UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Uma ferramenta de gestão de conhecimento de configuração de componentes de software segundo a metodologia de Projeto de Sistemas Orientados à Aplicação

Gustavo Fortes Tondello

Florianópolis
2004/1

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Curso de Bacharelado em Sistemas de Informação

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Sistemas de Informação.

Título: Uma ferramenta de gestão de conhecimento de configuração de componentes de software segundo a metodologia de Projeto de Sistemas Orientados à Aplicação

Autor: Gustavo Fortes Tondello

Orientador: Prof. Dr. Antônio Augusto Fröhlich

Banca Examinadora: Prof. Dr. Ricardo Pereira e Silva
Prof. M.Sc. Charles Ivan Wust

Palavras-chave: Engenharia de Software, Projeto de Sistemas Orientados à Aplicação, Componentes de Software, Gerenciamento de Conhecimento de Configuração, EPOS.

Florianópolis, 14 de junho de 2004.

Uma ferramenta de gestão de conhecimento de configuração de componentes de software segundo a metodologia de Projeto de Sistemas Orientados à Aplicação

Gustavo Fortes Tondello

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Sistemas de Informação, e aprovado em sua forma final pela Coordenadoria do Curso de Bacharelado em Sistemas de Informação.

_____

Prof. Dr. Antônio Augusto Fröhlich

Banca Examinadora

_____

Prof. Dr. Ricardo Pereira e Silva

_____

Prof. M.Sc. Charles Ivan Wust

À Deus, que me presenteou com a oportunidade de viver.

RESUMO

Estudos anteriores demonstraram que aplicações embutidas e móveis não encontram suporte de tempo de execução adequado em sistemas operacionais de propósito geral, já que estes sistemas geralmente incorrem em overhead desnecessário que tem impacto direto sobre a performance da aplicação. Cada classe de aplicações tem seus próprios requisitos quanto ao sistema operacional, e eles devem ser atendidos individualmente.

A metodologia de Projeto de Sistemas Orientados à Aplicação visa a criação de sistemas de suporte de tempo de execução para aplicações de computação dedicada. Um sistema operacional orientado à aplicação é criado a partir da composição de selecionados componentes de software, que são adaptados para preencher os requisitos de uma aplicação alvo de forma adequada. Isto é particularmente crítico para aplicações móveis embutidas, já que estas freqüentemente devem ser executadas em plataformas com severas restrições de recursos (por exemplo, microcontroladores simples, quantidade de memória limitada, etc).

No entanto, disponibilizar para cada aplicação um sistema de suporte de tempo de execução específico, além de requerer um conjunto de componentes de software bem desenhados, também necessita de um conjunto sofisticado de ferramentas para selecionar, configurar, adaptar e compor os componentes de forma correta. Ou seja, o gerenciamento da configuração se torna crucial para alcançar a customizabilidade pretendida.

Este trabalho terá como foco o desenvolvimento da base de conhecimentos e das ferramentas que permitirão a configuração de componentes de software para construir uma versão otimizada do EPOS, um sistema operacional orientado à aplicação. Iremos apresentar um modelo de especificação que guiará a construção da base de conhecimentos, e o desenvolvimento de uma ferramenta gráfica que será utilizada para descrever uma configuração específica e compilar uma versão otimizada do EPOS.

Palavras-chave: Engenharia de Software, Projeto de Sistemas Orientados à Aplicação, Componentes de Software, Gerenciamento de Conhecimento de Configuração, EPOS.

ABSTRACT

Previous studies have demonstrated that embedded and mobile application do not find adequate run-time support on ordinary all-purpose operating systems, since these systems usually incur in unnecessary overhead that directly impact application's performance. Each class of applications has its own requirements regarding the operating system, and they must be fulfilled accordingly.

The *Application-Oriented System Design* method is targeted at the creation of run-time support systems for dedicated computing applications. An *application-oriented operating system* arises from the proper composition of selected software components that are adapted to finely fulfill the requirements of a target application. This is particularly critical for mobile embedded applications, for they must often executed on platforms with severe resource restrictions (e.g. simple microcontrollers, limited amount of memory, etc).

Nonetheless, delivering each application a tailored run-time support system, besides requiring a comprehensive set of well-designed software components, also calls for sophisticated tools to select, configure, adapt and compose those components accordingly. That is, *configuration management* becomes a crucial to achieve the announced customizability.

This work will be focused on the development of the knowledge base and the tools that will allow the configuration of software components to build an optimized version of EPOS, an application-oriented operating system. We will present a specification model that will guide the construction of the knowledge base, and the development of a graphical tool that will be used to describe a tailored configuration and to compile an optimized version of EPOS.

Keywords: Software Engineering, Application-Oriented System Design, Software Components, Configuration Knowledge Management, EPOS.

SUMÁRIO

LISTA DE ABREVIATURAS

ADL	Architectural Description Language (linguagem de descrição arquitetural)
API	Application Programming Interface (interface de programação de aplicação)
CDL	Component Description Language (linguagem de descrição de componente)
DTD	Document Type Definition (definição de tipo de documento)
EPOS	Embedded Parallel Operating System (sistema operacional embutido e paralelo)
FIRST	Research Institute for Computer Architecture and Software Engineering (Instituto de Pesquisa em Arquitetura Computacional e Engenharia de Software)
GMD	German National Research Center for Information Technology (Centro de Pesquisa Nacional Alemão em Tecnologia da Informação)
GUI	Graphical User Interface (interface com o usuário gráfica)
IDE	Integrated Development Environment (ambiente de desenvolvimento integrado)
IDL	Interface Description Language (linguagem de descrição de interface)
SWT	Standard Widget Toolkit
UML	Unified Modeling Language (linguagem de modelagem unificada)
XML	Extensible Markup Language (linguagem de marcação extensível)

LISTA DE FIGURAS

1. INTRODUÇÃO

1.1. Tema

Cada vez mais vem crescendo a necessidade de técnicas de Engenharia de Software que sejam capazes de sustentar o desenvolvimento de software reutilizável e flexível, escalável e com boa performance.

Apesar de haver um crescente desenvolvimento destas técnicas na área de software aplicativo, existe uma dificuldade maior para que este desenvolvimento ocorra na área de sistemas operacionais e software básico. Dentro deste contexto, a metodologia de Projeto de Sistemas Orientados à Aplicação [Frö01] foi proposta para permitir a criação de Sistemas Operacionais para o domínio de aplicações embutidas. Um Sistema Operacional orientado à aplicação é totalmente construído a partir de componentes de software, que podem ser configurados estaticamente, gerando uma versão otimizada do sistema para cada aplicação específica que irá utilizá-lo. Isto é possível devido ao modelo de Projeto que é gerado pela aplicação da metodologia.

Este trabalho irá contribuir no desenvolvimento da metodologia, mais especificamente na criação de uma base de conhecimento sobre os componentes disponíveis para o desenvolvimento do Sistema Operacional e no desenvolvimento de uma ferramenta que seja capaz de gerenciar esta base, permitindo a criação e a compilação de uma configuração específica de acordo com as necessidades de uma determinada aplicação.

1.2. Delimitação do Tema

O trabalho será desenvolvimento sobre o Sistema Operacional EPOS [Frö01], que foi modelado utilizando a técnica de Projeto de Sistemas Orientados à Aplicação, com o objetivo de validar a metodologia. A ferramenta desenvolvida fornecerá um ambiente gráfico para que o usuário possa refinar a configuração pré-determinada por ferramentas automáticas e tomar as últimas decisões no sentido de adequar o Sistema à aplicação, disponibilizando os arquivos necessários para compilar o Sistema com a configuração escolhida.

1.3. Objetivo Geral

Pesquisar técnicas de Engenharia de Software para o gerenciamento de conhecimento de configuração de componentes de software, desenvolvendo um modelo para a especificação da base de conhecimentos sobre os componentes e uma ferramenta que permita ao usuário refinar a configuração do Sistema Operacional EPOS, compilando uma versão otimizada para uma aplicação específica.

1.4. Objetivos Específicos

- Pesquisar técnicas de Engenharia de Software para o gerenciamento de conhecimento de configuração de componentes de software;
- pesquisar técnicas para a configuração de sistemas flexíveis projetados com a metodologia de Sistemas Orientados à Aplicação;
- refinar o modelo existente para o gerenciamento do conhecimento dos componentes de software do Sistema Operacional EPOS;
- desenvolver uma ferramenta para a configuração do Sistema Operacional EPOS pelo usuário.

1.5. Motivação

A maioria dos sistemas aplicativos e sistemas operacionais são desenvolvidos atualmente de forma genérica: todos os componentes estão presentes no sistema, mesmo que um usuário possa não precisar de vários deles. Geralmente existem formas para que o usuário possa configurar seu sistema em tempo de execução (configuração dinâmica) escolhendo quais componentes irá utilizar, ou mesmo alguma forma automática de configuração.

Este método de configuração dinâmica funciona bem em sistemas operacionais voltados para desktops. Porém, no domínio de aplicações embutidas é diferente. Devido à baixa disponibilidade de recursos e a necessidade de alta performance, o uso de configuração estática possibilitaria a geração de um sistema mais leve e, além disso, a eliminação de rotinas complexas de configuração dinâmica diminuiria a ocorrência de falhas [Frö01].

Foi neste contexto que surgiu a metodologia de Projeto de Sistemas Orientados à Aplicação e o Sistema Operacional EPOS [Frö01].

Parte da configuração do Sistema Operacional EPOS é realizada por ferramentas automáticas, que escolhem os componentes adequados de acordo com a aplicação que irá utilizá-lo. Mesmo assim, uma outra parte da configuração ainda tem que ser feita pelo usuário.

Para que o gerenciamento de configuração do sistema por um usuário seja possível, é necessária uma ferramenta que apresente as opções de forma organizada, permita selecionar a configuração desejada ou alterar a configuração gerada automaticamente, e faça a validação da mesma antes de permitir a compilação do sistema customizado.

Além da aplicação nesta área de Sistemas Operacionais embutidos, outras aplicações poderão ser encontradas para a utilização de ferramentas e metodologias semelhantes na área de software aplicativo. Se bem que nesta área será preciso ter mais cuidado com relação a mudanças de requisitos, pois a necessidade de fazer o usuário recompilar o sistema para poder utilizar uma nova função pode ser inaceitável, mesmo assim poderão ser avaliadas as vantagens que seriam possivelmente alcançadas através da configuração estática de sistemas específicos para as aplicações de determinado usuário, eliminando o overhead e a complexidade em tempo de execução que comumente existe nos sistemas que se propõem a serem genéricos. Essa tarefa requereria um cuidadoso estudo e provavelmente a proposição de uma nova metodologia ou a adaptação da metodologia de Projeto de Sistemas Orientados à Aplicação, e já não faz mais parte do escopo deste trabalho. Ficará, no entanto, como sugestão para a realização de trabalhos futuros.

1.6. Técnicas e ferramentas

Todos os modelos definidos ao longo do trabalho serão documentados e apresentados utilizando os padrões mundiais UML [BRJ99] e XML [W3C98].

A ferramenta que será desenvolvida também será documentada utilizando diagramas de classe UML [BRJ99] para a especificação técnica. O desenvolvimento será realizado utilizando linguagem Java [SUN01], em virtude de sua orientação a objetos, portabilidade e disponibilidade de componentes para desenvolvimento de interfaces gráficas.

1.7. Estrutura do Trabalho

O próximo capítulo irá situar o desenvolvimento deste trabalho dentro do Projeto EPOS, apresentando uma breve descrição da metodologia de Projeto de Sistemas Orientados à Aplicação e do Sistema Operacional EPOS, assim como a descrição das necessidades de ferramentas de configuração.

O capítulo 3 apresentará uma revisão bibliográfica sobre as pesquisas mais recentes na área de especificação de componentes de software reutilizáveis.

O capítulo 4 descreverá os novos modelos desenvolvidos para especificação de componentes no contexto de Projeto de Sistemas Orientados à Aplicação.

O capítulo 5 apresentará o modelo utilizado para descrever uma configuração do sistema EPOS gerada pelo usuário e armazenada em um arquivo XML para futura recuperação.

O capítulo 6 descreverá a implementação da ferramenta EposConfig, a ferramenta gráfica que será utilizada pelos usuários para configurar e compilar uma versão específica do EPOS.

No capítulo 7 serão reunidas as conclusões alcançadas com o desenvolvimento do trabalho e as sugestões para pesquisa futura.

2. CONTEXTO

Antes de entender o contexto em que se encontra o desenvolvimento deste trabalho, será necessário ter uma compreensão básica a respeito da metodologia de Projeto de Sistemas Orientados à Aplicação, assim como um conhecimento básico da estrutura do Sistema Operacional EPOS e de seu estágio de desenvolvimento atual. Esses conceitos básicos serão apresentados a seguir de forma bastante abrangente. As descrições completas tanto da metodologia como do sistema estão publicadas em [Frö01].

2.1. Projeto de Sistemas Orientados à Aplicação

Segundo Fröhlich, os Sistemas Operacionais têm sido construídos através de abstrações mais convenientes ao hardware do que às aplicações. Desta forma, as aplicações têm que se adaptar ao sistema operacional. Além disso, esses sistemas não conseguem acompanhar com rapidez a evolução das aplicações e da engenharia de software.

A metodologia de Projeto de Sistemas Orientados à Aplicação [Frö01] permite projetar o sistema desde o começo mantendo o foco nas aplicações que irão utilizá-lo. Desta forma, todos os componentes são definidos pensando na reutilização e, ao mesmo tempo, criando uma estrutura que permite selecionar diferentes componentes para gerar o sistema com diferentes configurações, de acordo com a necessidade específica de uma aplicação.

Um Sistema Operacional orientado à aplicação deve conter exatamente as funcionalidades que serão utilizadas pela aplicação, nem mais, nem menos. O seguinte enunciado descreve um sistema orientado a aplicação:

"*Um sistema operacional orientado à aplicação é apenas definido em relação a uma aplicação correspondente, para a qual ele implementa o suporte de tempo de execução necessário e que é disponibilizado conforme solicitado.*" [Frö01]

Este conceito pode parecer estranho na computação geral, porém no domínio de sistemas dedicados é perfeitamente possível, pois estes sistemas geralmente executam um pequeno conjunto de aplicações que é previamente conhecido.

A seguir, apresentaremos de forma resumida os principais conceitos da metodologia. Quanto à representação gráfica, foi adotado o uso de UML [BRJ99], porém com a criação de algumas extensões, que serão também apresentadas ao longo desta seção.

2.1.1. Análise e decomposição de domínio

A princípio, poder-se-ia pensar em seguir as recomendações da orientação a objetos [Mey88, RBLP91, JCJO93, Boo94] para decompor o domínio. No entanto, optou-se pela utilização do conceito de projeto baseado em Famílias [Par76], que permite modelar abstrações independentes entre si como membros da mesma Família, eliminando a necessidade de trabalhar somente com Especialização como na orientação a objetos.

Uma segunda diferença é que, na análise de variabilidade proposta pela orientação a objetos, acabamos identificando diferenças que são inerentes à abstração, mas também diferenças que são dependentes do cenário em que a abstração será inserida. Para evitar o aumento excessivo da cardinalidade das especializações, optou-se por utilizar os conceitos propostos pela programação orientada a aspectos [KLM+97]. Desta forma, as abstrações são totalmente modeladas de forma independente ao cenário, e as características dependentes são modeladas como aspectos, que podem ser aplicados às abstrações através de adaptadores de cenário.

Portanto, identificar membros de famílias e aspectos são as duas atividades principais da decomposição de domínio orientada à aplicação. Ainda existe uma terceira, que é a identificação de características configuráveis. Estas são criadas quando se quer estender a funcionalidade de todos os membros de uma família, porém deseja-se evitar a duplicação da cardinalidade da família. Os aspectos também alteram a característica de todos os membros da família, mas a diferença é que os aspectos são transparentes, e as características configuráveis não são. Isto significa que a abstração é a responsável por implementar as características configuráveis e habilitá-las ou não dependendo da configuração do sistema.

2.1.2. Famílias de abstrações independentes de cenário (*Scenario-independent Abstractions*)

As abstrações do sistema são identificadas e agrupadas em famílias, de acordo com o que têm em comum. A independência de cenário garante que essas abstrações sejam genéricas o suficiente para compor qualquer sistema para qualquer ambiente e com qualquer aplicação. Posteriormente, essas abstrações são mapeadas em componentes que serão realmente implementados.

A implementação dos membros de uma Família não fica restrita ao uso de especialização como na orientação a objetos, embora ela possa ocorrer, quando for

conveniente. Os membros poderiam, por exemplo, serem implementados como classes disponibilizadas em conjunto em um pacote através de agregação ou composição. Além disso, em algumas famílias podem existir membros mutuamente exclusivos, ou seja, apenas um dos membros poderá estar presente na configuração do sistema.

Geralmente, irá existir na família uma classe ou conjunto de classes que implementará as características comuns a todos os membros, e onde também poderá estar a implementação das características configuráveis daquela família.

A Figura 2.1 mostra a notação utilizada para representar uma Família de abstrações.

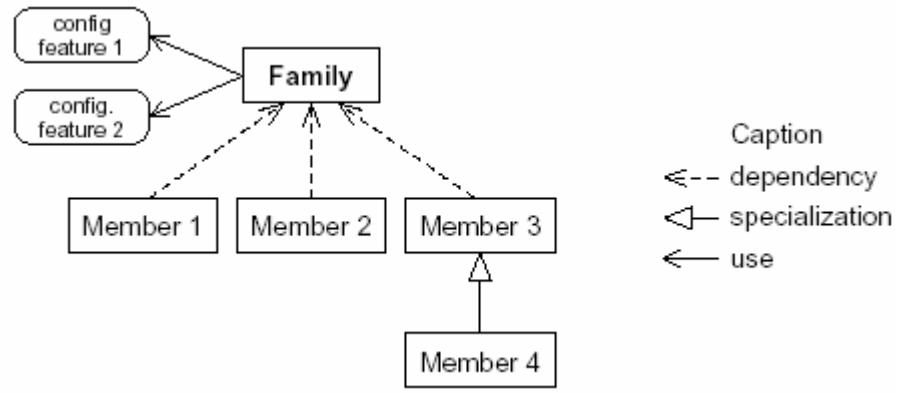


Figura 2.1. Notação utilizada para a representação de Famílias de Abstrações [Frö01]

2.1.3. Características Configuráveis (*Configurable Features*)

Quando é detectada uma característica que pode ser aplicada a todos os membros de uma Família, não é interessante modelar novos membros, o que iria duplicar a cardinalidade da Família. Neste caso, esta funcionalidade é modelada como uma Característica Configurável, que define o comportamento desejado e pode ser aplicado às abstrações de forma semelhante aos aspectos de cenário. Entretanto, diferentemente dos aspectos, a implementação dessas características fica por conta de cada membro da Família: uma Característica Configurável define apenas qual o comportamento que cada membro deve ter quando essa característica estiver ativada.

2.1.4. Aspectos de Cenário (*Scenario Aspects*)

Para evitar o crescimento do número de abstrações semelhantes para cenários diferentes, os aspectos dependentes de cenário são modelados separadamente. Para que esta estratégia funcione, é necessário que os aspectos possam ser aplicados às abstrações de forma transparente, isto é, sem que seja necessário modificá-las.

Para conseguir isto, os aspectos são selecionados para compor um cenário, que é depois aplicado às abstrações independentes de cenário através de um adaptador de cenário. Um adaptador de cenário engloba a abstração, intermediando sua comunicação com o cliente dependente de cenário. A Figura 2.2 mostra a estrutura geral de um adaptador de cenário.

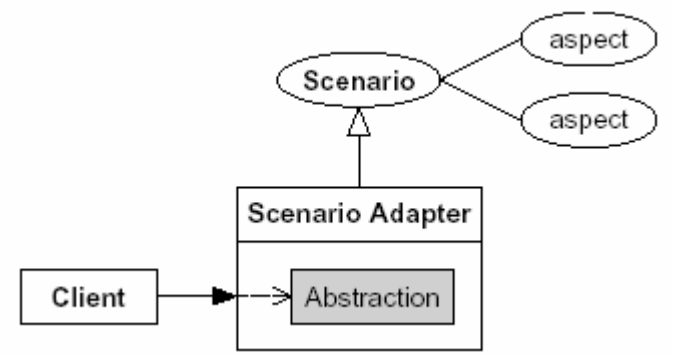


Figura 2.2. Estrutura geral de um Adaptador de Cenário [Frö01]

Existem dois tipos de aspectos: estrutural e comportamental. O primeiro altera a estrutura das abstrações, acrescentando estruturas de dados específicas para o cenário. O segundo altera o comportamento da abstração, acrescentando alguma semântica específica do cenário às operações da abstração.

2.1.5. Interfaces Infladas (*Inflated Interfaces*)

As Interfaces infladas combinam as características de todos os membros de uma Família, gerando uma visão única da Família como um "supercomponente". Isto permite que o programador da aplicação escreva seu código sempre em termos da interface inflada, postergando a decisão sobre qual membro da Família utilizar. Isto poderia, então, ser feito por ferramentas de configuração automáticas, que sejam capazes de identificar quais características da Família foram utilizadas e selecionar automaticamente uma realização que seja a menor possível para a implementação do subconjunto de características utilizadas,

através de uma ligação transparente da interface inflada a uma das realizações no momento da compilação. Para suportar este tipo projeto, duas novas relações foram propostas: realização parcial e realização seletiva. Uma realização parcial indica que apenas uma parte das características da interface inflada foi implementada pela realização. Uma realização seletiva indica que, dentre um conjunto de possíveis realizações, apenas uma pode ser ligada à interface por vez. A Figura 2.3 mostra a notação utilizada para representar as realizações.

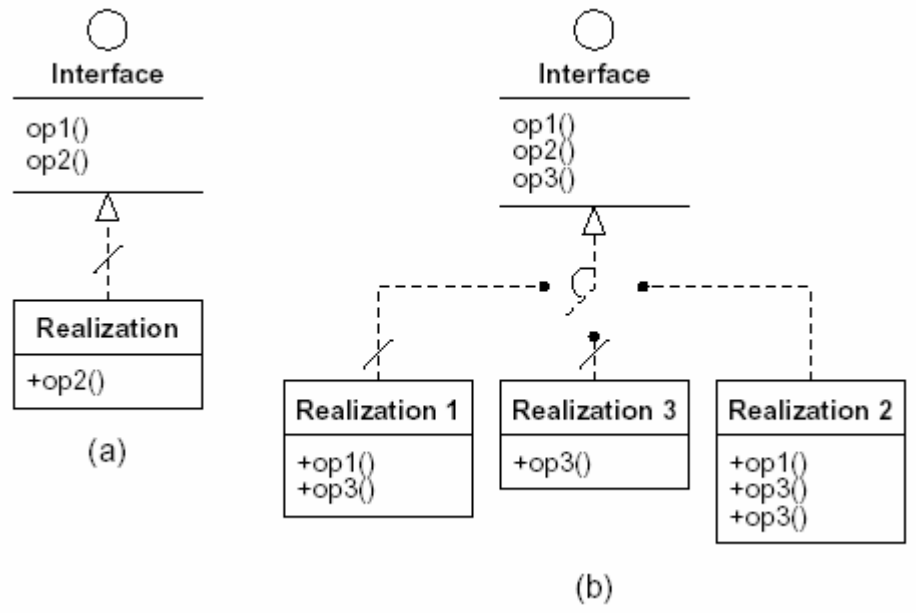


Figura 2.3. Notação utilizada para realização parcial (a) e realização seletiva (b) [Frö01]

Para que esta seleção funcione, as interfaces devem ser bem projetadas levando em conta a estrutura interna das Famílias. Existem quatro tipos de Famílias:

- *Uniforme*: uma família na qual todos os membros compartilham a mesma interface;
- *Incremental*: uma família na qual os membros seguem um projeto incremental, isto é, cada membro é uma extensão do anterior;
- *Combinada*: uma família na qual os membros não compartilham nenhuma característica, e podem ser combinados (via herança múltipla, por exemplo) para gerar novos membros com a soma das características;
- *Dissociada*: uma família que não se encaixa em nenhuma das categorias anteriores.

2.1.6. Arquiteturas reutilizáveis

Para que os diversos componentes modelados a partir do domínio tenham utilidade, é necessário reuni-los em uma arquitetura que faça sentido. A idéia é aproveitar a experiência adquirida com o desenvolvimento de diversos sistemas e modelar um framework que descreve como as abstrações interagem entre si, com o ambiente e com as aplicações. A utilização de um framework para o desenvolvimento do sistema diminui a ocorrência de erros, já que somente as composições pré-definidas pelos arquitetos do sistema poderão ser utilizadas.

Um framework de componentes orientado à aplicação seria uma composição de adaptadores de cenário. Cada adaptador definiria um "socket" para componentes de determinada família. Para implementar o sistema, bastaria conectar um componente a cada socket, escolhendo uma realização para cada uma das interfaces infladas presentes no framework.

A Figura 2.4 mostra a estrutura genérica de um framework orientado à aplicação.

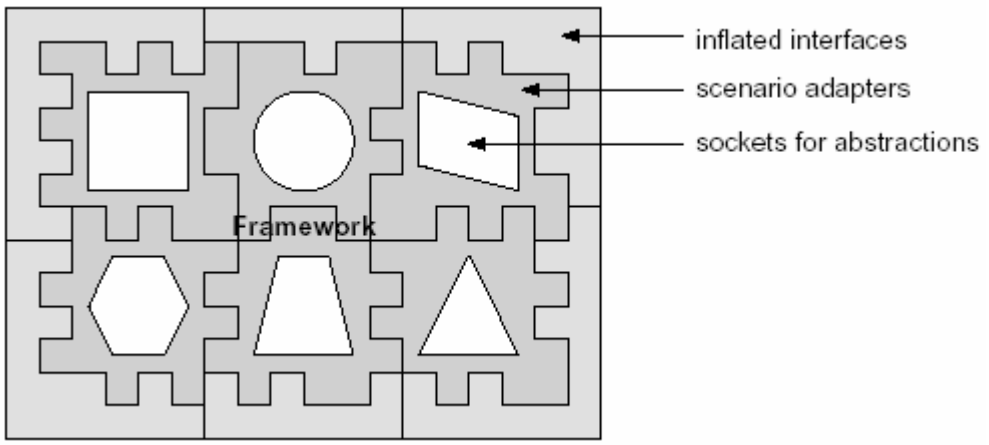


Figura 2.4. Estrutura genérica de um Framework orientado à aplicação [Frö01]

2.1.7. Visão Geral

A Figura 2.5 mostra uma visão geral da metodologia de Projeto de Sistemas Orientados à Aplicação: uma metodologia de projeto multi-paradigma que promove a construção de sistemas operacionais aplicando o processo de decomposição orientada à aplicação, modelando o domínio como Famílias de abstrações independentes de cenário e

reutilizáveis. Dependências de cenário são modeladas como Aspectos de cenário, que são aplicados às abstrações através de adaptadores de cenário. As Famílias são apresentadas às aplicações através das interfaces infladas, que são ligadas a uma das realizações no momento de geração do sistema. Arquiteturas de sistema reutilizáveis são capturadas em Frameworks de componentes descritos em termos de adaptadores de cenário.

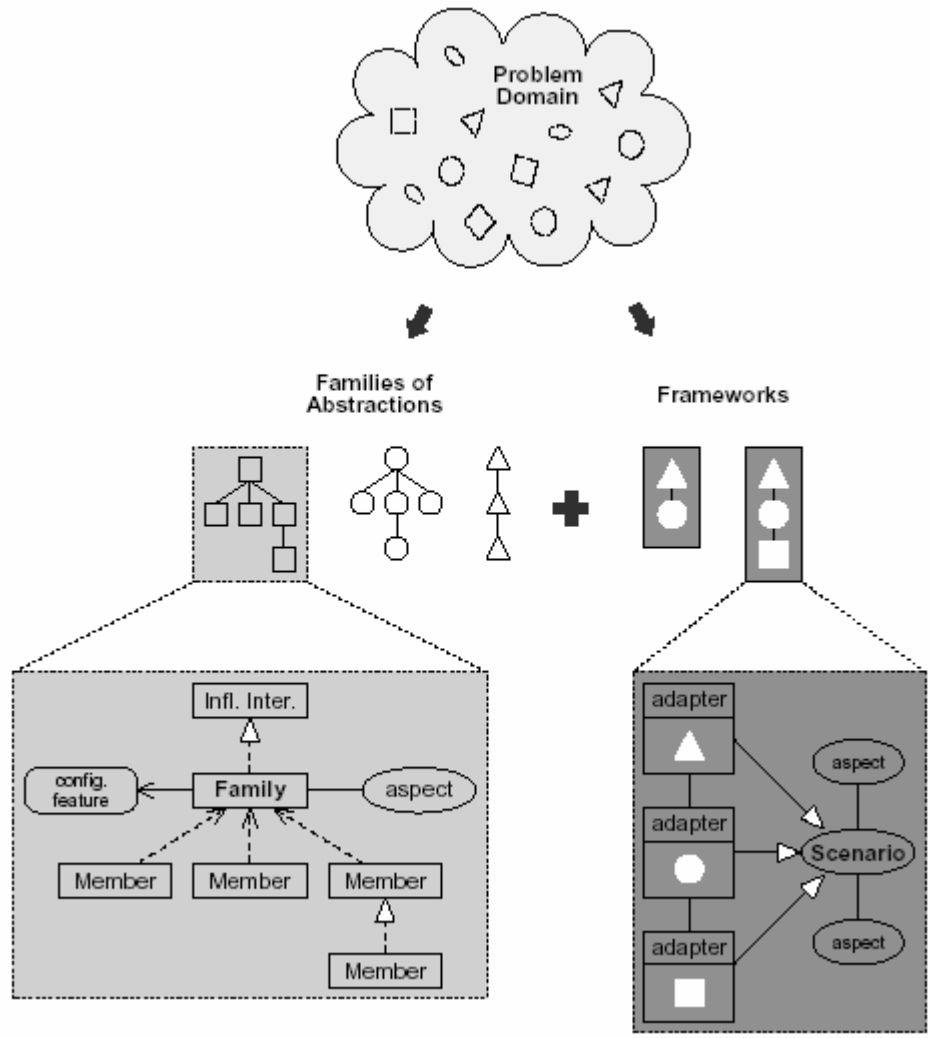


Figura 2.5. Visão geral da metodologia de Projeto de Sistemas Orientados à Aplicação [Frö01]

2.2. O Sistema Operacional EPOS

O Sistema EPOS [Frö01] foi criado em 1997 no Research Institute for Computer Architecture and Software Engineering (FIRST) do German National Research Center for Information Technology (GMD) como um projeto para experimentar os conceitos e mecanismos de Projeto de Sistemas Orientados à Aplicação. Portanto, EPOS é um sistema operacional orientado à aplicação e evoluiu juntamente com a criação da metodologia.

EPOS significa *Embedded Parallel Operating System*, ou seja, é um sistema operacional que pode ser utilizado por aplicações embutidas e aplicações paralelas, e está fortemente relacionado com computação dedicada e configuração estática.

Na descrição que segue, procuraremos entender de forma mais abrangente a arquitetura do sistema EPOS, concentrando maior atenção nos aspectos referentes ao modelo de configuração do sistema, que será a base para a realização deste Projeto.

2.2.1. Características básicas

O EPOS foi desenvolvido visando atender ao domínio de computação dedicada de alto desempenho, no qual as aplicações geralmente são executadas com exclusividade na plataforma e requerem um gerenciamento eficiente de recursos.

Desta forma, o EPOS foi desenvolvido com as seguintes metas:

- Funcionalidade: deve fornecer a funcionalidade necessária para executar aplicações dedicadas de alta performance.
- Customizabilidade: deve ser altamente customizável, de forma que uma instância possa se adequar especificamente a uma aplicação. Quando possível, essa configuração deve ser automática.
- Eficiência: deve disponibilizar os recursos para as aplicações com o mínimo overhead possível.

Para manter a escalabilidade, uma operação meticulosa de separação de interesses foi realizada. As abstrações foram modeladas independentes entre si, dos aspectos de cenário e dos componentes do framework.

A Figura 2.6 mostra uma representação de alto nível das Famílias de abstrações do EPOS. Elas foram organizadas em seis grandes grupos: gerenciamento de memória,

gerenciamento de processos, coordenação de processos, comunicação interprocessos, gerenciamento de tempo e gerenciamento de entrada/saída. Em cada um desses grupos foram definidas várias Famílias com suas Características Configuráveis e Aspectos de Cenário, porém não entraremos em descrições mais detalhadas sobre cada uma delas, já que isto não prejudicará a compreensão do presente trabalho.

As abstrações que são dependentes da arquitetura – *Processor* e *Node*, por exemplo – foram modeladas separadamente, como Mediadores de Hardware, isto é, abstrações dependentes do hardware, e servem para fazer a interface entre o hardware e o restante do sistema, escondendo as peculiaridades dos componentes de hardware freqüentemente utilizados pelo sistema.

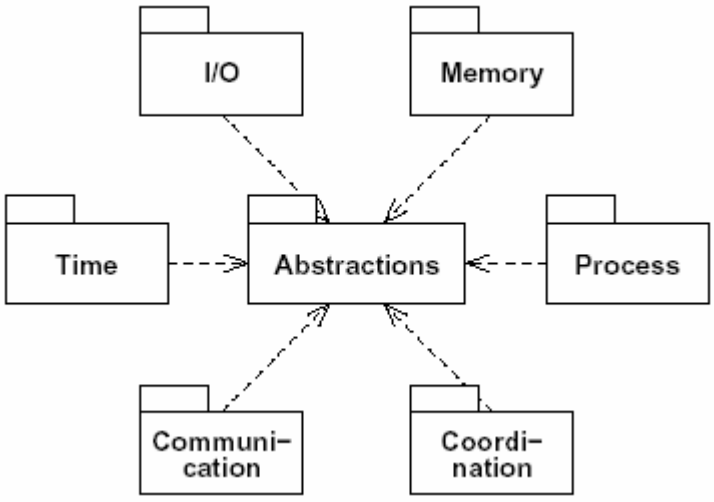


Figura 2.6. Grupos de famílias de abstrações do EPOS [Frö01]

2.2.2. Arquitetura do Sistema

O núcleo da arquitetura de software do EPOS está no Framework de Componentes, que especifica como as abstrações e aspectos devem ser combinados considerando o ambiente de execução estabelecido. Por ter relação direta e participar do processo de configuração estática do sistema, será apresentada a seguir a estrutura geral do Framework de Componentes do EPOS.

Logo após, será feita a descrição dos aspectos de portabilidade e inicialização do sistema, porém de forma mais abrangente, já que essa parte da arquitetura não é relevante para o processo de configuração estática do sistema.

2.2.2.1. Framework de Componentes

Um Framework de Componentes orientado à aplicação captura elementos de arquiteturas de sistema reutilizáveis, enquanto define como as abstrações podem ser organizadas em conjunto para formar um sistema funcional. O Framework de Componentes do EPOS foi modelado como um conjunto de Adaptadores de Cenário interrelacionados, formando um conjunto de sockets para abstrações e aspectos de cenário, que podem ser conectados ao framework via ligação das interfaces infladas com os componentes.

O Framework é implementado através de um metaprograma estático [Str97, Vel95, Pes97] e um conjunto de regras de composição. O metaprograma é responsável por adaptar as abstrações ao cenário de execução e juntá-las durante a compilação do sistema. As regras de composição regulam a execução do metaprograma, especificando restrições e dependências para a composição das abstrações. As regras de composição não fazem parte do metaprograma, elas são especificadas externamente por ferramentas para ajustar os parâmetros do metaprograma, permitindo que o framework possa atender a uma grande variedade de arquiteturas de software.

O metaprograma é executado durante a compilação do sistema. Ele realiza a composição das abstrações independentes de cenário, e depois disso ele adapta as abstrações e composições resultantes ao cenário especificado.

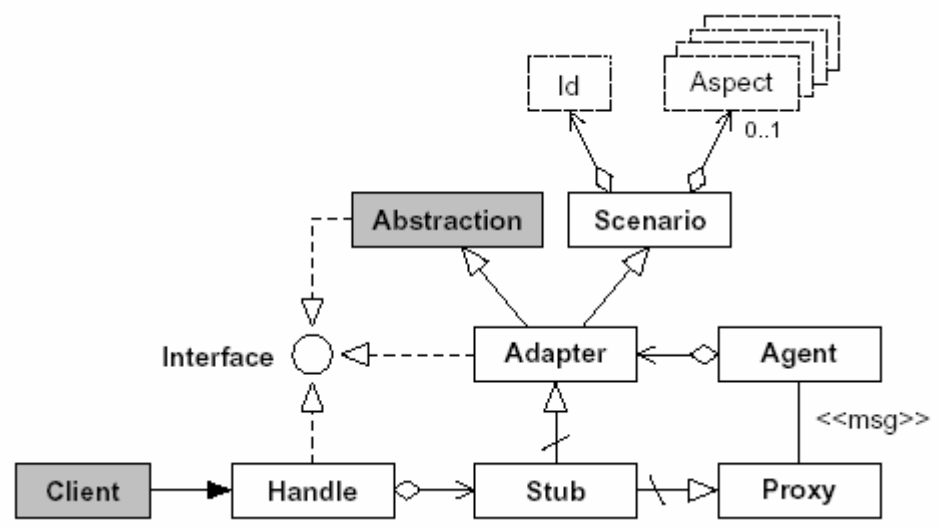


Figura 2.7. Uma visão geral do metaprograma estático do Framework [Frö01]

A Figura 2.7 mostra uma visão geral do metaprograma estático do framework.. A classe parametrizada Handle recebe uma abstração como parâmetro e funciona como um mediador, realizando sua interface de forma que as operações sejam direcionadas para o Stub. Esta classe parametrizada é responsável, por sua vez, de fazer a ponte entre o Handle e o adaptador da abstração ou o seu Proxy. Quando o cenário de execução é local, o Stub direciona a mensagem diretamente ao adaptador de cenário da abstração, quando o cenário de execução remota está ativado, o Stub encaminha a mensagem para o Proxy. Neste caso, o Proxy irá enviar a mensagem ao Agent, que irá então executar a mesma função que o Stub executa no cenário local: direcionar a mensagem para o adaptador de cenário. A classe parametrizada Adapter empacota a abstração, adaptando-a ao cenário que define os aspectos. O último elemento que configura o sistema é a classe parametrizada Scenario, que implementa as primitivas específicas do cenário na abstração. Para tanto, existe uma outra classe Traits, onde são especificados quais aspectos devem ser incluídos para uma abstração no cenário especificado.

A operação do metaprograma é coordenada pelas regras de composição, que reunem informações sobre a arquitetura capturadas durante o projeto. As regras especificam as restrições e dependências entre as abstrações, aspectos de cenário, mediadores de hardware, características configuráveis e requisitos não funcionais, permitindo a análise de uma determinada configuração do sistema e das composições criadas quanto à sua validade, impedindo a compilação de uma configuração inválida.

Para suportar a configuração externa através das regras de composição, os elementos do EPOS são marcados com uma chave de configuração (*configuration key*). Assim, quando uma chave é verificada, o elemento correspondente é inserido na configuração. Elementos que são organizados em Famílias são adicionados atribuindo uma chave de um membro à chave da Família, fazendo com que a interface inflada seja ligada com uma realização. Elementos que não possuem Famílias têm suas chaves verificadas de acordo com o elemento, por exemplo, atribuir a chave *True* a um aspecto ou a uma característica configurável irá habilitá-los.

Além disso, as regras de composição ainda defem pré e pós condições para as chaves de configuração. Por exemplo, uma pré-condição pode definir que, para que uma determinada abstração seja selecionada, uma outra também deverá estar previamente selecionada.

2.2.2.2. Portabilidade

O EPOS foi desenvolvido pensando em manter a portabilidade, e desta forma os componentes visíveis do sistema podem ser utilizados pelas aplicações sempre com a mesma sintaxe, independente da plataforma de execução.

Dois artefatos garantem a portabilidade do sistema: o utilitário Setup e os mediadores de hardware.

O utilitário Setup é executado antes do sistema operacional e prepara o hardware para a execução do mesmo. Este utilitário é altamente dependente da plataforma e não é o objetivo torná-lo portável, mas ele libera o sistema de uma grande fonte de problemas não portáveis.

Os mediadores de hardware são abstrações que, ao contrário das demais, são totalmente dependentes do hardware e também não são portáveis, sendo especificamente projetados e desenvolvidos para cada plataforma. Sempre que alguma abstração do EPOS precisa interagir com o hardware, ela o faz através de um mediador de hardware, promovendo desta forma a portabilidade.

2.2.2.3. Inicialização

A inicialização do EPOS ocorre em duas partes: a primeira é a configuração do hardware feita pelo utilitário Setup introduzido anteriormente, a outra é a inicialização dos componentes de software feita pelo utilitário Init.

O utilitário Init primeiro detecta quais são os componentes presentes no sistema para inicializá-los, e depois cria o primeiro (e talvez único) processo que irá ser executado. Ao final, o utilitário Init, assim como o Setup, libera completamente a memória que utilizou, disponibilizando-a para uso da aplicação.

A Figura 2.8 mostra uma visão geral da inicialização do EPOS: o boot invoca o utilitário Setup, que configura o hardware. Este termina liberando a memória que utilizou e chamando o utilitário Init, que inicializa os componentes utilizados pelo sistema, e termina também liberando a memória utilizada e carregando o executável especificado na imagem de boot para criar o primeiro processo.

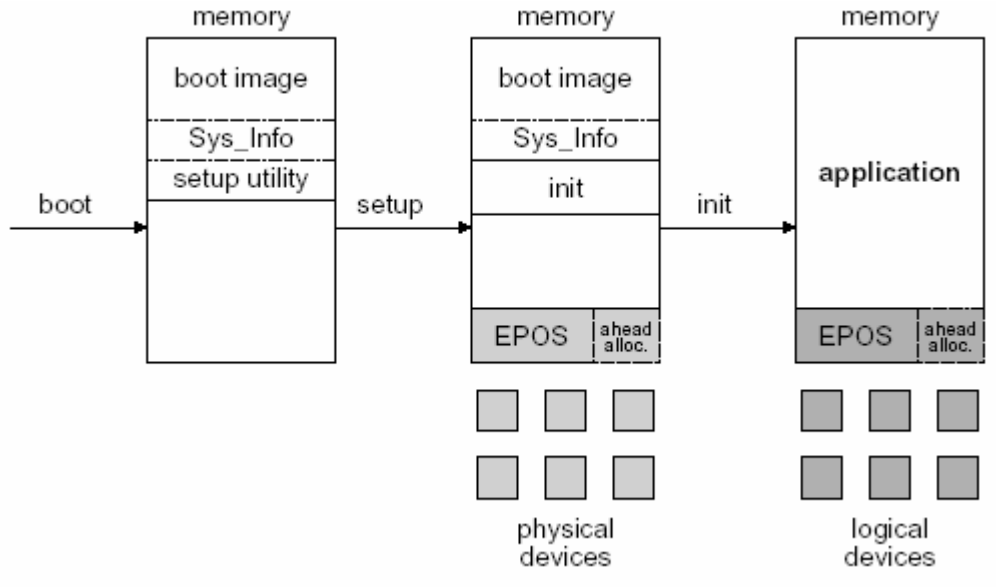


Figura 2.8. Uma visão geral da inicialização do EPOS [Frö01]

2.2.3 Configuração do EPOS

As regras de composição e os traits controlam o processo de configurar e moldar o EPOS para uma aplicação específica. O conjunto de chaves de configuração selecionado é validado e refinado pelas regras de composição, resultando em um conjunto de elementos que são consultados através dos traits de cada abstração pelo metaprograma do Framework de Componentes.

Para configurar um sistema baseado em componentes como o EPOS, definindo as chaves de configuração, basta entender os requisitos da aplicação. Devido à utilização das interfaces infladas, isto pode ser feito simplesmente lendo o código da aplicação e identificando quais características de cada Família foram utilizadas e, conseqüentemente, quais abstrações podem atender aos requisitos. Esta tarefa é realizada automaticamente por uma ferramenta (o *Analyser*) que gera uma lista das interfaces utilizadas.

Estes dados passam então para a segunda ferramenta (o *Configurator*), que consulta um catálogo com a lista de abstrações e as regras de composição, definindo a configuração específica para a aplicação. Durante esta fase, o Configurator procura escolher as abstrações mais simples para atender aos requisitos, usando como base o valor estimado de *overhead* gerado por cada abstração. A saída consiste em um conjunto de chaves de realização seletiva, que definem as ligações das interfaces infladas com as abstrações e os aspectos de cenário, e um conjunto de chaves de características configuráveis, que identificam quais

características configuráveis devem ser habilitadas para as abstrações e aspectos de cenário.

A última fase é realizada pela ferramenta *Generator*, que usa as chaves de configuração produzidas pelo Configurator para compilar uma versão customizada do EPOS. Estas chaves são traduzidas em estruturas *typedef* e *Traits* que controlam a operação do metaprograma estático do Framework de componentes durante a compilação.

A Figura 2.9 ilustra o processo de configuração do EPOS.

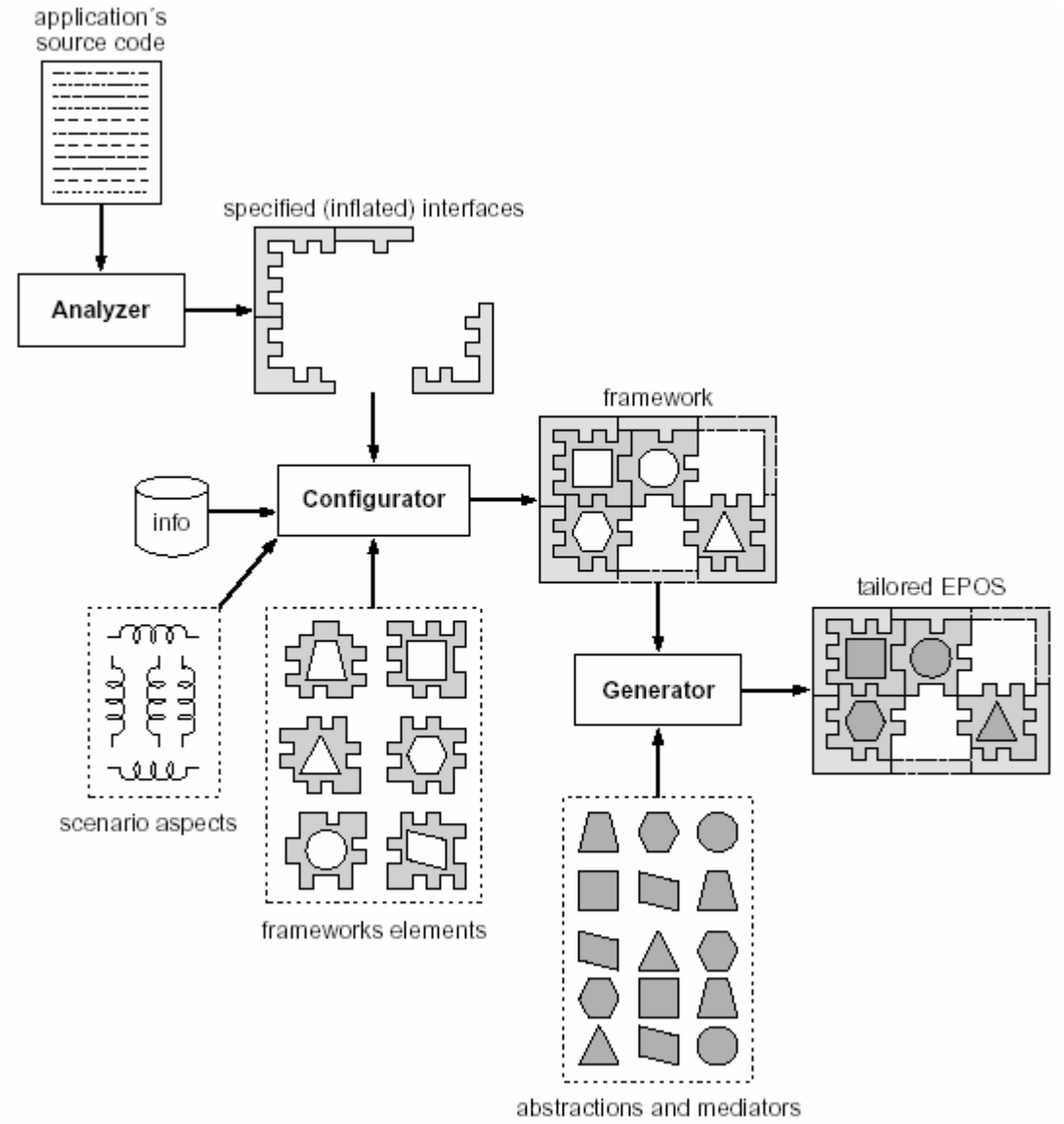


Figura 2.9. Ferramentas para configuração automática do EPOS [Frö01]

2.3. Necessidade de modelos de configuração

A geração de uma configuração otimizada a partir da análise de requisitos da aplicação necessita de modelos de especificação de componentes bastante exatos e detalhados, para que seja possível evitar a criação de configurações inválidas ou inconsistentes.

Além disso, embora a maior parte da configuração seja feita por ferramentas automáticas, às vezes estas não conseguem escolher a configuração ótima do sistema para uma determinada aplicação. Isto ocorre porque as ferramentas selecionam as abstrações através de apenas um critério, que é o overhead produzido por cada uma, ou seja, elas ligam a interface inflada ao membro com menor overhead e que inclua todas as características utilizadas. Porém, isto nem sempre é o ideal. Além disso, existem algumas informações que devem ser informadas pelo usuário como, por exemplo, as características do hardware que irá executar o sistema e a aplicação.

O presente trabalho tem o foco justamente na construção da base de conhecimentos que permitirá o desenvolvimento do Configurator e no desenvolvimento de ferramentas gráficas interativas que permitam ao usuário refinar e completar a configuração gerada pelas ferramentas automáticas, complementando o trabalho realizado pelo Configurator.

Esta ferramenta possibilitará ao usuário refinar uma configuração gerada automaticamente ou criar uma configuração personalizada manualmente. O usuário deverá ser capaz de configurar todos os elementos necessários: abstrações, aspectos de cenário, características configuráveis. A saída será o conjunto de chaves de configuração que estarão prontas para serem utilizadas pelo Generator para compilar a versão do EPOS moldada à aplicação e segundo as características informadas pelo usuário.

3. ESPECIFICAÇÃO DE COMPONENTES DE SOFTWARE

Para a definição exata do que é um componente de software ainda não existe um consenso. Em [Sil00], é apresentada a definição do WCOP 96, que descreve um componente como "uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas. Componentes podem ser duplicados e estar sujeitos a composição com terceiros." [Szy96], depois refinada no WCOP 97: "o que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de implementação específica. Assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Esta interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida." [Szy97]

Segundo Fröhlich [Frö01], para a metodologia de Projeto de Sistemas Orientados à Aplicação foi utilizada a definição mais ampla de componente extraída do Oxford English Dictionary [Oxf92]: "qualquer uma das partes do que algo é feito".

Mas o importante é decidir como iremos representar os componentes. Existem várias formas de representação de componentes, sendo que cada uma cumpre um objetivo específico e atende a um tipo de aplicação. A seguir apresentaremos uma breve descrição de algumas destas formas para basear o modelo de especificação para nossa base de conhecimentos, que será descrito no Capítulo 4.

3.1. Interfaces

A Interface de um componente é geralmente a descrição das assinaturas das operações que podem ser invocadas sobre ele. Outros autores, como Ólafsson [Ola96], propõem que a interface especifique também as assinaturas de operações que o componente invoca. Para Silva [Sil00], no caso geral, um componente possui uma interface, composta de um ou mais canais de comunicação, através dos quais o componente se comunica com o meio externo. Esses canais são geralmente bidirecionais, sendo a comunicação unidirecional um caso específico. A Figura 3.1 ilustra este conceito:

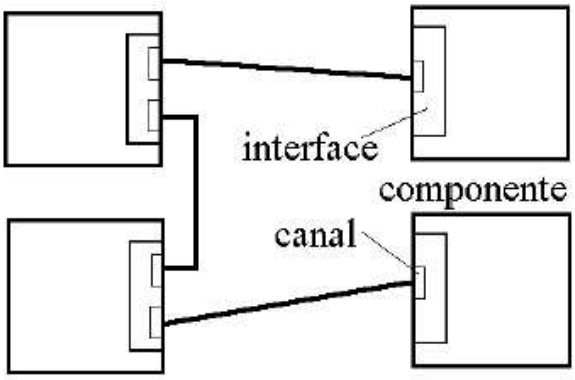


Figura 3.1. Um componente com interfaces formadas por canais bidirecionais [Sil00]

A Interface por si só não apresenta todas as informações necessárias para garantir a composição correta de componentes, pois não permite especificar nenhum aspecto semântico do componente. No entanto, as Interfaces constituem o mecanismo mais elementar para suportar composição de componentes, servindo de base para a criação de mecanismos mais complexos [Frö01].

3.2. Contratos

Um Contrato estende o conceito de Interface para incluir a especificação de aspectos comportamentais do Componente. Para Helm e Holland [HHG90], um Contrato define uma composição comportamental de um conjunto de participantes. Eles propõem uma especificação formal de um Contrato entre um conjunto de componentes, de forma que esta composição possa ser automaticamente verificada.

Já Larman [Lar00] propõe um modelo de especificação de Contratos mais informal. Neste caso, seria especificado um Contrato para cada operação de um Componente, geralmente em estilo declarativo. A especificação do Contrato definiria as mudanças de estados que a operação geraria no sistema através de pré-condições e pós-condições.

3.3. Darwin

Darwin [MDEK95] é uma linguagem declarativa para a descrição de arquiteturas de sistemas baseados em componentes (*Architectural Description Language* – ADL), cujo objetivo é ser uma notação de uso geral para a especificação da estrutura de sistemas

compostos de diversos componentes e utilizando diversos mecanismos de interação.

A linguagem Darwin inclui um complexo modelo formal criado a partir de uma teoria chamada π-calculus, que não iremos analisar. Mais interessante para nosso trabalho é o conceito que Darwin faz de um componente: através de serviços que ele provê para permitir que outros componentes interajam com ele e de serviços que ele requer para interagir com outros componentes. A Figura 3.2 mostra um exemplo de um componente especificado em Darwin nas suas duas formas (gráfica e textual).

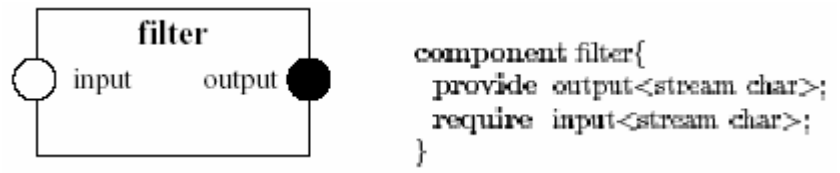


Figura 3.2. Exemplo de especificação de um componente em Darwin [MDEK95]

3.4. Koala

Koala [OLKM00, Omm02] é um modelo de componentes e também uma ADL, e foi criado com o objetivo de permitir o desenvolvimento de softwares para sistemas embutidos em equipamentos eletrônicos (Televisores, videocassetes, DVD players, etc.) utilizando componentes reutilizáveis.

O modelo Koala utilizou como base o Darwin, e, portanto a idéia de que componentes provêem e requerem interfaces, e ligam-se entre si para compor um sistema completo. Extensões foram criadas para permitir a adição de *glue code* entre componentes (código para adaptar interfaces não exatamente idênticas) e para permitir um mecanismo de parametrização de diversidade para a configuração de componentes e a otimização de código.

A definição de interfaces em Koala é feita através de uma IDL (*Interface Definition Language*) simples, listando os protótipos das funções em sintaxe C, como exemplificado na Figura 3.3.

```
interface ITuner
{
  void SetFrequency(int f);
  int  GetFrequency(void);
}
```

Figura 3.3. Exemplo da IDL do modelo Koala [OLKM00]

Um componente é descrito utilizando a CDL (*Component Description Language*), através das interfaces que ele provê e das que ele requer, como exemplificado na Figura 3.4. Na CDL, cada interface recebe um nome local, que especifica uma instância da interface, permitindo que um mesmo componente possa requerer mais de uma interface do mesmo tipo.

```
component CTunerDriver
{
  provides ITuner ptun;
           IInit  pini;
  requires II2c   ri2c;
}
```

Figura 3.4. Exemplo da CDL do modelo Koala [OLKM00]

Para conectar componentes, cada interface *requires* deve ser conectada a exatamente uma interface *provides*. Uma configuração de sistema é um conjunto de componentes interconectados onde todas as interfaces *requires* foram conectadas.

Os componentes podem ainda ser compostos, formando unidades maiores de reutilização. A Figura 3.5 mostra um exemplo de especificação de um componente composto, mostrando também como são definidas as ligações entre interfaces.

```
component CTvPlatform
{
  provides IProgram pprg;
  requires II2c slow, fast;
  contains
    component CFrontEnd cfre;
    component CTunerDriver ctun;
  connects
    pprg      - cfre.pprg;
    cfre.rtun - ctun.ptun;
    ctun.ri2c - fast;
}
```

Figura 3.5. Exemplo de definição de um componente composto em Koala [OLKM00]

Os componentes em si não têm conhecimento da configuração que o sistema completo terá, ou seja, quais são os outros componentes que serão ligados em suas interfaces. Depois de definida esta configuração, uma ferramenta também chamada Koala lê as descrições e gera um arquivo header com macros que renomeiam nomes lógicos de funções nas interfaces para nomes físicos dos componentes que irão implementá-las, fazendo as ligações necessárias. As ligações entre funções são feitas automaticamente quando elas têm nomes iguais, mas também podem ser explicitamente especificadas na configuração quando

os nomes forem diferentes. A ferramenta Koala procura fazer todas as ligações de forma estática quando possível. Em alguns casos, entretanto, pode ser especificado uma função ou um parâmetro que será resolvido em tempo de execução que define qual componente irá ser ligado a determinada interface, e neste caso são criadas funções simples como *if-then* para selecionar em tempo de execução a interface correta.

Para permitir a configuração de componentes, foi criado um mecanismo de parametrização. Porém, não foi utilizado o mecanismo padrão de funções *Get* e *Set* para o acesso a estes parâmetros; ao invés disto, um componente que deve ser parametrizado é especificado com uma interface *requires* que contenha os parâmetros necessários. Estas interfaces foram chamadas de *diversity interfaces*. A Figura 3.6 mostra um exemplo de uma destas interfaces: o componente CFrontEnd precisa de um parâmetro que será utilizado para saber qual componente utilizar para a implementação da interface ITuner. Este parâmetro está implementado em um módulo *m*. O resultado deste parâmetro é utilizado pela ferramenta Koala para fazer a ligação: se o valor for constante, esta ligação é feita em tempo de compilação e otimizada pelo compilador; se for uma função somente resolvida em tempo de execução, a ferramenta gera uma expressão que selecionará dinamicamente a ligação.

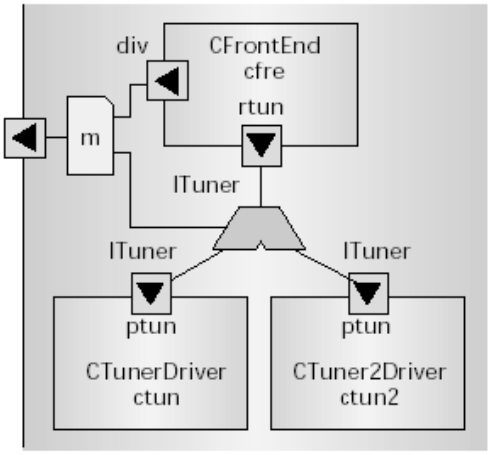


Figura 3.6. Exemplo de uma diversity interface em Koala [OLKM00]

Note que na representação de um Componente em Koala, ilustrada na Figura 3.6, as interfaces são representadas por caixas com setas, que indicam a direção da interface (seta para fora indica *requires*, seta para dentro indica *provides*). A forma entre os componentes representa um *switch* que faz a seleção de qual componente escolher para ligar a interface *requires* do componente CFrontEnd.

3.5. Ambiente SEA

O Ambiente SEA [Sil00] é voltado ao desenvolvimento e uso de artefatos de software reutilizáveis. Ele tem como base o uso de orientação a objetos e UML para possibilitar a utilização integrada das abordagens de desenvolvimento baseado em componentes e desenvolvimento baseado em frameworks, incluindo padrões de projeto, promovendo reuso tanto de implementação como de projeto. É importante notar que o termo framework é utilizado neste contexto com um significado diferente do adotado pela metodologia de Projeto de Sistemas Orientados à Aplicação. Para o Ambiente SEA, um framework orientado a objetos é "uma estrutura de classes inter-relacionadas, que corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas" [Sil00].

No ambiente SEA, componentes são definidos através de especificações orientadas a objetos. O que caracteriza uma estrutura de classes de componente é que esta reutiliza uma interface, originada de uma biblioteca de interfaces de componente. Ou seja, interfaces de componentes são definidas de forma independente dos componentes, para que possam ser reutilizadas para a construção de muitos componentes.

A interface de componentes é especificada quanto aos seus aspectos estruturais e comportamentais. Para a especificação estrutural é feita uma relação dos métodos fornecidos, dos métodos requeridos e das associações destes com cada canal da interface. Nem todos os métodos precisam estar acessíveis em todos os canais, e por isso a definição da interface é feita associando as assinaturas dos métodos com os canais, conforme ilustrado na Figura 3.7.

		Métodos requeridos					Métodos fornecidos				
		mA	**mB**	**mC**	**mD**	**mE**	**mF**	**mG**	**mH**	**mI**	**mJ**
Canais	**c1**	✓		✓	✓		✓	✓	✓		
	c2	✓	✓				✓			✓	✓
	c3		✓		✓	✓	✓		✓	✓	✓

Figura 3.7. Exemplo de especificação de relacionamento de canais e métodos de um componente no ambiente SEA [Sil00]

Para a especificação comportamental da interface, a questão a ser tratada é se há ou não restrição na ordem de invocação dos métodos. A inexistência de restrições significa que qualquer método fornecido ou requerido pode ser invocado a qualquer momento. Porém, se existirem restrições indicando que um método deve ser invocado antes de outro, esta informação precisa estar descrita na interface. Para isto, foi adotado o uso de Redes de Petri [Pet62], por ser um modelo baseado em formalismo algébrico, o que permite validar a especificação, e por possuir uma representação gráfica, o que facilita a compreensão.

A Figura 3.8 mostra um exemplo de especificação comportamental de componentes. A Rede de Petri ordinária, que foi o tipo utilizado, é composta por lugares (círculos), transições (retângulos), arcos que interligam lugares e transições (setas) e uma marcação inicial (caracterizada por uma quantidade de fichas em cada lugar da rede). Cada par (canal, método) da interface é associado a uma transição na rede, e os lugares representam os estados do componente. Um método somente pode ser invocado se houver pelo menos uma ficha em cada lugar de onde parte a transição correspondente.

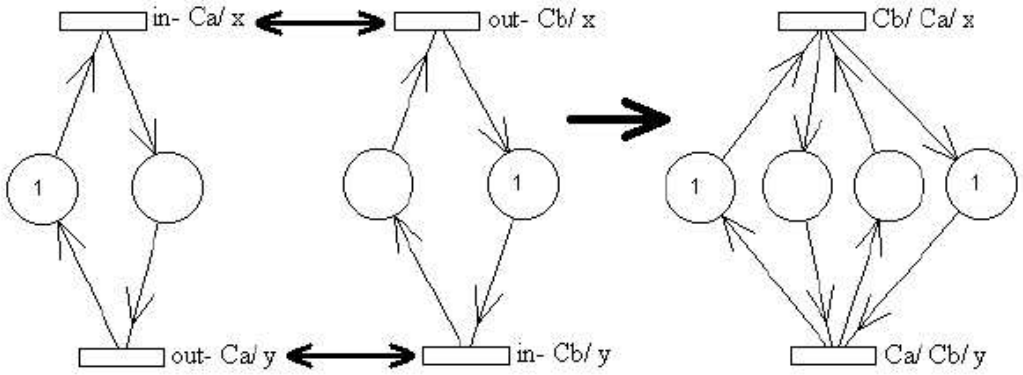


Figura 3.8. Exemplo de descrição comportamental de uma arquitetura de componentes no ambiente SEA [Sil00]

O comportamento de um conjunto de componentes interligados é obtido a partir da união das Redes de Petri. Nesta operação, os lugares, arcos e marcações iniciais permanecem inalterados, e os pares de transições correspondentes (método fornecido de uma interface com o método requerido de outra) são fundidos. A Figura 3.8 mostra o exemplo da união de dois componentes, sendo a rede resultante mostrada na parte direita da Figura. Nota-se que os dois componentes são estruturalmente compatíveis, pois foi possível unir os métodos fornecidos e

requeridos, mas são comportamentalmente incompatíveis, pois a marcação inicial na rede resultante mostra uma situação que não habilita o disparo de nenhuma transição (invocação de método), causando um *deadlock*.

4. REPOSITÓRIO DE COMPONENTES

Toda a estrutura de um Sistema Operacional orientado à aplicação é definida através da conexão de vários componentes básicos, de acordo com as necessidades da aplicação e da arquitetura do sistema definida pelo projetista.

Este capítulo irá apresentar os aspectos referentes à definição da estrutura da Base de Conhecimentos que será utilizada para descrever os componentes do repositório do EPOS.

4.1. Modelo atual

As ferramentas para configuração interativa desenvolvidas até o momento para o EPOS [Röm01] utilizam como entrada um arquivo de configuração e produzem como saída os arquivos com as chaves de configuração que serão utilizadas pelo compilador para parametrizar o framework de componentes. Neste modelo, todo o conhecimento sobre a arquitetura do sistema está neste arquivo de configuração, escrito em XML [W3C98]. O Anexo A apresenta a definição do mesmo, através do arquivo DTD correspondente.

As principais partes da estrutura deste modelo são:

- *Configuration*: é a raiz do arquivo XML, e especifica um nome descritivo para a configuração e os nomes dos arquivos de saída;
- *Domain*: separa as configurações de acordo com o domínio. Exemplos de domínio são Hardware e Sistema Operacional.
- *Section*: separa o domínio em seções lógicas. Cada seção tem um nome descritivo.
- *Interface*: representa um *socket* para a inserção de um componente no framework, através da seleção de uma realização para a interface. Geralmente possui um nome descritivo e a definição da realização padrão.
- *Realisation*: geralmente, um conjunto de realizações é disponibilizado para cada interface, para que o usuário escolha a que melhor se adapta a sua necessidade. Além de um nome descritivo, há a indicação do nome da classe que implementa esta realização e do nome do arquivo header onde se encontra esta classe.

- *Input*: assim como a parte *Interface*, é disponibilizada dentro de *Section* e permite ao usuário a entrada de parâmetros informativos sobre o sistema como, por exemplo, a quantidade de processadores presentes.
- *Pré e pós condições*: associadas com *Realization* ou *Input*, descrevem as restrições presentes nos componentes como, por exemplo, a condição de que um determinado componente deva estar presente no sistema para que outro também possa estar, ou que uma entrada somente será necessária se determinado componente estiver presente.

Além deste arquivo com as informações relativas à arquitetura do sistema, existem outras duas propostas de especificação.

A primeira especificação serve para descrever a máquina alvo do sistema. Nesta são entradas as informações relativas a maquina (processador, memória, barramentos e dispositivos), mapa de boot e mapa de memória, assim como o prefixo do compilador específico para a arquitetura. O DTD deste modelo está no Anexo B.

A segunda serve para a descrição das famílias de componentes. Neste modelo são descritos as interfaces infladas, o tipo e a classe de cada família, assim como as interfaces de cada componente membro da família, explicitando que parte da interface inflada ele realiza. As famílias podem ser de abstrações, aspectos ou mediadores de hardware. Além das interfaces, para os membros das famílias de mediadores de hardware ainda é necessário especificar a arquitetura alvo, já que eles são os componentes que são dependentes de plataforma no sistema. Para todos os membros de todos os tipos de família é informada uma estimativa de custo. Estas informações tornariam possível a seleção de componentes por ferramentas automáticas através da análise da aplicação, gerando blueprints de configuração, conforme descrito na seção 2.2.3. Todos os métodos são descritos através da assinatura completa. O Anexo C mostra o arquivo DTD que define este modelo.

4.2. Análise do modelo atual

A descrição da máquina alvo está bem estruturada. A descrição da interface dos componentes está representando bem as funções que eles oferecem, porém não existe nenhuma especificação do que o componente requer, informação que é identificada como

sendo de grande importância por vários autores, conforme descrito no Capítulo 3.

Não há nenhuma integração entre a especificação dos componentes e a especificação da arquitetura do sistema. Os programas de configuração interativa atuais lêem apenas o arquivo que descreve a arquitetura do sistema, não tomando nenhum conhecimento dos arquivos que descrevem os componentes. Por causa disto, a listagem dos membros existentes para cada família está redundante, já que é especificada nos dois modelos. Além disso, as restrições impostas pelos componentes, como dependência de um determinado membro de outra família ou necessidades de parâmetros (*inputs*) estão especificadas junto com a arquitetura do sistema, quando na verdade elas deveriam estar especificadas junto com as especificações dos componentes, formando uma base de conhecimento para a criação de várias arquiteturas.

4.3. Um novo modelo para a Base de Conhecimentos de Componentes

Nesta seção, vamos apresentar uma proposta para um novo modelo para a base de conhecimentos do EPOS, visando resolver as deficiências encontradas nos modelos atuais.

4.3.1. Especificação da máquina alvo

A especificação atual da máquina alvo já estava bastante completa e será totalmente reaproveitada, conforme apresentada no Anexo B.

4.3.2. Especificação de Componentes

Conforme estudado no Capítulo 3, é imprescindível que a especificação de um componente indique quais são os requisitos do mesmo, além do que ele fornece para possibilitar a composição automática. Desta forma, nossa especificação irá incluir a especificação das dependências de cada família e membros.

É importante notar a diferença existente entre os componentes na metodologia de Projeto de Sistemas Orientados à Aplicação e os outros modelos estudados, pois na metodologia orientada à aplicação os componentes estão organizados em Famílias. Desta forma, abstrações, aspectos e mediadores de hardware serão igualmente considerados componentes para efeitos de configuração, e as respectivas famílias serão consideradas

famílias de componentes. Alguns pontos serão tratados de forma semelhante para todas as famílias, outros serão tratados de forma diferenciada: o membro de um mediador de hardware é escolhido automaticamente em função da máquina alvo do sistema, e aspectos são aplicados às abstrações, sendo gerados para compilação como traits das famílias.

4.3.2.1. Famílias e Membros

A definição DTD do novo modelo de especificação de componentes está listada no Apêndice A. O modelo irá considerar o seguinte:

- Famílias: A seção *interface* conterá a descrição da interface inflada da família, ou seja, a união de todos os métodos, construtores, tipos e constantes fornecidos por todos os membros da família. A seção *common* especificará apenas o que é fornecido por todos os membros da família, por estar implementado na classe *Common* da mesma, o que geralmente ocorre apenas com tipos e constantes. As famílias terão ainda três novos elementos: *dependency* e *feature* serão utilizados para especificar as características e os requisitos da Família comuns a todos os membros (veja as seções 4.3.2.1 e 4.3.2.2) e *trait* para especificar as entradas necessárias (veja seção 4.3.3).

- Membros da Família: a definição de interface no modelo atual, ou seja, a assinatura dos construtores e métodos que ele realiza da interface inflada continuará sendo representado da mesma forma. O elemento *dependency* irá especificar quais características são necessárias para que o membro possa funcionar (veja as seções 4.3.2.1 e 4.3.2.2), e o *feature* irá especificar as características não funcionais que o membro implementa. Finalmente, o elemento *trait* servirá para indicar as entradas necessárias (veja seção 4.3.3). É importante destacar que a nomeação dos membros segue a convenção: ou é utilizado somente o nome do membro, ou `membro_família`, e que os nomes de membros devem ser únicos entre todas as famílias do repositório – isto é necessário porque o *Analyser* retorna apenas o nome da família e/ou membro sem informações de contexto, e é necessário identificar um membro único somente a partir de seu nome.

Existe uma ferramenta (*epos-newabs*) que lê as descrições de famílias e membros em XML e gera os arquivos de implementação em C++ (arquivos .h e .cc), facilitando o

desenvolvimento. Futuramente, poderá ser criada uma nova ferramenta que gere as descrições da interface inflada da família e dos métodos fornecidos pelo componente de volta para XML a partir do código fonte, permitindo manter ambos sincronizados automaticamente.

Estas definições são suficientes para descrever todos os aspectos estruturais dos componentes. No momento, não iremos trabalhar sobre a descrição comportamental, como sugerido por Silva [Sil00] (veja seção 3.5), porque introduziria uma grande complexidade no modelo e, por enquanto, não é o aspecto mais importante para a validação das composições de componentes.

4.3.2.2. Características (*features*)

Além da descrição estrutural do componente, identificamos a necessidade de especificar as características que o mesmo implementa. Seria uma forma de explicitar a semântica do componente, ou seja, que funcionalidades ou características significativas do domínio são implementadas pelo componente.

Resolvemos adotar um modelo semelhante ao conceito de Feature-Oriented Analysis [KCM+90]. Desta forma, cada componente irá implementar uma ou mais funcionalidades do domínio em questão. Cada Família será considerada a implementação de uma funcionalidade (por exemplo, a família Segment implementa a feature Segment), assim como cada Membro. Além disso, famílias e membros poderão implementar funcionalidades adicionais, especificadas através do elemento *feature*. Por exemplo, um aspecto poderia implementar a funcionalidade *reliable*. Uma *feature* também pode ter um valor, por exemplo, uma característica de um mediador de hardware poderia ser `consumo_energia < 10`.

Assim como as famílias e membros, as features também deverão ter nomes únicos dentro de todo o repositório de componentes.

4.3.2.3. Dependências

Para especificar as dependências funcionais entre famílias, teoricamente deveríamos criar um elemento no qual descrevêssemos a dependência completa, incluindo informações de tipos, constantes, métodos e construtores requeridos. No entanto, verificou-se na prática que, ao longo dos anos de existência do Projeto EPOS, nunca houve ocorrência de um caso em que uma dependência entre membros do repositório fosse definida em termos tão detalhados. Talvez isto ocorra porque a maior parte das famílias mais acessadas diretamente

por componentes do sistema sejam famílias de abstrações uniformes ou mediadores de hardware.

Devido a isso, resolvemos implementar a especificação de dependência através do modelo de funcionalidades (*features*). A seleção de famílias e membros específicos fica condicionada a requisitos encontrados na aplicação, a dependências por features entre as famílias e ao custo de cada membro.

A especificação de features e dependências é a solução criada para substituir o modelo de pré e pós-condições. Esta especificação visa atender às dependências semânticas entre dois componentes, ou seja, mesmo existindo mais de uma realização que fornece todos os métodos necessários por um componente, apenas uma dela é semanticamente adequada.

O elemento *dependency* irá conter apenas um atributo que indica a dependência que um membro tem por suporte à determinada funcionalidade ou cujo valor de determinada característica seja. Por exemplo, o membro Concurrent_Task da família Task tem uma dependência com a família Address_Space, mas não qualquer membro: deve ser um membro que realize mapeamento entre endereços lógicos e físicos. Esta dependência poderia ser especificada como sendo requisitada a funcionalidade *Mapped_AS*. Os membros de Address_Space que realizam o mapeamento conteriam a tag *feature* indicando que implementam a funcionalidade Mapped_AS. Isto impediria a associação entre Concurrent_Task e Flat_AS, funcionalmente (estruturalmente) válida, porém semanticamente inválida, porém permitiria a associação correta de Concurrent_Task com Paged_AS.

A vantagem desta abordagem sobre as pré e pós-condições é que permite especificar dependências em termos de características funcionais, ao invés de especificar diretamente o membro escolhido, priorizando a flexibilidade e expressividade, e gerando novas possibilidades de combinação. Um exemplo comparativo: a pré-condição de Concurrent_Task poderia ser: `Address_Space = Paged_AS`, o que a princípio resolveria a questão. Porém, se futuramente fosse implementado outro membro de Address_Space que também suportasse a Concurrent_Task, por exemplo, Paged_Segmented_AS, e a aplicação exigisse este novo membro por outros requisitos? Esta configuração seria válida, porém a pré-condição do membro Concurrent_Task estaria impedindo sua validação. O uso de *dependency* e *feature* resolve o problema, garantindo que tanto o membro Paged_AS como qualquer futura implementação que venha suportar essa propriedade possa ser utilizado com o Concurrent_Task.

A especificação de uma dependência poderia ainda ser composta, indicando a necessidade de um componente que implemente simultaneamente um conjunto de funcionalidades. Sendo assim, definimos a descrição de uma dependência como uma expressão lógica, contendo os operadores `&&` (e), `||` (ou), `!` (não), `=`, `<`, `<=`, `>`, `>=` (comparativos) e parênteses. Por exemplo, uma dependência poderia ser: `Wireless_Network && reliable`, indicando a necessidade de um componente que forneça a implementação de uma rede sem fio confiável.

É importante destacar ainda a participação dos aspectos no processo de solução de dependências: aspectos podem ser aplicados a abstrações, fornecendo a implementação de funcionalidades requeridas. No exemplo acima, provavelmente existiria uma família Wireless_Network, porém ela não teria suporte implícito a comunicação confiável. No entanto, poderia existir um aspecto que implemente a funcionalidade reliable. A aplicação deste aspecto à família Wireless_Network resultaria no componente desejado.

Existe apenas uma única questão não resolvida por este modelo: quando uma família ou membro é utilizado diretamente no código C++ da implementação de outro membro, a correspondente dependência será transportada para a descrição do componente. Porém, a dependência especifica apenas a funcionalidade requisitada. Teoricamente, esta funcionalidade pode ser implementada de qualquer forma: membro, família, aspecto. Porém, se o uso desta funcionalidade no código C++ do componente dependente for feito diretamente por chamadas às classes, isso obrigará a implementação da funcionalidade como uma família ou um membro com o mesmo nome da funcionalidade. Por exemplo, se a implementação de um membro de alguma família criasse uma instância de uma classe com o nome Mapped_AS, e a dependência especificasse que a funcionalidade Mapped_AS é requerida, o configurador poderia selecionar o membro Paged_AS, que implementa a feature Mapped_AS, e considerar a configuração válida. No momento da compilação, porém, um erro ocorreria, já que a classe Mapped_AS não existiria.

Este é um problema que até o momento não ocorreu, mas a metodologia proposta deixa a possibilidade de que ele ocorra. Caso essa possibilidade venha a se tornar realidade no futuro, com o crescimento do sistema, uma forma de tratá-la deverá ser desenvolvida.

4.3.3. Especificação das entradas de informações pelo usuário

Várias informações precisam ser entradas pelo usuário para que a configuração

possa ser gerada. Substituindo o elemento *input* anteriormente existente, foi criado o elemento *trait*, uniformemente utilizado sempre que é necessária a entrada de alguma informação pelo usuário (atualmente nos elementos *family* e *member* para a definição dos componentes). Estes traits também serão aproveitados para ser o lugar onde o usuário irá habilitar ou desabilitar características configuráveis das famílias. Por último, as ligações de aspectos com as famílias também serão convertidas para *traits,* facilitando o procedimento. Os valores de todos os traits recolhidos durante a etapa de configuração são disponibilizados para o metaprograma estático do framework de componentes no momento da compilação.

4.3.4. Descrição informal

Os elementos *family* e *member* terão um atributo *text* que será um espaço disponível para a descrição informal dos componentes. Esta descrição terá o objetivo de informar o usuário que operará a ferramenta gráfica, auxiliando-o na tarefa de configuração.

4.3.5. Exemplos

Apresentamos abaixo um exemplo de especificação de uma máquina alvo (target) para a arquitetura PC com processador IA32.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE target SYSTEM "target.dtd">

<target>
  <machine name="PC">
    <processor name="IA32" clock="950000000" word_size="32"
               endianess="little" mmu="true" />
    <memory base="0" size="33554432">
      <region base="1048576" size="32505856"/>
    </memory>

    <bus type="PCI"/>
    <device name="Myrinet" class="Network"/>
  </machine>

  <compiler prefix="/usr/share/cross-ia32/" />

  <bootmap
    boot =      "0x00007c00"
    setup =     "0x00100000"
    init =      "0x00200000"
    sys_code =  "0xaff00000"
```

```
    sys_data =  "0xaff40000"
    app_code =  "0x00000000"
    app_data =  "0x00400000"/>

  <memorymap
    base =      "0x00000000"
    top =       "0xffffffff"
    app_lo =    "0x00000000"
    app_code =  "0x00000000"
    app_data =  "0x00400000"
    app_hi =    "0x7fffffff"
    phy_mem =   "0x80000000"
    io_mem =    "0xd0000000"
    sys =       "0xafc00000"
    int_vec =   "SYS + 0x00000000"
    mach1 =     "SYS + 0x00001000"
    sys_pt =    "SYS + 0x00002000"
    sys_pd =    "SYS + 0x00003000"
    sys_info =  "SYS + 0x00004000"
    sys_code =  "SYS + 0x00300000"
    sys_data =  "SYS + 0x00340000"
    sys_stack = "SYS + 0x003c0000"
    mach2 =     "TOP"
    mach3 =     "TOP"/>
</target>
```

A seguir, um exemplo de especificação de uma família de componentes (Address_Space), com a especificação completa de dois membros.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE family SYSTEM "family.dtd">

<family name="Address_Space" type="abstraction"
                  class="uniform" text="Address Space family">
  <interface>
    <constructor />

    <method name="attach" return="Log_Addr">
      <parameter name="seg" type="Segment" />
      <parameter name="addr" type="Log_Addr" />
    </method>

    <method name="detach" return="int">
      <parameter name="seg" type="Segment"/>
      <parameter name="addr" type="Log_Addr"/>
    </method>

    <method name="physical" return="Phy_Addr">
      <parameter name="address" type="Log_Addr"/>
    </method>

    <type name="Log_Addr" type="synonym" />
```

```
    <type name="Phy_Addr" type="synonym" />
  </interface>

  <common>
      <type name="Log_Addr" type="synonym" value="MMU::Log_Addr"/>
      <type name="Phy_Addr" type="synonym" value="MMU::Phy_Addr"/>
  </common>

  <dependency requisit=”MMU” />
  <dependency requisit=”Segment” />

  <member name="Flat_AS" cost="1" type=”exclusive”
          text="Flat Address Space member of Family Address_Space">
    <constructor />

    <method name="attach" return="Log_Addr">
      <parameter name="seg" type="Segment" />
      <parameter name="addr" type="Log_Addr" />
    </method>

    <method name="detach" return="int">
      <parameter name="seg" type="Segment" />
      <parameter name="addr" type="Log_Addr" />
    </method>

    <method name="physical" return="Phy_Addr">
      <parameter name="address" type="Log_Addr" />
    </method>
  </member>

  <member name="Paged_AS" cost="3" type=”exclusive”
          text="Paged Address Space member of family Address_Space">
    <constructor />

    <method name="attach" return="Log_Addr">
      <parameter name="seg" type="Segment" />
      <parameter name="addr" type="Log_Addr" />
    </method>

    <method name="detach" return="int">
      <parameter name="seg" type="Segment"/>
      <parameter name="addr" type="Log_Addr"/>
    </method>

    <method name="physical" return="Phy_Addr">
      <parameter name="address" type="Log_Addr"/>
    </method>

    <feature name="Mapped_AS" />
  </member>
</family>
```

5. CONFIGURAÇÃO DO SISTEMA

Os modelos descritos no capítulo 4 possuem todas as informações necessárias para montar uma configuração do sistema EPOS. Uma configuração será composta pela descrição da máquina alvo (target) e por um conjunto de componentes (famílias ligadas aos membros selecionados para inclusão no sistema).

A configuração do EPOS será armazenada em um arquivo XML gerado automaticamente por ferramentas, sendo desnecessária a edição do arquivo pelo usuário. O propósito de existência deste modelo de arquivo é para armazenamento e reutilização de configurações. O DTD utilizado para geração do arquivo está no Apêndice B.

O elemento *target* do arquivo de configuração corresponde exatamente ao target definido no capítulo 4, e, portanto, é definido na DTD *configuration* como uma importação da DTD anterior. A configuração define um novo elemento *component*. Este elemento corresponde a um componente que irá integrar o EPOS, e é representado pela ligação de uma família com seus membros e valores definidos para os traits.

É importante notar que, embora seja possível incluir vários membros de uma mesma família na geração do sistema (respeitando as regras de tipos – inclusivo ou exclusivo – dos mesmos), a arquitetura, implementada em C++, da ligação entre as interfaces infladas e os membros [Frö01] só permite que um deles esteja ligado à interface inflada por vez. Isto pode criar dificuldades com famílias tipicamente dissociadas, como, por exemplo, Synchronizer: um membro Mutex pode ser utilizado em algum momento pela aplicação, e um membro Semaphore pode ser usado em outro. Se ambas as chamadas forem feitas em termos da interface inflada, o *Configurator* vai corretamente identificar que ambos os membros devem ser incluídos na configuração, porém apenas um deles poderá ser ligado à interface inflada, impossibilitando a criação de uma configuração que satisfaça esta aplicação. No momento, a única solução para isto seria solicitar ao programador que escrevesse suas chamadas aos membros de Synchronizer diretamente sobre as interfaces dos membros ao invés da interface inflada, forçando-o a explicitar sua decisão sobre quais membros utilizou.

A seguir, um exemplo de uma especificação de configuração do sistema EPOS de acordo com o novo modelo, gerada pela ferramenta em desenvolvimento EposConfig (veja capítulo 6).

```
<?xml version='1.0' ?>

<configuration>
  <target>
    <machine name="PC">
      <processor name="IA32" clock="950000000" word_size="32"
                 endianess="little" mmu="true" />
      <memory base="0" size="33554432">
        <region base="1048576" size="32505856"/>
      </memory>
      <bus type="PCI"/>
      <device name="Myrinet" class="Network"/>
    </machine>

    <compiler prefix="/usr/share/cross-ia32/" />

    <bootmap boot="0x00007c00" setup=”0x00100000" init="0x00200000"
             sys_code="0xaff00000" sys_data="0xaff40000"
             app_code="0x00000000" app_data="0x00400000"/>

    <memorymap base="0x00000000" top="0xffffffff"
               app_lo="0x00000000" app_code="0x00000000"
               app_dat ="0x00400000" app_hi="0x7fffffff"
               phy_mem="0x80000000"  io_mem="0xd0000000"
               sys="0xafc00000" int_vec="SYS + 0x00000000"
               sys_pt="SYS + 0x00002000" sys_pd="SYS + 0x00003000"
               sys_info="SYS + 0x00004000"
               sys_code="SYS + 0x00300000"
               sys_data="SYS + 0x00340000"
               sys_stack="SYS + 0x003c0000" mach1=”SYS + 0x00001000"
               mach2="TOP" mach3="TOP"/>
  </target>

  <component family="CPU"/>
  <component family="MMU"/>

  <component family="Address_Space">
    <selectedmember name="Paged_AS" bind="true"/>
  </component>

  <component family="Segment">
    <selectedmember name="Static_Segment" bind="true"/>
  </component>

  <component family="Thread">
    <selectedmember name="Concurrent_Thread" bind="true">
      <trait name=”busy_waiting” type=”boolean” value=”true”/>
    </selectedmember>
  </component>

  <component family="Id">
    <selectedmember name="Pointer" bind="true"/>
  </component>

</configuration>
```

6. A FERRAMENTA EPOSCONFIG

A ferramenta EposConfig tem o objetivo de permitir a edição de uma configuração do sistema operacional EPOS de forma gráfica, cumprindo todas as etapas – desde a análise do código fonte da aplicação até a compilação da versão específica do sistema. Neste trabalho, a implementação foi da etapa de configuração manual até a etapa de geração dos arquivos utilizados para a compilação (chaves de configuração e traits). As partes de análise do código fonte da aplicação e execução do processo de compilação serão implementadas em trabalhos futuros.

Neste capítulo veremos como foi desenvolvida a ferramenta e as principais características de sua arquitetura.

6.1. Protótipo

Inicialmente, um protótipo foi desenvolvido, visando amadurecer a especificação dos requisitos para a ferramenta a ser desenvolvida. Neste protótipo a interface geral foi pré-planejada, e apenas uma função foi implementada: a edição da máquina alvo. A figura 6.1 mostra a interface de edição da máquina alvo do protótipo.

6.2. Requisitos

Os requisitos definidos para o desenvolvimento da ferramenta foram:

- Edição das características da máquina alvo (target) do sistema. Estas informações podem ser importadas de um arquivo modelo de configuração da máquina alvo existente para cada arquitetura.
- Inclusão dos componentes (famílias) que serão incluídos na versão otimizada do sistema, incluindo seleção dos membros e ligação destes à interface inflada.
- Entrada das informações necessárias definidas pelos Traits.
- Geração do arquivo de chaves de configuração e do arquivo de traits utilizado pelo Generator para a compilação da versão otimizada do sistema.

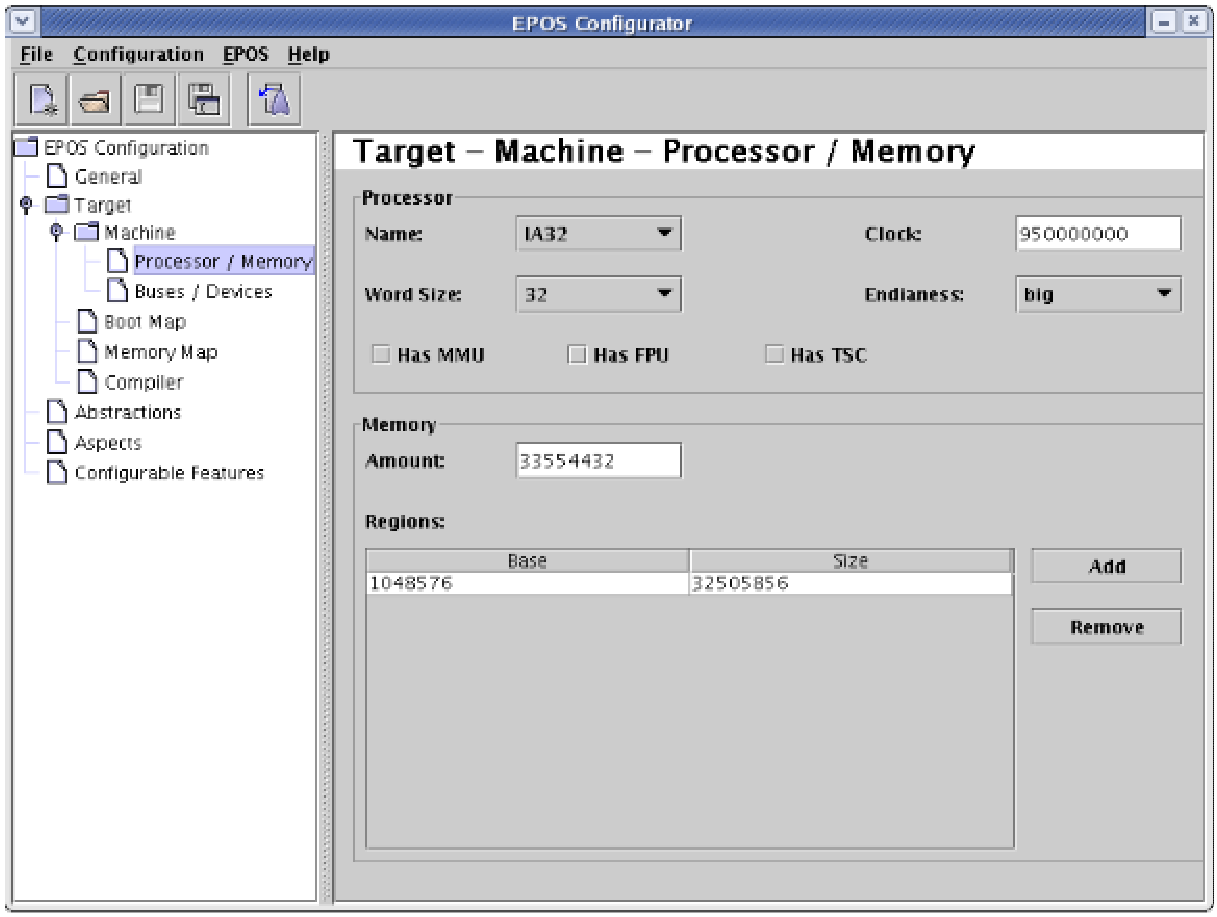


Figura 6.1. Interface do protótipo da ferramenta EposConfig

6.3. Técnicas e ferramentas

Conforme definição inicial, utilizamos a tecnologia Java [SUN03] e diagramas de classe UML [BRJ99] para o desenvolvimento do EposConfig. Escolhemos como ferramenta de trabalho o IDE Eclipse [Ecl04] por sua capacidade de extensão através de plugins: foram instalados plugins para edição de GUIs Swing (CloudGarden Jigloo SWT/Swing GUI Builder [Clo04]) e diagramas UML (Omondo EclipseUML [Omo04]).

Para a leitura e gravação de arquivos XML, utilizamos a biblioteca *commons betwixt* [Apa03] do projeto Jakarta da Apache Software Foundation. Esta biblioteca facilita bastante o desenvolvimento, pois traz APIs que transformam Java Beans em arquivos XML e vice-versa de forma bastante simplificada, utilizando apenas arquivos textuais de descrição dos Java Beans e mínima codificação. Devido à utilização desta biblioteca, o EposConfig não necessita o uso de DTDs – elas continuarão existindo apenas para o uso dos usuários e programadores que precisarem editar manualmente os arquivos XML e para documentação.

6.4. Diagramas de Classes

Para cada um dos modelos descrito nos capítulos 4 e 5, um Diagrama de Classes UML foi escrito. A construção dos modelos finais dos diagramas foi conseguida de forma incremental, começando com os atributos, e gradativamente acrescentando os métodos que foram se mostrando necessários durante a implementação da ferramenta.

A versão final de cada um dos diagramas é apresentada nos Apêndices C, D e E.

6.5. Implementação

A implementação da ferramenta EposConfig partiu da especificação dos diagramas de classes, incluindo-se as classes específicas necessárias para implementação da parte visual. A estrutura foi divida em pacotes Java, conforme abaixo.

- **br.ufsc.lisha.epos.config**: contém as classes principais da ferramenta – os Java Beans que armazenam a configuração do sistema (descrita no capítulo 5), conforme o Apêndice E, e a classe principal EposConfig, uma especialização de JFrame que corresponde à interface principal do programa.
- **br.ufsc.lisha.epos.config.gui**: contém as classes que implementam as interfaces secundárias da ferramenta – edição de máquina alvo e componentes. Estas classes foram implementadas como especializações de JInternalFrame para que pudessem ser exibidas como janelas filhas dentro da janela principal do programa.
- **br.ufsc.lisha.epos.config.gui.images**: contém os recursos gráficos utilizados (imagens utilizadas nos menus, barra de ferramentas e árvore de configuração).
- **br.ufsc.lisha.epos.config.repository**: contém os Java Beans que implementam o modelo de especificação de componentes descrito na seção 4.3.2 e no Apêndice D e a classe Repository, que representa a biblioteca de componentes do EPOS (um conjunto de famílias).
- **br.ufsc.lisha.epos.config.target**: contém os Java Beans que implementam o modelo de especificação da máquina alvo descrito na seção 4.3.1 e no Apêndice C.

6.6. Interface e Uso

Ao iniciar o EposConfig, uma nova configuração é criada. Ela permite abrir/salvar configurações em formato XML. A edição de uma configuração é realizada através da árvore de configuração, à esquerda, e das janelas de edição de configuração que abrem no espaço à direita.

Para editar a descrição da configuração, deve-se clicar duas vezes sobre o item *General* na árvore, ou clicar com o botão direito do mouse sobre ele e escolher a opção *Edit*. A figura 6.2 mostra a interface de edição da descrição da configuração.

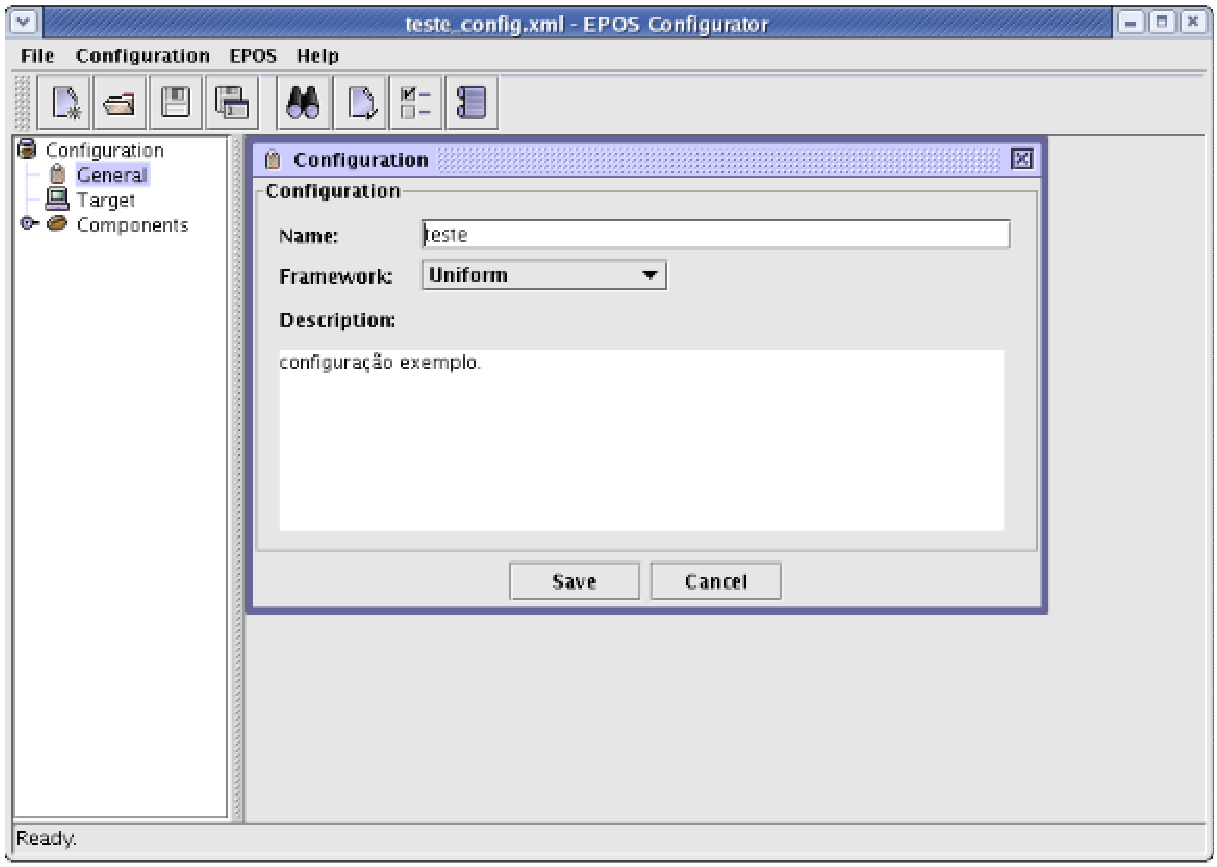


Figura 6.2. Interface do EposConfig – edição da descrição da configuração

De forma semelhante, para editar as configurações da máquina alvo, deve-se clicar duas vezes sobre o item *Target* na árvore, ou clicar com o botão direito do mouse e escolher *Edit*. A figura 6.3 mostra a interface de edição da máquina alvo. Ainda é possível importar ou exportar a configuração da máquina alvo de/para um arquivo XML.

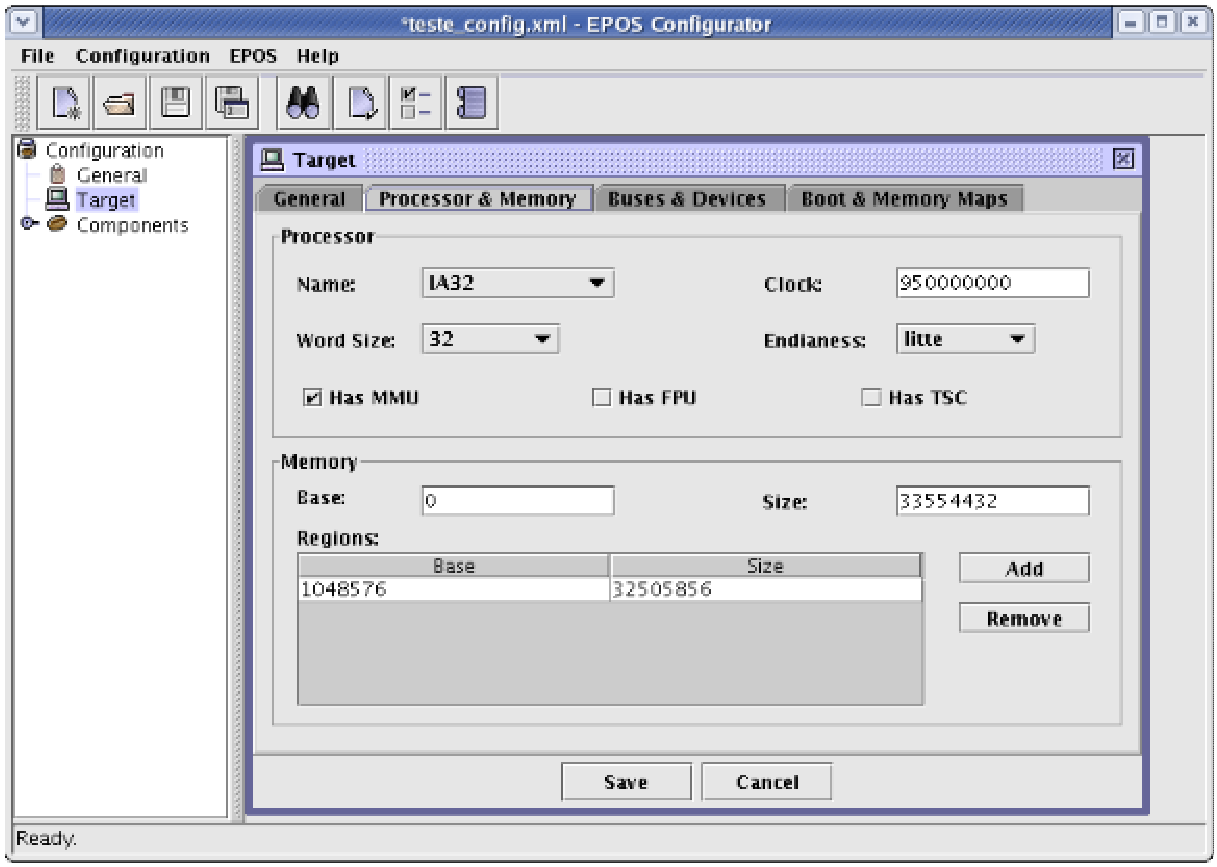


Figura 6.3. Interface do EposConfig – edição da máquina alvo

A maior parte do trabalho de edição de uma configuração é a seleção e configuração dos componentes. Para adicionar um novo componente na configuração, deve-se clicar com o botão direito do mouse no item *Components* na árvore e selecionar a opção *Add*. O programa irá apresentar uma janela na qual o usuário deverá selecionar qual componente deseja adicionar. A lista de componentes é lida do repositório (conjunto de arquivos XML descrevendo as Famílias) no momento em que o programa é iniciado.

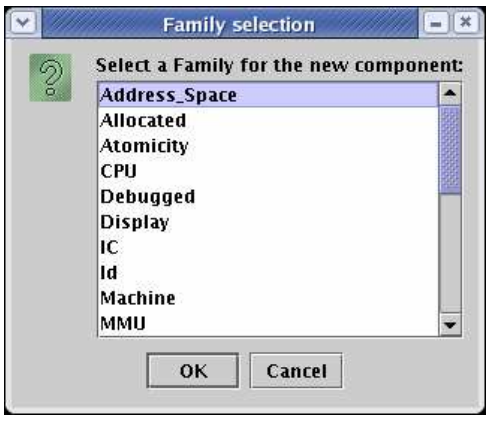


Figura 6.4. Interface do EposConfig – adição de um componente à configuração

Após adicionar um componente, deve-se clicar duas vezes sobre ele na árvore, ou clicar com o botão direito do mouse e selecionar *Edit*, para abrir a janela de edição de componente. Nesta janela será possível visualizar as seguintes características das Famílias: descrição, interface inflada, interface comum, features e dependências, e editar os traits. Também será possível, para cada um dos membros, visualizar tipo, custo, descrição, super-membros, interface, features e dependências, e editar os traits. Nesta janela deverá ser feita a seleção dos membros, ou seja, quais membros desta família deverão estar presentes na configuração final e qual deles deverá ser ligado à interface da família no momento da compilação. A figura 6.5 mostra a interface de edição de um componente (visualizando os membros).

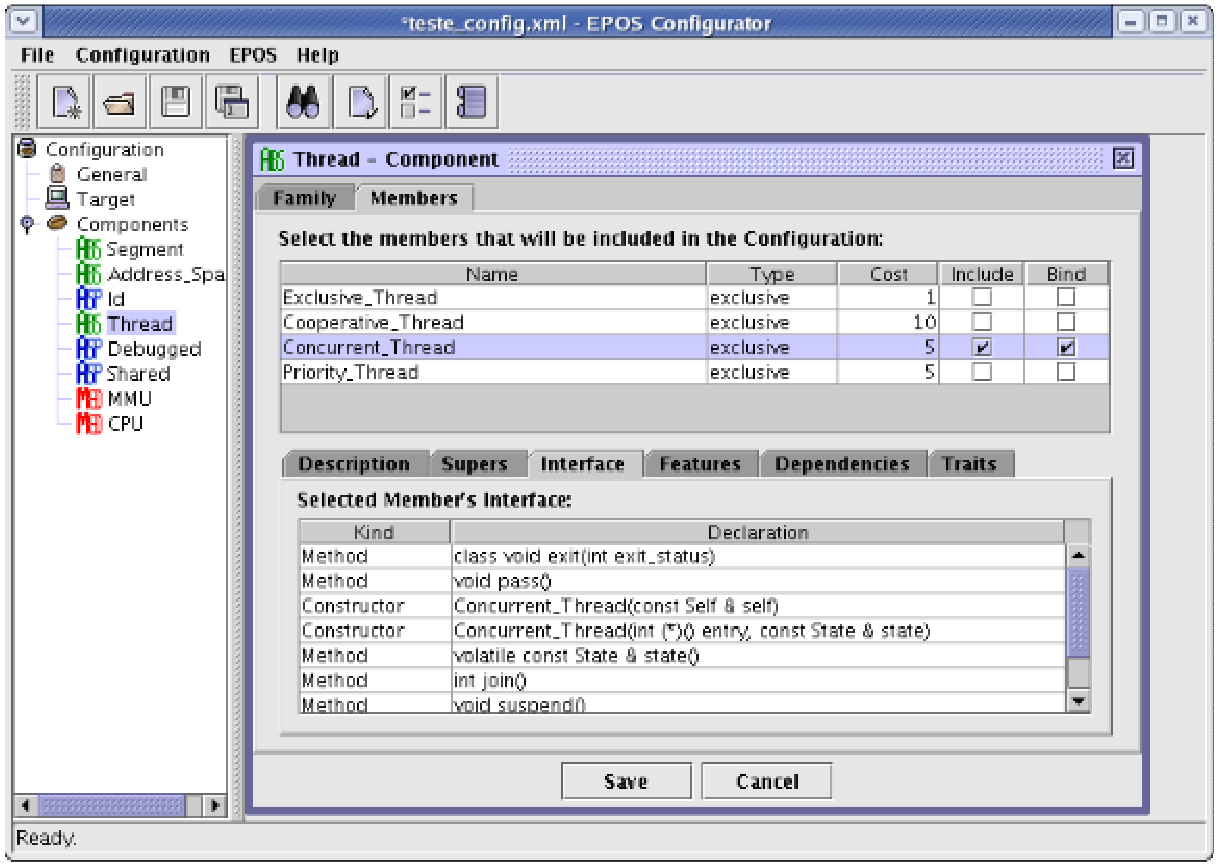


Figura 6.5. Interface do EposConfig – configuração de um componente

6.7. Arquivos de saída

Dois tipos de arquivos são gerados pelo programa: o arquivo de chaves de configuração e o arquivo de traits. Estes dois arquivos são utilizados pelo Framework do EPOS no momento da compilação.

6.7.1. Arquivo de chaves de configuração

O arquivo de chaves de configuração segue o seguinte modelo:

```
// This file was automatically generated by EposConfig.
#ifndef __default_keys_h
#define __default_keys_h

#assert CONF_FRAMEWORK     (Framework_type)
#assert CONF_ARCH          (Arch_name)
#assert CONF_MACH          (Machine_name)

// For each member included in the configuration:
#assert CONF_FAMILY_NAME   (Member_name)
// For each member bound to the family's interface:
#assert BIND_FAMILY_NAME   (Member_name)

#endif
```

6.7.2. Arquivo de Traits

O arquivo de traits segue o seguinte modelo:

```
// This file was automatically generated by EposConfig.
#ifndef __traits_h
#define __traits_h

#include <traits_def.h>
#include <system/config.h>

__BEGIN_SYS

// For each Family xor Family member:
template <> struct Traits<name>: public Traits<void>
{
     // For each trait:
     static const type name = value;
};


__END_SYS

#endif
```

7. CONCLUSÕES

O modelo definido para a especificação dos componentes é bastante completo e permitirá realizar as combinações necessárias para a geração de configurações válidas do EPOS. A ferramenta EposConfig é bastante funcional e permite realizar graficamente, e de forma simples, a criação de uma configuração completa. Esta é uma importante contribuição para a facilidade de trabalho com o EPOS, abrindo espaço para uma maior utilização do sistema no futuro.

As ferramentas utilizadas para desenvolvimento (XML, UML e Java) mostraram-se adequadas e favoreceram a culminação dos objetivos propostos.

Finalizada a primeira parte da construção do sistema completo de geração automática do EPOS, o próximo passo para o Projeto será a extensão do EposConfig, capacitando-o para fazer a seleção automática de componentes de acordo com os requisitos da aplicação. O programa deverá ser capaz de ler o arquivo com a descrição das interfaces requisitadas pela aplicação, gerado pelo *Analyzer*, identificando as features e realizações utilizadas e criando automaticamente a configuração. O trabalho do usuário será apenas refinar alguma seleção de componente e realizar a entrada das características da máquina alvo e dos traits.

Além desta necessária continuação para este trabalho, ainda ficam abertas duas linhas de pesquisa no âmbito da metodologia de Projeto de Sistemas Orientados à Aplicação, que poderão ser futuramente exploradas:

- Inclusão de especificações comportamentais no modelo de especificação de componentes. Esta especificação poderia cobrir restrições como: determinado método de um componente só pode ser chamado se o componente estiver em determinado estado. Este tipo de especificação teria que ser validada por um mecanismo formal, como uma máquina de inferência Prolog ou Redes de Petri.
- Evolução do mecanismo de seleção de membros através da performance. Hoje, isto é feito através da especificação de uma estimativa de custo de cada membro de cada família em forma de overhead, pelo programador. Mecanismos mais elaborados poderiam incluir uma forma automatizada de medir a performance real de cada membro em tempo de execução.

Os objetivos definidos para o trabalho foram plenamente atingidos. Conhecimento a respeito de técnicas de Engenharia de Software para especificação de componentes foram adquiridas tanto em nível individual como para o Projeto. Os modelos propostos e a ferramenta desenvolvida poderão ser imediatamente incorporados ao Projeto, tornando-se o mecanismo oficial de configuração e geração de uma versão otimizada do EPOS.

O término deste abre espaço para novos trabalhos de graduação ou mestrado que visem atender as necessidades ainda não satisfeitas para a criação do processo completo de geração automatizada, principalmente no que diz respeito à configuração automática baseada nos requisitos da aplicação e a integração da ferramenta EposConfig com o processo de compilação do EPOS.

Finalmente, as idéias propostas para o gerenciamento de configuração de sistemas operacionais poderiam ser transportadas, em trabalhos futuros, para outros domínios. O domínio de software aplicativo talvez não tenha uma característica importante do domínio alvo do EPOS – sistemas operacionais embutidos –, que é a possibilidade de definição dos requisitos do sistema operacional de forma estática. Porém, sistemas aplicativos também são desenvolvidos de forma genérica, fazendo com que a versão final sempre contenha muitas funcionalidades que não serão utilizadas por grande parte dos usuários, resultando em necessidades de recursos maiores do que o realmente utilizado. Futuros estudos poderão adaptar as idéias aqui descritas, inaugurando novas linhas de pesquisa que desenvolvam técnicas para configurar estaticamente um sistema aplicativo de acordo com os requisitos dos usuários.

REFERÊNCIAS BIBLIOGRÁFICAS

[Apa03] The Apache Software Foundation. *The Jakarta Project – commons betwixt*, on-line, 2003. http://jakarta.apache.org/commons/betwixt/

[Boo94] BOOCH, Grady. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2ª edição, 1994.

[BRJ99] BOOCH, Grady; RUNBAUGH, James e JACOBSON, Ivar. *The Unified Modeling Language User Guide*. Adison Wesley Longman, 1999.

[Clo04] Cloud Garden. *Jigloo GUI Builder (SWT and Swing) for Eclipse*, on-line, 2004. http://cloudgarden.com/jigloo/

[Ecl04] Eclipse Foundation. *Eclipse.org*, on-line edition, 2004. http://www.eclipse.org/

[Frö01] FRÖHLICH, Antônio Augusto. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001, 200 p., ISBN 3-88457-400-0. (Tese de Doutorado). On-line: http://epos.lisha.ufsc.br/

[HHG90] HELM, Richard; HOLLAND, Ian M. e GANGOPADHYAY, Dipayan. *Contracts: Specifying Behavioral Compositions in Object-oriented Systems*. ACM SIGPLAN Notices, 25(10):169-180, Outubro 1990.

[JCJO93] JACOBSON, Ivar; CHRISTERSON, Magnus; JONSSON, Patrik e OEVERGAARD, Gunnar. *Object-oriented Software Engineering: a Use Case Driven Approach*. Addison-Wesley, 1993.

[KCN+90] KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; e PETERSON, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, E.U.A.., Novembro 1990.

[KLM+97] KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina Videira; LOINGTIER, Jean-Marc e IRWIN, John. *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-oriented Programming'97, volume 1241 of Lecture Notes in Computer Science, páginas 220-242, Jyväskylä, Finlândia, Junho 1997.

[Lar00] LARMAN, Craig. *Utilizando UML e Padrões*. Bookman, Porto Alegre, 2000.

[MDEK95] MAGEE, Jeff; DULAY, Naranker; EISENBACH, Susan e KRAMER, Jeff. *Specifying Distributed Software Architectures*. Fifth European Software Engineering Conference, ESEC ’95. Barcelona, Setembro 1995.

[Mey88] MEYER, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Ola96] ÓLAFSSON, A. e DOUG, B. *On the need for “required interfaces” of components*. Em Special Issues in Object-Oriented Programming, Workshop of the ECOOP, 1996. Proceedings… Linz, 1996.

[OLKM00] VAN OMMERING, Rob; VAN DER LINDEN, Frank; KRAMER, Jeff e MAGEE, Jeff. *The Koala Component Model for Consumer Electronics Software*. IEEE Computer: 78-85, Março 2000.

[Omm02] OMMERING, Rob van. *Building Product Populations with Software Components*. ACM ICSE ’02:255-265, Orlando, Florida, EUA, Maio 2002.

[Omo04] Omondo. *Eclipse – Omondo – The live UML company*, on-line, 2004. http://www.omondo.com/

[Oxf92] Oxford University Press. *The Oxford English Dictionary*, segunda edição, 1992.

[Par76] PARNAS, David Lorge. *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, SE-2(1):1-9, Março 1976.

[Pes97] PESCIO, Carlo. *Template Metaprogramming: Make Parameterized Integers Portable with this Novel Technique*. C++ Report, 9(7):23-26, 1997.

[Pet62] PETRI, C. A. *Kommunikation mit automaten*. Bonn: Institut für Instrumentelle Mathematik, 1962. (Schriften des IIM Nr. 2).

[RBLP91] RUMBAUGH, James; BLAHA, Michael; LORENSON, William e PREMERLANI, William. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Röm01] RÖMKE, Sascha. *Ein XML-basiertes Konfigurationswekzeug für Betribssysteme am Baispiel EPOS*. Studienarbeit, Otto-von-Guericke-Universität, Magdeburg, Alemanha, 2001.

[Sil00] SILVA, Ricardo Pereira. *Suporte ao Desenvolvimento e Uso de Frameworks e Componentes*. Porto Alegre: UFRGS, 2000, 262 pág. (Tese de Doutorado).

[Str97] STROUSTRUP, Bjarne. *The C++ Programming Language*. Addison-Wesley, 3ª edição, 1997.

[SUN03] SUN Microsystems. *The Source for Java Technology*, online edition, 2003. http://java.sun.com/

[Szy96] SZYPERSKI, C. et al. *Summary of the First International Workshop on Component-Oriented Programming*. Em: International Workshop on Component-Oriented Programming (WCOP), 1º, 1996. Proceedings… Linz, 1996.

[Szy97] SZYPERSKI, C. et al. *Summary of the Second International Workshop on Component-Oriented Programming*. Em: International Workshop on Component-Oriented Programming (WCOP), 2º, 1997. Proceedings… Jyväskjylä, 1997.

[Vel95] VELDHUIZEN, Todd L. *Using C++ Template Metaprograms*. C++ Report, 7(4):36-43, Maio 1995.

[W3C98] World Wide Web Consortium. *XML 1.0 Recommendation*, online edition, 1998. http://www.w3c.org/

ANEXO A – Definição DTD do modelo atual de configuração da arquitetura do EPOS

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ENTITY % data.general
        'name    ID       #REQUIRED
        text     CDATA    #REQUIRED
        help     CDATA    #IMPLIED'>

<!ENTITY % data.extended
        '%data.general;
         pre     CDATA    #IMPLIED
         pos     CDATA    #IMPLIED'>

<!ELEMENT configuration     (domain)+>
<!ATTLIST configuration
          name     CDATA    #REQUIRED
          text     CDATA    #REQUIRED
          config_file CDATA  #IMPLIED
          keys_file   CDATA  #IMPLIED>

<!ELEMENT domain            (section)+>
<!ATTLIST domain
          %data.general;>

<!ELEMENT section
                  (interface|multiselect|input|(section)*)+>
<!ATTLIST section
          %data.general;
          pre      CDATA    #IMPLIED>

<!-- *************** REALISATION *************** -->
<!ELEMENT realisation       EMPTY>
<!ATTLIST realisation
          %data.extended;
          class    CDATA    #IMPLIED
          header   CDATA    #IMPLIED>

<!-- *************** INTERFACE *************** -->
<!ELEMENT interface         (realisation+,notneeded?)>
<!ATTLIST interface
          %data.extended;
          default  IDREF    #REQUIRED
          class    CDATA    #IMPLIED>

<!-- *************** MULTISELECT *************** -->
<!ELEMENT multiselect       (realisation)+>
<!ATTLIST multiselect
          %data.extended;
          default  CDATA    #IMPLIED>
```

```
<!-- *************** INPUT *************** -->
<!ELEMENT input             EMPTY>
<!ATTLIST input
          %data.extended;
          default   CDATA    #REQUIRED
          type      NMTOKEN  #REQUIRED
          range     NMTOKEN  #IMPLIED
          length    NMTOKEN  #IMPLIED>

<!-- ***** NOT-NEEDED ***** -->
<!ELEMENT notneeded       EMPTY>
<!ATTLIST notneeded
          name      CDATA    #FIXED 'NOT_NEEDED'
          text      CDATA    #FIXED 'Not needed'
          pre       CDATA    #IMPLIED
          pos       CDATA    #IMPLIED
          help      CDATA    #IMPLIED
          class     CDATA    #IMPLIED
          header    CDATA    #IMPLIED>
```

ANEXO B – Definição DTD do modelo atual de especificação da máquina alvo do EPOS

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT target (machine, bootmap, memorymap, compiler)>
<!ATTLIST target mode (kernel | builtin | library | linux)
                     #REQUIRED>

<!ELEMENT machine (processor, memory, bus*, device*)>
<!ATTLIST machine name (PC | Khomp | iPAQ | Myrinet | RCX |
                        AT90S | AT86RF | NDKStratix) #REQUIRED>

<!ELEMENT processor EMPTY>
<!ATTLIST processor name (IA32 | PPC32 | ARM | LANai | H8 | AVR8 |
                          NIOS16 | NIOS32) #REQUIRED
                    clock     CDATA #REQUIRED
                    word_size (64 | 32 | 16 | 8)   #REQUIRED
                    endianess (little | big) #REQUIRED
                    mmu       (true | false) #IMPLIED
                    fpu       (true | false) #IMPLIED
                    tsc       (true | false) #IMPLIED>

<!ELEMENT memory (region*)>
<!ATTLIST memory base CDATA #REQUIRED
                 size CDATA #REQUIRED>

<!ELEMENT region EMPTY>
<!ATTLIST region base CDATA #REQUIRED
                 size CDATA #REQUIRED>

<!ELEMENT bus EMPTY>
<!ATTLIST bus type  (ISA | PCI | GPIO) #REQUIRED
              clock CDATA #IMPLIED>

<!ELEMENT device EMPTY>
<!ATTLIST device name  CDATA #REQUIRED
                 class (Bridge | Network | Serial | Parallel)
                       #REQUIRED>

<!ELEMENT bootmap EMPTY>
<!ATTLIST bootmap boot     CDATA #REQUIRED
                  setup    CDATA #REQUIRED
                  init     CDATA #REQUIRED
                  sys_code CDATA #REQUIRED
                  sys_data CDATA #REQUIRED>
```

```
<!ELEMENT memorymap EMPTY>
<!ATTLIST memorymap base      CDATA #REQUIRED
                    top       CDATA #REQUIRED
                    app_lo    CDATA #REQUIRED
                    app_code  CDATA #REQUIRED
                    app_data  CDATA #REQUIRED
                    app_hi    CDATA #REQUIRED
                    phy_mem   CDATA #REQUIRED
                    io_mem    CDATA #REQUIRED
                    int_vec   CDATA #REQUIRED
                    sys       CDATA #REQUIRED
                    sys_pt    CDATA #REQUIRED
                    sys_pd    CDATA #REQUIRED
                    sys_info  CDATA #REQUIRED
                    sys_code  CDATA #REQUIRED
                    sys_data  CDATA #REQUIRED
                    sys_stack CDATA #REQUIRED
                    mach1     CDATA #REQUIRED
                    mach2     CDATA #REQUIRED
                    mach3     CDATA #REQUIRED>

<!ELEMENT compiler EMPTY>
<!ATTLIST compiler prefix CDATA #REQUIRED
          extra_flags CDATA #IMPLIED>
```

ANEXO C – Definição DTD do modelo atual de especificação de componentes do EPOS

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT family (interface, common, member+)>
<!ATTLIST family name ID #REQUIRED
                 type (abstraction | mediator | aspect) #REQUIRED
                 class (uniform | incremental | combined |
                        dissociated) #REQUIRED>

<!ELEMENT interface (constructor | method | type | constant)*>
<!ELEMENT common (type | constant)*>

<!ELEMENT member (super | constructor | method | type | constant |
                  trait)*>
<!ATTLIST member name ID #REQUIRED
                 type (exclusive | inclusive) "inclusive"
                 arch (IA32 | PPC32 | ARM | LANai | H8 | AVR8)
                      #IMPLIED
                 mach (PC | Khomp | iPAQ | Myrinet | RCX |
                       AT90S8515 | AT86RF401 ) #IMPLIED
                 cost CDATA #REQUIRED>

<!ELEMENT super EMPTY>
<!ATTLIST super name CDATA #REQUIRED>
<!ELEMENT constructor (EMPTY | parameter*)>
<!ELEMENT method (EMPTY | parameter*)>
<!ATTLIST method name CDATA #REQUIRED
                 return CDATA #IMPLIED
                 qualifiers (class | polymorphic | abstract)
                                                    #IMPLIED>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name CDATA #REQUIRED
                    type CDATA #REQUIRED
                    default CDATA #IMPLIED>

<!ELEMENT type EMPTY>
<!ATTLIST type name CDATA #REQUIRED
               type (class | structure | enumeration | union |
                     synonym | CDATA) #REQUIRED
               value CDATA #IMPLIED>

<!ELEMENT constant EMPTY>
<!ATTLIST constant name CDATA #REQUIRED
                   type CDATA #REQUIRED
                   value CDATA #REQUIRED>

<!ELEMENT trait EMPTY>
<!ATTLIST trait name CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED>
```

APÊNDICE A – Definição DTD do novo modelo de especificação de componentes do repositório do EPOS

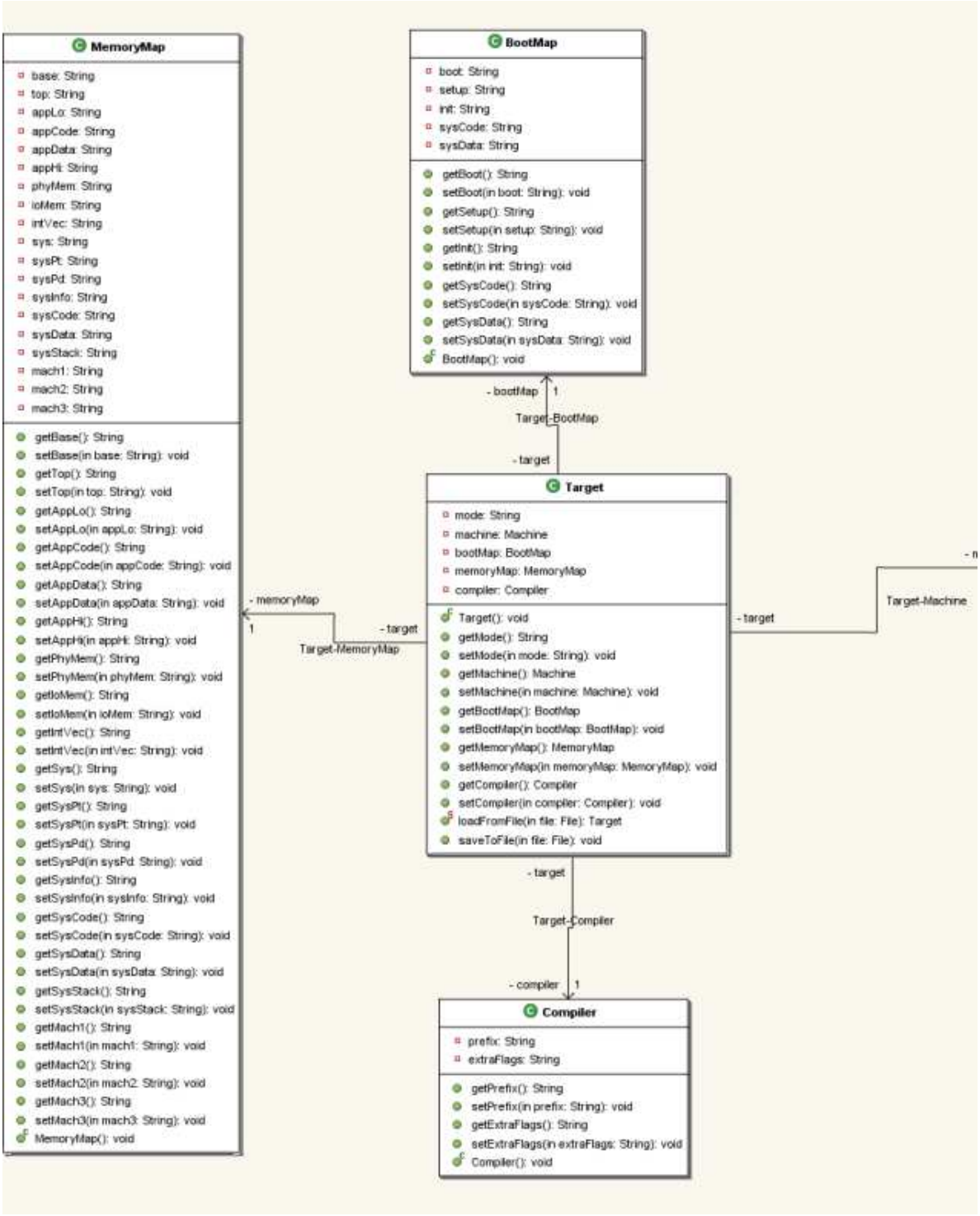
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT family (interface, common, member+,
                  (feature | dependency | trait)*)>
<!ATTLIST family name ID #REQUIRED
                 type (abstraction | mediator | aspect) #REQUIRED
                 class (uniform | incremental | combined |
                        dissociated) #REQUIRED>

<!ELEMENT interface (constructor | method | type | constant)*>

<!ELEMENT common (type | constant)*>

<!ELEMENT member (super | constructor | method | type | constant |
                  feature | dependency | trait)*>
<!ATTLIST member name ID #REQUIRED
                 type (exclusive | inclusive) "inclusive"
                 arch (IA32 | PPC32 | ARM | LANai | H8 |
                       AVR8 | ORBIS32 | NIOS16 | NIOS32 ) #IMPLIED
                 mach (PC | Khomp | iPAQ | Myrinet | RCX |
                       AT90S | AT86RF | OR1200 | NIOSDK16 |
                       NIOSDK32 ) #IMPLIED
                 cost CDATA #REQUIRED>

<!ELEMENT super EMPTY>
<!ATTLIST super name CDATA #REQUIRED>

<!ELEMENT dependency EMPTY>
<!ATTLIST dependency requisit CDATA #REQUIRED>

<!ELEMENT feature EMPTY>
<!ATTLIST feature name CDATA #REQUIRED
                  value CDATA #REQUIRED>

<!ELEMENT constructor (EMPTY | parameter*)>

<!ELEMENT method (EMPTY | parameter*)>
<!ATTLIST method name CDATA #REQUIRED
                 return CDATA #IMPLIED
                 qualifiers (class | polymorphic | abstract)
                            #IMPLIED>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name CDATA #REQUIRED
                    type CDATA #REQUIRED
                    default CDATA #IMPLIED>

<!ELEMENT type EMPTY>
<!ATTLIST type name CDATA #REQUIRED
```

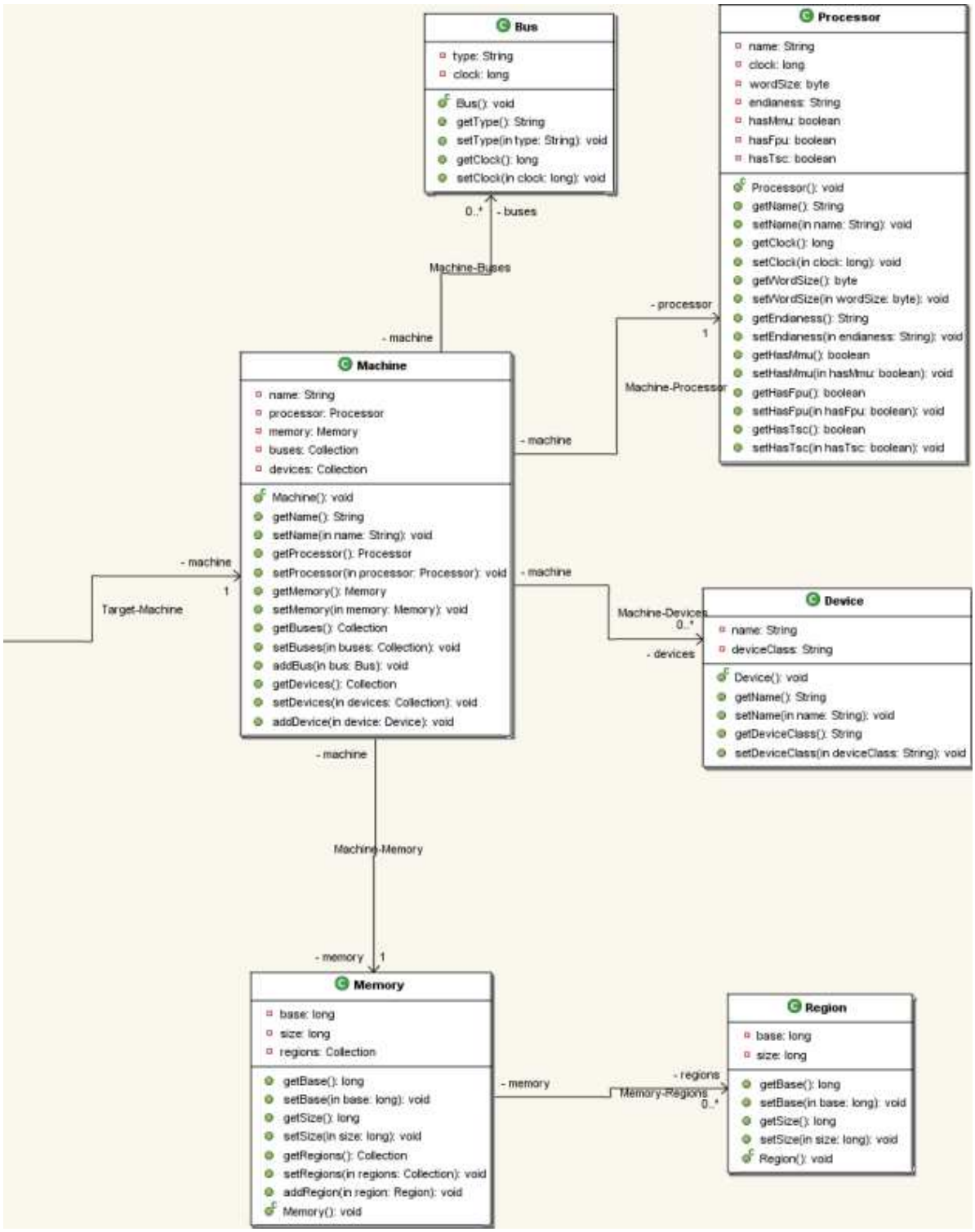
```
                type (class | structure | enumeration | union |
                      synonym | CDATA) #REQUIRED
                value CDATA #IMPLIED>

<!ELEMENT constant EMPTY>
<!ATTLIST constant name CDATA #REQUIRED
                   type CDATA #REQUIRED
                   value CDATA #REQUIRED>

<!ELEMENT trait EMPTY>
<!ATTLIST trait name CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED>
```

APÊNDICE B – Definição DTD do arquivo de armazenamento de Configuração do EPOS

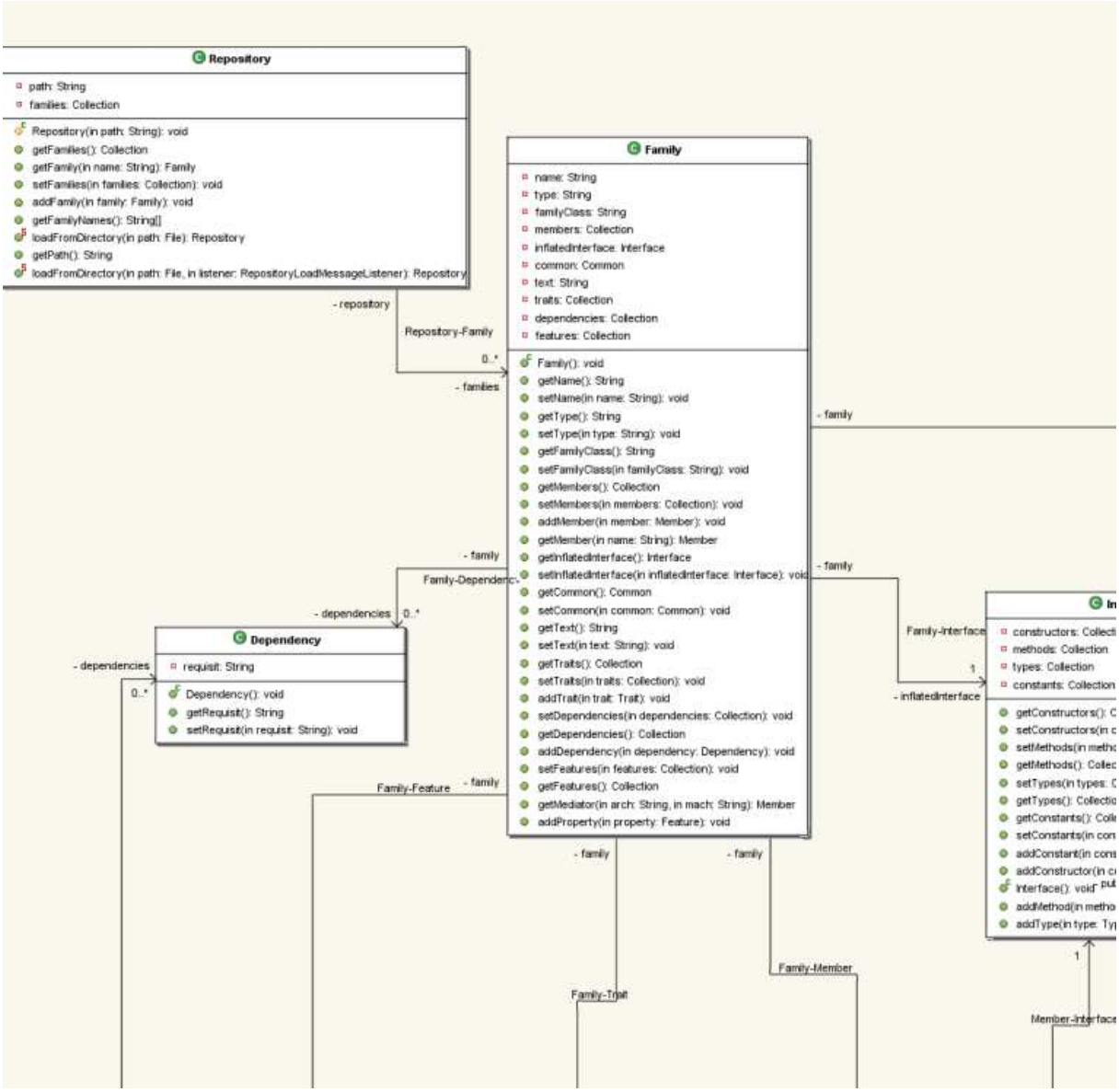
```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!ELEMENT configuration (target, component*)>
<!ATTLIST configuration name CDATA>

<!ENTITY % target SYSTEM "target.dtd">
%target;

<!ELEMENT component (EMPTY | (selectedmember*, trait*))>
<!ATTLIST component family CDATA #REQUIRED>

<!ELEMENT selectedmember (EMPTY | (trait*))>
<!ATTLIST selectedmember name CDATA #REQUIRED
                         bind (true | false) "false">

<!ELEMENT trait EMPTY>
<!ATTLIST trait name CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED>
```

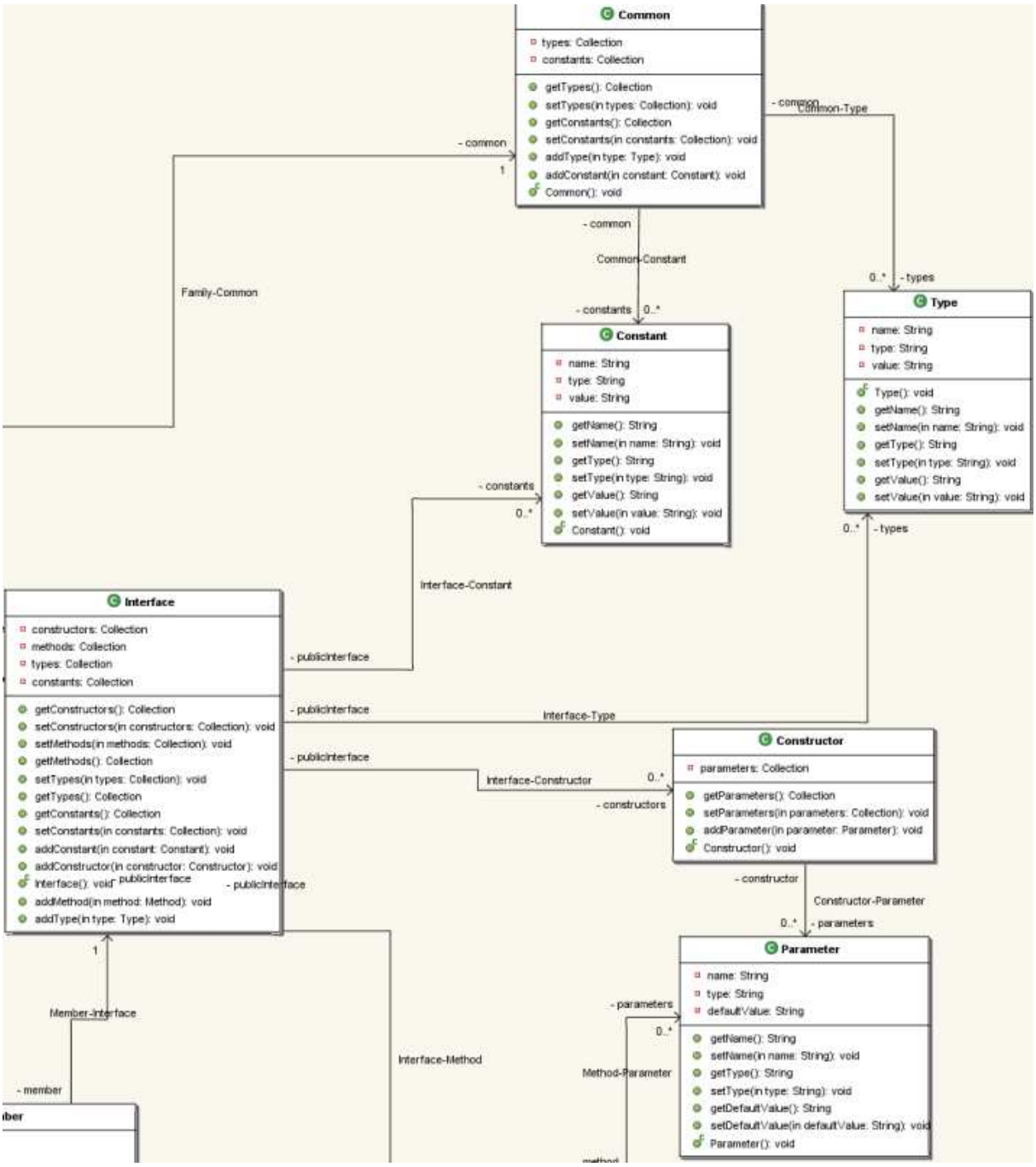
APÊNDICE C – Diagrama de Classes do modelo da máquina alvo do sistema

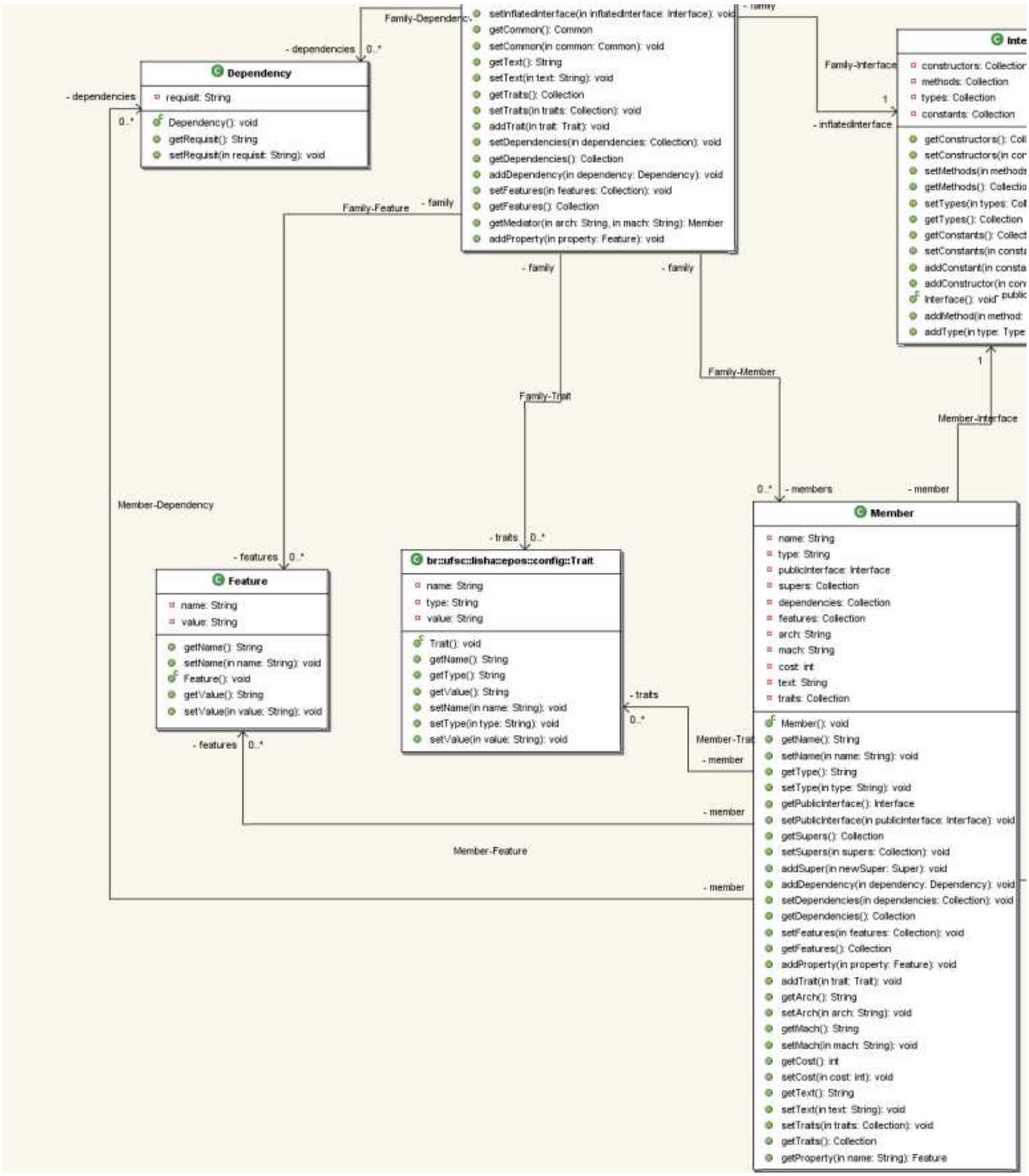


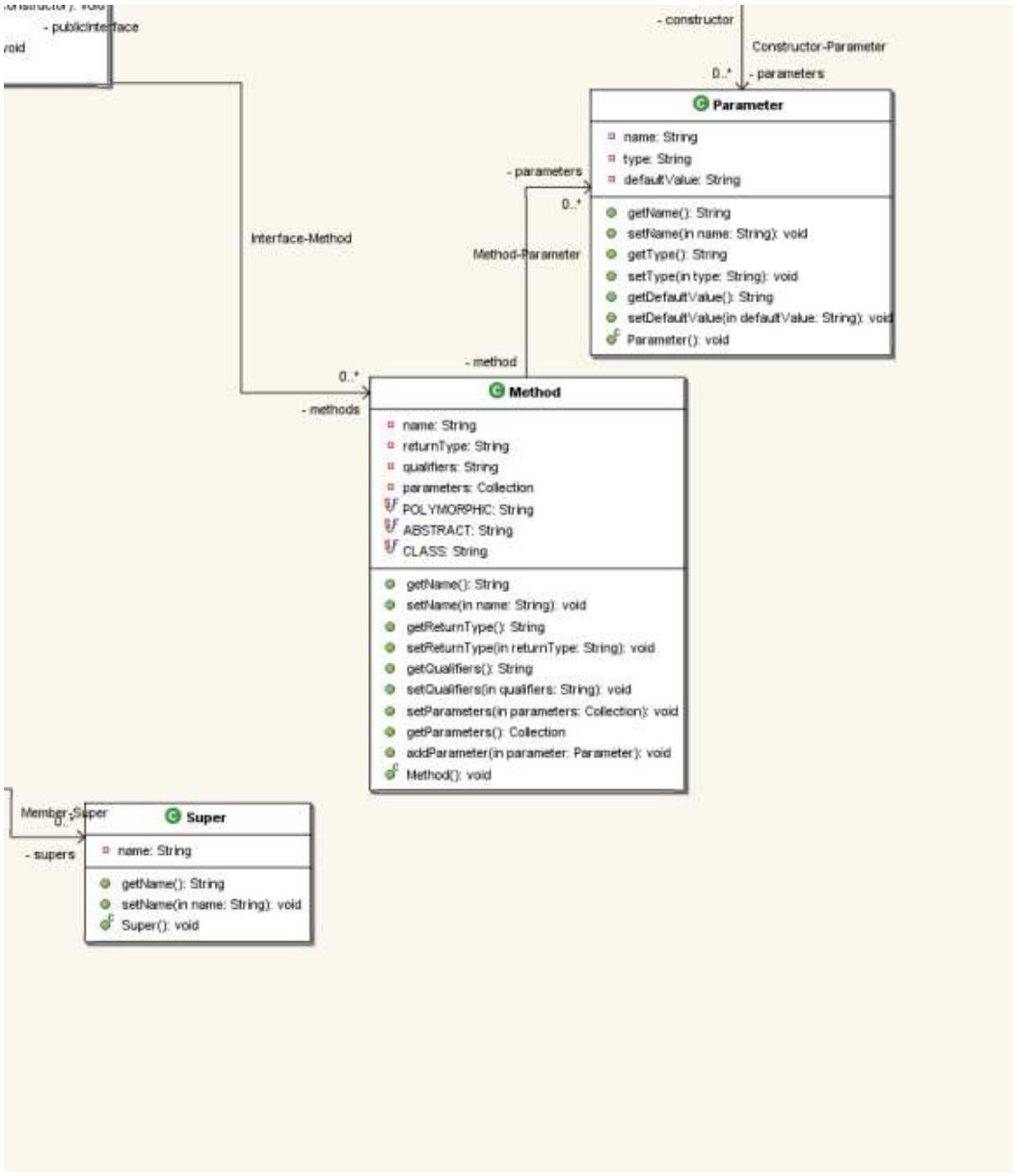

Bus
type: String
clock: long
Bus(): void
getType(): String
setType(in type: String): void
getClock(): long
setClock(in clock: long): void
Processor
name: String
clock: long
wordSize: byte
endianess: String
hasMmu: boolean
hasFpu: boolean
hasTsc: boolean
Processor(): void
getName(): String
setName(in name: String): void
getClock(): long
setClock(in clock: long): void
getWordSize(): byte
setWordSize(in wordSize: byte): void
getEndianess(): String
setEndianess(in endianess: String): void
getHasMmu(): boolean
setHasMmu(in hasMmu: boolean): void
getHasFpu(): boolean
setHasFpu(in hasFpu: boolean): void
getHasTsc(): boolean
setHasTsc(in hasTsc: boolean): void
0..*
- buses
Machine-Buses
- machine
- processor
1
Machine-Processor
Machine
name: String
processor: Processor
memory: Memory
buses: Collection
devices: Collection
Machine(): void
getName(): String
setName(in name: String): void
getProcessor(): Processor
setProcessor(in processor: Processor): void
getMemory(): Memory
setMemory(in memory: Memory): void
getBuses(): Collection
setBuses(in buses: Collection): void
addBus(in bus: Bus): void
getDevices(): Collection
setDevices(in devices: Collection): void
addDevice(in device: Device): void
- machine
1
Target-Machine
- machine
Machine-Devices
0..*
- devices
Device
name: String
deviceClass: String
Device(): void
getName(): String
setName(in name: String): void
getDeviceClass(): String
setDeviceClass(in deviceClass: String): void
- machine
Machine-Memory
- memory
1
Memory
base: long
size: long
regions: Collection
getBase(): long
setBase(in base: long): void
getSize(): long
setSize(in size: long): void
getRegions(): Collection
setRegions(in regions: Collection): void
addRegion(in region: Region): void
Memory(): void
- memory
- regions
Memory-Regions
0..*
Region
base: long
size: long
getBase(): long
setBase(in base: long): void
getSize(): long
setSize(in size: long): void
Region(): void

APÊNDICE D – Diagrama de Classes do modelo de especificação de componentes

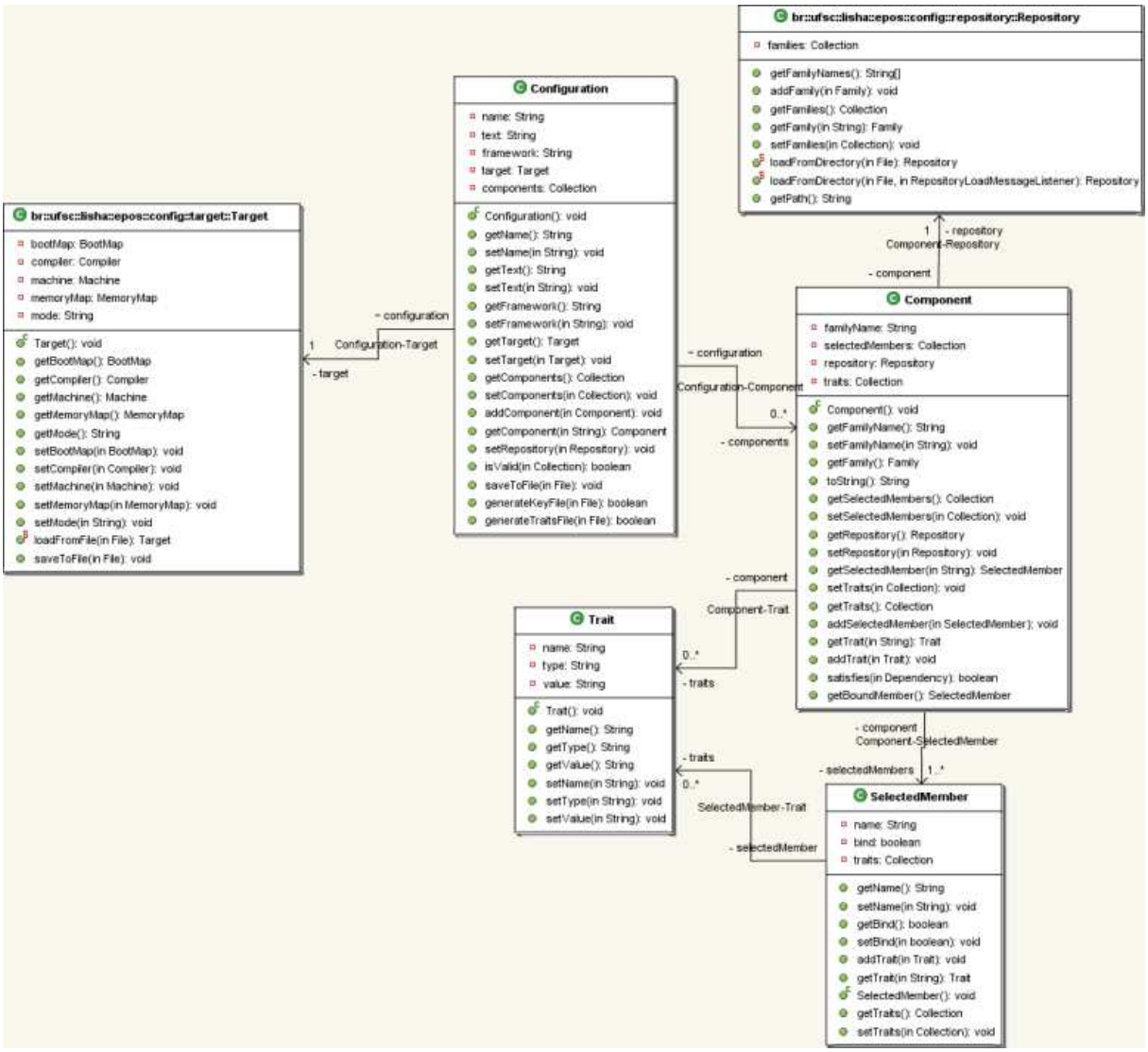



Common
types: Collection
constants: Collection
getTypes(): Collection
setTypes(in types: Collection): void
getConstants(): Collection
setConstants(in constants: Collection): void
addType(in type: Type): void
addConstant(in constant: Constant): void
Common(): void
- common
Common-Type
- common
1
- common
Common-Constant
Family-Common
0..*
- types
- constants
0..*
Type
name: String
type: String
value: String
Type(): void
getName(): String
setName(in name: String): void
getType(): String
setType(in type: String): void
getValue(): String
setValue(in value: String): void
Constant
name: String
type: String
value: String
getName(): String
setName(in name: String): void
getType(): String
setType(in type: String): void
getValue(): String
setValue(in value: String): void
Constant(): void
- constants
0..*
0..*
- types
Interface-Constant
Interface
constructors: Collection
methods: Collection
types: Collection
constants: Collection
getConstructors(): Collection
setConstructors(in constructors: Collection): void
setMethods(in methods: Collection): void
getMethods(): Collection
setTypes(in types: Collection): void
getTypes(): Collection
getConstants(): Collection
setConstants(in constants: Collection): void
addConstant(in constant: Constant): void
addConstructor(in constructor: Constructor): void
Interface(): void
addMethod(in method: Method): void
addType(in type: Type): void
- publicInterface
- publicInterface
- publicInterface
- publicInterface
Interface-Type
Interface-Constructor
0..*
- constructors
Constructor
parameters: Collection
getParameters(): Collection
setParameters(in parameters: Collection): void
addParameter(in parameter: Parameter): void
Constructor(): void
- constructor
Constructor-Parameter
0..*
- parameters
1
Member-Interface
- member
Interface-Method
Parameter
name: String
type: String
defaultValue: String
getName(): String
setName(in name: String): void
getType(): String
setType(in type: String): void
getDefaultValue(): String
setDefaultValue(in defaultValue: String): void
Parameter(): void
- parameters
0..*
Method-Parameter


Dependency
requisit: String
Dependency(): void
getRequisit(): String
setRequisit(in requisit: String): void
- dependencies
0..*
Family-Dependency
setInflatedInterface(in inflatedInterface: Interface): void
getCommon(): Common
setCommon(in common: Common): void
getText(): String
setText(in text: String): void
getTraits(): Collection
setTraits(in traits: Collection): void
addTrait(in trait: Trait): void
setDependencies(in dependencies: Collection): void
getDependencies(): Collection
addDependency(in dependency: Dependency): void
setFeatures(in features: Collection): void
getFeatures(): Collection
getMediator(in arch: String, in mach: String): Member
addProperty(in property: Feature): void
Family-Interface
1
- inflatedInterface
constructors: Collection
methods: Collection
types: Collection
constants: Collection
getConstructors(): Coll
setConstructors(in con
setMethods(in methods
getMethods(): Collectio
setTypes(in types: Col
getTypes(): Collection
getConstants(): Collect
setConstants(in consta
addConstant(in consta
addConstructor(in con
Interface(): void
addMethod(in method:
addType(in type: Type
Family-Feature
- family
Family-Trait
Family-Member
Member-Interface
- member
0..*
- members
Member-Dependency
Feature
name: String
value: String
getName(): String
setName(in name: String): void
Feature(): void
getValue(): String
setValue(in value: String): void
- features
br::ufsc::lisha::epos::config::Trait
name: String
type: String
value: String
Trait(): void
getName(): String
getType(): String
getValue(): String
setName(in name: String): void
setType(in type: String): void
setValue(in value: String): void
- traits
Member
name: String
type: String
publicInterface: Interface
supers: Collection
dependencies: Collection
features: Collection
arch: String
mach: String
cost: int
text: String
traits: Collection
Member(): void
getName(): String
setName(in name: String): void
getType(): String
setType(in type: String): void
getPublicInterface(): Interface
setPublicInterface(in publicInterface: Interface): void
getSupers(): Collection
setSupers(in supers: Collection): void
addSuper(in newSuper: Super): void
addDependency(in dependency: Dependency): void
setDependencies(in dependencies: Collection): void
getDependencies(): Collection
setFeatures(in features: Collection): void
getFeatures(): Collection
addProperty(in property: Feature): void
addTrait(in trait: Trait): void
getArch(): String
setArch(in arch: String): void
getMach(): String
setMach(in mach: String): void
getCost(): int
setCost(in cost: int): void
getText(): String
setText(in text: String): void
setTraits(in traits: Collection): void
getTraits(): Collection
getProperty(in name: String): Feature
Member-Feature


- publicInterface
- constructor
Constructor-Parameter
0..*
- parameters
Parameter
name: String
type: String
defaultValue: String
getName(): String
setName(in name: String): void
getType(): String
setType(in type: String): void
getDefaultValue(): String
setDefaultValue(in defaultValue: String): void
Parameter(): void
- parameters
0..*
Method-Parameter
Interface-Method
- method
0..*
- methods
Method
name: String
returnType: String
qualifiers: String
parameters: Collection
POLYMORPHIC: String
ABSTRACT: String
CLASS: String
getName(): String
setName(in name: String): void
getReturnType(): String
setReturnType(in returnType: String): void
getQualifiers(): String
setQualifiers(in qualifiers: String): void
setParameters(in parameters: Collection): void
getParameters(): Collection
addParameter(in parameter: Parameter): void
Method(): void
Member-Super
0..*
- supers
Super
name: String
getName(): String
setName(in name: String): void
Super(): void

APÊNDICE E – Diagrama de Classes do modelo de armazenamento de configuração do sistema



APÊNDICE F

Código Fonte

APÊNDICE G

Artigo

Configuração automática de Sistemas Operacionais Orientados à Aplicação

Gustavo Fortes Tondello e Antônio Augusto Fröhlich
Laboratório de Integração Software/Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 - 88049-900 Florianópolis - SC - Brasil
http://epos.lisha.ufsc.br/
{tondello | guto}@lisha.ufsc.br

Resumo

Este artigo apresenta uma abordagem para realizar a geração automática de sistemas de tempo de execução baseada na metodologia de Projeto de Sistemas Orientados à Aplicação. Nossa abordagem se baseia em um mecanismo de configuração estática que permite a geração de versões otimizadas do sistema operacional para cada uma das aplicações que irão utilizá-lo. Esta estratégia é extremamente importante no domínio de computação de alto desempenho, já que resulta em ganhos de performance e otimização do uso dos recursos.

1 Introdução

Estudos anteriores demonstraram que sistemas operacionais de propósito geral não fornecem suporte de tempo de execução adequado a aplicações embutidas e de alta performance, já que estes sistemas geralmente causam overhead desnecessário que tem impacto direto sobre a performance das aplicações [1, 15]. Cada classe de aplicações tem seus próprios requisitos quanto ao sistema operacional, e eles devem ser corretamente atendidos.

A metodologia de *Projeto de Sistemas Orientados à Aplicação* (AOSD) [6] é voltada para a criação de sistemas de tempo de execução para aplicações de computação dedicada e de alta performance. Um *sistema operacional orientado à aplicação* é construído a partir da composição de componentes de software que se adaptam perfeitamente aos requisitos da aplicação alvo. Desta forma, evitamos o tradicional efeito "obteve o que não pediu, mas não obteve o que necessitava" presente nos sistemas operacionais genéricos. Isto é particularmente crítico para aplicações embutidas de alta performance, já que elas geralmente devem ser executadas em plataformas com severas restrições de recursos (por exemplo, microcontroladores simples, quantidade limitada de memória, etc).

A metodologia de Projeto de Sistemas Orientados à Aplicação tem sido validada por vários experimentos conduzidos no escopo do projeto EPOS [7], incluindo um sistema de comunicação para clusters de estações interconectados em uma rede MYRINET que possibilitou a execução de aplicações paralelas com performance de comunicação sem precedentes—baixa latência para pequenas mensagens e máxima largura de banda para as maiores [9].

No entanto, para entregar a cada aplicação um sistema de suporte de tempo de execução específico, além de requerer um conjunto de componentes de software bem desenhados, também torna necessária a existência de ferramentas sofisticadas para selecionar, configurar, adaptar e compor estes componentes de forma consistente. Ou seja, o *gerenciamento de configuração* se torna crucial para alcançar a customizabilidade pretendida.

Este artigo descreve o gerenciamento de configuração em sistemas operacionais orientados à aplicação, tomando as estratégias e ferramentas atualmente desenvolvidas para o EPOS como um estudo de caso para a configuração automática de sistemas operacionais para aplicações embutidas e paralelas. As próximas seções descrevem os conceitos básicos da metodologia de Projeto de Sistemas Orientados à Aplicação, uma estratégia para configurar automaticamente um sistema baseado em componentes, uma estratégia de descrição de componentes para este fim, e os protótipos atualmente implementados. Finalmente, serão apresentados os próximos passos planejados para o projeto juntamente com as conclusões dos autores.

2 Projeto de Sistemas Orientados à Aplicação

A idéia de construir sistemas de suporte de tempo de execução através da agregação de componentes de software independentes tem sido usada, com grande sucesso, em uma série de projetos [3, 5, 14, 2]. Entretanto, a engenharia de componentes de software traz várias novas questões, por exemplo: como particionar o domínio do problema para modelar componentes de software realmente reutilizáveis? como selecionar os componentes do repositório que devem ser incluídos em uma instância do sistema específica para uma aplicação? como configurar cada componente selecionado e o sistema como um todo, para construir um sistema otimizado?

A metodologia de Projeto de Sistemas Orientados à Aplicação propõe algumas alternativas para proceder a engenharia de um domínio em busca de componentes de software. A princípio, a decomposição orientada à aplicação do domínio do problema poderia ser obtida seguindo as orientações da *Decomposição Orientada à Objetos* [4]. Entretanto, algumas importantes diferenças devem ser consideradas. Primeiramente, a decomposição orientada à objetos reúne objetos com comportamento similar em hierarquias de classes através da aplicação de análise de variabilidade para identificar como uma entidade especializa a outra. Além de levar ao famoso problema da "classe base frágil" [12], esta política assume que a especialização de uma abstração (isto é, *subclasses*) só são distribuídas com a presença de suas versões mais genéricas (isto é, *superclasses*).

A aplicação da análise de variabilidade utilizando *Projeto baseado em Famílias* [13], produzindo abstrações que podem ser distribuídas independentemente, modeladas como membros de uma família, pode evitar esta restrição e melhorar a orientação à aplicação. Certamente, alguns membros de famílias continuarão sendo modelados como especializações de outros, como no *Projeto Incremental de Sistemas* [10], mas isto não é mais uma regra imperativa.

Uma segunda diferença importante é relacionada com as dependências do ambiente. A análise de variabilidade executada pela decomposição orientada à objetos não enfatiza a diferenciação entre variações que pertencem à essência de uma abstração daquelas que emanam dos cenários de execução. Abstrações que incorporam dependências ambientais têm uma chance menor de serem reusadas em novos cenários, e, dado que um sistema operacional orientado à aplicação será apresentado a novos cenários sempre que uma nova aplicação é definida, permitir que estas dependências ocorram pode prejudicar seriamente o seu uso.

Felizmente, é possível reduzir estas dependências aplicando o conceito chave de *Programação Orientada à Aspectos* [11], isto é, separação de aspectos, ao processo de decomposição. Fazendo isto, é possível diferenciar variações que irão formar novos membros das famílias daqueles que irão gerar aspectos de cenário.

Baseada nestas premissas, a metodologia de Projeto de Sistemas Orientados à Aplicação leva a um procedimento de engenharia de domínio (veja Figura 1) que modela componentes de software com o auxílio de três construções básicas: famílias de abstrações independentes de cenário, adaptadores de cenário e interfaces infladas.

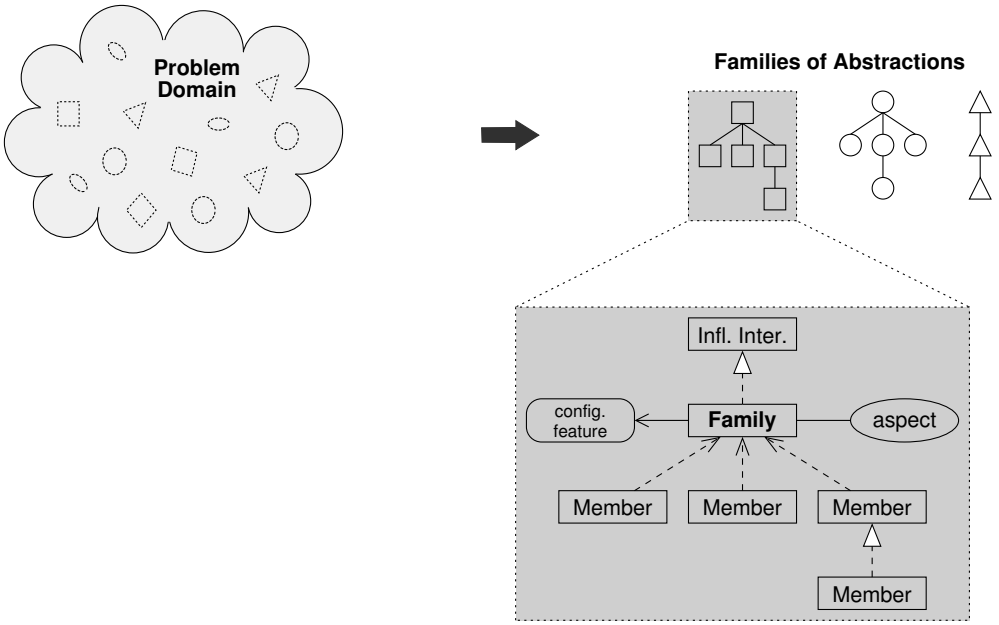


Figura 1: Visão geral da geração de abstrações através da decomposição do domínio orientada à aplicação.

Famílias de abstrações independentes de cenário

Durante a decomposição do domínio, as abstrações são identificadas a partir das entidades do domínio e agrupadas em famílias de acordo com o que têm em comum. Ainda nesta fase, é utilizada a técnica de separação de aspectos para modelar abstrações independentes de cenário, tornando possível sua reutilização em uma variedade de cenários. Estas abstrações são implementadas em seguida, finalmente criando os componentes de software.

A implementação dos membros de uma família de abstrações não fica restrira ao uso de especialização como faríamos na orientação a objetos, embora ela possa ocorrer, quando necessário. Por exemplo, os membros poderiam ser implementados como classes distribuídas em conjunto através de agregação ou composição. Além disso, algumas famílias podem conter membros mutuamente exclusivos, ou seja, apenas um dos membros pode estar presente na configuração do sistema.

Adaptadores de cenário

Como já explicado, a metodologia de Projeto de Sistemas Orientados à Aplicação indica a fatoração de dependências de cenário em *aspectos*, mantendo as abstrações independentes de cenário. No entando, para que esta estratégia funcione, deve existir um meio de aplicar os aspectos às abstraçoes de forma transparente. A abordagem tradicional seria o uso de um *aspect weaver*, porém um *adaptador de cenário* [8] tem as mesmas potencialidades sem necessitar uma ferramenta externa. Um adaptador de cenário encapsula a abstração, intermediando sua comunicação com os clientes dependentes de cenário para realizar as adaptações necessárias.

Interfaces infladas

Interfaces infladas sumarizam as características de todos os membros de uma família, criando uma visão única da família como um “super componente”. Isto permite que os programadores de aplicações escrevam suas aplicações baseadas nas interfaces bem conhecidas, adiando a decisão sobre qual membro da

família deve ser utilizado até que se tenha adquirido conhecimento suficiente sobre a configuração. A ligação de uma interface inflada a um dos membros da família pode então ser feita por ferramentas automáticas de configuração que identifiquem quais características de uma família foram usadas, podendo escolher a realização mais simples que implementa o subconjunto da interface necessário, no momento da compilação.

3 Configuração de Componentes de Software

Um sistema operacional projetado de acordo com as premissas de Projeto de Sistemas Orientados à Aplicação, além de todos os benefícios conseguidos através da engenharia de componentes de software, tem a vantagem adicional de ser adequado para geração automática. O conceito de interface inflada habilita um sistema operacional orientado à aplicação para ser automaticamente gerado a partir de um conjunto de componentes de software, já que as interfaces infladas servem como um tipo de especificação de requisitos para o sistema que deve ser gerado.

Uma aplicação escrita baseada em interfaces infladas pode ser submetida a uma ferramenta que pesquisa por referências às interfaces, descobrindo as características de cada família necessárias para suportar a aplicação em tempo de execução. Esta tarefa é realizada por uma ferramenta, o `analyzer`, que gera uma especificação de requisitos na forma de declarações parciais de interface de componente, incluindo métodos, tipos e constantes utilizados pela aplicação.

A especificação produzida pelo `analyzer` é utilizada para alimentar uma segunda ferramenta, o `configurator`, que consulta uma base de dados para criar a descrição da configuração do sistema. Esta base de dados contém informação sobre cada componetne no repositório, assim como as dependências e regras de composição que são usadas pelo `configurator` para construir uma árvore de dependência. Adicionalmente, cada componente no repositório está marcado com uma estimativa de "custo", para que o `configurator` possa escolhar a opção "mais barata" sempre que dois ou mais componentes satisfaçam a dependência. A saída do `configurator` consiste em um conjunto de chaves de configuração que definem as ligações das interfaces infladas às abstrações e ativam os aspectos de cenário identificados como necessários para satisfazer as restrições impostas pela aplicação alvo ou pelo cenário de execução.

O último passo do processo de geração é realizado pelo `generator`. Esta ferramenta traduz as chaves produzidas pelo `configurator` para parâmetros para um framework de componentes metaprogramado estaticamente, causando a compilação de uma instância específica do sistema. Uma visão geral do processo é apresentada na Figura 2.

4 Descrição de Componentes de Software

A estratégia utilizada para descrever componentes em um repositório e suas dependências é importante para tornar possível o processo de configuração descrito. A descrição dos componentes deve ser completa o suficiente para que o `configurator` seja capaz de identificar automaticamente quais as abstrações que melhor satisfazem os requisitos da aplicação, e isto sem gerar conflitos ou configurações e composições inválidas.

A estratégia proposta abaixo para descrever famílias pode sem dúvida ser considerada para especificar componentes, já que ela engloba muitas das informações necessárias para implementar componentes, incluindo suas interfaces e relacionamentos com outros componentes. Ela é baseada em uma linguagem declarativa de descrição, implementada sobre a *Extensible Markup Language* (XML) [16] e visa descrever individualmente as famílias de abstrações[1]. Os elementos mais significativos da lingua-

[1] Uma descrição completa do repositório de componentes é obtida pela mesclagem das descrições individuais das famílias.

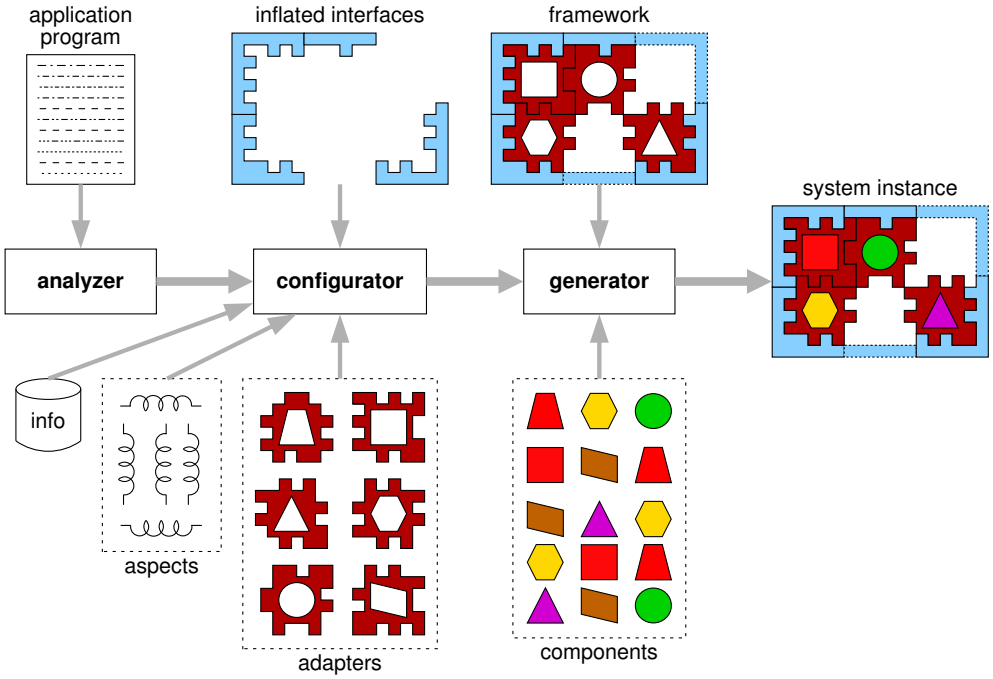


Figura 2: Uma visão geral das ferramentas envolvidas na geração automática do sistema.

gem serão explicados a seguir, tomando como base os fragmentos correspondentes do *Document Type Definition* (DTD) [16].

4.1 Famílias de abstrações

A declaração de uma família de abstrações em nossa linguagem consiste na declaração da interface inflada da família, um conjunto opcional de dependências, um conjunto opcional de features, um conjunto opcional de traits, seu pacote comum e um conjunto de membros (componentes de software), desta forma:

```
<!ELEMENT family (interface, dependency*, feature*, trait*, common, member
    +) >
```

A interface inflada de uma família, como explicada anterirmente, sumariza as características de toda a família, e é especificada como:

```
<!ELEMENT interface (type, constant, constructor, method) *>
```

O pacote comum de uma família é composto por declarações de tipos e constantes comuns a todos os membros da família. É especificado como:

```
<!ELEMENT common (type, constant) *>
```

O elemento `member`, mostrado abaixo, é usado para descrever cada um dos membros da família. Ele está no centro do processo de configuração automática, possibilitando às ferramentas fazer a seleção correta enquanto procuram por realizações para as interfaces infladas. Um membro de família é declarado como:

```
<!ELEMENT member (super, interface, dependency, feature, trait, cost) *>
```

O elemento `super` possibilita que um membro herde declarações de outros membros na família, permitindo a criação de famílias incrementais como no *Projeto Incremental de Sistemas* [10]. A interface de um membro designa uma realização parcial ou total da interface inflada da família em termos dos elementos `type`, `constant`, `constructor` e `method`. O elemento `trait`, que também pode ser especificado para a família como um todo, designa uma informação configuração que pode ser modificada pelos usuários, através de ferramentas de configuração, para influenciar a instanciação de um componente[2]. Um trait de um componente pode também especificar parâmetros de configuração que não podem ser deduzidos automaticamente, como o número de processadores em uma máquina alvo ou a quantidade de memória disponível.

Adicionalmente, cada membro da família está marcado com uma estimativa de custo relativa (`cost`), que é usada pelas ferramentas de configuração no caso em que múltiplos membros satisfaçam as restrições para realizar a interface inflada da família em um cenário de execução determinado. Esta estimativa de custo é bastante simplista, consistindo basicamente em uma estimativa de overhead feita pelo desenvolvedor do componente. Modelos de custo mais sofisticados, incluindo feed-back das ferramentas de configuração, são planejados para o futuro.

4.2 Dependências

Embora nós possamos usar o `analyzer` para descobrir as dependências da aplicação sobre as interfaces das famílias, esta ferramenta não pode ser usada para descobrir as dependências que a implementação de uma família tem por outras. Sendo assim, este tipo de dependência deve ser explicitado pelo programador através do elemento `dependency`. Esta dependência pode ocorrer para a família inteira ou apenas em membros, individualmente.

Escolhemos utilizar um modelo baseado em features para descrever as dependências entre componentes. Para satisfazer as dependências, cada família implementa uma feature com o mesmo nome da família, e cada membro de uma família implementa uma feature com o mesmo nome do membro. Famílias e membros podem ainda implementar features adicionais utilizando o elemento `feature`.

Por exemplo, considere uma família de abstrações de redes sem fio. Alguns membros podem declarar uma feature “reliable”, tornando-os aptos a suportar uma aplicações cujo cenário de execução demande comunicação confiável. Similarmente, membros de uma família de protocolos de comunicação poderiam especificar a dependência por uma infra-estrutura de rede sem fio confiável, enquanto outros poderiam implementar a feature eles mesmos.

Uma feature tem um nome e, opcionalmente, um valor. O nome deve ter relação com uma funcionalidade significativa no domínio da aplicação. Considerando o exemplo acima, poderíamos especificar a feature “reliable” de uma rede sem fio desta forma:

```
<family name="Wireless_Network">
  <interface>...</interface>
  <common>...</common>
  <member name="Wi-Fi">
    <interface>...</interface>
    <feature name="reliable" />
  </member>
</family >
```

e a dependência na família de protocolos assim:

[2]Traits são disponibilizados para os metaprogramas estáticos que formam o framework de componentes, no momento da compilação.

```
<family name="Wireless_Protocol">
  <interface>...</interface>
  <dependency feature="Wireless_Network"/>
  <common>...</common>
  <member name="Active_Message">...
    <interface>...</interface>
    <dependency feature="Wireless_Network && reliable" />
  </member>
</family>
```

É importante mencionar que o fato do membro `Active_Message` da família `Wireless_Protocol` requerer uma rede sem fio confiável não exclui automaticamente membros de `Wireless_Network` que não implementem esta feature: o `configurator` iria primeiro checar a disponibilidade de um aspecto de cenário que pudesse ser aplicado à rede não confiável para fazê-la funcionar como uma confiável. No caso do EPOS, este aspecto de cenário existe e habilitaria a integração correta dos componentes.

5 Ferramentas

No momento, temos protótipos de implementação do `analyzer` para aplicações escritas em C++ e JAVA. Estas ferramentas são capazes de analisar um programa de entrada e produzir uma lista das interfaces das abstrações do sistema (infladas ou não) utilizadas pelo programa, identificando quais métodos foram invocados e, no caso do JAVA, em qual escopo foram invocados.

Esta informação serve como entrada para o `configurator`, que está atualmente em desenvolvimento. O `configurator` é de fato implementado por duas ferramentas. A primeira é responsável por executar o algoritmo que seleciona quais membros de cada família devem ser incluídos na versão customizada do sistema. Este algoritmo consite em ler os requisitos encontrados pelo `analyzer` e compará-los com as interfaces de cada membro da família, especificadas no repositório. Sempre que um novo membro é selecionado, suas dependências são recursivamente verificadas, incluindo na configuração quaisquer membros de outras famílias necessários para satisfazê-las. A segunda parte do `configurator` é uma ferramenta gráfica que permite que o usuário visualize uma configuração gerada automaticamente, fazendo ajustes manuais, se necessário. Além disso, o usuário deverá entrar algumas informações importantes não descobertas automaticamente: a configuração da máquina alvo (arquitetura, processador, memória, etc) e os valores dos traits de cada componente.

Por último, as chaves de configuração geradas pelo `configurator` são utilizadas pelo `generator`, implementado sobre o *GNU Compiler Collection*, para compiler o sistema e gerar uma imagem de boot.

6 Limitações

O mecanismo de chaves de configuração utilizado para ligar as interfaces infladas às implementações reais de um de seus membros somente permite a conexão de cada interface inflada a um membro por vez.

Isto pode limitar o uso de famílias de componentes tipicamente dissociadas, como, por exemplo, a família `Synchronizer`. Uma parte do programa alvo poderia usar o membro `Mutex` da família, enquanto outra parte poderia usar o membro `Semaphore`. Neste caso, o `configurator` corretamente identificaria que dois membros teriam que ser selecionados para satisfazer os requisitos por esta família. No entanto, seria impossível conseguir isto, já que somente é possível ligar um deles à interface inflada da família `Synchronizer` por vez.

A única solução para esta situação existente no momento é solicitar ao programador da aplicação que escreva as chamadas aos sincronizadores diretamente sobre as interfaces dos membros, ao invés de usar

a interface inflada da família. Isto, no entanto, força o programador a explicitar suas decisões sobre qual tipo de sincronizador está usando.

7 Trabalhos futuros

Nós estamos finalizando a implementação do `configurator` que será capaz de gerar automaticamente a configuração de uma versão customizada do EPOS. Trabalhos futuros poderiam refinar a especificação e implementação do modelo de configuração do sistema em dois aspectos:

- Inclusão de especificação comportamental no modelo de descrição de componente. Esta especificação cobriria dependências como: algum método de um componente somente pode ser invocado se o componente está em determinado estado. Este tipo de especificação teria que ser validado por algum tipo de mecanismo formal, como uma máquina de inferência Prolog ou Redes de Petri.

- Evolução do mecanismo usado para selecionar membros pela performance. Hoje esta tarefa é realizada utilizando a especificação de uma estimativa de custo de cada membro de uma família pelo programador, em forma de overhead. Mecanismos mais elaborados poderiam incluir uma forma automatizada de medir a performance real de cada componente em tempo de execução.

8 Conclusões

Neste artigo apresentamos uma alternativa para realizar a geração automática de sistemas de tempo de execução tomando como base uma coleção de componentes de software desenvolvidos de acordo com a metodologia de Projeto de Sistemas Orientados à Aplicação. A abordagem proposta consiste em uma nova linguagem de descrição de componentes e um conjunto de ferramentas de configuração que são capazes de automaticamente selecionar e configurar componentes para montar um sistema de suporte de tempo de execução orientado à aplicação.

As ferramentas de configuração descritas estão na fase final de desenvolvimento e permitem a exposição das bibliotecas do sistema para os programadores das aplicações através de um repositório de componentes reutilizáveis descrito por suas interfaces infladas, as quais são automaticamente ligadas a uma realização em tempo de compilação. Isto é possível devido ao modelo de especificação de componentes que contém todas as informações necessárias para gerar configurações válidas e otimizadas para cada aplicação.

Esta arquitetura usa estratégias baseadas em componentes para gerar versões do sistema operacional otimizadas para as aplicações alvo, garantindo que os níveis de performance e otimização de uso de recursos para aplicações embutidas e paralelas sejam certamente melhores que aqueles conseguidos com a utilização de sistemas operacionas de propósito genérico.

Referências

[1] Thomas Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 92–94, Key Biscayne, U.S.A., April 1992.

[2] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[3] Danilo Beuche, A. Guerrouat, H. Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, St Malo, France, May 1999.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[6] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[7] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-oriented Operating Systems – the EPOS Aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, September 1999.

[8] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[9] Antônio Augusto Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations. In *Proceedings of 8th International Conference on High Performance Computing and Networking*, pages 417–426, Amsterdam, The Netherlands, May 2000.

[10] A. Nico Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[12] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

[13] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[14] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, U.S.A., October 2000.

[15] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[16] World Wide Web Consortium. *XML 1.0 Recommendation*, online edition, February 1998. [http://www.w3c.org].