamos quatro classes de estado básicas: *RUNNING*, *READY*, *BLOCKED*, e *TERMINATED*, que simbolizam respectivamente, um processo em execução, pronto a ser executado, bloqueado e finalizado. Na prática, essas classes de estado são representadas por objetos tais como filas de espera que efetuam o armazenamento dos processos. A partir dessas classes de estado e das peculiaridades dos escalonadores, estabelecemos um conjunto de transições de estado válidas para todo escalonador como ilustra o autômato da Figura 2.

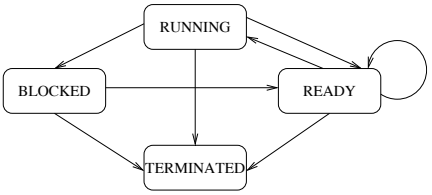

Figura 2: Autômato de mudança de estados de um processo em Bossa

Eventos do SO relativos ao escalonamento. No contexto do estudo realizado, notamos a dificuldade em modelar um escalonador como um programa sequencial, pois um escalonador deve ser invocado na ocorrência de diversos eventos do sistema. Uma maneira mais natural e compreensível consiste em descrever o funcionamento de um escalonador a partir das ações efetuadas na ocorrência dos eventos do sistema. Para tanto, identificamos tais eventos e, em seguida, escrevemos as políticas de escalonamento a partir das ações associadas aos tratadores desses eventos.

Assim, a interação entre o núcleo do SO e uma política de escalonamento se materializa através de um barramento de eventos.

Outras observações evidenciaram que certas políticas requerem a execução de ações pontuais quando da ocorrência de certos eventos (*e.g.*, aumentar a prioridade de um processo em espera de um evento de entrada-saída). Isso nos levou a organizar certos eventos de forma hierárquica como, por exemplo, os eventos de bloqueio e desbloqueio de processos. Como consequência, o desenvolvedor tanto pode definir ações *default* para todos os eventos da hierarquia (*e.g.*, todos os eventos de bloqueio) quanto especificar ações particulares para eventos específicos (*e.g.*, bloqueio devido à espera de dados do disco rígido).

Seleção e despacho de processos. Uma das tarefas fundamentais de um escalonador é a seleção e o despacho de um processo. A seleção é realizada geralmente em função dos atributos do processo tal como sua prioridade. O mecanismo de despacho é, por sua vez, específico ao sistema operacional. Atestamos que a seleção e a mudança do processo em execução é obrigatória em certas circunstâncias. Tais requisitos definem propriedades específicas do domínio que devem ser atendidas pelo framework. Por exemplo, caso o processador esteja livre e exista ao menos um processo pronto, o escalonador deve imperativamente escolher um novo processo. De maneira equivalente, quando inexistem processos prontos ou em execução, o escalonador deve relançar o processo *idle* (*loop* infinito) do sistema.

Estado de um escalonador. Ainda que certas infraestruturas utilizem os escalonadores de modo hierárquico considerando-os como processos virtuais, essa noção ainda não foi formalizada por outros autores. Verificamos que é possível tratar um escalonador como um processo convencional associando-lhe uma informação de estado (de modo similar ao estado de um processo). Para isso, definimos a noção de *estado de um escalonador* a partir do estado dos processos geridos pelo escalonador. Essa definição permite que um escalonador possa gerenciar outros escalonadores como se esses fossem processos convencionais, o que facilita a composição de políticas de escalonamento e permite a construção de infraestruturas hierárquicas de escalonamento [13]. Além disso, o estado de um escalonador é utilizado para identificar se o núcleo do SO deve efetuar a mudança do processo corrente.

4.2 Variações

As diferenças entre as políticas do domínio constituem o ponto central para identificar as variações entre as políticas. Nossa análise de domínio evidenciou as seguintes variações entre escalonadores de processo:

Atributos de processo. Existem dois tipos de atributos associados a um processo: atributos fornecidos pelo núcleo do SO (atributos de sistema) e atributos específicos à política de escalonamento. Um atributo de sistema é definido pelo núcleo como, por exemplo, o número de identificação de um processo (pid) ou o número de identificador do usuário (uid) que iniciou o processo. Um atributo de sistema corresponde a uma variação entre diferentes núcleos e não entre diferentes escalonadores porque um conjunto de políticas concebidas para um determinado núcleo deve dispor dos mesmos atributos de sistema. Por outro lado, cada política de escalonamento pode definir atributos específicos ao escalonador para atender a necessidades da estratégia de escalonamento escolhida. Exemplos de atributos específicos às políticas incluem prioridade e quantum de execução.

Critério de escalonamento. Todo escalonador classifica seus processos através de uma ordem de importância particular segundo os atributos dos processos. Essa relação de ordem é geralmente utilizada para eleger um novo processo. Além disso, identificamos que certas políticas usam essa relação de ordem em outras circunstâncias. Por exemplo, quando do desbloqueio de um processo, uma política de escalonamento baseada em prioridades geralmente compara a importância desse último com a importância do processo corrente para verificar se o processo corrente deve ser interrompido.

Ações associadas aos eventos de escalonamento. O comportamento de uma política de escalonamento depende sobretudo das ações efetuadas pelo escalonador quando do disparo dos eventos associados a seus processos. Exemplos dessas ações incluem notadamente a atualização de atributos do processo, a mudança de estado de um ou vários processos, e a preempção do processo corrente.

Definição de estados dentro das classes de estado. Ao passo que os estados de processo definidos pelo autômato da Figura 2 permitem a especificação de certas políticas, outras estratégias requerem a definição de estados adicionais a fim de poder tratar situações particulares. Por exemplo, certas políticas de escalonamento baseadas em tempo compartilhado, como a utilizada no Linux, efetuam tratamentos específicos para os processos que excederam o seu quantum de execução ou que liberaram o processador voluntariamente. Essas políticas induzem o programador a definir novos estados a fim de lidar com essa particulariedade.

Interface do escalonador. Um escalonador deve fornecer uma interface que os programadores possam utilizar para modificar ou consultar as informações relativas a execução dos processos. Para tanto, uma política deve definir um conjunto de funções que podem ser compiladas juntamente com as aplicações dos usuários do sistema.

5 Projeto do domí nio

A etapa de projeto do domínio envolveu a definição de três componentes: uma biblioteca de funções, um modelo de eventos e a sintaxe da linguagem dedicada. Em primeiro lugar, definimos uma biblioteca de funções para a implementação de escalonadores de processos escritos diretamente em C. Em seguida, definimos um modelo de eventos (máquina abstrata) que permite a execução de escalonadores dentro do núcleo do sistema operacional (descrito em [2]). O terceiro passo consistiu na definição da sintaxe da linguagem dedicada e a semântica estática e dinâmica de tal linguagem. Nesta seção descrevemos os aspectos principais dessas etapas e apresentamos trechos do código escritos na linguagem dedicada. Nosso objetivo não consiste em apresentar um tutorial sobre o desenvolvimento de políticas em Bossa, mas exemplificar como os conceitos identificados na análise de domínio foram concretizados em abstrações da linguagem dedicada fornecida pelo framework.

5.1 Definição de uma biblioteca de funções

O primeiro passo para implementação de escalonadores no Linux foi a definição de uma biblioteca de funções específicas. Essa biblioteca contém funções que gerenciam as filas de processos, definem a interface do escalonador com as aplicações e efetuam a inicialização de diversas estruturas internas do escalonador. Podemos ver essa biblioteca de funções como a interface de programação e os mecanismos de uma máquina virtual para a execução das políticas de escalonamento dentro

do núcleo do sistema operacional. Assim, podemos mapear as abstrações específicas fornecidas pelo framework para as funções definidas nessa biblioteca.

5.2 Definição de abstrações específicas ao domínio

A partir da identificação dos pontos comuns e das variações entre os escalonadores, descritos na seção 4, definimos novas abstrações que facilitam a escrita de políticas e capturam o conhecimento do domínio:

Classes de estado. Conceitualmente, as filas de espera e as variáveis são utilizadas para agrupar processos com características similares, notadamente processos com mesmo estado de execução. As implementações existentes, entretanto, não exploram essa relação. Por exemplo, por razões de eficiência, o escalonador Linux mantém o processo em execução e os processos que extinguiram seu quantum de execução na fila de processos prontos. Para reforçar essa relação, cada meio de armazenamento utilizado pelo escalonador é associado a uma classe de estado.

A partir da identificação das classes de estado (definidas na seção 4), a linguagem dedicada fornece abstrações correspondentes para a definição da política de escalonamento. As classes de estado `RUNNING`, `READY`,`BLOCKED` e `TERMINATED` representam o conjunto de processos em execução, prontos, bloqueados e finalizados, respectivamente. Tais classes de estado são instanciadas por filas de espera e variáveis. Para ilustrar essa abordagem, a Figura 3 mostra o código de uma política de escalonamento que define uma variável contendo o processo em execução (`corrente`), duas filas de espera para armazenar os processos prontos (`prontos` e `expirados`) e duas outras filas de espera servindo aos processos bloqueados (`bloqueados` e `bloqueados-disco`).

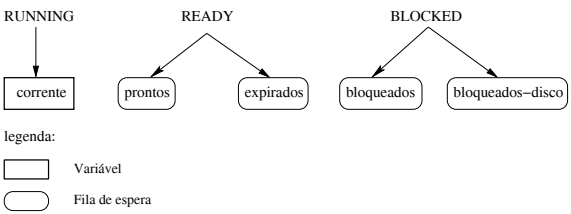

Figura 3: Utilização de classes de estado

A implementação desse exemplo na linguagem dedicada de Bossa corresponde ao código a seguir:

```
states = {
  RUNNING corrente : process
  READY prontos : select queue;
  READY expirados : queue;
  BLOCKED bloqueados, bloqueados-disco: queue;
}
```

No exemplo anterior, as classes de estado (*e.g.*, RUNNING, READY) são instanciadas como um conjunto de estados (*e.g.*, corrente, prontos), em que cada estado é implementado como uma fila de espera (`queue`) ou uma variável (`process`). O qualificador `select` identifica a fila de espera sobre a qual é feita a escolha do novo processo corrente (neste caso, a fila `prontos` é utilizada).

Declarações. Um conjunto de declarações permite especificar os atributos associados aos processos gerenciados por um escalonador assim como o critério utilizado para selecionar o processo pronto mais importante quando o escalonador deve eleger um novo processo. Os atributos de processo de um escalonador Bossa são definidos pelo tipo `process`. No trecho abaixo, típico de um escalonador *round-robin*, o escalonador associa a cada processo uma prioridade (`prioridade`) e um contador que armazena o tempo restante do quantum de execução (`ticks`).

```
process = {
  int prioridade; time ticks;
}
```

O critério de escalonamento é especificado pela declaração `ordering_criteria` que contém uma lista de atributos e para cada atributo uma ordem específica (`lowest`, `highest`). O exemplo a seguir determina que o escalonador escolhe o próximo processo na seguinte ordem: (i) o processo de mais alta prioridade e (ii) o processo que menos consumiu o seu quantum.

```
ordering_criteria = {
 highest prioridade, highest ticks
}
```

Tratadores de evento. O comportamento de um escalonador é definido por um conjunto de tratadores de evento a partir dos quais o desenvolvedor pode definir ações relativas à gestão das filas de espera ou a escolha de um novo processo. O exemplo a seguir decrementa o quantum de execução do processo corrente quando da passagem de um tic de relógio (evento `system.clocktick`). Caso o processo corrente tenha esgotado o seu quantum de execução, o seu quantum é recarregado e o processo é posto na fila de processos expirados através do operador *move* (`=>`).

```
On system.clocktick {
   corrente.ticks--;
   if (corrente.ticks <= 0) {
     corrente.ticks = 20;
     corrente => expirados;
   }
}
```

Operadores específicos. Abstrações específicas foram definidas para manipular as filas de espera e as variáveis. Por exemplo, a mudança de estado de um processo é feita através do deslocamento desse processo entre diferentes estados (representados por filas de espera e variáveis). Tal funcionalidade é realizada pelo operador *move* (`=>`), ilustrado no exemplo anterior. É importante notar que a utilização desse operador deve respeitar as transições válidas entre as classes de estado definidas pelo autômato da Figura 2.

5.3 Identificação de dependências entre as abstrações do domínio

As abstrações de Bossa fornecem os componentes básicos para a concepção de escalonadores. Entretanto, é igualmente importante identificar as relações de dependência entre essas abstrações a fim de verificar a coerência da política verificando se as abstrações são corretamente utilizadas. A Figura 4 apresenta as relações de dependência relativas às classes de estado, aos atributos de processo e às transições de estado codificadas nos tratadores de evento.

A noção de classe de estado é essencial, pois define o status do processo que pode ser armazenado nas filas de espera e nas variáveis, assim como serve para efetuar o cálculo do estado

do escalonador (fundamental para o funcionamento do suporte de execução). Os atributos do processo são combinados para definir o critério de escalonamento. Isso permite direcionar de forma automatizada e eficiente a implementação das funções que gerenciam as filas de espera, a exemplo da função que seleciona o processo mais importante. Enfim, nos tratadores de evento, as ações relativas à gestão das filas de espera e das variáveis são restringidas pelas transições permitidas entre as classes de estado e as transições específicas a cada evento. Essas últimas são especificadas por definições chamadas de *tipos de evento* [17]. Assim, é possível verificar transições incoerentes numa política de escalonamento (*e.g.*, não despachar um processo quando necessário).

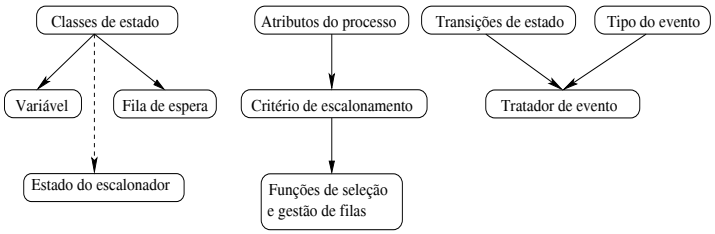

Figura 4: Dependências entre as abstrações de Bossa

6 Implementação do domí nio

O suporte de execução do framework Bossa foi implementado no núcleo do Linux a fim de permitir a instalação de uma política gerada pelo compilador Bossa. Esta implementação demandou várias modificações ao núcleo original do Linux. Por exemplo, diversos componentes do sistema tais como gerenciadores de dispositivos, chamadas de sistema e sistemas de arquivos foram modificados para notificar eventos ao suporte de execução. A título de informação, a versão 2.4 do núcleo Bossa/Linux possui 294 notificações de eventos espalhadas em 99 arquivos, o que evidencia a disseminação de código relativo ao escalonamento no núcleo do sistema. O funcionamento do suporte de execução do núcleo Bossa/Linux é descrito em detalhes em [2].

7 Avaliação

Nesta seção, é apresentada uma avaliação da experiência de utilização do framework Bossa mediante análise de aspectos qualitativos e quantitativos, que atestam a vialibidade na utilização concreta do framework. Em seguida, são relatados aspectos relativos ao aprendizado adquirido na concepção do framework.

7.1 Modularidade e facilidade de desenvolvimento

O framework Bossa fornece construções de alto nível que facilitam a implementação de políticas de escalonamento abstraindo os detalhes do sistema operacional (*e.g.*, estruturas de dados do núcleo, mecanismos de sincronização, comutação de processos). Assim, toda a complexidade relativa aos detalhes de programação do sistema operacional é capturada pelas abstrações da linguagem. Essa abordagem torna a programação modular, pois o comportamento de uma política de escalonamento encontra-se encapsulada em um único arquivo, o que contrasta com código dos

escalonadores convencionais, disseminados em diversas partes do núcleo. A facilidade de desenvolvimento e a proximidade das abstrações fornecidas em relação ao domínio proporciona o uso do framework Bossa como ferramenta de apoio no ensino prático de sistemas operacionais, adotado nos cursos de computação da UFBA e da École des Mines de Nantes, França.

7.2 Robustez no desenvolvimento e verificação de propriedades

Visto que o escalonador de processo é um componente crítico ao funcionamento do sistema operacional, é importante verificar a correção de políticas de escalonamento antes da sua utilização. A linguagem dedicada permite a verificação de propriedades específicas ao domínio, em geral, em tempo de compilação. Isso permite a descoberta prematura de erros no processo desenvolvimento, garantindo a correção de um escalonador antes da sua instalação no núcleo do sistema operacional.

Em Bossa, as propriedades são garantidas através de restrições sintáticas na linguagem ou através de técnicas de verificação, o que possibilita detectar incoerências e evitar situações nocivas de execução. Por exemplo, em Bossa, é impossível efetuar o reescalonamento de um processo bloqueado ou de um processo que já esteja em curso de execução. Como a linguagem dedicada não fornece laços infinitos nem funções recursivas, é possível garantir o término dos tratadores de evento de uma política de escalonamento e, por conseguinte, da execução do escalonador. A separação entre a política e os mecanismos de escalonamento garante a proteção dos dados do núcleo do sistema operacional. Por exemplo, diversas informações do núcleo sobre o contexto de execução de um processo (*e.g.*, lista de arquivos abertos) somente podem ser consultadas, nunca modificadas, por um escalonador escrito em Bossa.

A identificação e verificação de propriedades são inerentes ao desenvolvimento de frameworks dedicados como Bossa, já que nestes a preocupação com a robustez evidencia-se naturalmente desde a fase de projeto. Portanto, é fundamental que a análise de domínio seja orientada à descoberta das propriedades relevantes ao domínio.

7.3 Expressividade

A experiência prática na utilização do framework demonstra que o mesmo proporciona a especificação de diversos tipos de políticas de escalonamento. Alguns dos exemplos de escalonadores implementados incluem políticas voltadas aos sistemas de tempo-real (*e.g.*, *Earliest Deadline First* [19]), de tempo compartilhado (*e.g.*, Linux, baseadas na divisão proporcional de tempo (*e.g.*, *Stride* [28]), bem como extensões baseadas na noção de progresso de uma aplicação [25] e a construção de infraestruturas de escalonamento hierárquicas [13] através das quais pode-se fornecer serviços de escalonamento diferenciados à classes de aplicação específicas.

Tal diversidade de implementações atesta a viabilidade do uso efetivo do framework e a adequação das abstrações fornecidas, principalmente, considerando-se que grande parte da validação experimental de novas políticas de escalonamento é geralmente realizada através de simulações, em parte, devido à carência de ferramentas de desenvolvimento especializadas e especificidade e ao baixo grau de reutilização obtido das implementações *ad hoc* existentes.

7.4 Desempenho adequado

O framework Bossa foi validado no contexto de aplicações reais, a exemplo de aplicações de cálculo, interativas e multimídia [1]. A avaliação do desempenho dos escalonadores escritos em Bossa e do suporte de execução demonstra custo de execução aceitável (inferior a 3%) ou equivalente ao desempenho dos escalonadores originais [18]. Uma das vantagens da utilização da plataforma Bossa é que além de poder escolher entre diferentes políticas de escalonamento, os desenvolvedores podem adaptar as políticas de escalonamento fornecidas pelo sistema operacional no intuito de melhorar o desempenho de aplicações específicas como, por exemplo, na aplicação de vídeo `MPlayer` [1].

8 Aprendizado adquirido

Um dos grandes desafios na implementação do framework Bossa consistiu em lidar com a complexidade e tamanho do sistema operacional utilizado de forma a adaptá-lo ao modelo proposto pelo framework. A seguir destacamos aspectos importantes aprendidos ao longo do desenvolvimento do framework Bossa.

8.1 Adaptação da implementação do modelo de eventos para o Linux

Quando da implementação de um framework dedicado ao desenvolvimento de sistemas operacionais, é importante verificar se o modelo téorico validado na prancheta pode ser implementado literalmente nos componentes do sistema operacional escolhido. Os testes de execução das políticas de escalonamento Bossa mostraram que algumas das hipóteses relativas aos eventos estavam incorretas, levando o sistema ao colapso devido a tais situações incoerentes. Por exemplo, durante um evento de bloqueio, o processo a ser bloqueado estava por vezes na fila de processos prontos e não na variável que armazenava o processo corrente. Tal ocorrência provocava erros de execução assim que o tratador tentava colocar esse processo na fila de processos bloqueados. Tais erros de execução eram decorrentes da interferência entre as execuções dos tratadores de evento (*i.e.*, uma seqüência de tratamentos de eventos antes da intervenção do suporte de execução). A identificação e correção de tal anomalia demandou esforço considerável, sendo resolvida através da inserção de código adicional na política para tratar essas situações.

8.2 Importância da concisão das políticas

Uma maneira de melhorar a concisão do código a ser escrito em um framework dedicado consiste em suprimir procedimentos desnecessários (em geral, identificados na análise de domínio como pontos comuns) e em abstrair ao máximo os detalhes de implementação, tornando a linguagem a mais declarativa possível. Por exemplo, o cálculo do estado do escalonador presente na versão original da linguagem foi suprimido e passou a ser gerado automaticamente pelo compilador. Se por um lado, limitar a expressividade da linguagem pode restringir a abrangência da linguagem, por outro lado, isso pode favorecer estratégias de otimização do código gerado. Por exemplo, em Bossa, a função de seleção do processo mais importante (*select*), bem como as funções de comparação entre processos são geradas automaticamente pelo compilador com base no critério de escalonamento definido pela política Bossa.

8.3 Oportunidades de otimização

É importante avaliar detalhadamente as características e relações entre as abstrações fornecidas pelo framework de modo a identificar possibilidades de otimização. Por exemplo, para um tratador de evento em Bossa, o cálculo do estado do escalonador pode ser simplificado em função das filas de espera utilizadas. Tendo em vista que a gestão das filas de espera é realizada de forma implícita, alguns aspectos podem ser otimizados. Por exemplo, a fila de processos prontos (constantemente consultada para escolher um novo processo) pode ser total ou parcialmente ordenada em função das características dos atributos utilizados na definição do critério de escalonamento (atributos estáticos ou dinâmicos). Além disso, a escolha dos mecanismos de mais baixo nível pode ser feita considerando-se especificidades da máquina alvo.

9 Trabalhos correlatos

Parnas foi um dos pioneiros quanto a estruturação de um sistema operacional baseado no conceito de família de programas, considerando como domínio de aplicação o sistema de memória

virtual [22]. A utilização de linguagens de programação e frameworks dedicados têm sido igualmente importantes no provimento de novas abstrações de software e aumento da utilização efetiva de técnicas de verificação. Alguns exemplos de linguagens dedicadas incluem GAL [27], Devil [23] e ESP [16].

Os sistemas operacionais ditos *extensíveis*, tais como Exokernel [14] e SPIN [5], permitem aos programadores de sistema estender ou substituir as funcionalidades fornecidas pelo núcleo a fim de melhorar o desempenho das aplicações. Embora esses sistemas tenham demonstrado na prática como melhorar o desempenho de certas aplicações, sua utilização continua restrita a especialistas em sistemas operacionais. A programação de extensões em SPIN é feita por meio da linguagem Modula-2 que, apesar de facilitar o desenvolvimento e a verificação de certas politícas do núcleo, não fornece abstrações específicas no tocante ao escalonamento de processos. Além disso, existem restrições quanto à substituição do escalonador do núcleo.

Algumas abordagens inovadoras utilizam orientação a aspectos (*Aspect-Oriented Programming* ou AOP) na concepção de componentes ou na própria estruturação do SO. Coady *et al.* [8, 9] investigaram o uso de AOP na melhora da modularidade do sistema de arquivos do BSD. Outro exemplo da aplicação prática de AOP foi conduzida no desenvolvimento de tratadores de interrupção no sistema operacional PURE [20]. De fato, o framework Bossa pode ser considerado como uma abordagem simplificada da programação orientada a aspectos [3], pois os eventos representam pontos nos quais novos comportamentos podem ser introduzidos. A diferença fundamental entre Bossa e as abordagens tradicionais de AOP é que, em Bossa, o código dos tratadores de evento são passíveis de verificações específicas ao domínio.

10 Conclusão

A complexidade dos sistemas operacionais modernos e a crescente exigência por sistemas cada vez mais robustos e livres de erros são aspectos propulsores no uso de técnicas de engenharia de software nesse domínio. Neste artigo, descrevemos a experiência no desenvolvimento do framework Bossa que permite a construção de políticas de escalonamento de forma simplificada, robusta e eficiente através de uma engenharia de domínio.

O framework Bossa fornece ao programador um conjunto abstrações específicas ao domínio do escalonamento de processos que facilitam o desenvolvimento, teste e manutenção de políticas nesse contexto. A especificação de um escalonador em Bossa é modular, concisa e dispensa conhecimento aprofundado do funcionamento do sistema operacional. Dessa forma, o programador pode se concentrar no desenvolvimento de políticas ao invés de preocupar-se com detalhes de implementação. Além disso, o compilador Bossa verifica que uma política de escalonamento atende a um conjunto de regras de coêrencia, o que assegura um nível mínimo de segurança ao escalonador antes de sua integração no núcleo do sistema operacional.

A diversidade de políticas de escalonamento existentes abre várias possibilidades de extensão desse trabalho. Nesse sentido, alguns trabalhos estão sendo conduzidos quanto à implementação de políticas voltadas à sistemas multiprocessados e sistemas de tempo-real embarcados.

Disponibilidade A distribuição do framework Bossa e exemplos de políticas de escalonamento encontram-se publicamente disponíveis em `http://www.emn.fr/x-info/bossa`

Agradecimentos A realização deste trabalho conta com o apoio parcial do CNPq e da FAPESB (Projeto SEDRE).

Referências

[1] L. P. Barreto. *Conception aisée et robuste d'ordonnanceurs de processus au moyen d'un langage dédié*. Thèse de doctorat, Université de Rennes 1, France, June 2003.

[2] L. P. Barreto. Uma Arquitetura Baseada em Eventos para Desenvolvimento de Políticas de Escalonamento de Processos. In *I Workshop de Sistemas Operacionais (WSO'2004)*, July 2004.

[3] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. In *Proceedings of the 1st Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6, April 2002.

[4] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, 2002.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–284, December 1997.

[6] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 555–563, May 1999.

[7] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD´2003)*, pages 50–59, March 2003.

[8] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Aspect-oriented operating systems. *Communications of the ACM*, 44(10), October 2001.

[9] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, September 2001.

[10] C. Consel and R. Marlet. Architecturing software using A methodology for language development. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming (PLIP'98)*, Pisa, Italy, September 1998.

[11] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November/December 1998.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'2001)*, pages 57–72, Banff, Canada, October 2001.

[13] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 107–121, Berkeley, CA, USA, October 1996.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 52–65, October 1997.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[16] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: a language for programmable devices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2001)*, pages 309–320, Snowbird, UT, USA, June 2001.

[17] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW'2002)*, pages 54–61, Saint-Emillion, France, September 2002.

[18] J.L. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 80–91, August 2004.

[19] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[20] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 19–24, March 2005.

[21] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[22] D. L. Parnas, G. Handzel, and H. Würges. Design and Specification of the Minimal Subset of an Operating System Family. *IEEE Transactions on Software Engineering*, SE-2(4):301–307, 1976.

[23] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL Approach to Improve Productivity and Safety in Device Drivers Development. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'98)*, pages 101–109, Grenoble, France, September 2000.

[24] M. Simos. Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 196–205, April 1995.

[25] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, February 1999.

[26] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–38, December 1995.

[27] S. Thibault, R. Marlet, and C. Consel. Domain-Specific Languages: from Design to Implementation – Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.

[28] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-share Resource Management*. PhD Thesis, Massachusetts Institute of Technology, September 1995.

[29] D. A. Wheeler. More Than a Gigabuck: Estimating GNU/Linux's Size, jul 2002. http://www.dwheeler.com/sloc. Version 1.07.

[30] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages and Aplications (OOPSLA´92)*, pages 414–434, Vancouver, Canada, October 1992.