DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9707

Configuração Dinâmica de Sistemas

Dilma M. da Silva
Markus Endler

agosto 97

Configuração Dinâmica de Sistemas

Dilma M. Silva and Markus Endler
Departamento de Ciência da Computação,
IME, Universidade de São Paulo,
Rua do Matão 1010, 05508-900 São Paulo, Brasil
{dilma,endler}@ime.usp.br

Resumo

Este relatório apresenta um resumo do estado da arte em configuração dinâmica de sistemas de software. Em particular, são discutidos os principais problemas da configuração dinâmica de sistemas, e apresentados os principais mecanismos utilizados para realizá-la tanto no nível do núcleo de sistemas operacionais como no nível de programas de aplicação. Além disto são dados exemplos de sistemas operacionais e ambientes de desenvolvimento que dão suporte à configuração dinâmica de software.

Abstract

This technical report presents the state of the art in dynamic configuration of software systems. It contains a discussion of the main problems of dynamic configuration and shows the main mechanisms used to implement it, both at the level of operating system kernels and the level of application programs. Moreover, many examples of operating systems and software development environments supporting dynamic configuration are given.

Sumário

1 Introdução

Nesta seção daremos uma introdução ao conceito de configuração dinâmica enfatizando o contexto e as diferentes formas em que a mesma tem sido empregada. A seguir discutiremos as principais aplicações da facilidade de configuração dinâmica. Por fim, discutiremos as principais tarefas e problemas relacionados à sua implementação.

Reconfiguração dinâmica de um sistema pode ser definida como a alteração de algumas de suas partes durante a sua operação. Devido à longevidade e às exigências de alta disponibilidade e flexibilidade de certos sistemas, configuração dinâmica vem sendo incorporada em um número cada vez maior de sistemas. Configuração dinâmica faz parte de um conjunto mais amplo de tarefas associadas à manutenção e à evolução de software e está bastante relacionada com outras tarefas do ciclo de desenvolvimento de software, como a especificação e o projeto de sistemas, a integração de módulos, o controle de versões e o gerenciamento de sistemas.

Na forma mais rudimentar, reconfiguração dinâmica pode ser realizada através de *patches*, nos quais uma parte do código original (geralmente no nível de instruções assembler) é substituído por um novo código, ou através de *hooks*, que permitem adicionar código em pontos pré definidos do programa. Outra possível forma de configuração dinâmica é através de programas mutáveis, que podem ser implementados em certas linguagens, como por exemplo Lisp. No contexto de programação orientada a objeto, tal facilidade tem sido denominada *reflexão computacional*.

Uma terceira abordagem para a configuração dinâmica é derivada da noção de *programação em granularidade grande*(programming-in-the-large)[15], na qual se dá ênfase ao processo de integração ou interconexão de módulos de programa (implementados em uma ou diversas linguagens de programação) para a formação de sistemas maiores. Baseado nesta noção e devido a sua boa adequação a ambientes computacionais multiprocessadores e distribuídos, surgiu a idéia de restringir a configuração dinâmica ao nível da seleção, instanciação e interconexão de módulos. Comparado com as duas outras formas mencionadas acima, esta abordagem tem a vantagem de facilitar a identificação do efeito das mudanças sobre a parte restante do sistema.

1.1 Formas de Configuração Dinâmica

Quanto às suas características, configuração dinâmica pode ser *interativa*, quando as alterações a serem efetuadas são definida na interação com um usuário, ou *pre-definida*, quando a modificação é predeterminada por um programa (ou script) de reconfiguração. A reconfiguração também pode diferir com relação à granularidade e à natureza dos objetos que podem sofrer alterações. Estes podem variar desde trechos de código e procedimentos (em patches), variáveis ou estruturas de dados, objetos, processos, interfaces, indo até subsistemas completos. Quanto à descrição das reconfigurações, estas podem estar determinadas pelas instruções de uma linguagem de configuração, de uma linguagem de programação, e/ou por um conjunto de serviços disponíveis na forma de bibliotecas ou um sistema de apoio a execução. Além disto, distingue-se se a reconfiguração ocorre em código executando em modo protegido (*modo kernel*) ou em *modo usuário*. Com relação ao sistema que é o objeto da configuração, este pode ser um programa de aplicação, um serviço de sistema executando em modo usuário ou uma função/módulo do núcleo de um sistema operacional.

1.2 Aplicações

A utilidade de mecanismos de configuração dinâmica na maioria das vezes está ligada à necessidade de manter um sistema (ou programa) executando ininterruptamente, seja devido ao alto custo envolvido com sua parada ou com a necessidade de garantir uma disponibilidade contínua do sistema, como ocorre

por exemplo em sistemas de comunicação, sistemas de automação industrial ou em sistemas de apoio à navegação.

O uso de tal facilidade, no entanto, é tipicamente motivada por outros objetivos, como por exemplo, garantir a tolerância a falha do sistema, permitir uma adaptação dinâmica à carga imposta ao sistema, melhorar a utilização de recursos, possibilitar um monitoramento e/ou depuração flexível.

Configuração dinâmica tem sido usada para possibilitar uma adaptação de sistemas a mudanças de seus ambientes de execução, independente da velocidade e freqüência em que estas ocorrem. Por exemplo, pode-se considerar a necessidade de migração gradual de um sistema para uma nova infra-estrutura de processamento ou comunicação em uma escala de tempo de dias a meses. No outro extremo, pode se requerer uma troca imediata de um protocolo de comunicação por outro em função das condições de transmissão de dados do meio. Este último caso vem sendo pesquisado bastante no contexto de computação móvel.

O uso de configuração dinâmica pode estar relacionado também à necessidade de adaptação a variações na carga do sistema ou a uma mudança no perfil de seu uso. Por exemplo, é bastante útil que sistemas operacionais permitam uma configuração dinâmica dos seus serviços de acordo com as necessidades de cada aplicação, sem que haja necessidade de recarregar o sistema. Para uma adaptação a variações na carga de um sistema distribuído, pode-se fazer necessária uma migração de alguns servidores de um nodo da rede para outro, o que representa uma reconfiguração dinâmica. Por fim, pode-se usar configuração dinâmica também para implementar um balanceamento de carga, para fins de uma melhor utilização dos recursos computacionais.

Configuração dinâmica também vem exercendo um papel importante na implementação de novas tecnologias como "groupware" e "collaboratories"[39]. Para estas categorias de aplicações, é fundamental aliar flexibilidade funcional com eficiência no aproveitamento de recursos em ambientes de execução altamente dinâmicos e heterogêneos.

Tolerância a falha é talvez uma das mais importantes aplicações de configuração dinâmica. Assumindo que haja meios para detectar diferentes tipos de falha, para muitos sistemas é necessário que haja também formas de eliminar dinamicamente as mesmas e restabelecer o curso normal de execução. Isto pode envolver desde uma simples criação de um processo em outra máquina, redirecionamento de referências até complexos mecanismos de restabelecimento de estado otimistas ou pessimistas.

Por fim, pode se usar configuração dinâmica também para incorporar dinamicamente código para o monitoramento da execução de partes do sistema, para efeito de depuração ou análise e melhora de desempenho.

1.3 Principais Tarefas e Problemas

Apesar de sua grande variedade de tipos e aplicações, tentaremos caracterizar aqui as principais tarefas e problemas relacionados com a configuração dinâmica.

Independente da forma de reposição ou inclusão dinâmica de uma parte do sistema, precisa-se garantir sempre uma compatibilidade total entre a (nova) parte inserida e o restante do sistema. Esta compatibilidade é composta de uma *consistência sintática* e uma *consistência semântica*. A primeira apenas garante que a forma de interação com a nova parte, seja ela uma chamada de procedimento, o acesso a uma variável ou uma comunicação, é compatível com a implementação das partes não alteradas. A consistência semântica, por sua vez, deve garantir que a nova parte é compatível com o restante do sistema em termos de sua funcionalidade e de possíveis efeitos colaterais, tais como a utilização adequada dos recursos compartilhados.

Devido à dificuldade de tratar de forma geral o problema da consistência semântica, no melhor dos casos é dado suporte a uma checagem da compatibilidade sintática nos sistemas com a capacidade de configuração dinâmica. Para permitir tal verificação, faz-se necessária a definição de interfaces entre

as componentes do sistema, cujos atributos são comparados antes de uma reconfiguração.

Além da verificação da compatibilidade de interfaces, é necessário também *manter informações sobre as dependências mútuas* entre as componentes de um sistema. Isto é necessário tanto para uma reconexão das componentes, como para identificar os elementos que devam ser colocados em estado consistente (ou passivo) durante a reconfiguração.

Outra tarefa indispensável é garantir a *transparência da interação* com a parte sendo incluída tanto durante quanto após uma reconfiguração. Em outras palavras, deve-se garantir o que resto do sistema em execução não "perceba" que está ocorrendo uma modificação e que, a menos de uma possível deterioração temporária da eficiência, não sofra qualquer tipo de falha ou problema durante ou depois da mudança.

Para permitir um controle efetivo da configuração de um sistema, muitas vezes é necessário *instrumentá-lo*, isto é, gerar código adicional que implementa funções de monitoramento e de controle da configuração. Estas podem servir por exemplo para detectar a disponibilidade de uma componente, obter o seu status, manipular o seu estado, ou suprimir temporariamente interações com outras componentes.

Um dos problemas mais complexos relacionados à configuração dinâmica é o de obter um sincronismo entre a reconfiguração e a execução normal do sistema, ou seja determinar *quando* uma reconfiguração deve ser feita. No caso de um sistema executando em um uniprocessador este problema se reduz ao problema de garantir a consistência semântica da reconfiguração. Em sistemas paralelos e distribuídos, no entanto, devido a simultaneidade de ações, este problema requer medidas adicionais que garantem que a semântica do acesso à componente sendo reconfigurada não sofra prejuízo durante a reconfiguração. Devido a complexidade deste problema e a diversidade de soluções propostas, dedicaremos a seção7 somente a este assunto.

2 Principais Mecanismos e Custos Associados

Nesta seção apresentaremos os principais mecanismos usados para viabilizar a configuração dinâmica de sistemas, identificando as suas particularidades quando aplicados ao contexto de sistemas operacionais e sistemas distribuídos. Além disto, discutiremos os custos associados a cada mecanismo para cada um dos contextos.

2.1 Alteração e Redirecionamento de Referências

A maioria dos sistemas modernos consiste de componentes (ou módulos) que interagem com as demais componentes de forma bem definida. Para isto cada componente possui referências a pontos de entrada (ou objetos) bem definidos em outras componentes. Assim, uma forma simples de reconfiguração pode consistir de uma alteração e/ou o redirecionamento destas referências.

No caso de sistemas operacionais, é comum que a noção de módulos esteja intrinsecamente ligada a de domínios de proteção, de forma que a alteração dinâmica de referências pode vir a conflitar com requisitos de eficiência e segurança. O sistema Lipto (seção 3.3) propõe como solução o desacoplamento dos conceitos de proteção e modularidade. A modularidade é definida através da hierarquia de classes que descreve os objetos do núcleo, enquanto que domínios são definidos através de composição. O efeito de alteração e redirecionamento de referências é obtido através da alteração da composição de um serviço, que está sujeita ao controle do núcleo via listas de acesso. Outra forma comum de redirecionamento de referências do núcleo é o uso de *interposição*, que permite a associação de código do usuário aos serviços oferecidos pelo núcleo sem que a interface deste seja alterada[33, 53]. Através do código interposicionado o serviço pode ser alterado sem envolvimento direto do núcleo, permitindo que sistemas operacionais que não prevêm configuração possam ser adaptados dinamicamente. Esta

técnica é limitada pela extensão em que a funcionalidade de um serviço pode ser alterada sem envolver manipulação das estruturas de dados internas ao núcleo.

Em sistemas distribuídos, uma referência remota é armazenada em uma estrutura de dados no núcleo do sistema operacional (ou no sistema de apoio à execução), que pode ser alterada pelo próprio processo de aplicação possuidor de um handle para tal estrutura ou por um agente externo, como por exemplo um programa gerenciador de configuração.

No primeiro caso, quando se trata de um redirecionamento, é a própria componente da aplicação que precisa obter o novo endereço a ser armazenado como referência, o que pode envolver a consulta a um serviço de nomes ou a leitura de um arquivo. Fala-se neste caso de uma facilidade de *binding direto*, como é encontrado em ambientes como o Concert/C e Arjuna.

No segundo caso, o sistema de apoio à execução, geralmente na forma de um daemon que chamaremos de *runtime executive* (seção 5.2), coloca à disposição um conjunto de funções remotas (ou portas) bem conhecidas através das quais um agente externo pode alterar uma referência remota mantida por uma componente de aplicação, fazendo com que um redirecionamento seja totalmente transparente para esta componente. Sistemas como Conic, Darwin e Polylith adotam este mecanismo, que é chamado de *binding indireto*. Neste caso têm-se o custo adicional da interação com o agente externo e de um eventual bloqueio da comunicação durante a fase de atualização da mesma, como ocorre no sistema Polylith.

2.2 Criação Remota

Um outro mecanismo importante é o da instanciação remota de componentes ativas (processos) ou passivas (objetos/módulos) a partir de uma outra componente executando no sistema. Para que a criação remota seja possível, é necessário que em cada nó da rede execute um *runtime executive* que deve poder criar uma componente em resposta a um pedido remoto. Além disto, é necessário que este runtime executive tenha acesso ao código executável da nova componente, que pode se dar através do acesso a um sistema de arquivos compartilhado ou através da transmissão do código da componente na própria mensagem de requisição da criação remota. De qualquer forma, os custos associados com a obtenção/seleção do código, a operação local de criação e o eventual redirecionamento dos streams de E/S são bastante altos.

2.3 Mecanismos de Sincronização

Mecanismos de sincronização são usados para evitar que durante uma configuração dinâmica partes operantes de um sistema interajam com componentes ainda não disponíveis ou prontas para a execução. Estes mecanismos são necessários principalmente para a configuração dinâmica em sistemas multiprocessadores e distribuídos. Os mecanismos mais utilizados consistem no bloqueio do envio de mensagem e no uso de protocolos para garantir que certas partes do sistema permaneçam passivos ou semi-passivos. Em qualquer um dos casos o custo em termos de tempo de execução associado a estes mecanismos é geralmente bastante alto. Devido à complexidade e a relevância do problema da sincronização, trataremos o mesmo separadamente na seção 7.

2.4 Monitoramento

O monitoramento da execução de programas é um elemento fundamental que fornece a base para qualquer decisão de configuração. No caso de configuração dinâmica automática (em que as ações de reconfiguração são disparadas pela própria aplicação, é essencial que as informações fornecidas pelo monitoramento descrevam com precisão o estado de execução. O mecanismo específico que é usado

e seu grau de acoplamento com os demais mecanismos de reconfiguração depende muito do tipo e do domínio de aplicação da configuração dinâmica.

Para sistemas operacionais, o monitoramento tipicamente analisa o padrão de uso dos recursos e serviços por uma aplicação, gerando como resultado valores dos parâmetros de configuração a serem usados pelo serviço de configuração.

Em outros ambientes, o monitoramento é realizado através do teste periódico (*polling*) da disponibilidade de componentes da aplicação, ou então através da geração de notificações e de sua posterior análise. Dependendo da natureza do mecanismo de monitoramento usado têm-se diferentes custos de execução e de instrumentação da aplicação.

2.5 Instrumentação Automática de Programas

Para a maioria das formas de configuração dinâmica faz-se necessária uma instrumentação do sistema a ser configurado com funções adicionais, como por exemplo para o monitoramento, o redirecionamento de referências, o controle de execução e outros. Existem duas dificuldades principais na instrumentação de programas. A primeira é conseguir que um agente capture informação específica da aplicação sem causar uma pertubação na execução que interfira no andamento da aplicação ou na acuridade da informação coletada. A segunda dificuldade é a identificação de pontos da execução de uma aplicação em que instrumentação possa ser adicionada sem danificar o funcionamento da aplicação. Esta questão é um caso particular do problema de sincronização (seção7).

A instrumentação pode ter diferentes graus de automação e pode ser realizada de diversas formas, indo desde a simples disponibilização de procedimentos ou classes de uma biblioteca até o uso de sofisticados compiladores ou pré-processadores. Por exemplo, em sistemas como Conic, Regis/Darwin e Polylith (seção 6) existe um compilador para uma linguagem de configuração que gera arquivos de definição de estruturas e de interface, para as componentes compostas e primitivas, respectivamente. Já outros ambientes, como o DCE, provêm um compilador para uma linguagem de descrição de interfaces, que apenas gera stubs cliente e servidor. No sistema KTK (seção 4.4) instrumentação está disponível para as camadas de baixo nível do suporte de execução, através de uma biblioteca de *threads* cujo código foi cuidadosamente instrumentado, e para a camada dos objetos da aplicação, através de ferramentas para especificação e compilação de *sensores* e *atuadores*.

2.6 Base de Dados sobre a Configuração

Para que configuração dinâmica seja possível é necessário se manter uma base de dados sobre a configuração atual de um sistema. A natureza da informação contida nesta base de dados e sua localização varia conforme a forma de configuração implementada. Enquanto em sistemas operacionais configuráveis esta informação é geralmente mantida no próprio kernel, alguns ambientes para sistemas distribuídos provêm programas gerenciadores de configuração (ou servidores de nome especiais) que disponibilizam funções para a consulta e a modificação desta base de dados através de operadores e dos próprios programas de aplicação. Também neste caso, há um custo relativo à disponibilização e manutenção da base de dados, bem como um custo com a consulta à mesma.

2.7 Interface com Serviço de Configuração

O grau de sofisticação da interface com o serviço de configuração dinâmica também varia muito de acordo com o seu uso.

Como a configuração dinâmica de um kernel é tipicamente requisitada diretamente por um programa de aplicação ou por um monitor de execução (portanto sem uma intervenção humana), a sua interface

geralmente consiste de um simples procedimento ou um ponto de acesso ao código que efetua o carregamento e/ou o redirecionamento de referências. No entanto, para algumas novas aplicações já existem interfaces que disponibilizam para o operador a configuração dinâmica de serviços do sistema operacional. Por exemplo, para *servidores de vídeo* pequenas anomalias nas medidas de qualidade de serviço (QoS) poderiam ser tratadas através de configuração automática, enquanto que casos mais críticos poderão requerer a intervenção direta de um operador na modificação do protocolo de comunicação.

Já outros ambientes que permitem a um usuário (ou operador) determinar as reconfigurações de seu sistema de forma interativa geralmente disponibilizam um interpretador de comandos para a entrada textual de consultas e comandos. Para alguns ambientes, como por exemplo o Conic e Darwin, além disto também se implementou uma interface gráfica para o monitoramento e a alteração dinâmica de um sistema.

2.8 Geração Dinâmica de Código

A geração de código durante a execução (*on the fly*) tem a vantagem de ter disponível informações sobre a situação *atual* dos recursos a serem utilizados pela aplicação no momento. Sob este aspecto, a técnica de geração dinâmica de código é a que mais tem potencial para geração de um sistema idealmente especializado para um ambiente específico que aproveite de forma ótima os recursos. Mas a obtenção de um código "ótimo" para uma situação particular da execução só é proveitosa se os ganhos em desempenho forem maiores do que os gastos envolvidos na própria geração de código. Além disso, a geração dinâmica é uma técnica difícil de ser empregada por várias razões:

1. o código a ser sintetizado em geral é complexo, tornando o processo de geração difícil de ser verificado e mesmo testado;

2. a disponibilidade dos recursos em muitos casos (*locks* por exemplo) varia muito, tornando necessário que as decisões tomadas durante a geração sejam freqüentemente reavaliadas;

3. o código gerado para uma situação específica pode gerar erros no sistema se o estado dos recursos mudar. Com um código muito geral (e portanto robusto a tais mudanças), o ganho de desempenho tem menos chance de compensar o custo de geração. Com um código altamente especializado a confiabilidade do sistema pode ficar comprometida.

A seção 3.1 descreve o sistema Synthesis, onde a técnica foi explorada com bons resultados. Mais recentemente, o projeto Synthesis vem explorando formas de especialização *otimista* de código.

3 Configuração Dinâmica de Sistemas Operacionais

Configuração dinâmica do núcleo de sistemas operacionais manifestou-se inicialmente através de pequenos trechos de código que se auto modificavam. Com os avanços do hardware e aumento da complexidade das aplicações e dos próprios sistemas operacionais, este tipo de solução *ad hoc* se tornou impraticável sob vários aspectos, em especial pelo comprometimento da segurança e confiabilidade do sistema, ausência de gerenciamento das mudanças e escalabilidade. A popularização de mecanismos para link-edição dinâmica (*dynamic linking*) ajudou a tornar os serviços providos pelos sistemas operacionais um pouco mais flexíveis, mas sem se adequar a adaptações dinâmicas de comportamento, já que aplicações clientes acabam acoplados após o tempo de disparo (*launch time*) a uma implementação específica, sem que seja fornecida a possibilidade para redirecionamentos transparentes para serviços suplementares ou alternativos.

As mudanças na organização e implementação de sistemas operacionais introduzidas pela arquitetura de *micro-kernels* têm sido muito importantes no desenvolvimento de sistemas operacionais que sejam

capazes de se adaptarem dinamicamente a diferentes demandas de serviços e disponibilidade de recursos do ambiente de execução. O sistema pode ser alterado via seus dispositivos de integração dos módulos ou substituição de componentes básicos, mas esta forma de configuração enfrenta o mesmo problema da própria arquitetura de micro-kernels: degradação do desempenho. Sendo eficiência um requisito primordial para sistemas operacionais, as técnicas desenvolvidas para configuração dinâmica de aplicações e ambientes de programação (cap. 5) em geral não podem ser empregadas diretamente. Novos modelos de arquitetura de sistemas operacionais (por exemplo o Exokernel, descrito na seção 4.1) que tornam mais viável a adaptação e configuração dinâmica com sobrecarga aceitável têm sido propostos. Sistemas como Paramecium[71] empregam uma arquitetura de software projetada para permitir fácil reconfiguração.

Sistemas operacionais orientados a objetos tem sido apresentados como uma solução para o desenvolvimento de sistemas flexíveis e adaptáveis[38]. Foram propostos esquemas para a configuração pelo usuário em tempo de execução de serviços de pequena granularidade[43] (tais como *locks*, *threads*), sistemas de arquivos distribuídos customizáveis[7], gerenciadores de memória que lidam com páginas de tamanho configurável, ferramentas para rastreamento de chamadas do sistemas (*system calls*) e referência a arquivos[33], etc.

Há também o enfoque de se obter configuração através de um esquema flexível para a definição e manipulação de *eventos*, de uma maneira fortemente acoplada com o compilador sendo utilizado. Em [56] é definido um esquema para invocações baseado em eventos com o objetivo de se obter um *binding* dinâmico em sistemas operacionais que seja flexível, transparente, seguro e eficiente. Instalando-se um *handler* em um evento, um código representando uma extensão ou especialização de um serviço pode ser executado para atividades com a granularidade de uma chamada de procedimento. Estes *handlers* podem ser dinamicamente adicionados ou removidos de um evento, com um número qualquer deles podendo estar associado a um único evento. É definido um conjunto de permissões estáticas e dinâmicas de acesso de forma a garantir que eventos não sejam utilizados de forma errônea ou insegura pelas extensões. Eventos podem ser executados síncrona ou assincronamente, mas o emprego destas idéias no sistema operacional SPIN[4] permite execução assíncrona somente nos casos em que a semântica de uma chamada de procedimento não seja violada.

Nas próximas seções serão descritos mecanismos para configuração oferecidos por alguns sistemas operacionais, destacando-se as características que viabilizam o uso efetivo da flexibilidade oferecida. Os sistemas diferem tanto em forma e grau de disponibilização da configuração dinâmica, quanto na arquitetura do núcleo e sua implementação. Na seção 4 são descritos sistemas que oferecem uma estrutura básica para o desenvolvimento de kernels configuráveis.

3.1 Synthesis/Synthetix

Synthesis[58] é um sistema operacional distribuído que combina chamadas eficientes do núcleo com uma interface de alto nível e ortogonal. Pu e Massalin propõem um sintetizador de código no núcleo que gera código especializado (portanto curto e rápido) de rotinas do núcleo para situações específicas. Eles provaram que os ganhos durante a execução podem ultrapassar os custos de geração de código *on-line*. Por exemplo, rótinas do Synthesis (geradas durante a execução) contém entre 20 e 30 instruções, enquanto que o *read* do 4.3 BSD Unix contém cerca de 500 linhas de código em C. Eles empregaram três métodos de geração de códigos, todos definidos de forma elegante, em um nível de abstração mais elevado do que usualmente se encontra neste tipo de pesquisa. Mais recentemente Pu vem explorando o uso de avaliação parcial incremental[11] na obtenção de sistemas operacionais modulares portáteis de alto desempenho (projeto Synthetix). Em [57] é descrita uma técnica para geração otimista de código do kernel para estados do sistemas que são bem prováveis de ocorrer, mas não garantidamente ("quasi-invariants"). Corretude é preservada usando-se "guards", isto é, o código é substituído por um que

lide com a situação não otimista. Em [12] é apresentada uma técnica eficiente para dynamic linking, de forma a viabilizar a substituição de versões.

As idéias deste projeto vem sendo aplicadas em um sistema operacional do mundo real. O sistemas de arquivos do HP-UX foi re-implementado usando os mecanismos do Synthesis, com experimentos demonstrando que o custo da leitura de um byte pelo núcleo foi reduzido em mais de uma ordem de magnitude.

3.2 Spring

O sistema operacional Spring[37, 29] introduziu a noção de *subcontratos* como base de suporte para computação distribuída flexível. Este conceito foi derivado da experiência inicial com os "object adaptors" do CORBA[61]. A motivação é permitir que os programadores controlem aspectos fundamentais do gerenciamento de objetos, em especial possibilitando que mecanismos diferentes para chamadas de procedimentos remotas fossem integrados de forma consistente. Um subcontrato está envolvido em todos os eventos importantes da vida de um objeto: criação, reprodução, destruição, transferência de espaço de endereçamento e invocação de seus métodos. Orientação a objeto é explorada (em particular herança de interfaces, que são especificadas em IDL[54]) como base para o desenvolvimento de software em que evoluções sejam encorporadas naturalmente. O projeto Spring desenvolveu um sistema de arquivos extensível, uma arquitetura de *caching* unificado, serviço de persistência, *dynamic linking* eficiente entre espaços de endereçamento, interface flexível de paginação, suporte para operação disconectada e um serviço de nomes flexível.

3.3 Lipto

Lipto[17] é um sistema operacional orientado a objetos. Uma característica interessante de sua arquitetura é que ela não explora especialização em subclasses, e sim composição de objetos. A estrutura do sistema Lipto coloca à disposição apenas um conjunto pequeno de funções no kernel, com os outro sistemas localizados nos domínios de proteção das aplicações. A substituição de partes de um objeto composto — ou seja, uma reconfiguração do serviço — é feita através de uma "meta-interface" que contém um grafo de especificação da composição[16]. O aspecto inovador deste sistema é o total desacoplamento da modularização dos domínios de proteção.

O esquema oferecido para configuração neste sistema se adequa mais em termos de configuração para pré-instalação do sistema. Por exemplo, um subsistema consistindo de um conjunto de módulos pode ser configurado com cada um dos módulos em um domínio de proteção diferente durante a fase de testes (já que a detecção de erros é assim facilitada), e em uma fase posterior do ciclo de vida do software os módulos podem ser combinados em um único domínio por razões de desempenho. Configuração em tempo de execução foi também explorada, mas em um caso específico: o mecanismo utilizado pelo serviço de RPC (chamada remota de procedimento) pode ser dinamicamente selecionado[31].

3.4 Kea

Kea[72] é um núcleo de sistema operacional cujo desenvolvimento foi totalmente direcionado para a obtenção de máxima flexibilidade em termos de recuperação dinâmica e extensibilidade, sem que haja degradação demasiada do desempenho. Kea fornece uma forma de RPC bem geral, em que o destino de uma chamada remota pode ser alterada independentemente da origem da chamada. Pontos de entrada inter-domínios podem ser remapeados em tempo de execução.

4 Ambientes de Desenvolvimento de Núcleos

Além dos exemplos de sistemas operacionais envolvendo configuração dinâmica apresentados na seção anterior, existem sistemas projetados de forma a servir de base para a construção de sistemas operacionais com kernels configuráveis. Estes sistemas definem uma arquitetura com apoio explícito para as atividades de configuração dinâmica, e em geral fornecem a especificação e implementação de alguns componentes básicos que podem ser reaproveitados.

4.1 Exokernel

Exokernel[21] é uma arquitetura base para desenvolvimento de sistemas operacionais flexíveis que sejam construídos como bibliotecas. Exokernel fornece a interface através da qual as bibliotecas implementam objetos do núcleo e políticas de uso. A idéia básica é que um núcleo *bem* pequeno multiplexa de forma segura os recursos de hardware disponíveis entre as aplicações. A separação da proteção dos recursos do gerenciamento dos mesmos permite uma configuração das abstrações fundamentais de um sistema operacional através da extensão, especialização e substituição das bibliotecas. Ao invés de fornecer para cada aplicação sua própria máquina virtual, o Exokernel exporta os recursos do hardware sem emulá-los. O exokernel Aegis foi construído com base nesta arquitetura, servindo de base para o ExOS, um sistema operacional no estilo de biblioteca. ExOS gerencia aspectos fundamentais como processos e memória virtual no nível da aplicação.

4.2 Agentes de Interposição

O trabalho desenvolvido por Jones[33] define uma ferramenta para interposição transparente de código da aplicação nas chamadas do kernel. Esta ferramenta permite que o código a ser interposto entre o cliente de um serviço e a implementação deste serviço disponível no kernel seja escrita em termos de objetos de alto nível providos pela interface do kernel. Estão disponíveis objetos para *pathnames*, descritores, processos, grupo de processos, arquivos, diretórios, *pipes*, sockets, sinais, etc.

4.3 FLEX e Flux

FLEX[8] é uma ferramenta para construção de sistemas flexíveis e eficientes que foi desenvolvida na Universidade de Utah. A ferramenta foi projetada de forma a prover interfaces para sistemas operacionais que sejam poderosas e eficientes. A técnica explorada neste projeto é a construção de módulos especializados de implementação em tempo de execução. Uma diferença entre a especialização oferecida por FLEX e outros sistemas como Synthesis (seção 3.1) é a granularidade dos serviços cuja implementação pode ser gerada em tempo de execução. Enquanto o sistema operacional Systhesis especializa as chamadas do sistema como "read", FLEX teve como objetivo permitir que o usuário adicione serviços completos (incluindo vetores de interrupção associados) ao núcleo do sistema operacional. A ferramenta pode ser usada para estender dinamicamente um sistema operacional de forma controlada, permitindo que o usuário construa a implementação mais conviente para seu uso específico de um serviço. O usuário tem acesso a dados e dispositivos privilegiados. FLEX funciona lendo módulos, manipulando-os conforme especificado em arquivos de manipulação, e escrevendo o executável resultante em um arquivo ou em um espaço de endereçamento do usuário.

A ênfase e os objetivos do grupo que desenvolveu a ferramenta FLEX foram se alterando, e hoje FLEX é visto apenas como o embrião de um projeto maior: o sistema operacional Flux. Os objetivos do projeto Flux são (1) prover uma infra-estrutura para sistemas baseados em componentes que sejam altamente eficientes e (2) prover controle flexível e transparente de todos os recursos que possam vir a ser usados por outros subsistemas. Há uma grande ênfase no desenvolvimento de software que possa

ser distribuído e que seja efetivamente útil no desenvolvimento de software flexível. *Fluke*[26] é a arquitetura que serve de base para o projeto. Fluke é descrita como uma *arquitetura virtualizável*: permite que máquinas virtuais recursivas (máquinas virtuais que executam em outras máquinas virtuais) sejam implementadas eficientemente por microcódigos em hardware genérico. A premissa básica da arquitetura é que combinando-se conceitos de *microkernels* com máquinas virtuais seja possível implementar no nível de processos do usuário tanto a funcionalidade comumente oferecida por um sistema operacional (gerenciamento de processos, paginação, etc) como características desejáveis como tolerância a falhas. Desta forma, o sistema operacional pode ser configurado para prover demandas específicas de algumas aplicações, tais como distribuição e gerenciamente de executáveis não confiáveis. O *Flux OS Toolkit*[25] foi desenvolvido com o objetivo de fornecer componentes reutilizáveis (que sejam "limpas" e bem documentadas) que possam ser usadas como peças básicas no desenvolvimento de sistemas operacionais e ambientes de execução. Outra contribuição do projeto é o *Flick*[18], um compilador para IDL que suporta vários dos padrões existentes para a linguagem, diversos códificações dos dados e mecanismos de transporte. Experimentos mostraram que código gerado por Flick é mais eficiente do que outras implementações disponíveis. Com a tendência atual do uso de IDL para especificações de interface (inclusive interfaces de configurações) as técnicas exploradas pelo compilador Flick podem ser úteis em outros projetos.

4.4 KTK/CTK

KTK (*Kernel Toolkit*)[53] fornece mecanismos para uso, inclusão e configuração dinâmica de abstrações alternativas para elementos do núcleo de um sistema operacional. Por exemplo, usando-se KTK foi implementado um esquema para *locks* adaptativos. Mostrou-se que para obter-se ganhos de desempenho é necessário que os mecanismos de adaptação lidem tanto com adaptação síncrona quanto assíncrona em relação à aplicação[52].

CTK (*Configuration Toolkit*)[62] estendeu o modelo apresentado por KTK de forma a incluir a configurações de abstrações mais genéricas. CTK é uma biblioteca para a construção de abstrações configuráveis baseadas em objetos que sejam parte de aplicações multiprocessadas ou sistemas operacionais. A biblioteca explora configuração dinâmica om o objetivo de obter melhoria de desempenho em programas paralelos. O modelo de programação oferecido por CTK facilita a expressão e implementação de configurações do programa e o sistema de execução eficiente possibilita ganhos de desempenho através da configurações de componentes durante a execução. A configuração dos programas é expressa sem que seja prejudicado o encapsulamento e o reuso de abstrações, pois a funcionalidade de um objeto que seja dependente do seu tipo é explicitamente separada de propriedades sujeitas à configuração como desempenho, confiabilidade e características temporais.

Programas em CTK são configurados usando-se *atributos* e *policies*. Atributos podem ser associados com classes, objetos, variáveis de estado, operações e invocações. Em tempo de execução estes atributos são interpretados pelas *policies*, que podem ser variadas separadamente do objeto que está sendo configurado. Atributos funcionam como "botões" que controlam a configuracão e podem ser manipulados em tempo de execução, enquanto que *policies* implementam as mudanças resultantes destas manipulações.

5 Ambientes de Desenvolvimento de Aplicações Distribuídas

Além de uma infra-estrutura básica para a realização de configuração dinâmica muitos sistemas também fornecem ferramentas para o desenvolvimento de aplicações paralelas e distribuídas com capacidade de configuração dinâmica. Estas ferramentas vão de simples pre-processadores para instrumentação de programas até compiladores para novas linguagens de programação e configuração.

Os ambientes disponíveis podem ser classificados quanto a existência ou não de uma linguagem de configuração distinta da linguagem de programação das componentes básicas.

Entre os sistemas mais conhecidos que primeiro adotaram a chamada *programação de configuração* baseada neste bi-linguismo está o sistema Conic [48], desenvolvido no Imperial College de Londres no inicio dos anos 80. A este se seguiram sistemas como Durra[2], desenvolvido na CMU, e Regis/Darwin[47], este último como evolução do Conic. A principal argumentação a favor desta linha é que ela induz a uma programação estruturada que favorece uma melhor modularização e reutilização de componentes de uma aplicação, além de permitir um gerenciamento melhor de sistemas complexos que consistem de um grande número de processos.

Em paralelo, surgiram linguagens que incorporavam na própria linguagem de programação os elementos necessários para uma configuração dinâmica do sistema. Entre os sistemas pioneiros nesta linha pode-se citar a linguagem NIL[68, 67] desenvolvida na IBM, e Argus[44, 6], desenvolvida no M.I.T. A partir destes, surgiram muitas outras linguagens e sistemas com funcionalidade semelhante, tais como Hermes[69], Concert/C[22], e outros[1]. O principal argumento a favor destes sistemas é o seu alto grau de flexibilidade de programação, uma vez que nestas linguagens as referências a portas de comunicação são objetos de primeira ordem que podem ser enviadas entre processos.

Com o advento da programação orientada a objeto, surgiu também uma outra linha de desenvolvimento de programas reconfiguráveis *orientados a objetos* (ao contrário dos baseados em processos ou objetos ativos[9]), no qual é possível instanciar objetos remotos e acessar os seus métodos como se fossem locais. Além da capacidade de configuração dinâmica induzida pela criação dinâmica de objetos, alguns sistemas ainda permitem uma reconfiguração no nível do próprio código que implementa um objeto, através de métodos *reflexivos*, isto é, que são capazes de modificar a funcionalidade do próprio objeto do qual fazem parte. Entre os ambientes orientados a objeto reflexivos mais conhecidos estão o sistema Apertos[77] e SOM/DSOM[14].

A seguir veremos os principais elementos que compõem um ambiente de desenvolvimento para programas configuráveis dinamicamente.

5.1 Linguagem de Programação

Dentre os principais elementos de um ambiente de desenvolvimento está a linguagem de programação das componentes básicas, sejam estas processos ou objetos. No que diz respeito à configuração dinâmica, as linguagens de programação se distinguem quanto às facilidades para controlar outras componentes e as referências remotas. Enquanto que linguagens como Conic e Regis não permitem criar processos (ou objetos) remotos através da linguagem de programação, isto é possível por exemplo em Concert/C e em linguagens orientadas a objeto distribuídas, como Arjuna e outras[9]. Uma outra diferença diz respeito ao grau de transparência de comunicação. Por exemplo, em Conic e Durra, processos só interagem com os demais processos através de suas *portas* e *referências para portas* declaradas localmente. Ou seja, um processo desconhece completamente a identidade da componente ligada momentaneamente às sua portas e referências remotas. Já em outros ambientes, como o DCE, o Regis ou Arjuna o processo (ou objeto) cliente tem como procurar a componente desejada através da consulta a um *serviço de nomes*, o qual retorna o endereço de acesso correspondente. Tendo este endereço, é possível então acessar a componente remota independentemente de sua localização. Uma facilidade presente na maioria das linguagens de programação em questão é a capacidade de enviar tais endereços em mensagens para outras componentes, o que de certa forma também constitui uma alteração dinâmica da topologia de interconexão das componentes.

5.2 Runtime Executive

O runtime executive é um processo (com um ou mais fluxos de controle) que possibilita a criação remota de processos de aplicação, efetua o controle destes processos, monitora o estado dos mesmos e se encarrega do redirecionamento da E/S para a máquina de onde originou o pedido de criação remota.

Este processo precisa executar (na forma de um daemon) em todas os nós da rede que devam fazer parte do conjunto de máquinas disponíveis para a aplicação.

5.3 Serviço de Nomes

O serviço de nomes é necessário particularmente para os sistemas que visam permitir o desenvolvimento de aplicações abertas, isto é aquelas que necessitam interagir com outros serviços e programas de aplicação. O serviço de nomes é realizado através de um ou mais processos servidores que têm a tarefa de traduzir nomes para endereços de portas de comunicação. Estes endereços são geralmente armazenados em um espaço de nomes hierárquico, similar à estrutura de sistemas de arquivos em sistemas operacionais.

5.4 Definição de Interfaces

Interfaces desempenham um papel fundamental no projeto e implementação de sistemas distribuídos, em particular de sistemas configuráveis dinamicamente. É através destas que são definidos os tipos básicos de dados e a forma de interação (síncrona, assíncrona, RPC) a ser usada na comunicação entre as diferentes componentes. Assim, indiretamente, as mesmas também definem os subconjuntos de processos que podem interagir entre si e quais podem ser trocados uns pelos outros.

Quanto à forma de definição de interfaces, existe a possibilidade desta fazer parte da linguagem de programação, como é o caso em Regis[13] e em Concert/C[23] ou na forma de uma linguagem independente da linguagem de programação, para qual se provê geradores de código para as diversas linguagens alvo. Neste segundo caso, se enquadram uma série de linguagens de definição de interface - *Interface Definition Language (IDL)* - que são adotadas em ambientes de programação distribuída como o Distributed Computing Environment da OSF ou o CORBA. A cada IDL geralmente está associado um gerador de stubs que gera o código necessário para o estabelecimento de conexões de comunicação e a conversão dos tipos de dados usados na comunicação. Estes stubs são então ligados aos códigos de um processo cliente e de um processo servidor.

O sistema Polylith[59] propõe uma solução parecida que também se baseia em uma especificação da interface independente da linguagem de programação. Neste sistema existe uma ferramenta chamada *packager*, que tem como função gerar o código necessário para a chamada dos procedimentos e a conversão dos tipos de dados de diversas linguagens de programação para os tipos básicos do *software bus*, que é o elemento que efetua a comunicação entre processos de aplicação, para uma variedade de formas de interação.

5.5 Linguagem de Configuração

Linguagens de configuração, também chamadas de linguagens de controle, meta-linguagens, ou linguagens de interconexão de módulos, são usadas para descrever a estrutura de um sistema (ou subsistema) e efetuar as alterações sobre esta. Em alguns ambientes como Conic e Durra são chamadas de *componentes compostas* aquelas implementadas com a linguagem de configuração.

Dentre as linguagens de configuração adotadas nos sistemas baseados na separação entre algoritmos e estrutura do programa, existem uma série de diferenças relativas aos seguintes aspectos:

O Paradigma de Execução A grande maioria das linguagens é ou procedural ou baseada em regras. No primeiro caso, toda a configuração e as possíveis reconfigurações de um sistema são descritas na forma de scripts de configuração, que executam seqüencialmente e que podem ser chamados de outros scripts ou invocados pelo próprio usuário. Exemplos de linguagens deste tipo são Conic, Darwin, Durra. No segundo caso, usa-se a idéia de que qualquer reconfiguração, incluindo a configuração inicial, é descrita por regras que são disparadas automaticamente na ocorrência de certos eventos especificados na própria regra. A linguagem Lomita, adotada em Meta[76] é um exemplo de uma linguagem baseada em regras. Nesta linguagem, as condições de disparo são baseadas em valores de sensores que estão associados aos processos da aplicação.

Objetos e Tipos primitivos Uma outra diferença que distingue as linguagens de configuração são os tipos básicos de objetos que estas manipulam. Além dos tipos mais comuns como *componente*, *interface (ou porta)*, *referência para interface* e *canal de comunicação*, algumas linguagens como Darwin também permitem o uso de variáveis de tipos escalares como *inteiro*, *caracteres*, *ponto-flutuante*, permitindo a formação de expressões que são avaliadas em tempo de instanciação da componente. Outras linguagens, como a Module Interconnection Language (MIL) do sistema Polylith[59] e Durra permitem também a associação de atributos quaisquer às instâncias. Estes especificam, por exemplo, o nodo em que a instância deve ser executada, o nome e o diretório do executável, e o conjunto de funções exportadas e importadas por cada instância, ou seja, os pontos de interação entre as mesmas.

Composicionalidade Outro diferencial entre as linguagens de configuração está na capacidade de descrever estruturas hierárquicas de programas, isto é, onde instâncias primitivas (processos) podem ser agrupadas em ditas *componentes compostas*, que são tratadas como as componentes primitivas, em um nível superior da descrição estrutural. Esta característica é a que indica se uma linguagem é adequada para o desenvolvimento de programas contendo um grande número de processos. Exemplos de linguagens com tal propriedade incluem Darwin, MIL/Polylith e Durra, enquanto que Lomita/Meta não permite tal forma de estruturação.

Estruturas de Controle Linguagens de configuração variam muito com relação às estruturas de controle que que provêem. Linguagens usadas para (re)configurações pré-definidas tendem a ter um conjunto bastante sofisticado de estruturas de controle. A linguagem Darwin, por exemplo, incorpora comandos de repetição *forall* e de seleção *when*, que permitem definir estruturas compostas de componentes replicadas (vetores de componentes) e estruturas variantes, cujos elementos dependem de parâmetros da componente. Linguagens como Durra e Darwin também incorporam estruturas de controle que possibilitam a descrição de configurações recursivas. Em uma linguagem experimental chamada Gerel[19], estendeu-se a linguagem Conic com comandos de seleção que permitem a programação de scripts de configuração genéricos através de um mecanismo para a atribuição dinâmica de variáveis de configuração e a avaliação de pré-condições.

Atributos de Componentes Em algumas linguagens de configuração é possível associar atributos a componentes e canais de conexão. Um dos atributos mais comuns associados a componentes são a máquina em que deverão executar, qual é o arquivo executável e os argumentos a serem usados em sua criação. Em outras linguagens, como Durra, é possível associar requerimentos funcionais e temporais, que são usados para verificar a compatibilidade semântica entre componentes antes de sua conexão. Nesta mesma linguagem é também possível descrever as características de um canal de conexão, como por exemplo se este é usado para uma comunicação síncrona ou assíncrona, e neste segundo caso, qual será o tamanho de seu buffer.

5.6 Interface Gráfica para Configuração

É notória a importância de uma boa interface gráfica para ambientes de desenvolvimento. Em particular, logo percebeu-se que a manipulação dinâmica de uma configuração introduz um grau maior de complexidade em um sistema que requer ferramentas para o monitoramento e gerenciamento da configuração, e que este gerenciamento pode ser muito facilitado se esta ferramenta tiver uma interface gráfica. Conic foi um dos primeiros sistemas a prover uma tal ferramenta[42], que executa na plataforma Macintosh. Através desta é possível não só mostrar graficamente a estrutura interna de um programa distribuído (executando em uma rede Unix), como também modificar esta estrutura de forma análoga ao uso dos comandos de reconfiguração CONIC. Uma das funções gráficas mais interessantes e inovadoras para a época era a função de *redesenho otimizado*, que ajustava automaticamente a posição dos retângulos (representando as componentes do sistema) de tal maneira a obter um número mínimo de intersecções entre as linhas conectando os mesmos. Baseados nesta experiência, foi desenvolvida então uma outra ferramenta[35, 36] integrada para o projeto e o desenvolvimento de um programa em Regis/Darwin. Além das funções da ferramenta anterior, ela também permite uma visualização integrada em forma textual e gráfica do código e da configuração de um programa, possui uma função de geração automática de código e de gravação de decisões de projeto e configuração.

Outros exemplos de ferramentas para a *programação visual* que dão suporte a uma manipulação gráfica de estruturas representando programas são o STILE[66] e o Designer's Notepad[28].

6 Exemplos de Ambientes de Programação

Nesta seção serão apresentados alguns ambientes de programação conhecidos que dão suporte à programação de aplicações reconfiguráveis dinamicamente. A escolha dos sistemas foi feita de modo a apresentar uma gama de diferentes paradigmas de meta-programação utilizados. Para cada ambiente, apresentaremos as suas principais características, e discutiremos os principais pontos positivos e negativos. Quando necessário, apresentaremos pequenos exemplos para ilustrar as facilidades e características dos sistemas.

6.1 Regis/Darwin

Regis[13] e Darwin[47] são linguagens que compõem um ambiente de programação desenvolvido no Imperial College de Londres e podem ser vistos como evoluções do sistema Conic.

Ao contrário da linguagem de configuração em Conic, Darwin é uma linguagem de configuração genérica projetada para ser usada com diversas linguagens de programação de componentes básicas, tais como C, C++ e Pascal. Suas principais características são a possibilidade de definição de estruturas de programas hierárquicas, parametrizáveis, auto-configuráveis, recursivas e com a facilidade de instanciação deferida ('lazy instantiation'). Esta última característica permite que uma componente seja criada somente quando ocorre uma chamada (isto é, um envio de mensagem) para uma de suas portas. Outra diferença com relação a Conic é que Darwin só pode ser usada para configuração dinâmica programada, isto é, nas quais as eventuais alterações estão pre-definidas no código Darwin das componentes compostas. Uma reconfiguração arbitrária (*ad-hoc*) do programa só pode ser feita no nível das macro-componentes de um sistema.

Regis é uma extensão de C++ para a programação das componentes primitivas de um programa. Ela contém uma série de classes para a criação, o controle e a manipulação de objetos *processo* (implementados como threads), objetos para comunicação síncrona e assíncrona, e objetos *address*, que podem conter qualquer endereço, incluindo endereços IP.

Além destas duas linguagens, o ambiente contém uma série de serviços básicos de apoio à execução. Um deles é o *remote execution daemon*, que permite criar processos remotamente, redirecionando a entrada e saída do processo remoto para o terminal do processo local, e permitindo especificar os valores iniciais para as variáveis de ambiente do novo processo. O outro é um servidor de nomes que permite que programas em Regis/Darwin interajam com outros programas através da exportação e importação de referências a pontos de comunicação próprios ou serviços externos.

6.2 Polylith

O Sistema Polylith foi desenvolvido pelo Prof. James Purtilo e seus alunos na Universidade de Maryland at College Park, como uma ferramenta de auxílio para a interconexão e execução de programas escritos em diferentes linguagens e para sistemas distribuídos heterogêneos. O conceito central em Polylith é o *software bus*[59], que é usado como uma interface (um elemento intermediário) na conexão de qualquer duas componentes do sistema. Entre outros, o software bus realiza a conversão de dados entre diferentes representações de máquina, encapsula todos os detalhes relativos aos protocolos de comunicação específicos sendo usados, e permite a instanciação de processos nos nodos da rede.

Além disto, Polylith dá suporte a uma instrumentação automática de programas para uma interconexão com outros módulos. Esta instrumentação consiste da geração de stubs (que acessam procedimentos do software bus) para diferentes formas de comunicação, incluindo RPC, comunicação assíncrona, pipes locais, etc. Esta instrumentação e a instanciação das componentes de um programa distribuído é determinada pele descrição da configuração do sistema usando a Module Interconnection Language (MIL). Através desta linguagem de configuração é possível associar atributos às componentes, tais como nome do arquivo executável ou a máquina destino, definir as interfaces de cada componente e interconectá-las umas às outras.

Baseado nesta infra-estrutura foi desenvolvida a ferramenta Surgeon[30], que instrumenta programas para reconfiguração dinâmica. Esta instrumentação consiste essencialmente da geração de procedimentos que permitem controlar o envio de mensagens entre componentes e possibilitam a transferência do estado entre processos criados a partir do mesmo código fonte.

6.3 Arjuna

Arjuna[60] é um ambiente para a programação orientada a objeto de aplicações distribuídas e tolerantes à falha baseada no paradigma de transações atômicas aninhadas e objetos persistentes. O sistema foi desenvolvido na University of Newcastle upon Tyne para redes Unix, implementado em C++ e se utiliza um mecanismo de invocação remota de objetos similar à chamada remota de procedimentos (RPC).

Arjuna implementa uma série de mecanismos para a criação remota de objetos, execução de ações atômicas, acesso a objetos persistentes e replicação de objetos.

Em Arjuna todo objeto é persistente no sentido em que o seu estado e o código binário para a execução de seus métodos ficam armazenados em um repositório de objetos não volátil denominado *object store*. Uma invocação de um método deste objeto (até então em estado passivo), causa uma requisição ao seu *servidor de objetos* associado, que pode ser localizado através de um serviço de nomes. Este servidor, tendo as informações e as permissões de acesso necessárias, carrega então o objeto (o seu estado e código) em sua memória e executa o método requisitado, retornando o resultado ao objeto cliente. Além disto todo programa distribuído em Arjuna é composto de uma ou mais transações atômicas concorrentes que manipulam conjuntos de objetos, cujos estados só são modificados no object store caso a transação correspondente tenha terminado com sucesso. Este modelo de processamento é parecido com aquele adotado no sistema Argus[45].

Como parte do sistema Arjuna, desenvolveu-se também o ambiente *Gandiva*[74], que permite a

configuração dinâmica de classes. O princípio básico utilizado é a separação de um programa em *classes de interfaces* e *classes de implementação*. Através do mecanismo de delegação, invocações de métodos da classe interface são redirecionados para métodos correspondentes da classe implementação associada. Além disto é possível redefinir dinamicamente a relação entre instâncias destas classes, permitindo assim uma alteração dinâmica do código que implementa um dado objeto. O ambiente permite também o uso de *classes de controle*, que podem ser associadas a classes de implementação e que podem controlar aspectos não funcionais das mesmas, como por exemplo a utilização de recursos ou a semântica do acesso concorrente aos métodos, etc.

6.4 Meta/Lomita

O Meta Toolkit[76, 50] foi desenvolvido no contexto do projeto ISIS[5] na Universidade de Cornell. O principal objetivo deste trabalho foi o de explorar o controle de sistemas reativos e distribuídos através de uma linguagem baseada em regras, chamada Lomita. A idéia básica em Meta é a de instrumentar um programa de aplicação com *sensores* e *atuadores*, através dos quais é possível monitorar e controlar um programa. Sensores são usados para mostrar o estado corrente de um processo, através da leitura do valor de variáveis em um processo. Atuadores, por sua vez são instruções que permitem a manipulação do valor de variáveis de cada processo, podendo assim ser usados para manipular o estado de execução do programa como um todo. No nível de controle, é possível então descrever regras que disparam sempre que suas condições (descritas em termos de valores lidos por sensores) sejam satisfeitas. Além disto, implementou-se em Meta um mecanismo de avaliação distribuída destas condições que é baseado nas facilidades de ordenação de eventos distribuídos presentes em ISIS.

Mesmo tendo sido desenvolvido principalmente para o controle e o gerenciamento de aplicações com vistas à tolerância a falha das mesmas, este sistema pode também ser utilizado para a reconfiguração dinâmica de programas.

7 O Problema da Sincronização

Um dos problemas inerentes à configuração dinâmica é a dificuldade em sincronizar as reconfigurações de um sistema com o processamento normal do mesmo. Embora este problema também surja de forma atenuada em sistemas uniprocessados, sua complexidade é maior em ambientes paralelos e distribuídos, devido a simultaneidade de ações. Assim, nesta seção iremos abordar o problema para esta classe de sistemas e apresentar algumas das soluções encontradas na literatura.

Para programas paralelos (e distribuídos), o problema da sincronização pode ser decomposto nos seguintes sub-problemas:

- Determinar os estados *seguros* do programa, isto é, nos quais uma reconfiguração não corrompe a consistência do sistema.
- Levar o programa (ou parte dele) para um destes estados *seguros*.
- Garantir que o programa não consegue entrar em um estado *não-seguro* durante a reconfiguração.
- Após a reconfiguração, garantir que o programa conseguirá retomar o processamento normal, o que eventualmente pode requerer a manipulação do estado interno de processos.

Devido a notória impossibilidade de se identificar e controlar os estados globais de um programa distribuído, pode-se ter uma idéia da dificuldade prática de solucionar os problemas acima mencionados.

Apesar disto, existem vários trabalhos que apresentam propostas de solução para o problema da sincronização, dos quais apresentaremos e discutiremos os mais interessantes.

Kramer e Magee[41, 40] propõem um algorítmo que determina, para cada tipo de reconfiguração estrutural, os conjuntos de processos de uma aplicação que precisam ser levados a um estado *passivo*, no qual não emitem e não recebem mensagens, e a um estado *semi-passivo*, no qual os processos são capazes de receber mensagens mas não iniciar novas interações com outros processos.

O principal problema desta solução é que para alguns algorítmos distribuídos isto requer uma paralisação de todo o conjunto de processos. Além disto, a proposta não leva em conta uma eventual necessidade de manipulação do estado interno dos processos.

O trabalho de Hofmeister[30], por sua vez, está baseado na infra-estrutura de comunicação e controle implementada pelo *software bus* de Polylith. Através desta é possível reconectar processos dinamicamente sem que as mensagens enviadas pela conexão desfeita sejam perdidas. Em vez disto, as mensagens são armazenadas nos stubs gerados até que a conexão seja restabelecida. Além desta facilidade, o ambiente Polylith permite definir funções para a codificação e decodificação do estado de processos e interfaces para a transferência destes estados de um processo para outro. Assim pode-se inicializar um novo processo a ser incluído em um grupo de processos exatamente com o estado de outro processo ativo ou desativado. A principal objeção a esta solução está na sua falta de generalidade, dada a sua forte dependência da infra-estrutura de comunicação provida no projeto Polylith.

Estes duas abordagem são classificadas como gerenciamento ativo de reconfiguração (*active change management*), em oposição ao gerenciamento passivo de reconfiguração (*passive change management*), no qual se associa as alterações de uma configuração à ocorrência de padrões de eventos nas interfaces das componentes envolvidas, tal como é descrito no trabalho de Etzkorn[24]. O problema desta abordagem é que ela implicitamente assume que o sistema irá espontaneamente atingir um estado seguro e que irá permanecer neste estado um tempo suficientemente longo para que a reconfiguração possa ser feita sem que haja uma corrupção da consistência.

Uma outra solução para os problemas de sincronização mencionados acima é basear todo o processamento em transações atômicas e objetos persistentes, tal como ocorre nos ambientes Arjuna e Argus. Neste modelo, toda reconfiguração pode ser naturalmente descrita como uma transação atômica, e devido ao próprio mecanismo de controle de concorrência de transações garante-se uma sincronização entre esta *transação de reconfiguração* e as demais transações no sistema. Tendo como base este modelo, Wheater e Shrivastava[75] descrevem um serviço para a restruturação dinâmica do agrupamento de objetos para efeito de controle de concorrência no acesso a grupos de objetos. Apesar do modelo de ações atômicas e objetos persistentes garantir automaticamente um sincronismo entre reconfiguração e processamento normal, deve-se ter em mente o elevado custo (em termos de desempenho) do controle de concorrência realizado. O controle de concorrência também é usado como base para o esquema de sincronização de configurações no sistema KTK (seção 4.4), que oferece a possibilidade de que um objeto seja replicado (com as chamadas para o objeto original sendo redirecionadas para a réplica) durante o processo de reconfiguração. A replicação é iniciada via requisição explícita do plano de configuração e envolve custos muitos altos, tendo sida empregada para reconfigurações de objetos de alta granularidade, em especial para objetos cujos fragmentos estão distribuídos em dezenas de processadores.

O modelo proposto pelo sistema operacional Spring (seção 3.2), baseado em subcontratos, define os pontos possíveis para manipulação de um objeto ou invocações, limitando desta forma onde e quando uma atividade de alteração do comportamento de um objeto pode ser acionada.

O projeto Synthetix (seção 3.1) define um esquema de reconexão dinâmica de rotinas especializadas cujos objetivos são invocação eficiente das rotinas especializadas e controle de concorrência entre os *threads* lidando com reconexão e os *threads* da aplicação. A solução se baseia nos conceitos de "quasi-invariantes" e "gatilhos vigilantes"[12].

Além destes, iremos discutir também outros trabalhos recentes sobre este assunto, como por exemplo

os trabalhos de Coatta[10] e Warren[73].

8 Problemas em Aberto

Nesta seção discutiremos outros problemas relacionados com reconfiguração dinâmica. Eles representam tópicos importantes para a área de configuração dinâmica de sistemas distribuídos e sistemas operacionais, para os quais soluções satisfatórias ainda não foram obtidas ou comprovadas.

8.1 Heterogeneidade

Dado que uma das principais vantagens da configuração dinâmica é o de permitir uma auto-adaptação de um sistema a *diferentes* ambientes de execução, a questão da heterogeneidade de ambientes precisa ser necessariamente considerada, principalmente em sistemas distribuídos. Nestes sistemas, heterogeneidade pode manifestar-se em diversos níveis, desde diferenças nos recursos disponíveis em cada nó da rede, como por exemplo processadores distintos, presença ou não de disco local ou características particulares de controladoras de dispositivos, até diferenças entre os sistemas operacionais e bibliotecas disponíveis em cada nó.

Ou seja, para que uma configuração dinâmica de um programa de aplicação seja possível em um ambiente heterogêneo teoricamente é necessário que se possa selecionar e/ou compor o código das componentes para cada nó do sistema. Isto requer de um lado uma descrição das características de cada nó da rede e por outro lado requer uma integração da infra-estrutura de configuração dinâmica com um sistema de controle de versões e variantes. No entanto, devido à complexidade e ao custo envolvido no processo de seleção e de link-edição quando comparados aos tempos de execução, praticamente não existem ambientes que tenham implementado tal integração. Ou seja, na maioria dos ambientes assume-se a existência prévia do código objeto (ou executável) das componentes para os diferentes tipos de arquiteturas em uma rede , apesar das limitações impostas por esta solução.

8.2 Concorrência e Conflitos entre Reconfigurações

Uma vez que sistemas podem ser usados simultaneamente por múltiplas aplicações, é preciso determinar como configuração dinâmica se comporta neste contexto. Em particular, o problema surge quando se trata da configuração de um sistema compartilhado, tal como um sistema operacional ou uma biblioteca compartilhada, uma vez que as propriedades exigidas por cada aplicação podem indicar a presença de componentes conflitantes. Uma das maneiras de resolver este problema poderia ser a de associar prioridades a classes de aplicações para configurar um sistema.

Quando as configurações dinâmicas são realizadas de forma descentralizada, surge o problema adicional da concorrência entre reconfigurações iniciadas por diferentes usuários ou agentes no sistema. O problema aqui é garantir que reconfigurações concorrentes não deixem o sistema com uma configuração inconsistente. Uma das possíveis soluções para este problema seria a de implementar reconfigurações na forma de transações atômicas, como é o caso em Arjuna, que eventualmente poderiam utilizar um protocolo de confirmação em duas fases sendo que na primeira fase seria verificada a possibilidade de realizar a reconfiguração e seriam tomadas as as devidas ações preparatórias, e em na segunda fase, seriam então instanciadas e conectadas as novas componentes do sistema.

Com exceção de poucos sistemas, poucos trabalhos por enquanto trataram destes dois problemas.

8.3 Configuração de Sistemas Existentes

Como foi mencionado anteriormente, configuração dinâmica requer que a aplicação ou o sistema com esta facilidade seja implementado em uma linguagem de programação específica ou que seja previamente instrumentado com chamadas a bibliotecas especiais. Isto naturalmente cria um certo problema para os sistemas já existentes, para os quais uma recodificação é economicamente inviável ou é até impossível, por exemplo devido a inexistência do código fonte ou do compilador correspondente. Neste caso, a única solução é a programação de componentes que realizem a interface (p.ex. através de canais de entrada e saída) com as partes do sistema original. Estas componentes, chamadas de *proxys* ou *wrappers*, permitem assim integrar estas partes pré-existentes de forma a obter um sistema configurável.

A comunidade de engenharia de software vem explorando o uso de reengenharia e engenharia reversa na manipulação de sistemas existentes (*legacy software*). É possível que resultados desta linha de pesquisa sejam futuramente usados na reestruturação de partes de um sistema existente de forma a inserir suporte para configuração dinâmica no código.

8.4 Multiplicidade e Escalabilidade

Outro grande problema diz respeito ao tamanho, à complexidade e à multiplicidade de interdependências entre componentes de sistemas atuais. Ou seja, está se tornando cada vez mais difícil fazer reconfigurações sólidas, dado que em tais sistemas muitas vezes a alteração em uma parte do sistema pode ter implicações indiretas na funcionalidade e no desempenho de muitas outras partes do sistema. Ao que tudo indica, este problema só tem solução através de uma estruturação hierárquica de sistemas complexos e do confinamento das reconfigurações a sub-configurações de tal estrutura. No entanto, são poucos os sistemas que têm uma arquitetura hierárquica reconfigurável. Além disto, ainda existem muitos problemas em aberto com relação à interação de reconfigurações entre diferentes níveis de uma hierarquia.

A configuração dinâmica de aplicações distribuídas apresenta o problema adicional de que diferentes partes do sistema podem estar executando em domínios de gerenciamento distintos com diferentes padrões de segurança, e interagindo através de redes de longo alcance (WANs), que podem apresentar características distintas de largura de banda e de confiabilidade, além de um tráfego muito variável. Por estes motivos muitos sistemas se limitam ao suporte a aplicações em redes locais homogêneas.

8.5 Uso em Sistemas Críticos e Sistemas de Tempo Real

O uso de configuração dinâmica para sistemas críticos apresenta uma série de problemas, que vão desde a dificuldade de garantir uma perfeita compatibilidade (consistência semântica) e não-interferência por exemplo entre uma componente sendo adicionada e o restante do sistema, até o problema de estimar a duração de uma reconfiguração. Este segundo é um grande problema especialmente para sistemas de controle de processos em tempo real, visto que durante o período de reconfiguração um sistema tipicamente apresenta um desempenho mais fraco não só pela interrupção temporária do funcionamento de algumas de suas partes como também pelo custo de execução do próprio mecanismo de reconfiguração. Isto pode eventualmente fazer com que os limites de tempo para a reação a eventos externos sejam ultrapassados, comprometendo assim todo o controle do processo. Por isto configuração dinâmica nestes sistemas têm sido adotada apenas de forma bem restrita e controlada.

Por outro lado, em algumas aplicações de tempo real são inúmeras as oportunidades para melhoria do desempenho via configuração, já que um esquema flexível de uso de recursos pode resultar em mais tarefas sendo terminadas dentro dos limites de tempo impostos. Existem sistemas projetados visando aplicações de tempo real onde os objetos podem ser configurados em termos do tipo de invocação (periódica, por eventos, assíncronas, com disparo automático de recuperação quando limites de tempo

são ultrapassados, etc) ou da forma de escalonamento de tarefas, mas neles os problemas acima descritos ainda não foram solucionados.

Os resultados nesta área devem avançar significativamente em consequência da atenção atual por parte de pesquisadores e da indústria que vem recebendo o desenvolvimento de aplicações como teleconferência e simulação distribuída para aplicações militares e de entretenimento (realidade virtual distribuída com nodos interagindo através de redes de longo alcance e tráfego variável).

8.6 Segurança

À medida que configuração dinâmica vá sendo incorporada a mais e mais sistemas, isto deverá também fazer crescer a preocupação com aspectos de segurança. Por permitirem a alteração de módulos e conexões, os elementos de reconfiguração representam uma nova oportunidade para a violação da integridade do sistema por agentes invasores. Nos últimos meses foram propostas técnicas para extensão segura do núcleo via autenticação (sem chaves criptográficas) em tempo de execução de rotinas a serem executadas em modo kernel.

Se no passado a presença de erros e limitações na implementação de serviços como *mail* foram responsáveis por problemas de segurança "históricos" como a *Internet Worm*[65], deficiências na implementação do suporte à configuração dinâmica podem eventualmente levar a sistemas com uma vulnerabilidade adicional inaceitável.

8.7 Implicações para o Projeto de Sistemas

Durante o desenvolvimento de software o projetista tem focalizado tipicamente nno desenho das componentes: a funcionalidade do sistema é decomposta em módulos/classes, representações para as componentes são escolhidas e interfaces são definidas. A escolha da forma de interação entre as componentes é frequentemente feita de modo implícito, com o esquema presente em uma linguagem de programação ou ambiente de execução sendo utilizado.

As técnicas para configuração dinâmica de sistemas abordadas neste texto baseiam-se, de uma forma ou de outra, no princípio de reconexão de elementos do software. Para que sistemas sejam concebidos levando-se em consideração as mudanças na configuração, é necessário que as fases de especificação e projeto de um sistema passem a incorporar explicitamente as decisões de interação entre componentes. Este é um dos pontos abordados na pesquisa sendo desenvolvida em *arquitetura de software*[27], em que o planejamento de futuras alterações no comportamento do sistema é uma das motivações principais da área. Há também trabalhos que pregam a necessidade de explicitar formas de interação entre componentes via elementos adicionais (*conectores*) em linguagens de configuração[70]).

A área de arquitetura de software e de outros tópicos relacionados com o desenvolvimento de software altamente flexível (como *patterns*) é recente, e ainda não é possível saber se haverá impacto sobre o desenvolvimento de software básico configurável, já que tradicionalmente estes sistemas foram considerados um domínio muito específico, em que as técnicas de engenharia de software nem sempre se aplicam. Além disto, ainda não se sabe como o desenvolvimento de componentes é influenciado pela necessidade de reconfiguração, ou seja, se a forma usual de projeto e implementação de serviços básicos precisará ser adaptada a estas novas exigências.

9 Desenvolvimentos Futuros

Se durante os anos 80 configuração dinâmica era ainda essencialmente um assunto de pesquisa que teve como resultado o desenvolvimento de diversos ambientes de programação e protótipos com esta facilidade, a partir dos anos 90 grande parte da experiência adquirida com esta pesquisa foi utilizada

para o desenvolvimento de sistemas e aplicações que utilizam de forma extensiva esta facilidade. Em paralelo, houve uma preocupação em sofisticar os mecanismos e as linguagens de programação e de configuração, bem como em solucionar o problema da sincronização mencionado na seção 7.

Além disto, passou-se a usar configuração dinâmica no contexto mais amplo de gerenciamento de aplicações, que nos últimos anos têm se tornado um tópico de crescente pesquisa e desenvolvimento, envolvendo inclusive esforços de padronização e de definição de arquiteturas e modelos gerais para o gerenciamento, tais como o ODP[32] e o DME[55] da Open Software Foundation e outros[3]. Neste contexto de gerenciamento de aplicações configuração dinâmica é combinada e integrada com outros serviços ou mecanismos tais como por exemplo o monitoramento de processos e recursos, serviços de nomes, o processamento de eventos, e o gerenciamento de réplicas.

Com a identificação dos mecanismos básicos de configuração necessários e sua implementação, passou-se também a estudar mais as estratégias de reconfiguração para os diferentes tipos de sistemas de acordo com os requisitos específicos de desempenho, tolerância a falha e adaptabilidade[63]. Em particular, surgiram pesquisas correlatas sobre o gerenciamento de serviços replicados[34, 46], domínios de gerenciamento[64], gerenciamento integrado de configuração, e linguagens para a descrição de estratégias de configuração[20, 51, 49].

Neste sentido pesquisa nesta área aponta para o desenvolvimento de sistemas auto-reguláveis que empregam configuração dinâmica como forma de adaptação automática às características variáveis da infra-estrutura computacional e dos requisitos de seus usuários. Tal flexibilidade vem se tornando uma característica cada vez mais exigida de sistemas de propósito geral, tais como sistemas operacionais, sistemas de apoio à execução em arquiteturas paralelas e distribuídas e subsistemas de comunicação. O objetivo principal é sempre o de fazer um uso otimizado dos recursos disponíveis para cada tipo de aplicação, garantindo o máximo de confiabilidade, segurança e transparência com relação a tecnologia sendo usada.

Ambientes e sistemas orientados a objeto certamente terão um papel cada vez mais relevante neste desenvolvimento, dado que o próprio modelo de orientação a objeto facilita uma adaptação e especialização de um sistema às necessidades particulares de cada aplicação. Isto pode ser verificado pelo grande número de sistemas operacionais adaptáveis que se baseiam neste modelo. À medida que o uso do paradigma de orientação a objetos no desenvolvimento de sistemas distribuídos foi ganhando mais força através de consórcios da indústria (OMG, OSF), as restrições iniciais ao uso do modelo devido a uma suposta ineficiência na execução foram desaparecendo.

Uma outra vertente cada vez mais importante em computação é a meta-programação, isto é, uma programação voltada para a configuração e a integração de sistemas compostos de um grande número de componentes eventualmente heterogêneas. Neste contexto, configuração dinâmica deve se tornar cada vez mais popular, uma vez que para um número cada vez maior de aplicações críticas se requer uma disponibilidade e confiabilidade muito alta, e ao mesmo tempo há a necessidade da introdução gradativa de nova funcionalidade, correções e/ou otimizações. No entanto, neste aspecto ainda existem alguns problemas resolvidos apenas parcialmente. Estes incluem por exemplo como tratar a evolução dinâmica de interfaces entre as componentes e como garantir a consistência de uma aplicação durante e após uma configuração. Em particular, este último problema é difícil porque envolve não só a consistência sintática entre as interfaces, mas também a semântica das operações e dos objetos compartilhados, que no entanto é predominantemente dependente da aplicação em questão. Ou seja, enquanto que o problema de interfaces evolutivas poderá vir a ser resolvido de forma genérica, este último problema deverá ter sempre soluções baseadas no conhecimento da aplicação em questão.

Referências

[1] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[2] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, M.J. Gardner, and R.W. Lichota. Building Fault Tolerant Distributed Applications with Durra. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 128–139. IEE, March 1992.

[3] M.A. Bauer, P.J. Finnigan, J.W. Hong, J. Pachl, and T.J. Teorey. An Integrated Distributed System Management Architecture. In *Proc. of the CASCON '93*. IBM Canada, October 1993.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[5] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. Tech. Report 1216, Dept. of Computer Science, Cornell University, Ithaca, USA, 1993. ftp.cs.cornell.edu:/pub/isis.

[6] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts 02139, March 1983.

[7] Peter Bosch and Sape Mullender. PFS: A distributed and customizable file system. In *Proc. of IWOOOS'96 (International Conference on Object-Oriented Operating Systems*, October 1996.

[8] John B. Carter, Bryan Fork, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: a tool for building efficient and flexible systems. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 198–202, Napa, CA, October 1993.

[9] R.S. Chin and S.T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[10] T. Coatta and G. Neufeld. Distributed Configuration Management Using Composite Objects and Constraints. In *Proc. of the 2nd Int. Workshop on Configurable Distributed Systems*, CMU, Pittsburgh, May 1994. IEEE.

[11] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1993.

[12] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 108–115. IEEE Computer Society Press, May 1996.

[13] S. Crane, N. Dulay, and K. Twidle J. Magee. *Regis Programmers Manual.* Imperial College London, release 0.6.8 edition, August 1995.

[14] Scott Danforth and Ira Forman. Reflections on metaclass programming in SOM. In *Proc. of OOPSLA '94*, pages 440–452. ACM Press, October 1994.

[15] F. DeRemer and H. H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, 1976.

[16] Peter Druschel. Efficient support for incremental customization of OS services. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, December 1993.

[17] Peter Druschel, Larry Peterson, and Norman Hutchinson. Beyond microkernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th Internacional Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[18] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proc. of Conference on Programming Languages Design and Implementation*, June 1997.

[19] M. Endler. *A Language for High-Level Programming of Dynamic Reconfiguration*. PhD thesis, Technische Universität Berlin, Franklinstraße 28/29, 1000 Berlin 12, Germany, November 1992. (published as GMD Bericht Nr. 210, by R. Oldenbourg Verlag).

[20] M. Endler and A. D'Souza. Supporting Distributed Application Management in Sampa. In *Proc. 3rd International Conference on Configurable Distributed Systems, Annapolis, USA*, pages 177–184. IEEE, May 1996.

[21] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th Symposium on Operating Systems Principles*. ACM Press, December 1995.

[22] J.S. Auerbach et al. High-Level Language Support for Programming Distributed Systems. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 320–330. IEEE, April 1992.

[23] J.S. Auerbach et al. Concert/C Tutorial and User Guide: An Introduction to a Language for Distributed C Programming. Technical report, IBM T. J. Watson Research Center, January 1995.

[24] G. Etzkorn. Change programming in distributed systems. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 140–151. IEE, March 1992.

[25] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable components for OS implementation. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997.

[26] Bryan Fork, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symposium on Operating Synstem Design and Implementation(OSDI)*, October 1996.

[27] David Garlan and Mary Shaw. An introductin to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 2, pages 1–39. World Scientific Publishing Company, 1993.

[28] N. Haddley and I.Sommerville. Integrated support for system design. *IEE Software Engineering Journal*, 5(6):331–338, November 1990.

[29] Graham Hamilton, Michael Powell, and James Mitchell. Subcontract: A flexible base for distributed computing. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79. ACM Press, December 1993.

[30] C. Hofmeister, E. White, and J. Purtilo. SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 164–175. IEE, March 1992.

[31] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[32] ISO/IEC/JTC1/SC21/WG7. *Basic Reference Model of Open Distributed Processing, parts 1-5*, 1992.

[33] M. Jones. Interposition agents: Transparently interposing user code at the system interface. *Operating System Review*, 27(5):69–79, December 1993. Proc. 14th ACM Symp. on Operating Systems Principles.

[34] C. Karamanolis and J.N. Magee. A Replication Protocol to Support Dynamically Configurable Groups of Servers. In *3rd Int. Conference on Configurable Distributed Systems(ICCDS96)*, pages 161–168, May 1996.

[35] Jeff Magee Keng Ng, Jeff Kramer and Naranker Dulay. The Software Architect's Assistant - A Visual Environment for Distributed Programming. In *Proc. of the Hawaii Int. Conference on System Sciences (HICSS-28)*, pages 254–263. IEEE, Computer Society Press, January 1995.

[36] Jeff Magee Keng Ng, Jeff Kramer and Naranker Dulay. *A Visual Approach to Distributed Programming*, pages 7–31. Kluwer Academic Publishers, February 1996.

[37] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–13. ACM PRESS, Dec 1993.

[38] Gregor Kiczales and John Lamping. Operating systems: Why object oriented? In *International Workshop in Object Oriented Operating Systems*, pages 25–30, December 1993.

[39] Richard T. Kouzes, James D. Myers, and W. A. Wulf. Collaboratories: Doing science on the internet. *IEEE Computer*, 29(8):40–46, August 1996.

[40] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro90)*, Tel Aviv, Israel, May 1990.

[41] J. Kramer, J. Mage, and A. Young. A Refined Model of Change Management in Distributed Systems. In *Proc. of the 3rd Int. Workshop on Large Grain Parallelism*. SEI/CMU, October 1989.

[42] J. Kramer, J. Magee, and K. Ng. Graphical configuration programming. *IEEE Computer*, 22(10):53–65, October 1989.

[43] Willy S. Liao, See-Mong Tan, and Roy H. Campbell. Fine-grained, dynamic user customization of operating systems. In *Proc. of IWOOOS'96 (International Conference on Object-Oriented Operating Systems*, October 1996.

[44] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[45] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[46] M.C. Little and S.K. Shrivastava. Using Application Specific Knowledge for Configuring Object Replicas. In *Proc. of the 3rd. Int. Workshop on Configurable Distributed Systems*, pages 169–176. IEEE, May 1996.

[47] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 102–117. IEE, March 1992.

[48] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, SE-15(6), June 1989.

[49] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, Department of Computing, Imperial College London, December 1995.

[50] K. Marzullo, R. Cooper, M.D. Wood, and K.P. Birman. Tools for Distributed Application Management. *IEEE Computer*, 24(8):42–51, August 1991.

[51] N.H. Minsky. Governing Distributed Systems: from Protocols to Laws. In B.D. Shriver, editor, *Proc. of the 24th Hawaii International Conference on System Sciences*, pages 418–427. IEEE Computer Society Press, January 1991.

[52] Bodhisattwa Mukherjee and Karsten Schwan. Evaluation of the adaptation tecniques in Kernel Tool Kit (KTK). In *Proc. of the Third International Conference on Configurable Distributed Systems*, pages 228–235. IEEE Computer Society Press, May 1996.

[53] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, 1:259–270, 1994.

[54] Object Management Group. The Common Object Request Broker: Architecture and Specification. Available from the OMG Web site (http://www.omg.org), July 1995.

[55] Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142. *OSF Distributed Management Environment Architecture (DME)*, 1992.

[56] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proc. of the 2nd Symposium on Operating System Design and Implementation (OSDI)*, October 1996.

[57] Calton Pu, Tito Autrey, and Andrew Black. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, Dec 1995.

[58] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[59] J. Purtilo. The polylith software bus. Tech. Report TR-2469, University of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, University of Maryland, September 1990.

[60] S.K. Shrivastava and S.M. Wheater. Implementing Fault-Tolerant Distributed Applications using Multi-coloured Actions. In *Proc. Int. Conference Distributed Computing Systems*, pages 1–9. IEEE Computer Society, June 90.

[61] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[62] Dilma Menezes da Silva and Karsten Schwan. CTK: configurable object abstractions for multiprocessors. Technical Report GIT-CC-97-03, Georgia Institute of Technology, Atlanta, GA 30332, January 1997. Submitted to IEEE Transactions on Software Engineering.

[63] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4):333–360, February 1994.

[64] M. Sloman and J. Moffett. Managing Distributed Systems. Domino Report A1/IC/1.2, Imperial College London, August 1990.

[65] E. H. Spafford. The internet worm: Crisis and aftermath. *Communications of the ACM*, 32:678–687, June 1989.

[66] M.P. Stovsky and B.W. Weide. STILE: A GRaphical Design and Development Environment. In *COMPCON Spring '87*, February 1987.

[67] R. Strom and S. Yemini. The NIL Distributed System Programming Language: A Status Report. Research Report RC 10864, IBM T.J. Watson Research Center, April 1984.

[68] R.E. Strom. NIL: An Integrated Language and System for Distributed Programming. *SIGPLAN Notices*, 18(6):73–82, June 1983.

[69] R.E. Strom. Hermes: A High-Level Process-Based Language for Reliable Distributed Computing. In *Proc. of the 3rd Int. Workshop on Large Grain Parallelism.* SEI/CMU, October 1989.

[70] Bala Swaminathan and Kenneth J. Goldman. Data handles and virtual connections. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 19–26, May 1996.

[71] Leendert van Doorn, Philip Homburg, and Andrew S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proc. of the IEEE Workshop on Hot Topics In Operating Systems*, 1995.

[72] Alistair C. Veich and Normal C. Hutchinson. Kea – a dynamically extensible and configurable operating system kernel. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 236–242, May 1996.

[73] I. Warren and I. Sommerville. A Model for Dynamic Configuration which Preserves Application Integrity. In *Proc. 3rd International Conference on Configurable Distributed Systems, Annapolis, USA*, pages 81–88. IEEE, May 1996.

[74] S.M. Wheater and M.C. Little. The Design and Implementation of a Framework for Extensible Software. Broadcast project tech. report, Dept. of Computer Science, Univ. of Newcastle upon Tyne, http://arjuna.ncl.ac.uk/arjuna/papers.html, 1995.

[75] S.M. Wheater and S. Shrivastava. Exercising Application Specific Run-time Control Over Clustering of Objects. In *Proc. of the 2nd Int. Workshop on Configurable Distributed Systems*, March 1994.

[76] M.D. Wood. *Fault-tolerant Management of Distributed Applications using the Reactive System Architecture.* PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, January 1992.

[77] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. of Object-Oriented Programming Systems, Languages and Applications*. ACM Press, October 1992.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1994 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

Flavio S. Corrêa da Silva
AN ALGEBRAIC VIEW OF COMBINATION RULES
RT-MAC-9401, Janeiro de 1994, 10 pp.

Flavio S. Corrêa da Silva e Junior Barrera
AUTOMATING THE GENERATION OF PROCEDURES TO ANALYSE BINARY IMAGES
RT-MAC-9402, Janeiro de 1994, 13 pp.

Junior Barrera, Gerald Jean Francis Banon e Roberto de Alencar Lotufo
A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9403, Janeiro de 1994, 28 pp.

Flavio S. Corrêa da Silva
ON THE RELATIONS BETWEEN INCIDENCE CALCULUS AND FAGIN-HALPERN STRUCTURES
RT-MAC-9404, abril de 1994, 11 pp.

Junior Barrera; Flávio Soares Corrêa da Silva e Gerald Jean Francis Banon
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES
RT-MAC-9405, abril de 1994, 15 pp.

Valdemar W. Setzer; Cristina G. Fernandes; Wania Gomes Pedrosa e Flavio Hirata
UM GERADOR DE ANALISADORES SINTÁTICOS PARA GRAFOS SINTÁTICOS SIMPLES
RT-MAC-9406, abril de 1994, 16 pp.

Siang W. Song
TOWARDS A SIMPLE CONSTRUCTION METHOD FOR HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9407, maio de 1994, 13 pp.

Julio M. Stern
MODELOS MATEMATICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9408, maio de 1994, 50 pp.

Imre Simon
STRING MATCHING ALGORITHMS AND AUTOMATA
RT-MAC-9409, maio de 1994, 14 pp.

Valdemar W. Setzer e Andrea Zisman
*CONCURRENCY CONTROL FOR ACCESSING AND COMPACTING B-TREES**
RT-MAC-9410, junho de 1994, 21 pp.

Renata Wassermann e Flávio S. Corrêa da Silva
TOWARDS EFFICIENT MODELLING OF DISTRIBUTED KNOWLEDGE USING EQUATIONAL AND ORDER-SORTED LOGIC
RT-MAC-9411, junho de 1994, 15 pp.

Jair M. Abe, Flávio S. Corrêa da Silva e Marcio Rillo
PARACONSISTENT LOGICS IN ARTIFICIAL INTELLIGENCE AND ROBOTICS
RT-MAC-9412, junho de 1994, 14 pp.

Flávio S. Corrêa da Silva, Daniela V. Carbogim
A SYSTEM FOR REASONING WITH FUZZY PREDICATES
RT-MAC-9413, junho de 1994, 22 pp.

Flávio S. Corrêa da Silva, Jair M. Abe, Marcio Rillo
MODELING PARACONSISTENT KNOWLEDGE IN DISTRIBUTED SYSTEMS
RT-MAC-9414, julho de 1994, 12 pp.

Nami Kobayashi
THE CLOSURE UNDER DIVISION AND A CHARACTERIZATION OF THE RECOGNIZABLE Z-SUBSETS
RT-MAC-9415, julho de 1994, 29pp.

Flávio K. Miyazawa e Yoshiko Wakabayashi
AN ALGORITHM FOR THE THREE-DIMENSIONAL PACKING PROBLEM WITH ASYMPTOTIC PERFORMANCE ANALYSIS
RT-MAC-9416, novembro de 1994, 30 pp.

Thomaz I. Seidman e Carlos Humes Jr.
SOME KANBAN-CONTROLLED MANUFACTURING SYSTEMS: A FIRST STABILITY ANALYSIS
RT-MAC-9501, janeiro de 1995, 19 pp.

C.Humes Jr. and A.F.P.C. Humes
STABILIZATION IN FMS BY QUASI- PERIODIC POLICIES
RT-MAC-9502, março de 1995, 31 pp.

Fabio Kon e Arnaldo Mandel
SODA: A LEASE-BASED CONSISTENT DISTRIBUTED FILE SYSTEM
RT-MAC-9503, março de 1995, 18 pp.

Junior Barrera, Nina Sumiko Tomita, Flávio Soares C. Silva, Routo Terada
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY PAC LEARNING
RT-MAC-9504, abril de 1995,16 pp.

Flávio S. Corrêa da Silva e Fabio Kon
CATEGORIAL GRAMMAR AND HARMONIC ANALYSIS
RT-MAC-9505, junho de 1995,17 pp.

Henrique Mongelli e Routo Terada
ALGORITMOS PARALELOS PARA SOLUÇÃO DE SISTEMAS LINEARES
RT-MAC-9506, junho de 1995, 158 pp.

Kunio Okuda
PARALELIZAÇÃO DE LAÇOS UNIFORMES POR REDUÇÃO DE DEPENDÊNCIA
RT-MAC-9507, julho de 1995, 27 pp.

Valdemar W. Setzer e Lowell Monke
COMPUTERS IN EDUCATION: WHY, WHEN, HOW
RT-MAC-9508, julho de 1995, 21 pp.

Flávio S. Corrêa da Silva
REASONING WITH LOCAL AND GLOBAL INCONSISTENCIES
RT-MAC-9509, julho de 1995, 16 pp.

Julio M. Stern
MODELOS MATEMÁTICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9510, julho de 1995, 43 pp.

Fernando Iazzetta e Fabio Kon
A DETAILED DESCRIPTION OF MAXANNEALING
RT-MAC-9511, agosto de 1995, 22 pp.

Flávio Keidi Miyazawa e Yoshiko Wakabayashi
POLYNOMIAL APPROXIMATION ALGORITHMS FOR THE ORTHOGONAL Z-ORIENTED 3-D PACKING PROBLEM
RT-MAC-9512, agosto de 1995, pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON COLLECTIONS OF CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MORPHOLOGICAL MACHINES
RT-MAC-9513, agosto de 1995, 84 pp.

Marco Dimas Gubitoso e Jörg Cordsen
PERFORMANCE CONSIDERATIONS IN VOTE FOR PEACE
RT-MAC-9514, novembro de 1995, 18pp.

Carlos Eduardo Ferreira e Yoshiko Wakabayashi
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS COMBINATÓRIOS: TEORIA, ALGORITMOS E APLICAÇÕES
RT-MAC-9515, novembro de 1995, 45 pp.

Markus Endler and Anil D'Souza
SUPPORTING DISTRIBUTED APPLICATION MANAGEMENT IN SAMPA
RT-MAC-9516, novembro de 1995, 22 pp.

Junior Barrera, Routo Terada,
Flávio Corrêa da Silva and Nina Sumiko Tomita
*AUTOMATIC PROGRAMMING OF MMACH'S FOR OCR**
RT-MAC-9517, dezembro de 1995, 14 pp.

Junior Barrera, Guillermo Pablo Salas and Ronaldo Fumio Hashimoto
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMING OF MMACH'S
RT-MAC-9518, dezembro de 1995, 14 pp.

Daniela V. Carbogim and Flávio S. Corrêa da Silva
FACTS, ANNOTATIONS, ARGUMENTS AND REASONING
RT-MAC-9601, janeiro de 1996, 22 pp.

Kunio Okuda
REDUÇÃO DE DEPENDÊNCIA PARCIAL E REDUÇÃO DE DEPENDÊNCIA GENERALIZADA
RT-MAC-9602, fevereiro de 1996, 20 pp.

Junior Barrera, Edward R. Dougherty and Nina Sumiko Tomita
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY DESIGN OF STATISTICALLY OPTIMAL OPERATORS IN THE CONTEXT OF COMPUTATIONAL LEARNING THEORY.
RT-MAC-9603, abril de 1996, 48 pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MMACH'S
RT-MAC-9604, abril de 1995, 66 pp.

Kunio Okuda
CYCLE SHRINKING BY DEPENDENCE REDUCTION
RT-MAC-9605, maio de 1996, 25 pp.

Julio Stern, Fabio Nakano e Marcelo Lauretto
REAL: REAL ATTRIBUTE LEARNING FOR STRATEGIC MARKET OPERATION
RT-MAC-9606, agosto de 1996, 16 pp.

Markus Endler
SISTEMAS OPERACIONAIS DISTRIBUÍDOS: CONCEITOS, EXEMPLOS E TENDÊNCIAS
RT-MAC-9607, agosto de 1996, 120 pp.

Hae Yong Kim
CONSTRUÇÃO RÁPIDA E AUTOMÁTICA DE OPERADORES MORFOLÓGICOS E EFICIENTES PELA APRENDIZAGEM COMPUTACIONAL
RT-MAC-9608, outubro de 1996, 19 pp.

Marcelo Finger
NOTES ON COMPLEX COMBINATORS AND STRUCTURALLY-FREE THEOREM PROVING
RT-MAC-9609, dezembro 1996, 28 pp.

Carlos Eduardo Ferreira, Flávio Keidi Miyazawa e Yoshiko Wakabayashi (eds)
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS DE CORTE E EMPACOTAMENTO
RT-MAC-9610, dezembro de 1996, 65 pp.

Carlos Eduardo Ferreira, C. C. de Souza e Yoshiko Wakabayashi
REARRANGEMENT OF DNA FRAGMENTS: A BRANCH-AND-CUT ALGORITHM
RT-MAC-9701, janeiro de 1997, 24 pp.

Marcelo Finger
NOTES ON THE LOGICAL RECONSTRUCTION OF TEMPORAL DATABASES
RT-MAC-9702, março de 1997, 36 pp.

Flávio S. Corrêa da Silva, Wamberto W. Vasconcelos e David Robertson
COOPERATION BETWEEN KNOWLEDGE BASED SYSTEMS
RT-MAC-9703, abril de 1997, 18 pp.

Junior Barrera, Gerald Jean Francis Banon, Roberto de Alencar Lotufo, Roberto Hirata Junior
MMACH: A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9704, maio de 1997, 67 pp.

Julio Michael Stern e Cibele Dunder
PORTFÓLIOS EFICIENTES INCLUINDO OPÇÕES
RT-MAC-9705, maio de 1997, 29 pp.

Junior Barrera e Ronaldo Fumio Hashimoto
COMPACT REPRESENTATION OF W-OPERATORS
RT-MAC-9706, julho de 1997, 13 pp.

Dilma M. Silva e Markus Endler
CONFIGURAÇÃO DINÂMICA DE SISTEMAS
RT-MAC-9707, agosto de 1997, 35 pp