UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Troca Dinâmica de Versões de Componentes de Programas no Modelo de Objetos

por

WERNER HAETINGER

Dissertação submetida à avaliação, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Drª Maria Lúcia Blanck Lisbôa
Orientadora

Porto Alegre, julho de 1998

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Haetinger, Werner

Troca Dinâmica de Versões de Componentes de Programas no Modelo de Objetos / por Werner Haetinger - Porto Alegre: CPGCC da UFRGS, 1998.

83 f.: il.

Dissertação de Mestrado - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre - RS, 1998. Orientadora: Lisbôa, Maria Lúcia Blanck

1. Tolerância a Falhas, 2. Orientação a Objetos, 3. Reflexão Computacional. 4. Substituição Dinâmica de Versões de Software. I. Lisbôa, Maria Lúcia Blanck. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Maria Panizzi
Pró-Reitor de Pós-Graduação: Prof. Philippe Olivier Alexandre Navaux
Coordenador do CPGCC: Profa.: Carla Maria Dal Sasso Freitas
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Chegando ao fim deste trabalho, mais do que uma conquista pessoal, este documento representa a cooperação de várias pessoas que contribuíram no seu desenvolvimento. Muito obrigado a todas as pessoas que contribuíram, direta ou indiretamente, para o desenvolvimento deste trabalho. De forma especial:

À professora Maria Lúcia Lisboa pela orientação e pelo auxílio despendido sempre que necessário.

Aos demais colegas do grupo de tolerância a falhas pelo interesse, pela troca de idéias e pelas valiosas contribuições.

À UNISC - UNIVERSIDADE DE SANTA CRUZ DO SUL pela oportunidade que me foi dada.

À CAPES pela bolsa de estudos.

Sumário

Lista de Abreviaturas

JDK - Java Development Kit
LPOO - Linguagens de Programação Orientadas a Objetos
MO - Meta-objeto
MOTF - Meta-objetos Tolerantes a Falhas
MVJ - Máquina Virtual Java
O-O - Orientação a Objetos
OV - Objeto de Validação
POO - Programação Orientada a Objetos
PV - Programa de Validação
RC - Reflexão Computacional
RPC - Remote Procedure Call
STR - Sistemas de Tempo-Real
TF - Tolerância a Falhas

Lista de Figuras

Lista de Quadros

Resumo

A manutenção de software é uma realidade presente em todos os sistemas de computação, gerando a necessidade de novas versões que alterem as funcionalidades existentes no software ou adicionem novas. Particularmente, sistemas de tempo-real nem sempre podem ser descontinuados tornando-se indisponíveis para realizar a instalação de uma nova versão. Tais sistemas evidenciam a necessidade de substituição de componentes, representados por funções, procedimentos, módulos ou objetos, durante o processo de execução do programa ou sistema. Outrossim, após ser realizada a substituição da versão, o componente não pode apresentar falha sob pena de comprometer o fornecimento dos seus serviços. Portanto, constata-se a importância de novas técnicas de manutenção de software que não prejudiquem a sua disponibilidade e confiabilidade. A abordagem aqui proposta é utilizar uma arquitetura reflexiva aliada a técnicas típicas do domínio da tolerância a falhas para promover a separação entre as atividades de substituição e validação de componentes e as funcionalidades executadas pelo próprio componente. No decorrer deste trabalho são apresentados diversos cenários de sistemas que podem se beneficiar da troca dinâmica de componentes e abordadas várias facetas do problema de substituição. A proposta é apoiada por um estudo de caso, implementado na linguagem de programação Java e seus diferentes protocolos de reflexão computacional.

PALAVRAS-CHAVE: Tolerância a Falhas, Orientação a Objetos, Reflexão Computacional, Substituição Dinâmica de Versões de Software.

Abstract

Software maintenance is a present reality in all computational systems. This demands the frequent installation of new versions. Usually, real-time systems cannot be interrupted to install a new version. For such systems, the replacement of components, represented by functions, procedures, modulus or objects, must be performed during the execution of the program or system. Even when the old version has been replaced, the new one should not contain faults that could invalidate its services. Therefore, we need new software maintenance techniques that can mantain the system availability and realibility. The approach proposed here consists in using a reflective architecture along with techniques which are typical of the fault tolerant domain. The procedure is carried out by keeping a clear separation between validation activities and the functions executed by the component itself. We present several scenarios to which the dynamical exchange of components can be applied. Different aspects of the replacing issue are also addressed. The proposal is supported by a specific application which has been implemented in the Java language and its different protocols of computational reflection.

KEYWORDS: Fault Tolerance, Object Orientation, Computational Reflection, Dynamic Software Version Change.

1 Características do Problema

Este capítulo define o contexto do problema e introduz a abordagem adotada para a sua solução.

1.1 Introdução

Todo e qualquer software é desenvolvido, independente de plataforma e tecnologia, segundo uma seqüência básica de grandes atividades, denominada ciclo de vida. Esta seqüência de atividades compreende seis etapas [PRE88]: planejamento, análise, projeto, codificação, teste e manutenção. Enquanto as primeiras etapas são cumpridas durante o processo de desenvolvimento, a etapa de manutenção refere-se a alterações do software feitas após a sua liberação como um produto pronto para ser usado por um ou mais clientes.

Após desenvolvido, o software pode ser multiplicado de forma trivial, por simples cópia; é justamente cada uma das cópias deste produto que será usada para prestar serviços a seus usuários. Segundo considerações de Ghezzi [GHE91], uma definição geral de um produto de software compreende todos os subprodutos gerados durante o processo de desenvolvimento: a especificação de requisitos, o projeto, o código fonte, os dados de teste, etc. Esta definição torna possível a produção de diferentes versões de um mesmo produto, de forma a atender a diferentes usuários e adequar-se à variedade de ambientes de execução. A esse respeito, diz Juran, citado em [FER95], "Qualidade é a adequação ao uso do ponto de vista do cliente".

No decorrer de sua existência como produto, a sobrevivência de um software é diretamente relacionada com a sua maleabilidade. A possibilidade de alterar o produto, sem necessariamente alterar o seu projeto, pode ser explorada de forma positiva, objetivando a evolução e adaptação do software. Sob o ponto de vista operacional, qualquer software pode ser facilmente alterado, visto que consiste de linhas de código na forma de um texto. Porém, sob o ponto de vista funcional, a alteração de código pode implicar a alteração de seu projeto, do qual esse software é apenas uma instância.

Das breves considerações acima, depreende-se que um software é um produto que, inevitavelmente, sofre alterações durante o seu ciclo de vida. E ainda que, a etapa denominada de manutenção difere do sentido usual de manutenção de produtos, visto que compreende todas as possíveis alterações realizadas no software, para fins de correção de imperfeições, adaptações, extensões ou para introduzir melhorias, podendo resultar em um produto diferente do inicialmente concebido em seu projeto. Como conseqüência, a etapa de manutenção pode exigir técnicas de implementação distintas das utilizadas durante a fase de desenvolvimento.

1.2 Manutenção de software

A utilização de métodos de especificação formal, avanços na tecnologia de ferramentas de apoio ao desenvolvimento de software, o uso de técnicas de engenharia de software, ferramentas para a prototipação, assim como as técnicas de testabilidade de software, auxiliam na construção de sistemas sempre maiores e mais complexos.

Mas estes sistemas não são livres de erros e além disso, só conseguem suprir aquelas necessidades que lhes foram antecipadas. Assim, apesar dos avanços, o software ainda necessita ser modificado depois de colocado em operação. Estas alterações ocorrem em maior quantidade no início e no fim da vida de um sistema, podendo representar de 50 a 70% do esforço total dedicado a um software [PRE88].

Outro aspecto a considerar é a possibilidade de antecipar, ainda na fase de projeto, os tipos de alterações que o software poderá sofrer durante a sua fase operacional. Alguns exemplos de tipos de alterações são assim identificadas em [GHE91]: alteração em algoritmos, em representação de dados, na máquina virtual, em dispositivos periféricos, e no ambiente social (por exemplo, em legislação). A antecipação permite construir componentes de software que isolem os aspectos sujeitos a alterações, de forma a facilitar a sua substituição. Para atender a este aspecto, o modelo de objetos é particularmente adequado: permite confinar, encapsular, esconder e proteger estruturas de dados e correspondente código de manipulação, oferecendo seus serviços por meio de interfaces bem definidas e estáveis. A estabilidade das interfaces pode ser assegurada mesmo que sejam feitas alterações em qualquer das propriedades – dados ou código – do componente.

As alterações de um software podem ser caracterizadas como manutenção corretiva e manutenção adaptativa. A manutenção corretiva compreende a remoção de erros latentes introduzidos a nível de projeto e desenvolvimento do sistema que são detectados durante a operação; ou as alterações visando estender o sistema acrescentando-lhe melhorias na eficiência e no desempenho de sua funções. A manutenção adaptativa consiste em fazer a adição de novas funções à aplicação ou a alteração de determinadas funções do software visando adequá-lo à transformações do mundo real.

A manutenção adaptativa pode ser vista como um processo de evolução do software: a partir do produto originado pelo processo de desenvolvimento, novas versões são geradas ao longo de seu ciclo de vida, sendo que cada nova versão se distingue da anterior por conter novas funcionalidades ou por substituir a implementação de uma funcionalidade já existente.

A evolução de um software é o tema central deste trabalho, cuja preocupação é a continuidade da disponibilidade dos serviços durante a transição de uma versão de software, que se encontra em operação, para outra versão mais atualizada.

1.3 A Transição

Depois que um sistema entrou em regime de operação normal e precisa sofrer uma alteração, a abordagem mais comum é descontinuar o programa que está em execução e então substituí-lo pela nova versão. O sistema já alterado é então reinicializado, passando a estar novamente disponível ao usuário. O tempo que passa entre o cancelamento do programa antigo até a disponibilização do programa novo é chamado de tempo transiente que, muitas vezes não é desprezível.

Esta interrupção ocasiona um atraso de tempo e necessariamente implica a indisponibilidade do serviço aos usuários do software, e, à medida que os usuários

tornam-se mais dependentes de um sistema, também vão se tornando mais intolerantes a estas interrupções.

Em aplicações críticas estas interrupções são inaceitáveis. Para muitas companhias o custo da desativação de um sistema para manutenção pode se tornar proibitivo. Por exemplo desabilitar o sistema de processamento de transações bancárias pode ter conseqüências econômicas significativas, particularmente se a companhia tem reputação de prover alta disponibilidade dos serviços. Em empresas de telecomunicações os padrões de indisponibilidade do sistema são menores que 2 horas em 40 anos [WEB96].

Se o software a ser modificado faz parte de um sistema distribuído, onde os programas interagem em redes geograficamente distribuídas, os problemas de substituição do software podem se tornar mais complexos. Além do sistema ficar indisponível, a substituição de um programa deve ser coordenada. Se um computador começar a executar a nova versão do software enquanto os outros continuam a executar a versão antiga, pode haver troca de dados incorretos.

Constata-se que há necessidade de novas técnicas para manutenção que não interrompam a operação do sistema por longos períodos. Uma destas técnicas é utilizar um sistema que faz a substituição da versão de um programa sem interrompê-lo. Um sistema de modificação dinâmica 'on-the-fly' de software elimina a etapa de desativação do sistema para instalar uma nova versão do software, tornando mais rápidos os procedimentos de fazer reparos ou incrementar as funcionalidades da aplicação que está executando, provendo assim, o serviço contínuo aos usuários do sistema e eliminando os custos associados a um cancelamento do sistema.

Constata-se dois efeitos práticos da substituição dinâmica de componentes de software: um de cunho técnico, referente à disponibilidade e confiabilidade do sistema; e o outro de cunho psicológico, visto ser mais fácil convencer as pessoas da confiabilidade do sistema com o argumento que o software pode sofrer manutenção sem prejuízo de seus serviços.

A transição dinâmica de versões de software encontra sua maior aplicabilidade em sistemas de tempo-real, que controlam dispositivos físicos, devido a estes sistemas possuírem alta exigência de confiabilidade e disponibilidade. Usualmente desenvolvidos para atuar em ambientes concorrentes, distribuídos e/ou de tempo-real, a exemplo de automação bancária, comercial e industrial, estes sistemas devem ser projetados com características de adaptabilidade.

1.4 As Premissas

As premissas básica adotadas no contexto deste trabalho são:

- Um software é estruturado como um conjunto de componentes, onde cada componente é responsável por uma ou mais funcionalidades da aplicação.
- A evolução de um software implica a substituição de um ou mais componentes.
- A nova versão não altera a arquitetura do software da versão anterior, ou seja, é mantida a forma de organização e de interação entre os componentes do programa.

Estas premissas servem de base para as especializações a seguir:

- O software foi desenvolvido no modelo de objetos, no qual um objeto é considerado um componente atômico de uma aplicação, possui uma funcionalidade bem definida e encapsulada.

- Os componentes a serem substituídos são similares, ou seja, oferecem o mesmo serviço, podendo diferir na suas respectivas implementações.

- A unidade de substituição é uma classe de objetos; uma classe pode ser substituída por outra similar, porém não pode ser retirada ou completamente modificada sem comprometer o produto final.

O conjunto de serviços de um objeto é representado por um conjunto de interfaces aqui denominadas de protocolo do objeto. Se dois objetos A e B possuem o mesmo protocolo e o mesmo comportamento externo, estes objetos podem ser considerados equivalentes, independentemente de sua estrutura interna. Diz-se que ambos objetos possuem equivalência funcional. Portanto a substituição de um objeto A por B não causará problemas à aplicação, desde que o mesmo serviço seja realizado [LIS95c]. Por outro lado, existindo a similaridade de serviços, mas não de protocolo, a equivalência ainda pode ser obtida através de conversão de interfaces.

As hipóteses de similaridade não consideram que os componentes são instâncias de mesma classe (réplicas), pois não faz sentido substituir réplicas de componentes idênticos, mas sim de classes distintas (versões), implicando a existência de dois componentes diversificados A e B, sendo A o componente primário que será substituído pelo componente B, denominado de componente alternativo. Ambos fornecem serviços similares, porém são instâncias de classes distintas.

1.5 Abordagem Proposta

A abordagem aqui proposta é utilizar técnicas típicas do domínio de tolerância a falhas para efetivar a troca dinâmica, em tempo de execução, de componentes em programas orientados a objetos, utilizando uma arquitetura reflexiva [LIS97].

O domínio de tolerância a falhas pode contribuir para autorizar a troca dinâmica de componentes com técnicas concebidas para detecção de falhas em tempo de execução, preservação da confiabilidade de programas por redundância de componentes e reconfiguração de programas distribuídos em tempo de execução.

Baseadas fortemente em redundância de componentes, as técnicas e mecanismos típicos do domínio de tolerância a falhas em software agregam significativo conhecimento sobre gerenciamento de componentes redundantes e preservação do estado da computação, essenciais para a efetivação da troca dinâmica de componentes, mantendo a confiabilidade e a disponibilidade do sistema como um todo.

A reflexão computacional será utilizada como técnica de programação - a programação em meta-nível - com a finalidade de separar o programa original e a

nova versão do programa do programa responsável pelo gerenciamento da troca de versões.

1.6 Organização da Dissertação

Em razão da grande extensão e abrangência do assunto abordado neste trabalho, foi feito um estudo genérico do problema em seu aspecto mais amplo para chegar a um estudo mais detalhado do aspecto específico. Entre os tópicos abordados de forma genérica encontram-se os sistemas de tempo-real, sistemas distribuídos e a programação com objetos distribuídos. Este trabalho vai se dedicar com maior ênfase aos aspectos internos da substituição dinâmica da versão de um componente de software, ficando em segundo plano os detalhes externos à substituição como a comunicação entre os objetos distribuídos e a sincronização em tempo-real.

No capítulo 2 é feito um resumo sobre os principais conceitos relacionados com linguagens de programação; no capítulo 3 é feito uma descrição sobre a técnica de reflexão computacional; no capítulo 4 são apresentados os trabalhos correlatos nesta área; o capítulo 5 faz as considerações sobre a troca de versões de componentes em seus aspectos formais e são apresentados problemas e soluções para efetivar a substituição; no capítulo 6 é apresentado um estudo de caso; e no capítulo 7 são apresentadas as conclusões.

2 Projeto de Programas

Dividir um sistema grande e complexo em unidades menores é uma prática essencial em engenharia de software. No modelo de programação orientado a procedimentos, o código era dividido em funções, depois as funções foram agrupadas em módulos, e, no modelo de objetos, em objetos e classes [GOL97].

Neste capítulo serão introduzidos os conceitos e idéias ligados ao projeto de programas nos modelos imperativo e orientado a objetos e a programação concorrente em ambos os modelos.

2.1 Modelo Imperativo

As linguagens imperativas baseiam-se na arquitetura de Von Neumann, onde os conceitos fundamentais são a execução em seqüência das instruções armazenadas na memória. A máquina de Von Neumann foi o modelo de todos os computadores digitais por muitos anos. A máquina básica de Von Neumann apresenta dois componentes: uma memória e um único processador.

A estrutura e as operações do computador têm forte influência na programação em linguagens imperativas. Existe uma proximidade deste modelo computacional em relação ao modelo de hardware. A essência nessa programação é o endereçamento simbólico (dar nome aos endereços de memória); alteração de valores na memória (atribuir valores a posições de memória); e o seqüenciamento dos comandos (controle das ações elementares).

Os programas, assim como os dados estão armazenados na memória e dela são recuperados para execução pelo processador. O processador pode realizar dois tipos de funções: acessar qualquer posição de memória para recuperar ou modificar seu conteúdo, e executar instruções através de operações de lógica simples. Essas operações são efetuadas uma de cada vez em uma seqüência definida pelo programa.

A maioria das linguagens de programação existentes atualmente pertencem à classe das linguagens imperativas. Neste tipo de linguagem a descrição do algoritmo é a preocupação principal, visto que a partir de um estado inicial, o algoritmo descreve uma seqüência de ações sobre os dados que são modificados através de comandos precisamente escolhidos.

As linguagens de programação Von Neumann utilizam variáveis para endereçar as células de armazenamento, estruturas de controle realizam os desvios e comandos de atribuição efetuam a busca 'fetching' e armazenamento 'storing'. Os valores podem ser atribuídos às variáveis, modificando assim seu conteúdo. Um dos comandos imperativos mais poderosos é a atribuição [LIS94], pois seu efeito é alterar o valor armazenado em uma variável. O valor é um dos atributos principais de uma variável e a execução de um programa consiste em alterar os conteúdos das suas posições de memória.

Os valores das variáveis em um dado instante descrevem o estado da computação, naquele momento. Para conhecermos o estado da computação necessitamos informações sobre o conteúdo das posições de memória após a execução

de cada um dos comandos de um programa. Neste modelo de operação, que muda o estado, calcula e altera valores de variáveis, os algoritmos descrevem a seqüência de ações responsáveis pelas mudanças de estado.

2.2 O Modelo Orientado a Procedimentos

A construção do programa é monolítica, mas o programa é dividido em unidades que possam ser entendidas independentemente, ter um tamanho manuseável e conter as ações que vão ser repetidas por muitas vezes.

Um programa grande, mesmo dividido em partes ou composto de muitos módulos, pode dificultar a sua visualização completa e ultrapassar os limites da percepção humana.

Neste ponto cabe fazer uma distinção entre as formas de abstração. Cada um destes módulos representa uma abstração, com a finalidade de esconder detalhes de implementação. As abstrações podem ser organizadas na forma de:

a) Funções: abstrações de valores, visto que da sua execução sempre resulta um valor.

b) Procedimentos: abstração de comandos, visto que representam ações que executam em conjunto.

c) Unidades 'packages' ou 'units': unidades de compilação independente, usadas para agrupar um subsistema ou módulo de uma aplicação.

Cada função ou procedimento tem objetivos bem definidos. O controle sempre retorna à função que a chamou. Existe um hierarquia, onde as funções chamadas são hierarquicamente inferiores às funções chamadoras.

No modelo orientado a procedimentos os programas baseiam-se fortemente nos dados globais, que são aqueles dados voláteis, existentes na memória principal durante a execução de um programa, e que são acessíveis por qualquer parte do programa, inclusive pelas funções. Estes dados são criados no início da execução do programa, passam a existir durante o tempo de existência do programa e após são destruídos. Incluem-se nestes tipos de dados aqueles criados explicitamente pelo programador.

Cada um destes dados, depois de criado, pode ser alterado por qualquer rotina do programa, que deve ser responsável por garantir sua integridade. Este modelo falha em garantir a consistência destes dados. Uma das metas a perseguir num programa procedural é preservar a integridade dos dados globais. Os mesmos dados são utilizados em várias rotinas; alterando uma rotina poderemos alterar a integridade de outra.

As funções têm visibilidade irrestrita no programa procedural, podendo ser invocadas de qualquer ponto do programa, dificultando a pesquisa e a referência por parte do programador, bem como a percepção sobre a inter-relação entre as mesmas e não se podendo avaliar o quanto uma alteração irá afetar as demais rotinas. Os ambientes de programação oferecem ferramentas para auxiliar o programador nesta visualização através de geradores de referências cruzadas 'cross reference'.

O reúso de código a nível de função também fica dificultado devido às referências que as funções fazem aos dados globais.

2.2.1 Programação no Modelo Orientado a Procedimentos

O programador precisa entender e visualizar todo o programa antes de começar a programá-lo. Foram desenvolvidas muitas técnicas de programação estruturada para serem usadas pelos programadores no modelo de programação procedural. Estas técnicas referem-se à organização interna de um programa, visando a facilitar a programação e manutenção.

Algumas destas técnicas consistem basicamente em certas orientações aos programadores como: construir funções pequenas visando facilitar sua compreensão, não utilizar desvios incondicionais para frente e para trás, usar padronização dos nomes de variáveis globais, adoção de regras para a distribuição de funções aninhadas dentro de um programa.

Um programa no modelo procedural tem sua estrutura básica mostrada na figura 2.1.

FIGURA 2.1 - Programa no Modelo Procedural

2.3 Modelo Orientado a Objetos

Atualmente a busca por melhores linguagens de programação parece estar concentrada na Programação Orientada a Objetos, onde técnicas e ferramentas incluem linguagens, sistemas, interfaces, ambientes de desenvolvimento, bases de dados, bibliotecas de classes, etc. Linguagens de programação O-O possuem diversas características que facilitam sua utilização frente a outras linguagens imperativas. Nesta seção, é descrito o paradigma da orientação a objetos, suas características e principais conceitos.

O modelo de objetos utiliza conceitos já conhecidos nas áreas de linguagens de programação como ocultamento de informação, tipo abstrato de dados, etc. Estendendo estes conceitos criou-se um novo estilo de programação. Um programa O-O consiste em um conjunto de classes que definem objetos de estrutura e comportamento idênticos. Esta característica é semelhante ao tipo de dados de uma linguagem convencional, como o tipo integer do Pascal que define variáveis com as

propriedades do tipo inteiro; a classe é um tipo que define objetos com as propriedades com que foi construída.

Uma das principais características do modelo de objetos é o suporte para abstração de dados, a habilidade de definir tipos novos de objetos cujo comportamento é descrito de forma abstrata, sem se fazer referência a detalhes de implementação, como as estruturas de dados e algoritmos usados para representar os objetos. Com a abstração de dados os objetos podem ser manipulados somente através de requisições de uma operação pública do objeto (troca de mensagens) sugerindo que seja precisamente definida a interface entre os objetos.

O encapsulamento apresenta muitas vantagens, como a simplicidade para a compreensão dos programas, maior facilidade para a modificação dos mesmos e maior garantia de integridade dos dados. O ideal é que nenhuma parte de um sistema complexo dependa dos detalhes internos de qualquer outra parte. Deve-se procurar minimizar a exposição de detalhes internos de implementação nas interfaces. A interface deve ser a mínima possível. É possível construir aplicações com bons níveis de segurança e confiabilidade através do encapsulamento de dados e funções dentro de objetos, que também representam a unidade básica de execução um uma linguagem O-O.

2.3.1 Componentes de Programas

Os componentes de programas no modelo de objetos são classes e objetos.

Classes

Programas escritos segundo o paradigma de O-O são montados a partir de classes. Uma classe define um conjunto de objetos que compartilham uma estrutura de dados e um comportamento comum [BOO94]. Do ponto de vista de linguagens de programação fortemente tipadas, uma classe corresponde a um tipo definido pelo usuário.

Classes de objetos podem ser subdivididas em subclasses, com o intuito de especializar o comportamento de seus objetos, constituindo desta forma uma hierarquia de objetos. O mecanismo que permite criar novas classes a partir de classes já existentes, pela especificação das diferenças entre a nova classe e a classe original, chama-se herança. O mecanismo de herança permite a construção de sistemas na forma incremental e evolutiva, possibilitando a reutilização de códigos já escritos. Pequenas modificações podem ser facilmente testadas e efetuadas partindo da criação de novas classes que herdam os comportamentos de classes já consagradas. Uma classe herda a estrutura de dados e as operações de uma classe de nível hierárquico mais elevado (a superclasse) e, da mesma forma, pode ter sua estrutura de dados e operações herdadas ou transmitidas para outra classe (a subclasse). O mecanismo de herança introduz duas vantagens:

- permite que uma alteração em algum dos atributos da classe seja assimilado por todas as suas subclasses derivadas.

- encoraja o reúso de código existente.

Um modelo genérico de classe contém os seguintes itens:

- nome da classe
- lista de superclasses que transferem propriedades a esta classe
- declaração das variáveis que irão compor o estado interno de cada objeto desta classe
- declaração das operações que irão compor os métodos que definirão o comportamento dos objetos desta classe.

O estilo de programação segundo este paradigma introduz características bastante interessantes, a mais imediata delas é a modularidade encontrada no conceito de objeto que encapsula dados e funções.

Objetos

Objetos são as entidades básicas de um sistema O-O, e sob o ponto de vista de projeto, objetos modelam as entidades no domínio da aplicação. Os objetos são instâncias de classes que são extensões da notação de tipos abstratos de dados [LIS94]. O conjunto de objetos que contém a mesma estrutura de dados e operações pertencem à mesma classe. Cada objeto compõe-se de uma estrutura de dados, a qual ocupa espaço em memória, e um conjunto de métodos associados que implementam as operações realizadas por aquele objeto e que manipulam os seus dados. A figura 2.2 representa o modelo genérico de um objeto.

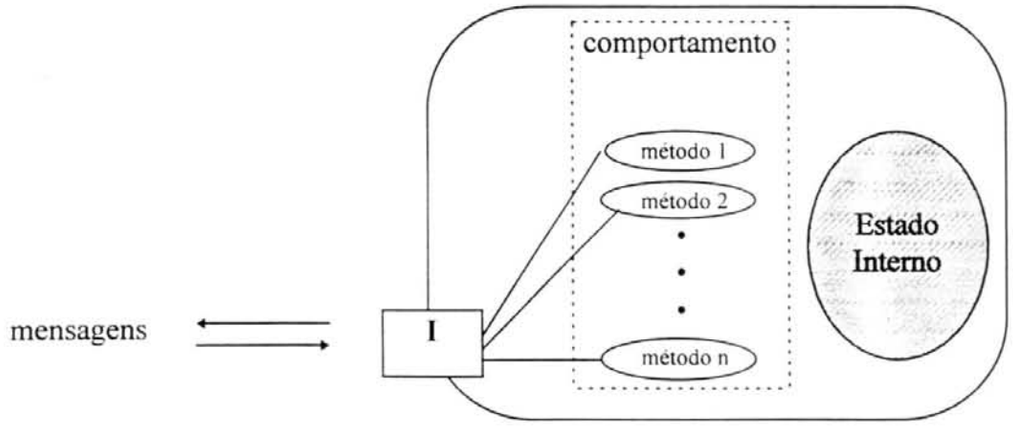

FIGURA 2.2 - Modelo genérico de um objeto

Estado interno: compreende a memória interna ao objeto que mantém as informações durante a vida útil do objeto. As variáveis mantidas nesta área de memória somente são acessíveis através dos métodos do próprio objeto [CAV94].

Comportamento do objeto: corresponde a um conjunto de métodos que são individualmente disparados pelo recebimento de uma mensagem. As ações que os métodos executam são programadas na classe que instanciou o objeto.

Método: é uma função ou procedimento contido no objeto e que executa as suas ações. O método está associado a uma interface do objeto e é quem recebe e envia as mensagens.

Interface: é a parte do objeto que está visível ao mundo externo. Através da interface circulam as mensagens recebidas e enviadas pelo objeto. A interface é identificada na figura 2.2 pela letra "I".

Mensagem: constitui-se do objeto destinatário, do seletor do método e, opcionalmente, de um conjunto de argumentos. É o principal modo de comunicação entre objetos.

O ocultamento de informação é importante por assegurar a integridade dos dados. O estado de um objeto está contido em variáveis privadas, visíveis apenas dentro do escopo do objeto, e podendo ser modificadas e manipuladas apenas por um conjunto de procedimentos restrito ao escopo do objeto. Como as variáveis internas de um objeto não são acessadas externamente, uma interface cuidadosamente projetada permitirá a modificação do estado interno desse objeto.

Segundo Booch [BOO94], um cliente é um objeto que utiliza os recursos de outro objeto, conhecido como servidor. Caracterizamos a função de um objeto considerando os serviços que ele oferece. Esta visão força a nos concentrarmos na visão externa de um objeto e nos leva a um modelo de contrato de programação. A visão externa de cada objeto define um contrato sobre o qual outros objetos podem depender.

Polimorfismo é uma técnica em que o método implementa a ação de uma forma particular dependendo da mensagem que recebe, podendo gerar respostas diferentes para mensagens diferentes. O polimorfismo exige a amarração de alguns nomes em tempo de execução, assim o código executável de um objeto possui uma estrutura dinâmica, dificultando a verificação e validação de programas [LIS94].

Uma mensagem enviada a um objeto provoca um determinado comportamento no objeto receptor, assim, um objeto ativa um processo em outros objetos. Daí surge o modelo de objeto ativo e objeto passivo [MAN93].

Objetos são criados e destruídos de forma dinâmica durante a execução do programa. Cada instância de um objeto dá origem a uma nova entidade com estruturas de dados próprias e comportamentos bem definidos. Não existe a noção de estado global em um programa O-O, mas é mais natural falar no estado independente de cada um dos objetos.

2.3.2 Programação no Modelo de Orientação a Objetos

Ao fazer a programação neste modelo o desenvolvedor não precisa entender ou visualizar todo o programa. É necessário de fato identificar componentes que são as classes e suas instâncias. Estes componentes apresentam uma interação típica cliente/servidor. O programador programa uma classe visualizando estritamente a sua funcionalidade específica e a sua interface pública com o mundo externo. Os detalhes da sua implementação ficam escondidos. Em um nível mais elementar, o escopo que o desenvolvedor precisa enxergar é interno a uma classe.

No nível mais abstrato o desenvolvedor visualiza a interação entre as instâncias das classes. Ele manipula os objetos focalizando apenas os serviços por eles prestado e fazendo invocações a estes serviços através da troca de mensagens por intermédio unicamente de uma interface pública.

2.4 Comparação entre o Modelo Orientado a Procedimentos e o Modelo de Objetos

Migrar de uma linguagem orientada a procedimentos para uma orientada a objetos, requer algo mais do que a compreensão de uma nova sintaxe. Quando é projetado um programa orientado a procedimentos, programadores tendem a pensar sobre o que o programa faz; para um programa O-O, o programador precisa pensar sobre o que e como o programa está manipulando, ou seja, esse modelo ressalta a importância dos dados. Muitas vezes esta mudança de perspectiva causa problemas aos programadores acostumados a uma abordagem de programação orientada a procedimentos[LAD96].

Uma mudança na maneira de pensar pode ser desconcertante e assim muitas pessoas tentam açucarar a programação O-O utilizando-a em termos da programação orientada a procedimentos que lhes é mais familiar. Por exemplo, "C++ é C com extensões O-O", é uma afirmação comum que é correta, mas enganosa. É confortante acreditar que migrar de C para C++ é apenas uma questão de dominar uma nova sintaxe. Ainda mais errônea é a difundida crença de que o uso de técnicas de O-O resultarão em programas que são mais rápidos, menores e mais fáceis de manter [LAD96].

A diferença entre programação orientada a procedimentos e orientada a objetos é que programadores precisam mudar de projetar programas baseados em ações para projetar programas em torno de tipos de dados e suas interações. Compreender esta diferença é uma necessidade quando se deseja extrair o máximo do modelo de objetos.

2.5 Modelos Concorrentes

Visando obter um melhor desempenho na execução de operações, foram buscadas alternativas para o modelo de computador de Von Neumann. Em ambientes monoprocessados foram introduzidos conceitos de multiprogramação. E em ambientes multiprocessados foi introduzida a programação paralela. Com a evolução das idéias envolvidas no paradigma da O-O, foram agregadas extensões a esse paradigma, como distribuição e concorrência, visando adequá-lo às aplicações específicas.

2.5.1 Paralelo

Subentende-se por computação paralela, a execução de mais de uma tarefa por um ou mais processadores, no mesmo intervalo de tempo. Dois processos são ditos paralelos quando são executados de forma concorrente e independente, exceto em pontos em que eles se comunicam ou sincronizam [FOS95].

Podem ser definidos dois tipos de sistemas de computação que permitem executar tarefas paralelas:

• sistemas logicamente paralelos: as várias tarefas são executadas em um único processador. Tem-se a ilusão de que todas as tarefas estão sendo executadas nos mesmo instante de tempo, mas é feita a divisão do tempo de processamento do processador entre as várias tarefas, sendo cada uma executada em uma fatia de tempo. O principal objetivo destes sistemas é aproveitar o tempo de ociosidade do processador durante as operações de I/O de uma tarefa, para executar uma outra tarefa, e aumentar assim o desempenho do sistema. Sistemas com essa característica são chamados de multiprogramados ou multitarefa.

• sistemas fisicamente paralelos: são chamados de sistemas multiprocessadores, e ocorre execução de várias tarefas em vários processadores, no mesmo instante de tempo. O desempenho do sistema pode ser aumentado adicionando hardware para processar um maior número de tarefas simultaneamente. Os múltiplos processadores compartilham a memória primária, o que não ocorre nos sistemas distribuídos.

Neste contexto surgiu a programação paralela, com o objetivo de possibilitar o desenvolvimento de aplicações que possam tirar o máximo proveito de tais sistemas.

Multicomputadores se assemelham a multiprocessadores pois ambos executam diferentes tarefas sobre conjuntos diferentes de dados, porém multiprocessadores compartilham uma área de memória comum, enquanto multicomputadores possuem memória distribuída entre os diversos processadores interligados por um sistema de comunicação e não compartilham um relógio comum.

No que se refere à cooperação entre as tarefas visando resolver um problema comum, estes detalhes são bastante significativos. Nos sistemas multiprocessadores a memória serve como elemento de comunicação e sincronismo entre os processadores. Nos multicomputadores estas operações são realizadas entre os processadores independentes, por meio de troca de mensagens através da rede de comunicação.

Estendendo este conceito, a utilização de sistemas distribuídos também consiste em uma forma de exploração de concorrência, uma vez que os sistemas distribuídos baseiam-se na execução cooperativa e simultânea de processos.

As linguagens de programação paralelas devem atender algumas exigências básicas, tais como controle de processos, controle de acesso a dados compartilhados e mecanismos de sincronização. Para isso, devem oferecer mecanismos sintáticos que implementam unidades de paralelismo, permitindo a execução de diferentes unidades do programa em processadores distintos.

Os processos paralelos são interativos e suas ações podem manter uma relação de competição ou cooperação. Existe competição quando um certo recurso é compartilhado por dois processos, e ambos disputam por ele através de exclusão mútua. Existe cooperação quando processos têm uma relação de dependência entre si, e cooperam para atingir um objetivo comum [FRA96b].

Os recursos compartilhados devem ser protegidos contra acessos indevidos, através de mecanismos de controle para proteção de código e para proteção de dados. A proteção de código compreende tornar exclusivo um trecho crítico de código. A proteção e integridade dos dados é particularmente importante nos sistemas

concorrentes, onde deve ser evitado o acesso simultâneo aos dados para atualização. Por exemplo, não permitir que um vetor seja consultado durante um processo de atualização. Nestes casos as operações de atualização devem ser serializadas.

No âmbito deste trabalho as noções de programação paralela são importantes porque nos casos de sistemas de tempo-real com pequeno tempo de resposta, ambas as versões (a antiga e a nova) podem executar paralelamente para ser conhecido o resultado de ambas num tempo menor.

2.5.2 Distribuído

Sistemas distribuídos consistem basicamente de módulos de uma aplicação executando em elementos processadores independentes. Estes módulos compartilham informações através de mensagens para a solução de um único problema. Nos dispositivos de armazenamento primário, o tempo necessário para acessar os dados em um local é o mesmo tempo necessário para acessá-los em outro local do dispositivo

Para ambientar o problema de processamento distribuído, neste trabalho é utilizada a mesma definição de sistema distribuído adotada por Jalote [JAL94]:

"Um sistema computacional distribuído consiste de nodos autônomos conectados por uma rede de comunicação. Cada nodo consiste de um processador, uma memória privada que é inacessível aos outros processadores e um relógio privado. Os nodos são fracamente acoplados, não têm memória compartilhada e se comunicam através da troca de mensagens".

Segundo esta definição, cada processador em um sistema distribuído executa seu próprio fluxo de instruções sobre seus dados locais, devendo tanto os dados como o programa estar armazenados em sua memória independente. As trocas de informação entre os processadores são realizadas através de mensagens enviadas pela rede de comunicação que os interliga.

Redes de grandes dimensões, que se estendem por regiões dispersas geograficamente e possuem taxas de transferência relativamente baixas, não são muito bem controladas do ponto de vista tempo-real por apresentar atrasos e baixa confiabilidade.

Sistemas distribuídos podem também ser configurados com processadores dedicados a tarefas específicas, portando-se como prestadores de serviços. A interação nestes sistemas se dá pela requisição dos serviços prestados pelos diferentes processadores.

Redes que controlam sistemas críticos são normalmente de pequena escala, vão de confiáveis a muito confiáveis, estendem-se por no máximo poucas centenas de metros e apresentam taxas de transferências elevadas.

A utilização de sistemas distribuídos também consiste em uma forma de exploração de concorrência, pois os sistemas distribuídos também baseiam-se na execução cooperativa e simultânea de processos. De qualquer forma, a programação

com linguagens distribuídas difere da programação em linguagens seqüenciais por utilizar múltiplos processadores cooperando entre si.

Atualmente tenta-se construir sistemas distribuídos que combinam a facilidade de acesso, coerência e vantagens de manutenção dos sistemas centralizados, com as vantagens de compartilhamento, custo e autonomia dos sistemas em rede. Aliado a isso são buscadas segurança e alta disponibilidade [MUL90].

A melhora no desempenho de aplicações é obtida diminuindo o seu tempo de execução, esta é uma das razões mais importantes para o uso de sistemas distribuídos. Efetua-se o particionamento de uma tarefa em subtarefas menores que são alocadas em diferentes processadores. Estes, por sua vez, executam as subtarefas e enviam o resultado de volta para o cliente.

Outras aplicações utilizam replicação de dados e/ou de processamento visando garantir a integridade dos dados manipulados e das operações realizadas. Aplicações típicas de tolerância a falhas são encontradas em sistemas de bancos de dados distribuídos. Manter a consistência dos dados em um meio concorrente e não confiável é um dos principais desafios para muitas aplicações.

Sistemas distribuídos também podem ser configurados como processadores dedicados a tarefas específicas, portando-se como prestadores de serviços. A interação nestes sistemas se dá pela requisição dos serviços a serem prestados pelos diferentes integrantes do sistema. O sistema operacional Amoeba [MUL90] possui esta característica, utilizando diversos servidores, como servidor de arquivo e servidor de impressão instalados em nodos de processamento específicos.

Podem ser adotados dois tipos de comportamento na comunicação entre processos:

- síncrono: o processo envia sua mensagem através da rede e fica suspenso, aguardando a chegada da resposta.

- assíncrono: enquanto é aguardada a resposta à uma solicitação, o processo prossegue com o seu processamento, uma operação assim é dita não-bloqueante.

Um mecanismo síncrono de interação entre processos é a chamada remota de procedimentos (RPC - Remote Procedure Call). Este mecanismo é uma abstração das chamadas locais de procedures, permite que seja invocado um procedimento remoto fazendo a transmissão de parâmetros, sem empregar uma memória compartilhada.

Sistemas distribuídos são importantes neste trabalho porque muitas aplicações de tempo-real são naturalmente distribuídas e porque a programação no modelo de objetos é facilmente adaptada aos sistemas distribuídos.

2.6 Tempo-Real

A utilização de computadores no controle e monitoramento de processos críticos, nas mais variadas áreas de aplicação, vem se tornando cada vez mais uma realidade. Tais sistemas computacionais, também chamados sistemas de tempo-real, também requerem novas técnicas para apoiar a substituição de versões de seus módulos integrantes, que devem ser empregadas para prover confiabilidade e

disponibilidade à aplicação principalmente pelo caráter crítico assumido por certas aplicações. As técnicas de TF já consagradas, podem ser utilizadas para assegurar uma correta substituição dinâmica em sistemas de tempo-real, mas encontram os limites temporais impostos por estas aplicações. Muitas aplicações de tempo-real são naturalmente distribuídas facilitando sua utilização em sistemas distribuídos.

O interesse em programação em tempo-real tem crescido consideravelmente nos últimos anos. Um dos motivos é a disponibilidade de hardware relativamente barato e robusto que geram uma tendência de mudar as funções de controle de tempo-real de dispositivos mecânicos para dispositivos eletrônicos. Além disso, os dispositivos eletrônicos estão auxiliando pessoas em um crescente número de áreas que demandam responsabilidade em tempo-real pela necessidade de substituição de controle humano em situações complexas e/ou críticas, onde máxima confiabilidade e segurança são exigidas. Outro aspecto que precisa ser mencionado é a confiabilidade crescente dos sistemas de computação.

Mas hardware é apenas uma parte do contexto, a outra parte é o software. Antigamente as aplicações em tempo-real eram freqüentemente executadas por programas difíceis de serem alterados, expandidos ou mantidos. As técnicas necessárias para manusear estes problemas ainda não estão bem resolvidas. A complexidade de tolerância a falhas aumenta quando precisa ser considerado também o aspecto temporal. O ambiente com o qual estes sistemas interagem é caracterizado por sensores, processos físicos, atuadores, dentre outros, com a ressalva de que precisam gerar dados corretos em limites bastante restritos de tempo. Aplicações de tempo-real geralmente são formadas por processos que executam continuamente e que fornecem uma resposta rápida em relação a algum estímulo.

Com isso os sistemas de tempo-real tornaram-se grandes incentivadores do uso de tolerância a falhas. Além da confiabilidade normalmente exigida nestas aplicações, que envolvem grandes quantias de dinheiro em equipamentos, bem como vidas humanas, a disponibilidade torna-se outro fator crítico a ser levado em consideração. Justificam-se os elevados custos envolvidos em garantir um sistema confiável e altamente disponível.

O objetivo de controlar foi o grande impulsionador no surgimento dos STR, cuja finalidade é exercer controle em ambientes que possuem uma alta dinâmica composta de tarefas exaustivas e críticas, notadamente o ponto mais fraco do trabalho humano. Os avanços nesta área foram incentivados com o objetivo de projetar sistemas que apresentem um comportamento confiável no controle de tarefas críticas.

Antes de prosseguir, vamos definir o termo sistema de tempo-real mais precisamente. Utiliza-se o termo sistema de tempo-real (STR) na descrição de sistemas controlados por software e hardware que devem executar todas suas funções de processamento dentro de certos limites de tempo pré-definidos. Existem muitas interpretações sobre a natureza exata de um sistema de tempo-real, mas todas têm em comum a noção de tempo de resposta - o tempo necessário para o sistema gerar uma saída a partir de uma entrada associada. Podemos encontrar algumas definições em [BUR96]:

"Sistema de tempo-real é aquele em que o tempo em que a saída é produzida é significativo". "Sistema de tempo-real é qualquer atividade de processamento de informação ou sistema que tem que responder à estímulos gerados externamente dentro de um período de tempo finito e especificado". "Sistema de tempo-real é um sistema que deve reagir a estímulos do meio ambiente (inclusive à passagem do tempo físico) dentro de intervalos de tempo ditados por este ambiente".

"Sistemas de tempo-real caracterizam-se pela necessidade fundamental de manter um sincronismo entre o ambiente e o processo". "A correção de um sistema de tempo-real não depende apenas dos resultados lógicos da computação, mas também do tempo em que estes resultados são produzidos".

Outra restrição se aplica se o STR for uma aplicação crítica, tal como o controle de uma usina termoelétrica: aplicação exige alta confiabilidade, isto é, além de imediata a resposta deve estar correta, pois qualquer erro pode trazer resultados catastróficos. Os STR diferem dos sistemas tradicionais pelo fato de terem restrições de tempo e tratarem em muitos casos com situações críticas. Ainda segundo [BUR96], os sistemas de tempo-real podem ser classificados, de acordo com os requisitos de tempo, em:

- tempo-real rígido ou crítico: aqueles em que é absolutamente imperativo que a resposta ocorra dentro do limite de tempo especificado. O fato de não cumprirem com precisão as restrições temporais pode levar a conseqüências imprevisíveis no ambiente em que estes sistemas estão embutidos. Nesta classe encontram-se aplicações como sistemas de controle de bordo em aviação, controle industrial, controle de tráfego em ferrovias, controle médico.

- tempo-real brando: aqueles em que o tempo de resposta é importante, mas o sistema ainda vai funcionar corretamente se o limite de tempo é ocasionalmente quebrado. A aplicação pode tolerar pequenos atrasos com relação ao instante em que devem ser produzidos os resultados. É possível uma ressincronização em caso de falha temporal. Exemplos desta classe são sistemas de aquisição de dados, transações bancárias, reservas de passagens, teleconferências.

É importante comentar que existe uma variação significativa no tempo de resposta entre diferentes STR. Em aplicações como o controle de bordo de aeronaves o tempo é da ordem de frações de segundo, enquanto nos sistemas de transações bancárias o tempo pode ser de minutos ou até mesmo horas.

O projeto de um programa de tempo-real, compreende um conjunto de processos concorrentes que operam assincronamente para atender às restrições individuais de tempo dos vários componentes do sistema. Estes processos usualmente intercomunicam-se e são incluídos pontos de sincronização no projeto para possibilitar a troca de mensagens e a proteção de dados compartilhados.

Nos sistemas de tempo-real, o computador trabalha em paralelo como o dispositivo que está controlando, monitorando-o e acionando-o. Assim, uma das necessidades dos STR é manterem-se constantemente sincronizados com o processo controlado.

Devido ao tipo de utilização dos STR, são características desejáveis destes sistemas:

- fornecer o resultado dentro do tempo previsto

- ser confiável

- ter alta disponibilidade, durante o tempo de operação. A disponibilidade pode ficar comprometida se for necessário realizar uma manutenção no software de tempo-real.

Um STR é constituído por diversos módulos ou procedimentos (componentes de software), cada um encarregado de executar uma atividade (tarefa) específica, como ler um dado de um sensor ou ativar um acionador. Cada um destes módulos pode ser visto como um servidor que executa as solicitações de serviço do seu cliente, o programa de tempo-real.

2.6.1 Características de Objetos de Tempo Real

Aqui será apresentado um enfoque de STR programados com objetos. Os objetos de tempo-real devem apresentar algumas características básicas que serão comentadas aqui.

O modelo de objetos [BOO94] é adequado a aplicações para controle em tempo-real, pois estes sistemas, via de regra, apresentam a característica de serem distribuídos, suas partes componentes trabalharem numa estrutura cliente-servidor e comunicarem-se através de mensagens, à semelhança do modelo de objetos. Apesar dos benefícios da abordagem de objetos, a aplicação testa técnica à programação em tempo-real representa uma área ainda pouco explorada.

A idéia apresentada neste trabalho é embutir os serviços básicos de tempo-real dentro de objetos e desenvolver um programa coordenador que gerencia estes objetos visando realizar precisamente as atividades da aplicação de tempo-real. Através do paradigma da O-O torna-se possível também a reutilização de software de tempo-real. Software para tempo-real tem sido tradicionalmente reescrito para cada nova aplicação. Isto porque praticamente inexistem ferramentas que tornem possível a reutilização deste tipo de software.

Um objeto é a unidade básica de execução para uma aplicação em tempo-real orientada a objetos. Um sistema de tempo-real recebe as chamadas do mundo exterior através de interrupções que ativam eventos dentro do programa de tempo-real. No STR O-O quem ativa os eventos no programa coordenador são as mensagens enviadas pelo objeto servidor. Assim o programa coordenador é um programa orientado a eventos (do tipo estímulo-reação). Atualmente a tendência das linguagens de programação é a orientação a eventos, exemplos são Java, Delphi, ADA, Visual Basic.

Cada objeto servidor executa uma pequena parte da aplicação total, trabalhando em paralelo com outros objetos (tarefas). Em vários pontos os objetos devem coordenar suas atividades.

"Coordenação é o bloqueio de uma tarefa até que alguma condição especificada seja encontrada" [RIP93]. De acordo com esta definição um objeto pode

se coordenar com eventos físicos, como também consigo mesmo e com outros objetos. Por exemplo, quando um objeto sofre uma pausa, ele é bloqueado durante um determinado tempo especificado. O objeto está portanto coordenado com o relógio físico. Um objeto também pode ser desbloqueado através da requisição de outro objeto.

Freqüentemente um objeto coordena-se com outro objeto, é a condição na qual o objeto A fica aguardando a resposta de outro objeto B. Momentos antes de uma substituição, o objeto O1 que vai ser substituído, é bloqueado para então ser cancelado. Após ser instalada a nova versão O2 o objeto precisa se coordenar novamente com o programa coordenador e com os outros objetos.

A comunicação entre o programa coordenador e os objetos servidores pode ser feita através de um objeto separado chamado trocador de mensagens. Sua implementação é uma fila pública para a qual qualquer objeto pode enviar uma mensagem e da qual, qualquer objeto pode receber uma mensagem. A vantagem é a coordenação das mensagens. No instante da substituição o objeto servidor O1 vai ser cancelado e não poderá receber mensagens, mas através do trocador de mensagens elas poderão ser disponibilizadas e processadas assim que a nova versão do objeto servidor O2 estiver ativa.

2.6.2 Características de Tolerância a Falhas nos Sistemas de Tempo Real

Geralmente nos STR são exigidas características como disponibilidade, confiabilidade e integridade para que o sistema mantenha-se em operação normal, mesmo na ocorrência de eventuais falhas internas, nestes sistemas é fundamental que haja tolerância a falhas. "Tolerância a falhas é a capacidade que um sistema tem de produzir resultados coerentes mesmo na presença de falhas e de se recuperar destas falhas" [AND81]. Aqui serão apresentadas de um modo geral as técnicas de Tolerância a Falhas mais usuais empregadas nos STR.

A tolerância a falhas é importante nos sistemas críticos, onde falhas podem levar a grandes prejuízos além de colocar em risco vidas humanas. A detecção e localização de erros deve ser rápida, pois geralmente o tempo disponível é pequeno num sistema de tempo-real crítico. Às vezes é impossível voltar atrás e refazer operações visando corrigir os erros detectados, caso ocorra uma falha não mascarável o sistema deve ir para um estado seguro [WEB96].

Não é usual fazer recuperação por retrocesso em caso de falha num sistema de tempo-real, usa-se recuperação por avanço [AND81], cuja idéia é conduzir o sistema a um novo estado, não ocorrido anteriormente. Este enfoque é bastante eficiente, mas é específico de cada sistema e também é difícil ser implementado através de mecanismos genéricos. A técnica de recuperação por avanço mostra-se adequada apenas quando os danos podem ser previstos antecipadamente, de forma razoavelmente acurada.

Ao fazer a programação de objetos de tempo-real tolerantes a falhas, as estruturas de controle para implementar as técnicas de TF podem confundir-se com as funcionalidades da aplicação. Procurando separar estes dois subsistemas é adotada a técnica de reflexão computacional [LIS95c].

Nos STR a violação do tempo passa a ser a preocupação mais importante. Dois mecanismos principais podem ser usados para fazer a avaliação do tempo [LIS94] a partir de um relógio local:

Vigilância 'watchdog timer': monitora o recebimento de uma resposta a intervalos de tempo determinados. Não se aplica a STR orientados a eventos, pois os eventos podem ocorrer a intervalos de tempo irregulares.

Expiração de tempo 'time-out': suspende a execução de um processo após ocorrer a expiração do período limite de tempo. É exigido um mecanismo de sincronização de relógios para exercer este controle de tempo de forma confiável em sistemas distribuídos [VER96].

O processamento tempo-real no modelo cliente-servidor implica em restrições de tempo tanto no cliente quanto no servidor da aplicação. Quando da ativação de uma requisição de serviço, estas restrições são especificadas na forma de um 'time-out' do lado do cliente e quando o serviço é requisitado do lado do servidor, são especificadas na forma de um 'deadline' [FRA96a].

2.7 Objetos Distribuídos

A união dos paradigmas de POO e concorrente provê os recursos desejados pelos projetistas de software, tanto no que tange ao desenvolvimento como no desempenho de execução de aplicações.

É uma tendência o crescimento das aplicações distribuídas em praticamente todas as áreas. Há sistemas de tempo-real naturalmente distribuídos, como, por exemplo, controles de reservas de passagens aéreas, sistemas bancários, automação industrial. Nestes, o uso de sistemas distribuídos é praticamente imprescindível. Os componentes remotos integrantes de uma aplicação distribuída também são passíveis de requerer uma substituição dinâmica. Torna-se importante que esta substituição seja tolerante a falhas, pois os sistemas distribuídos são um campo muito propício a falhas e onde está ocorrendo muita pesquisa na área de tolerância a falhas [JAL94].

Além disso, a literatura consultada na área de substituição dinâmica, apresenta poucas soluções que sejam aplicáveis também a sistemas distribuídos.

A solução neste trabalho foi proposta dentro do modelo de objetos que é um modelo naturalmente portável para os sistemas distribuídos, devido às grandes semelhanças existentes entre ambos os modelos [CAV94]. Objetos possuem tanto área de dados como o código de execução independentes e inacessíveis aos demais. Os acessos aos dados encapsulados internos de um objeto somente são possíveis através de mensagens explícitas trocadas entre objetos. Objetos trabalham numa arquitetura cliente/servidor que pode ser plenamente estruturada num sistema distribuído.

Sistemas distribuídos provêem recursos físicos para implementação de sistemas O-O, uma vez que ambos possuem características similares. A independência a nível de execução de módulos de sistemas distribuídos pode ser provida por objetos encapsulando dados e funções. Os recursos para implementação do protocolo de acesso aos serviços dos módulos distribuídos, é provida pela abstração de dados existente no paradigma da O-O.

A interação entre os objetos se dá através do envio de mensagens e do respectivo retorno de dados.

A programação em linguagens distribuídas difere da programação em linguagens seqüenciais por utilizar múltiplos processadores cooperando entre si. Programar uma aplicação distribuída é uma tarefa bastante complexa, cabendo ao programador, além de codificar a aplicação, controlar a comunicação entre as partes do programa. Há pesquisa em sistemas de apoio ao desenvolvimento de software distribuído que fazem os controles de sincronismo/recuperação que visam a dar suporte à concorrência, encapsulamento de ações em transações e fazer manipulação de dados atômicos. Estas rotinas são implementadas através de reflexão computacional [STR95], tornando mais simples o desenvolvimento de aplicações distribuídas e liberando o programador destes detalhes de comunicação, sincronização e TF, podendo o mesmo deter-se exclusivamente nas funcionalidades da aplicação em si.

A concorrência encontrada em objetos distribuídos pode ser separada em dois níveis:

inter-objetos, os objetos executam de forma concorrente entre si. Objetos operando neste modelo podem tratar apenas uma mensagem em um determinado instante de tempo permitindo a concorrência somente a nível de objetos. O objeto encontra-se em "repouso" quando não estiver processando uma mensagem recebida e encontra-se "aguardando" quando enviou uma solicitação de serviço a outro objeto e está a espera de uma resposta. Este objeto é chamado *passivo*.

intra-objeto, é feita a execução concorrente de diferentes fluxos de execução em um mesmo objeto. Sua implementação é conjunta ao modelo de concorrência inter-objeto. A modelagem do objeto deve conter mecanismos de exclusão mútua, como semáforos ou monitores, para fazer o controle de acesso ao seu estado interno. O objeto que processa algum serviço sem que tenha recebido uma mensagem ou que prossegue a execução após enviar resposta a uma mensagem recebida é chamado objeto *ativo*. O objeto ativo pode executar várias tarefas em um mesmo instante de tempo, independente do recebimento ou não de mensagens.

A concorrência inter-objetos já é garantida naturalmente pelo sistema distribuído, e através da introdução do esquema de concorrência intra-objeto, atinge-se um grau maior de concorrência no sistema.

Disso decorrem dois modelos de comunicação entre objetos:

comunicação síncrona: um objeto A envia uma mensagem a outro objeto B e permanece aguardando pela resposta. O processamento do objeto A só é retomado após ter recebido a resposta do objeto B. Este modelo implica que em um determinado instante, apenas um objeto esteja processando. Um objeto pode ter apenas uma mensagem enviada esperando tratamento.

comunicação assíncrona: este modelo permite que um objeto A envie uma mensagem a outro objeto B e retome o seu processamento imediatamente, sem aguardar pela resposta de B. As mensagens são mantidas em uma fila de espera em B até serem atendidas. Desta forma A e B executam suas tarefas de forma concorrente.

As mensagens enviadas devem ser de alguma forma endereçadas aos destinatários, as formas mais comuns são o endereçamento indireto e o direto. O endereçamento indireto se dá através do envio da mensagem a um elemento intermediário, normalmente chamado de caixa-postal, o qual é acessado eventualmente pelo destinatário. O endereçamento direto provê o envio de mensagens ao processo destinatário sem intermediários.

RPC (chamada remota de procedimentos) possui sintaxe e semântica similar a uma chamada de procedimento local, permitindo envio de dados através de parâmetros a procedimentos remotos e receber o retorno dos resultados. Este mecanismo causa o bloqueio do processo de origem da invocação até o término do procedimento invocado.

A programação com objetos distribuídos é uma evolução natural da programação O-O convencional. Novas linguagens e ferramentas [CAV94, YAN96] que disponham de mecanismos de comunicação e sincronismo estão sendo pesquisadas para permitir que os diferentes processos de uma mesma aplicação possam cooperar entre si.

2.8 Conclusões

Uma aplicação compreende um conjunto de componentes interrelacionados. Estes componentes são abstrações organizadas na forma de funções, procedimentos, módulos ou objetos, de acordo com o modelo ou mecanismos de abstração disponíveis na linguagem de implementação.

As linguagens de programação imperativas sempre foram o paradigma para experimentação de novas técnicas e modelos de programação, inclusive na área de substituição dinâmica.

As linguagens orientadas a procedimentos não oferecem mecanismos para restringir as operações sobre os dados, o que poderia garantir maior segurança e integridade aos dados, reduzindo-se as chances do surgimento de uma operação errônea. O uso de dados globais pode ser visto como um mecanismo para economia de memória, pois as mesmas variáveis são utilizadas em diferentes partes do programa, em rotinas diferentes. O intenso emprego de dados globais é um fator de dificuldade em se executar a substituição dinâmica de uma função [GUP96, SEG93].

O paralelismo tem entre seus objetivos aumentar o desempenho de aplicações. A agilidade do processo de substituição dinâmica pode ser incrementada quando a linguagem de programação adotada oferece facilidades de execução concorrente de processos com mecanismos de sincronização e controle de dados.

Sistemas distribuídos, sendo apoiados em multicomputadores, apresentam uma vantagem adicional em relação aos sistemas centralizados: a distribuição física dos equipamentos, o que lhes permite exercer controle também sobre dispositivos remotos, possibilitando construir aplicações de tempo-real distribuídas.

A orientação a objetos introduz a facilidade do encapsulamento das classes de objetos e a possibilidade de execução distribuída e concorrente de objetos, além de trabalhar numa arquitetura cliente-servidor, o que propicia um ambiente favorável a

uma natural substituição de versões. O mecanismo de herança permite o reúso de código.

Sistemas de tempo-real são, de maneira geral, empregados em aplicações de controle de processos que muitas vezes são críticos e introduzem restrições temporais que devem ser consideradas pelo sistema de substituição dinâmica.

Este novo ambiente aberto de programação é a tendência atual. A solução proposta neste trabalho tem condições de realizar substituições de versão tolerantes a falhas também neste ambiente de programação, sem fazer exigências quanto a um hardware ou a um sistema operacional especializado.

3 Reflexão Computacional

O paradigma reflexivo permite que um sistema execute processamento sobre si mesmo, para o ajuste ou a modificação de seu comportamento. Neste capítulo são abordados os principais conceitos de reflexão computacional e terminologia associada e apresentadas algumas linguagens reflexivas.

3.1 Conceitos Básicos e Terminologia

As linguagens de programação convencionais carecem de um mecanismo para exercer um controle externo às instruções ou modificar o funcionamento de uma instrução. Uma vez que o fluxo de execução tenha iniciado o programador não tem poder para exercer controle para intervir no modo como uma instrução opera. As linguagens tradicionais ocultam do programador as suas implementações, e o impedem de alterar o comportamento dos comandos da linguagem.

Muitas vezes é interessante e vantajoso que o programador exerça um controle adicional sobre o fluxo de execução dos comandos da linguagem ou que tenha informações internas detalhadas da linguagem tais como estrutura de classes e informações sobre métodos. Este controle adicional é o local adequado para introduzir ou tomar ações que agem de forma oculta à aplicação. Tais ações paralelas são úteis para averiguar o correto andamento da execução da aplicação.

A idéia básica contida na reflexão é que o programa pára de executar a aplicação durante um momento e reflita sobre o processo, analisando seu estado, se está processando corretamente, se prosseguindo a execução vai conseguir atingir o seu objetivo satisfatoriamente, se não precisa mudar a sua estratégia ou algoritmo de execução, se o valor armazenado em suas variáveis é o valor esperado, ou de maneira análoga: se o valor armazenado nas variáveis não está de fato incorreto e fora do domínio.

Uma linguagem de programação que possui a habilidade de modificar e/ou estender sua própria implementação usando recursos da própria linguagem é chamada de reflexiva [SIL97]. Estas linguagens provêem mecanismos para manipular entidades internas da linguagem. Através do mecanismo de reflexão computacional o sistema pode manipular as estruturas internas que representam o seu próprio estado.

A reflexão computacional (RC) é introduzida no modelo de objetos através da abordagem de meta-objetos. A cada objeto da aplicação, denominado objeto de nível base, pode associado um meta-objeto, denominado de objeto de meta-nível. Um meta-objeto é um objeto que contém informação sobre outro objeto (o objeto do nível base) e/ou controla a execução deste objeto.

As facilidades de abstração e encapsulamento das linguagens O-O possibilitam construir objetos no nível base e meta-objetos no meta-nível de forma mais independente. Segundo esta abordagem os aspectos funcionais e não-funcionais são separados com o uso de "objetos-base" e "meta-objetos", respectivamente. Os objetos-base implementam a funcionalidade da aplicação e os meta-objetos associados executam os procedimentos de controle que determinam o comportamento dos seus objetos-base correspondentes.

Numa arquitetura reflexiva os objetos estão relacionados com a solução de problemas do domínio da aplicação e os meta-objetos ficam relacionados com a solução de problemas administrativos do próprio sistema e fazem o armazenamento de informações específicas sobre o nível base [SIL97].

A adoção da reflexão torna acessíveis ao programador as políticas que controlam a execução da aplicação. Mudanças de política a nível-meta permitem alterar o comportamento do sistema sem ter que modificar o código da aplicação a nível base.

A reflexão torna-se útil também para a estruturação de atividades administrativas da aplicação, tais como mecanismos de controle temporal para sistemas de tempo-real [FRA96a], estatísticas de desempenho, depuração de erros e procedimentos de TF [LIS95c], rotinas de otimização e políticas de segurança, suporte a replicação em sistemas distribuídos [FRA96b], comportamento inteligente para dispositivos controladores [LIS96b], implementação de tipos de dados atômicos para transações [STR95] etc. A RC possibilita a inclusão destas atividades de maneira a não interferir na aplicação em si.

A interligação entre os dois níveis de execução de um programa – o nível base e o meta-nível – pode ser feita através de interceptação de chamadas de métodos. As chamadas aos métodos do objeto-base são desviadas com intuito de ativar meta-métodos que permitem modificar o comportamento do objeto-base ou adicionar funcionalidades a ele.

3.2 Linguagens com Características Reflexivas

Algumas linguagens de programação, a exemplo de SmallTalk e Java, já incorporam alguns mecanismos que dão suporte à reflexão [ROM97]. Para outras linguagens foram desenvolvidas extensões que propiciam o uso de reflexão como OpenC++ [CHI93], Guaraná [OLI97]. A reflexão se subdivide em duas modalidades:

Reflexão estrutural: é uma modalidade em que são obtidas informações sobre a estrutura estática de um objeto como: nome de classes ascendentes e descendentes, nome e modificador de métodos, tipo de variáveis, tipo de parâmetros passados aos métodos, etc. A reflexão estrutural é passiva e não interfere no fluxo de execução do programa. A reflexão estrutural não oferece um protocolo de meta-objetos intercessional que se interponha à chamadas de métodos, mas apenas provê habilidades introspectivas ao sistema, sem incorporar mecanismos para interceptação de mensagens. Este tipo de reflexão é encontrado no ambiente Smalltalk e no ambiente nativo da linguagem Java.

Aqui são apresentados alguns métodos da classa Reflect da linguagem Java:

```
import java.lang.reflect.*;

public static void print_class(Class c)
{ Constructor[]    constructors = c.getDeclaredConstructors();
  Field[]          fields       = c.getDeclaredFields();
  Method[]         metodos      = c.getDeclaredMethods();
  Class parameters[], exceptions[];
```

O seguinte trecho exibe os construtures da classe e seus parâmetros:

```
  for (int i = 0; i < constructors.length; i++)
    { parameters = constructors[i].getParameterTypes();
      exceptions = constructors[i].getExceptionTypes();
      System.out.println(constructors[i].getName()+"\n" + "quant. de
parametros do construtor="+parameters.length);     }
```

O seguinte trecho exibe as variáveis da classe, seu tipo e nome:

```
System.out.println("*** Quantidade de campos=" + fields.length);
  for(int i = 0; i < fields.length; i++)
    { System.out.println (Modifier.toString(fields[i].getModifiers())
+ " " + fields[i].getType() + " " + fields[i].getName());      }
```

O seguinte trecho exibe os métodos da classe, seu tipo de retorno, nome e argumentos:

```
System.out.println("*** Quantidade de metodos=" + metodos.length);
  for(int i = 0; i < metodos.length; i++)
    { System.out.print (Modifier.toString(metodos[i].getModifiers())
+ " " + metodos[i].getReturnType() + " " + metodos[i].getName());
      parameters = metodos[i].getParameterTypes();
      if (parameters.length > 0)
         { System.out.print("(");
           for (int j = 0; j < parameters.length; j++)
             { if (j > 0) System.out.print(", ");
               System.out.print (parameters[j].getName()); }
           System.out.print(")");
        } } }
```

Reflexão comportamental (ou computacional): esta modalidade oferece recursos de outra natureza e que permitem que haja uma interferência ativa na execução do programa. Este tipo de reflexão permite que sejam interceptadas dinamicamente as chamadas a procedimentos, que se tenha acesso ao conteúdo de variáveis, que se intercepte os parâmetros enviados para os métodos. Podem ser executados novos procedimentos através destas interceptações.

A seguir serão brevemente descritas, com base em [LIS97a] as facilidades reflexivas encontradas nas linguagens OpencC++ e Java. Estas duas linguagens de programação foram inicialmente selecionadas para a implementação deste trabalho: a primeira, tendo em vista a continuidade de trabalhos anteriores desenvolvidos no CPGCC/UFRGS [LIS95c,VEN97]. A segunda, por sua atualidade, seu emprego em trabalhos recentes envolvendo tolerância a falhas [FRA96a] e possibilidade de exploração do modelo de objetos com carga dinâmica de classes.

3.2.1 Meta-objetos de Open C++

A linguagem C++ [STR91] não oferece, por concepção, facilidades reflexivas. Para possibilitar acesso, em tempo de execução, às informações a respeito das classes e objetos do programa é necessário modificar a implementação da linguagem ou fazer extensões baseadas em pré-processadores. O pré-processador OpenC++ - Versão 1.2 [CHI93] adota um modelo de reflexão computacional no qual

chamadas de métodos e acessos a variáveis de instância são tornados disponíveis ao programador, em tempo de execução. O protocolo possui as seguintes características:

a) Meta-objeto: objetos selecionados do programa podem ser instanciados como objetos reflexivos, associados a um meta-objeto; outros objetos da mesma classe podem ser instanciados como objetos não reflexivos

b) Relação: a associação é única entre objetos e meta-objetos, isto é, um meta-objeto não pode ser compartilhado por mais de um objeto;

c) Controle: um meta-objeto pode controlar mensagens dirigidas a seus referentes (chamadas de funções-membro) e acesso à variáveis de instância (variáveis membro);

d) Todas as classes usadas para instanciar os meta-objetos descendem da classe MetaObj;

Em tempo de execução, mensagens dirigidas a um objeto de nível base são desviadas para o seu correspondende meta-objeto, tendo início a computação de meta-nível. A figura 3.1 esquematiza o mecanismo de interceptação de chamada. Quando uma mensagem é dirigida a um método reflexiva, o meta-ojjeto recebe as informações contidas nesta mensagem: o nome do método chamado e os argumentos fornecidos pelo cliente. De posse dessas informações, o meta-objeto pode re-enviar a mensagem ao objeto de nível base, executar pré e pós-processamento.

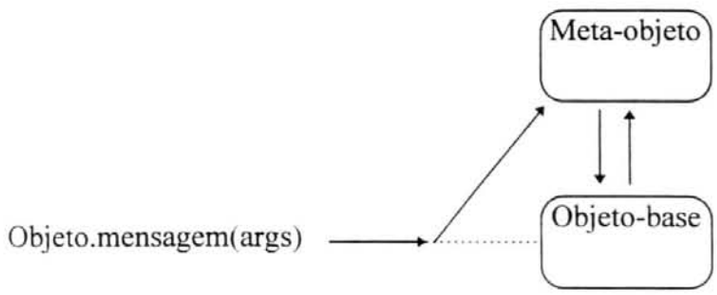

FIGURA 3.1 - Reflexão comportamental em OpenC++

Meta-objetos podem ser criados como classes comuns de C++, tendo como classe ancestral a classe MetaObj, a qual define os métodos que um meta-objeto pode usar para controlar seu referente. Por ser uma classe comum de C++, um meta-objeto pode receber herança múltipla, i. e., ter uma ou mais classes ancestrais além de MetaObj, e pode sofrer especializações, i.e., pode ser usado como classe base de outras classes descendentes, assim transmitindo indiretamente o protocolo herdado da classe MetaObj.

A computação no meta nível é realizada por redefinição dos métodos adquiridos da classe MetaObj e pelos métodos particulares (definidos ou herdados) da classe do meta-objeto em questão.

3.2.2 Máquina Virtual Java

O modelo abstrato da Máquina Virtual Java - MVJ determina que todas as classes e interfaces são objetos do tipo Class. Objetos do tipo Class são automaticamente criados quando novas classes são carregadas para serem executadas, e contém informações sobre o nome da classe, a super-classe, a lista de interfaces implementadas e o carregador de classes utilizado.

As informações de identificação, como os nomes da classe e da super-classe de um objeto são disponibilizados ao programador através de chamadas de métodos que se referem ao próprio objeto (por exemplo, this.getClass()). Portanto, por concepção, a linguagem Java possui características reflexivas, onde informações sobre objetos do programa são fornecidas em tempo de execução pelo seu ambiente. Introspecção é o termo que melhor caracteriza este modelo de reflexão de Java: métodos permitem examinar os processos internos de objetos da aplicação.

Mais recentemente, a partir de 1996, a linguagem Java passou a incluir novos mecanismos de reflexão computacional, através de protocolos de meta-objetos inseridos em seu ambiente de execução.

O protocolo MetaJava: Desenvolvido por Golm [GOL97], introduz uma arquitetura reflexiva para o ambiente Java e que apresenta as seguintes características:

- procura não afetar o desempenho do sistema no nível-base, o nível meta não impõe sobrecarga quando não é usado.

- permite separar o código funcional e não-funcional da aplicação tornando mais fácil compreender, dar manutenção e depurar o programa Java uma vez que o programador focaliza no domínio da aplicação.

- os programas no meta-nível e no nível-base são reusáveis, permitindo substituir a parte não funcional, se o sistema precisa ser adaptado para um novo ambiente de execução.

- a arquitetura é genérica uma vez que permite resolver vários problemas como persistência, distribuição, replicação e sincronização.

O protocolo Guaraná: Em desenvolvimento por Oliva [OLI97], é uma arquitetura reflexiva baseada em Java e que trabalha sem requerer nenhuma modificação na linguagem em si. Um objeto no meta-nível é dividido em um conjunto de refletores 'reflectors' dinamicamente modificáveis. O refletor é um objeto no meta-nível responsável por implementar o comportamento reflexivo. A meta-configuração de um objeto é independente da classe do objeto. Guaraná também provê um mecanismo de autenticação extensível, para prevenir acesso total aos objetos do nível-base por objetos arbitrários no meta-nivel..

Ambos os protocolos MetaJava e Guaraná requerem modificação da Máquina Virtual Java (MVJ). MetaJava é uma modificação do ambiente JDK, enquanto Guaraná é implementado sobre Kaffe (um interpretador Java livre) que provê compatibilidade com JDK 1.1.

Em ambos os protocolos os métodos e variáveis que devem ser manipulados pelo metaobjeto devem ser identificados como parte da operação sobre o objeto. Para fazer a interceptação na invocação de métodos, a identificação dos métodos e os seus parâmetros precisa ser informada ao metaobjeto. Da mesma forma para uma operação de leitura ou atualização sobre variáveis, o metaobjeto precisa receber a identificação destas variáveis.

3.3 Conclusões

A utilização de uma linguagem de programação com característica reflexivas possibilita uma nova estruturação de programas no modelo de objetos, separando dois níveis de abstração: o nível base e o meta-nível. O programa de nível base é composto pelos objetos da aplicação responsáveis pelas funcionalidades correspondentes ao domínio do problema. No programa de meta-nível são implementados os objetos que fornecem os serviços não diretamente relacionados ao domínio do problema, ou seja, os objetos reponsáveis por atividades administrativas da aplicação.

O propósito maior da RC é adicionar comportamento sem interferir na aplicação em si. Dessa forma o programador fica liberado dos aspectos de controle sobre o comportamento da aplicação. Também existe menor chance de incorrer em algum erro, pois fica reduzida a complexidade da aplicação, através da separação de requisitos funcionais e não-funcionais.

Na programação convencional o programa de aplicação é repleto com algoritmos complicados de policiamento. O que torna difícil compreender, manter, depurar e validar o programa. A separação entre do algoritmo reflexivo do algoritmo base permite ao programador de aplicação focalizar-se mais no domínio da aplicação. Através da reflexão as funcionalidades de controle e as funcionalidades da aplicação ficam isoladas uma da outra. Separar o código funcional do não-funcional torna reusáveis ambas as partes do sistema.

Dentre os protocolos de reflexão aqui abordados, destacam-se as seguintes vantagens e desvantagens, tendo em vista a sua utilização na implementação de troca dinâmica de componentes:

OpenC++: este protocolo fornece a facilidade de implementação de mensagens, porém a linguagem C++ dificulta a carga de classes em tempo de execução.

Java: permite apenas reflexão estrutural não podedo ser aplicado a este caso.

Meta-Java: implementa reflexão comportamental.

Guaraná: implementa reflexão comportamental, porém este protocolo ainda se encontra em desenvolvimento, dificultando o seu emprego imediato.

4 Trabalhos de Referência

São apresentados aqui alguns trabalhos de referência que contextualizam o problema e as suas soluções. As soluções podem ser baseadas em hardware ou em software. Serão também apontados os requisitos que precisam ser satisfeitos para que a substituição dinâmica possa ser realizada.

Substituições arbitrárias para o software não podem ser toleradas, se quisermos transferir informação sobre o estado de um processo antigo para um novo. Por exemplo, substituição dinâmica de software não é possível se a nova versão do software é um programa completamente diferente realizando computações totalmente diferentes.

A idéia básica é que se quer substituir o programa P, sendo executado por um processo, por um outro programa P' e enquanto ele estiver em execução. Assumimos que ambos, P e P', são programas corretos no sentido que eles executam propriamente e produzem os resultados esperados para as entradas fornecidas a eles. Isto é, o comportamento de ambos, P e P', é bem conhecido para seus respectivos domínios de entrada. O problema aqui abordado é o comportamento de um processo para o qual, durante a execução, P é substituído por P'.

4.1 Considerações Sobre Estado da Computação

Gupta [GUP93, GUP96] apresenta as considerações teóricas envolvendo o problema de substituição dinâmica de software para um modelo de programa imperativo ou procedural. Destas considerações merecem destaque as seguintes:

Um ambiente para modificações dinâmica precisa lidar com programas, processos, e o estado dos processos. Um programa no modelo procedural é constituído por uma seqüência de comandos.

• Um processo é um programa em execução, e apresenta dois componentes: o seu código e o seu estado. Geralmente o código do programa que está em execução permanece o mesmo durante o tempo de vida do processo, enquanto o estado do processo sofre alterações que dependem de uma função de transição que especifica o próximo estado dependendo do estado atual do programa que está executando.

• O estado de um processo é a caracterização completa de um ponto na vida do processo. Um processo é geralmente executado a partir de um estado inicial e passa por uma porção de estados durante a sua execução. Cada um dos estados intermediários pelos quais o processo passa é dito um estado alcançável ou estado válido do processo. Porém, não necessariamente um processo precisa ser executado a partir do seu estado inicial, mas pode ser executado a partir de qualquer estado alcançável ou estado válido do processo.

Como o estado é a caracterização completa de um processo, o comportamento de um processo, executado a partir de um estado intermediário válido, é idêntico ao comportamento que ele teria se tivesse sido executado partindo do estado inicial. O seu comportamento é independente de como aquele estado alcançável foi obtido, especificamente, além disso o estado alcançável pode ter sido criado por um

outro processo. Enquanto é feita a troca dinâmica de versão do programa, é muitas vezes desejável, e de fato necessário, mapear o estado do processo, seja para inicializar uma variável adicionada à nova versão, para modificar o tipo de uma variável, etc.

Uma substituição dinâmica é conceitualmente equivalente ao seguinte conjunto de passos: (1) A execução do programa P é iniciada com a pré-condição de P verdadeira. (2) No instante t, a execução é interrompida e o código do programa substituído pelo código P' (o estado do processo executando o programa permanece o mesmo). (3) Então a execução é retomada a partir do estado existente (mas com o novo código).

A tarefa é encontrar as condições suficientes no estado do processo que garantam a validação em uma substituição dinâmica. O sistema de substituição dinâmica precisa aguardar até que estas condições sejam satisfeitas pelo estado do processo antes de proceder com a instalação da substituição. Isto não pode ser feito testando as condições em pontos randômicos no tempo nem a intervalos regulares de tempo, pois o sistema poderá nunca produzir um ponto de parada para o processo em um estado que satisfaça as condições requeridas.

Outra solução possível é verificar estas condições após a execução de cada instrução do programa, mas isto tornaria o 'overhead' muito grande para uma substituição dinâmica. Numa abordagem mais prática o sistema de substituição dinâmica pode simplesmente aguardar até que a execução atinja um destes pontos de controle usando um mecanismo similar aos 'breakpoints' nos depuradores.

Intuitivamente pode-se perceber que, do ponto de vista do controle, uma substituição é válida se puder ser garantido que a execução, após o retorno da substituição, tenha restauradas todas as variáveis que foram afetadas pela substituição, antes de qualquer uso.

Gupta [GUP96] apresenta um framework para modelar as substituições dinâmica, onde sua validação é baseado no conceito de estados de processos. Foi escolhida esta abordagem porque a noção de validação pode ser usada em uma grande variedade de modelos de programas diferentes, incluindo os modelos de programas distribuídos.

Um aspecto importante do problema da substituição dinâmica é a validação da substituição. O objetivo da validação é similar ao da testabilidade do programa, visto que desejamos verificar (observando o comportamento do programa após a substituição) que a substituição é válida. Esta verificação não pode ser realizada efetuando-se a substituição no sistema real, pois uma substituição inválida pode levar a uma quebra 'crash' do sistema, anulando todo o propósito da substituição dinâmica. A validação de uma substituição dinâmica é portanto um importante tópico inexplorado ou aberto [GUP96].

Alguns sistemas para os quais a substituição dinâmica pode ser desejada na prática são de natureza de tempo-real. Para tais sistemas qualquer noção de validação precisa também garantir que nenhuma restrição de tempo-real seja violada. O framework apresentado por [GUP96] é incapaz de manipular tais restrições.

Novamente Gupta [GUP93] diz que uma substituição é dita completa se duas condições forem satisfeitas para todas as funções que serão modificadas:

A primeira condição diz que a funcionalidade da nova versão de uma função precisa ser um superconjunto daquilo que o cliente espera. Se isto não for verdadeiro, então a substituição não é transparente ao cliente que precisa conseqüentemente ser modificado. A segunda condição requer que a nova versão possa ser capaz de realizar suas funcionalidades sob as mesmas condições nas quais o cliente invoca a versão antiga. Se isto não for verdadeiro, então o cliente precisa saber disso e precisa ser modificado para chamá-la sob as novas condições.

Uma substituição dinâmica feita dentro do modelo procedural, sobre o programa P num instante t é válida se (1) ela for uma substituição completa e (2) a função que será substituída não estiver na pilha de execução do programa no instante t da substituição.

Sistemas de modificação dinâmica baseados no modelo procedural exigem um ambiente de execução especializado. Por exemplo, para saber qual a função que está executando é feita a observação direta da pilha do programa [GUP96].

Mas a pilha de um ambiente de execução normal não mantém informação sobre qual a função que está executando em cada instante, mantém apenas informação sobre o endereço de retorno, isto é, a próxima instrução que deve ser executada quando a função terminar. Para saber qual a função que está em execução em cada instante através da observação da pilha, é preciso adicionar um ID (identificador) da função.

Pilha

ID da função chamadora
parâmetros
endereço de retorno

A função principal do programa nunca pode ser modificada, pois ela está sempre na pilha de execução do programa [GUP93]. Felizmente, num programa bem estruturado, a maior parte do trabalho é feito por funções de nível mais baixo, e assim não requerem de fato que a função principal seja alterada.

Segal [SEG93] aponta 7 características principais para um sistema de substituição dinâmica.

- Preservar a correção do programa
- Minimizar a intervenção humana
- Suportar substituição de programas em vários níveis, desde uma substituição de apenas um módulo por outro implementado de uma maneira diferente mantendo a mesma interface, até substituições complicadas que incluem modificação da interface dos módulos, tendo que preservar o estado do módulo entre a substituição, alterando a implementação do seu estado interno e modificando suas

variáveis de estado. Alterar a implementação das variáveis de estado freqüentemente ocorre em programas que implementam estruturas de dados na forma de tipos abstratos de dados.

- Suportar reestruturação de código: a substituição deve permitir atualizar programas quando novos módulos são adicionados; módulos existentes são deletados e a funcionalidade é migrada entre módulos.

- Atualizar programas distribuídos

- Não necessitar de hardware específico ou especializado

- Não impor restrições à linguagem e ao ambiente

Um sistema de tal sofisticação é de construção complexa e a literatura consultada não refere nenhum exemplo de sistema que atenda a todos estes requisitos.

4.2 Soluções Baseadas em Software

A vida de um programa pode ser separada em tempo de compilação, tempo de ligação, tempo de carga e tempo de execução, sendo estas as alternativas possíveis para realizar uma substituição de versão.

4.2.1 Substituição de Processos

Nesta abordagem de software estruturado na forma de processos, a unidade de substituição é um processo: a nova versão distingue-se da anterior quanto à implementação de pelo menos um processo. Quando uma nova versão do software deve ser instalada, um processo adicional que implementa a nova versão é criado pelo sistema.. A substituição do processo operacional pela nova versão requer a transferência de estado entre os processos envolvidos. Esta abordagem foi adotada no microsatélite brasileiro [ANA97] e na proposta descrita em [GUP93].

Na proposta de Gupta [GUP93], um processo pode conter uma ou mais funções. Quando nenhuma das funções contidas nos processos envolvidos na substituição está na pilha de execução, o sistema transfere o estado do processo antigo (executando o software antigo) para o novo processo (executando o novo software) e elimina ('kill') o processo antigo. O usuário não percebe descontinuidade no serviço do software e apenas um tempo adicional marginal é imposto pelo sistema. A maior vantagem desta abordagem é que ela conduz a uma implementação muito simples e eficiente e não requer nenhum compilador ou ligador especial.

Este sistema de substituição dinâmica de processos foi implementado num nodo único, mas pode ser facilmente modificado para substituição dinâmica de componentes de um programa distribuído comunicando-se por mecanismos de chamada remota de procedimentos - RPC. RPC suporta uma interação do tipo cliente/servidor semelhante à utilizada na abordagem de processos. Numa implementação RPC tipicamente o cliente e o servidor são dois processos independentes executando em máquinas diferentes.

Gupta utilizou na sua implementação uma camada de modificação que atua entre o usuário e o sistema. Esta camada recebe comandos do usuário, o qual deve

solicitar a troca de componentes, e os executa. Todos comandos relacionados à substituição dinâmica passam por esta camada.

O mecanismo de carga dinâmica de processos aplicado ao microsatélite brasileiro [ANA97] também usa este princípio. O computador de bordo do Primeiro Microsatélite de Aplicações Científicas Brasileiro (SACI 1) conta com uma facilidade para efetuar o carregamento dinâmico de programas a partir da Terra. A idéia de carregar processos dinamicamente garante maior confiabilidade ao sistema e lhe dá maior flexibilidade, possibilitando pequenas correções nos aplicativos e até instalação de novas aplicações após o sistema ter sido embarcado.

O SACI 1 utiliza um sistema de carga dinâmica de processos baseado numa procedure de carga *Kernel Run* e em três tabelas: tabela de carga de processos, tabela de redirecionamento de processos e tabela de processos carregados [ANA97]. O hardware é baseado em transputers e programado na sua linguagem nativa, OCCAM. A linguagem OCCAM já provê rotinas para a carga dinâmica e execução de um código compilado separadamente.

A comunicação do SACI 1 utiliza um roteador que centraliza o controle de todas as comunicações no sistema. Todas as mensagens que são trocadas entre os aplicativos passam por este roteador.

A estratégia de carga usada pelo carregador prevê a existência de um determinado número de processos inicialmente inativos chamados processos SPARE. Um programa que é carregado dinamicamente, assume o lugar de um dos processos SPARE vagos. As entradas da tabela de processos que não estão sendo utilizadas ficam sendo ocupadas por processos SPARE. Quando um novo aplicativo precisa ser carregado, outro processo SPARE é ativado para executar o código carregado. A função do SPARE é ocupar o lugar de um futuro aplicativo.

O roteador consulta uma tabela de redirecionamento de processos para cada mensagem que trafega por ele, e se existir uma entrada nesta tabela para o processo destino a mensagem é redirecionada para ele. A existência desta tabela permite a substituição de um aplicativo por outro causando um mínimo de impacto para o roteador e deixando a substituição transparente para os demais aplicativos.

Durante o processo de carga um novo programa é enviado da Terra para o satélite através de tele comandos e é instalado numa porção livre da memória RAM do computador de bordo. A carga pode tomar dois caminhos: o código recebido pode ser o substituto de um outro aplicativo, ou pode ser um novo aplicativo.

No caso de substituição o carregador altera a tabela de processos e em seguida cria uma entrada na tabela de redirecionamento de processos para que o aplicativo possa receber suas mensagens. No caso do código recebido ser um novo aplicativo, não é necessário alterar a tabela de redirecionamento de processos. Um ponteiro é por fim passado para a tabela de processos ativos informando o endereço onde se localiza o código para a execução do novo aplicativo carregado.

Não é feita referência quanto ao uso de técnicas de tolerância a falhas a nível de substituição de versões no projeto.

4.2.2 Substituição de Procedimentos

No sistema PODUS (Procedure-Oriented Dynamic Updating System) [SEG93] um programa é atualizado fazendo a carga de uma nova versão do programa e substituindo cada procedimento da versão antiga pelo seu procedimento correspondente na versão nova, durante a execução. Atualizar um procedimento 'procedure' implica em manter ambas as verões na memória e então mudar a amarração ou ligação do procedimento na sua versão antiga para a nova versão. O sistema aguarda para realizar a substituição quando o procedimento não estiver em execução.

PODUS impõe duas restrições à estrutura do programa que será atualizado. (1) Os programas precisam ser escritos numa metodologia 'top-down' e os procedimentos mais baixos na hierarquia são atualizados por primeiro e depois são atualizados os mais altos na hierarquia. (2) Para garantir integridade de dados durante a substituição de uma procedure, PODUS exige que todos os dados globais manipulados pelas procedures sejam acessados através de tipos abstratos de dados.

Os procedimentos são carregados através de 'sockets' e o sistema executa em ambiente UNIX.

4.2.3 Ligação Dinâmica

Ligação dinâmica do código objeto é outro mecanismo para de fazer a modificação de um sistema em execução [FRA97]. A idéia original buscada com a ligação dinâmica era de compartilhar módulos comuns entre diversas aplicações. Os módulos constituintes de um sistema são compilados separadamente e são mantidos distintos durante todo o seu tempo de vida. Os módulos permanecem isolados até o instante da carga, assim eles podem ser transportados e reutilizados independentemente um do outro.

Sistemas extensíveis podem ser obtidos com esta técnica, permitindo que mais módulos sejam adicionados ao sistema, mesmo no topo da hierarquia de módulos. Com isso as aplicações podem aumentar suas funcionalidades em tempo de execução.

Os módulos extras devem registrar sua presença junto à aplicação original, que pode então usá-los sem a necessidade de realizar uma ação explícita de importação. Ao invés disso os módulos são ativados por mecanismos indiretos de chamada de procedimentos 'callback'. Prover estes mecanismos de uma forma confiável requer uma linguagem que suporta esta capacidade explicitamente, como Oberon [FRA97].

A maioria dos sistemas que suportam ligação dinâmica atualmente utilizam um carregador/ligador que modifica endereços diretamente no código do programa objeto em tempo de carga, antes de iniciar a execução ao invés de manipular tabelas. Além da modificação de código em tempo de execução ser um processo complexo, ele invalida freqüentemente o cache de instruções.

Uma abordagem alternativa a esta é utilizar chamadas a tabelas de endereçamento indireto para ligar diferentes módulos. Estas tabelas são alteradas

dinamicamente, em tempo de execução, se um módulo precisa ser substituído por uma nova versão. Para ligar um módulo A que está em execução, com outro módulo B importado da biblioteca o ligador simplesmente substitui as referências simbólicas existentes na tabela de ligação de A pelas posições de memória atuais dos procedimentos 'procedures' correspondentes. Para isso ser efetivado, todas as chamadas externas contidas no código objeto devem ser indiretas, e serem feitas através da tabela de ligação. Através deste esquema de ligação, todos os módulos são livremente relocáveis na memória. Se um módulo precisa ser movido para outro endereço mais tarde, apenas as entradas na tabela de ligação dos seus clientes e sua própria entrada na tabela precisam ser atualizadas.

A alteração da tabela é feita de tal maneira que não cause nenhuma inconsistência, mas esta estratégia acrescenta uma sobrecarga ao sistema. A maior desvantagem desta abordagem é que os mecanismos usados para atualização dinâmica são complicados e requerem que novos compiladores e ligadores sejam escritos [GUP93].

É especialmente atrativo ao desenvolvedor de software poder modificar a implementação de um módulo sem ter que recompilar seus clientes. Modificar a implementação de uma função em um módulo pode alterar seu comprimento e com isso afetar o endereço de início das funções que a sucedem no código objeto. Com ligação dinâmica e fazendo uso de referências indiretas esta alteração não afeta os módulos clientes porque ligações inter-módulos referem apenas a posições numa tabela cujos endereços de início são obtidos da tabela do código objeto carregado da biblioteca.

Franz [FRA97] apresenta diferentes abordagens para ligação dinâmica em sistemas modulares, mas não faz referência sobre a preservação (salvamento/restauração) do estado interno dos módulos que são carregados e descartados dinamicamente. Devido à sua abordagem estar dentro do modelo procedural que permite apenas substituição de código executável, não há sentido na preservação dos dados locais criados e manipulados pelas funções, pois estes dados são locais e portanto voláteis, só existem no curto tempo de execução da própria função.

4.2.4 Compilação em Tempo de Carga

Duas novas estratégias muito promissoras para realizar a carga dinâmica de módulos são a geração de código em tempo de carga e a compilação total em tempo de carga.

Até recentemente era universalmente aceito que o esforço requerido para uma compilação era tão grande que ela precisaria ser feita separadamente 'off-line'. Mas constata-se que o poder de processamento dos processadores está aumentando muito mais rapidamente que a velocidade das operações de I/O. A geração dinâmica de código 'on-the-fly' e em tempo de carga está hoje se tornando viável, pois reduz a quantidade de dados que precisam ser transferidos do armazenamento externo.

Através do uso de uma representação intermediária de programas altamente compacta chamada de 'slim-binary' ao invés de fazer a carga diretamente de arquivos

com o código objeto nativo da máquina, reduz-se dramaticamente o tempo de carga por I/O. O tempo economizado é então gasto na geração de código. Arquivos objeto são usualmente muito maiores do que o necessário [FRA97]. A representação em 'slim-binary' é tipicamente de duas e meia a três vezes mais compacta que o código objeto para microprocessadores comuns e pode se tornar ainda menor após a aplicação de técnicas de compressão de dados.

'Slim-Binaries' apresentam outra vantagem: além de ocuparem menos espaço, são independentes da máquina em que irão executar, o que simplifica a manutenção da biblioteca de módulos reduzindo o custo da produção de software.

Porém estas estratégias ainda não foram estudadas extensivamente pois o poder de processamento que viabiliza isto está começando a se tornar disponível apenas agora.

Um esquema semelhante foi recente proposto para acelerar a execução da Máquina Virtual Java através da compilação 'Just-in-time' que faz uma ligação do código gerado dinamicamente 'on-the-fly' numa operação tardia que é aplicada sobre um procedimento de cada vez.

A compilação total a tempo de carga não utiliza 'slim-binaries', ao invés disso faz a compilação diretamente a partir do código fonte. Esta abordagem também está em fase de estudo, um dos problemas em aberto é que podem ocorrer erros de compilação durante a carga, que comprometeriam a eficiência do processo.

4.3 Soluções Baseadas em Hardware

Soluções baseadas em hardware são caras e têm aplicação restrita [SEG93]. Em um sistema que usa atualização dinâmica baseada em hardware, um equipamento contendo um programa completo em execução é dinamicamente substituído por um segundo equipamento idêntico ao que está executando o programa. Devido ao segundo sistema de computador com a nova versão do programa ser carregado enquanto o primeiro computador continua executando a versão antiga, os programas podem ser atualizados com o mínimo tempo de indisponibilidade e máxima flexibilidade em reestruturação. Para realizar a substituição o primeiro computador deve ser interrompido em um ponto seguro na execução do programa e simultaneamente iniciado o segundo.

A principal desvantagem desta técnica é seu custo substancial. Ela é principalmente utilizada em sistemas que usam redundância de hardware para prover TF, como por exemplo sistemas de telecomunicações. Mesmo assim construir um sistema de computador redundante é difícil e dispendioso. Não apenas o hardware precisa ser interconectado, mas o software compartilhado entre os sistemas e as informações tais como em banco de dados precisam ser mantidas consistentes. Além disso, uma abordagem baseada em hardware redundante exige forte acoplamento entre os sistemas e não é adaptável a um ambiente distribuído. Por estas razões o enfoque dado neste trabalho é voltado para um sistema baseado em software.

4.4 Quadro Comparativo

Além dos trabalhos correlatos aqui descritos, vários outros sistemas podem ser encontrados na literatura. A seguir é mostrado um quadro comparativo entre sistemas de substituição dinâmica, baseado em [SEG93], onde se encontram em destaque os trabalhos abordados neste capítulo.

QUADRO 1 – Quadro comparativo de sistemas de troca dinâmica de componentes

	Argus	Conic	DAS	DMERT	DYMOS	Ligação dinâmica	Substituição de tipos	MTS	PODUS	SCP
tipo de atualização	cliente-servidor	módulos	procedimento	procedimento	procedimento	procedimento	tipo abstrato de dado	cliente-servidor	procedimento	Hardware
granula-ridade	todo o servidor	módulo	procedimento	procedimento	procedimento	procedimento	tipo abstrato de dado	programa	procedimento	programa
status/93	pesquisa	pesquisa/ produto	pesquisa	produto comercial	pesquisa	produto comercial	pesquisa	pesquisa/ produto	pesquisa	produto comercial
S.O. especializadoo	-	-	Sim	Sim	-	Sim	Sim	Sim	-	Sim
ambiente execução	Sim	Sim	-	-	Sim	-	-	-	-	-

Destes ambientes apenas o sistema PODUS não necessita de um sistema operacional especializado ou de um ambiente de execução. Os procedimentos são carregados através de sockets e o sistema executa em ambiente UNIX.

4.5 Conclusões

As soluções estudadas são unânimes no aspecto da importância de uma substituição dinâmica para garantir a continuidade do serviço, mas não oferecem mecanismos para tratar falhas no novo componente caso elas se manifestem.

As soluções baseadas em hardware facilitam a substituição dinâmica de componentes, com um bom grau de independência do domínio da aplicação. Já as soluções baseadas em software envolvem preocupações com o domínio da aplicação e a estrutura do programa.

Considerando é um consenso na área de engenharia de software que os programas podem apresentar falhas durante a sua execução, o principal problema ainda continua sendo garantir a validação da substituição, ou seja, garantir que o novo componente é funcionalmente correto.

Os projetos pesquisados na literatura enfocam apenas os aspectos ligados à carga e substituição dinâmica de módulos em si, mas não apresentam mecanismos de TF para a detecção e recuperação de falhas durante o processo de substituição. Todos adotam a hipótese de que a nova versão do programa é uma versão correta e que executará precisamente as suas funções depois de substituída no sistema. Nenhuma das abordagens pesquisadas apresenta execução de ambas versões (a antiga e a nova) durante um período de validação, todas elas descartam a versão antiga imediatamente após a substituição.

As implementações encontradas na literatura enfocam sempre o modelo orientado a procedimentos que é uma abordagem mais limitada, pois implica em ajuste de endereços de código em memória, apresenta dificuldade para modificar dados globais, requer programação em baixo nível (para monitorar o estado da pilha do programa) e assim são requeridas soluções específicas para as diferentes arquiteturas.

Por outro lado, uma substituição no modelo de objetos apresenta possibilidade de realizar trocas também nos tipos de dados, desde os tipos simples (por exemplo, de inteiro para real), como tipos abstratos (uma pilha implementada com array passa a ser implementada através de um lista encadeada). No modelo orientado a procedimentos uma modificação nos dados globais é muito mais complexa.

No modelo orientado a procedimentos não há necessidade de monitorar o estado da função a ser substituída. Isto porque as funções criam apenas dados locais e a substituição ocorre quando a função não está executando, assim não há dados locais que a função antiga necessita transferir para a nova. Mas para realizar uma substituição no modelo de objetos deve haver preocupação com o estado do objeto, cujos valores das variáveis instanciadas pelo objeto antigo precisa ser mapeados para os valores correspondentes no novo domínio (reestruturação de dados).

5 A Proposta na Prática, Problemas e Soluções

É apresentada neste capítulo a descrição de uma proposta de sistema de substituição de versões de programas tolerante a falhas.

Como premissa fundamental para a possibilidade e viabilidade de uma solução mais simples para o problema da substituição dinâmica será definido o seguinte predicado ou hipótese:

"Os resultados e as soluções apresentadas neste trabalho somente terão aplicação ou validade para um sistema novo, ou seja, um sistema que já tenha sido projetado desde a sua fase inicial, prevendo uma futura substituição dinâmica de algum dos seus componentes".

Esta definição é um ponto de partida para o trabalho aqui desenvolvido. Isto significa que os sistemas antigos e que não foram previstos para substituição dinâmica, não irão se beneficiar das estratégias e técnicas aqui apresentadas, ficando estes sistemas no aguardo do surgimento de uma solução genérica.

O modelo é implementado segundo o estilo de interações cliente-servidor. O sistema consiste de um programa coordenador que é visto como cliente dos serviços prestados pelos objetos servidores da aplicação. Estes objetos executam a quase totalidade das funcionalidades da aplicação. A figura 5.1 esquematiza a arquitetura básica do sistema adotado como modelo.

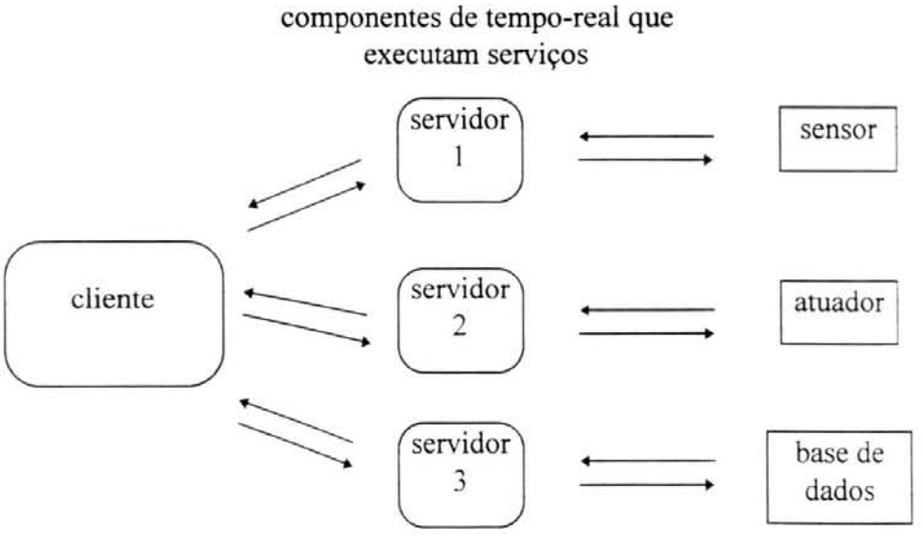

FIGURA 5.1 - Componentes de um STR no modelo de objetos

Cada objeto é visto como um elemento ou componente integrante do programa coordenador, mas é externo a ele. O programa coordenador não consegue operar ou funcionar de forma plena sem o serviço prestado por estes módulos externos. A atribuição mais importante do programa coordenador é gerenciar e coordenar a execução dos objetos servidores. Esta arquitetura torna viável uma

substituição de um destes objetos servidores por outro com funcionalidades equivalentes, ou seja, uma substituição da versão do objeto. Os objetos são instanciados externamente para viabilizar sua substituição sem exigir o cancelamento do programa coordenador. E evitando as operações de relocação interna de endereços [SEG93, FRA97] que surgiriam caso a substituição dos componentes fosse feita internamente ao programa coordenador.

É interessante que o programa coordenador tenha uma participação mínima na execução das tarefas específicas da aplicação, deixando a execução destas tarefas para os objetos prestadores de serviço. Realizar a substituição da versão do programa coordenador implica no seu desligamento ou cancelamento temporário, o que é inviável em sistemas críticos. Quanto maior o grau de serviço prestado pelos objetos, maior o grau de substituição que o sistema admite. Quem é passível de substituição dinâmica são os objetos servidores, mas não o programa coordenador.

Utilizando esta arquitetura para o STR não é necessário fazer a adoção de um sistema operacional especializado desenvolvido com a finalidade de fazer a carga dinâmica destes componentes. O programa coordenador é um elemento de gerência de execução que realiza o controle dos objetos da aplicação, invocando estas operações básicas tornando o sistema mais flexível com relação ao tipo de sistema operacional em que ele irá rodar. Sua tarefa de controle de objetos compreende a criação e remoção de objetos durante a execução da aplicação. O programa coordenador também não se preocupa com os detalhes de implementação dos seus objetos servidores externos, ficando responsável apenas pela utilização destes objetos (figura 5.2).

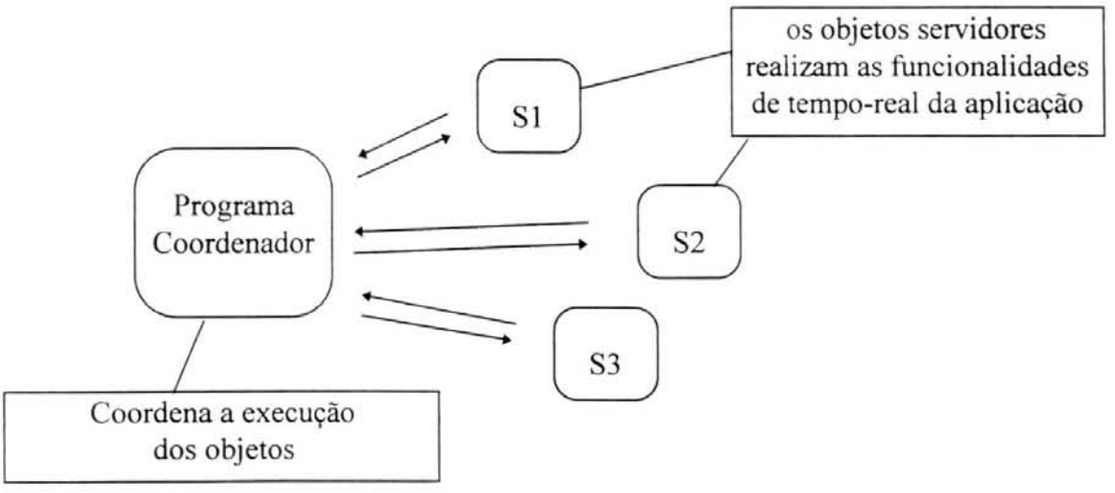

FIGURA 5.2 - Arquitetura do Sistema

Surgem as características como a rigidez nos limites de tempo e sincronismo, peculiares aos sistemas de tempo-real, como limitadores no uso de estratégias convencionais de substituição de versões de software. Portanto, faz-se necessário um levantamento e um estudo claro das características e principalmente das limitações impostas pelos STR para a aplicação de novas técnicas de substituição.

Assim este trabalho visa analisar o uso de técnicas de TF para realizar a substituição de versões de software em sistemas de tempo-real distribuídos, tendo como objetivo principal os aspectos de disponibilidade e confiabilidade. Todo este estudo pretende criar condições de avaliar a potencialidade de futuramente serem gerados sistemas operacionais ou ferramentas que permitam realizar a alteração suave na versão de aplicações de tempo-real críticas.

As técnicas de tolerância a falhas podem tornar o objeto muito grande, complexo e tornar a sua programação muito difícil. Visando simplificar a programação e isolar as estruturas e técnicas de TF das funcionalidades da aplicação tem sido adotada a técnica da reflexão computacional [LIS95b, LIS95c, FRA96a]. Os objetos da aplicação são controlados por meta-objetos que atuam em um meta-nível entre o objeto e o usuário externo deste objeto. Baseado neste conceito foi constituída a entidade do Meta Objeto Tolerante a Falhas - MOTF [LIS95c].

MOTF são abstrações de objetos confiáveis que utilizam reflexão computacional aliada à técnicas de TF com objetivo de prover ferramentas para a construção de software confiável a ser utilizado em aplicações críticas.

O sistema proposto tem por base um conjunto de objetos servidores que realizam operações e tarefas básicas da aplicação. Os resultados destas tarefas servem como elemento de entrada de dados para as etapas mais altas na hierarquia do sistema. Assim estabelece-se uma relação cliente-servidor. Vamos impor também que o sistema é crítico e portanto não pode sofrer interrupções, nem mesmo para instalar uma nova versão dos seus componentes servidores básicos.

A idéia de solução aqui proposta se aplica tanto a um sistema centralizado (multitarefa) como a um sistema distribuído, a abordagem será a mesma em qualquer dos casos, não sendo necessária uma mudança de paradigma. Caso seja desenvolvida uma aplicação distribuída, os objetos servidores ficarão instalados em nodos remotos executando suas funcionalidades e prestando seus serviços através da troca de mensagens pela rede de comunicação. Os clientes dos serviços são programas instalados em nodos distintos da rede, que solicitam a execução do serviço igualmente através da troca de mensagens com os servidores.

A seguir são mostrados alguns exemplos de sistemas críticos e de serviços executados pelos objetos servidores remotos e nos quais a solução proposta é aplicável. Todos os exemplos assemelham-se pela utilização da mesma estratégia de substituição, mas diferem pelo contexto em que se inserem e pelo tipo de serviço executado pelos servidores.

- a gravação de dados críticos em um arquivo ou banco de dados para uma transação bancária. O sistema crítico que não pode sofrer interrupção é o sistema coordenador de controle de operações bancárias (cliente) e os objetos servidores são igualmente críticos porque precisam executar com confiabilidade as suas operações de gravação dos dados. Nem o sistema de controle de operações bancárias e nem os objetos servidores remotos podem deixar de prestar seus serviços. O problema surge quando necessita-se fazer a substituição da versão de um dos objetos servidores. O objeto servidor não pode ser desativado para passar por um período de testabilidade da

nova versão. Todos os usuários dos seus serviços ficariam sem dispor dos seus serviços, comprometendo e tornando indisponível o sistema todo. Figura 5.3.

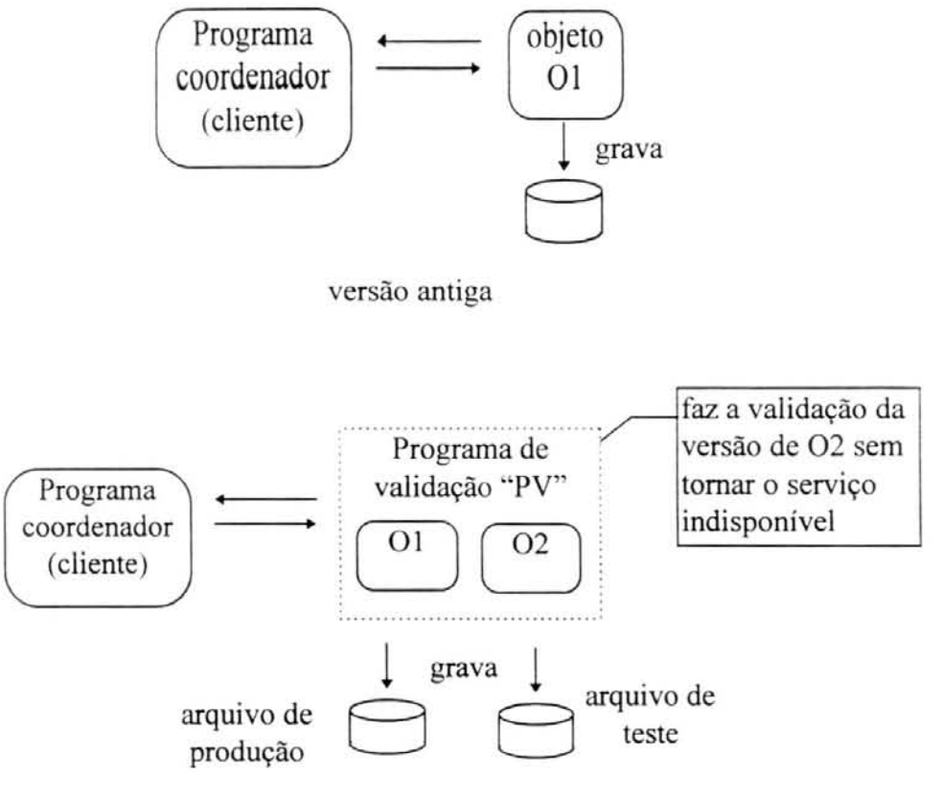

FIGURA 5.3 - Cenário de uma transação bancária

• um aeroporto movimentado que possui seu sistema de radares controlado por um programa servidor que realiza, por exemplo, o acionamento mecânico da antena do radar e ainda realiza a coleta dos dados e envia ao cliente que é um computador central coordenador do processamento. Vamos supor que se deseja efetuar uma substituição da versão do software ou/e hardware instalado neste servidor. O servidor do radar não poderá permanecer desativado por todo um período longo de testabilidade da nova versão do software. Não existe certeza de que a nova versão apresente funcionamento correto após a substituição. Um ambiente de teste para este tipo sistema teria um custo elevado e não conseguiria fazer uma representação fiel da realidade. Caso surja uma falha na nova versão, deve ser providenciada a reinstalação imediata da versão antiga. Tal controle de substituição não pode ficar ao encargo de um operador que realiza um acompanhamento visual e ininterrupto do desempenho da

nova versão. Este é outro exemplo são necessárias técnicas de TF para validar a substituição. Figura 5.4.

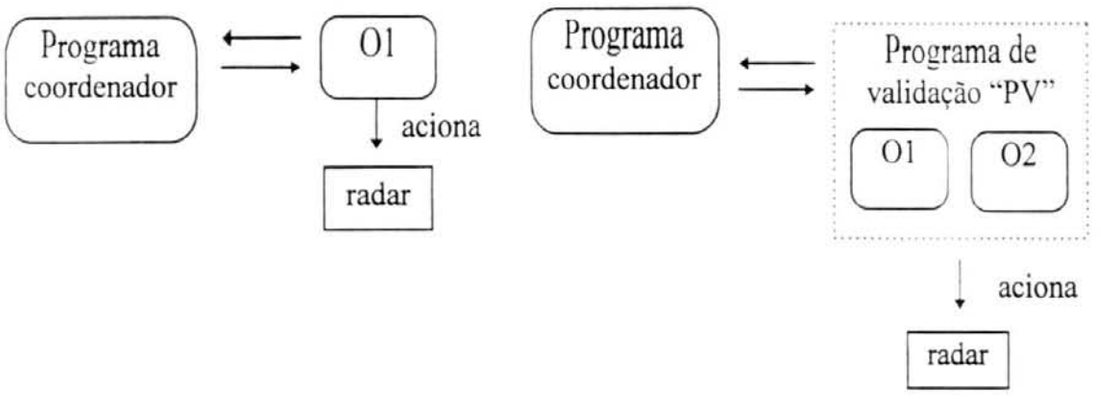

FIGURA 5.4 - Cenário de um sistema de radares

• um sistema de controle de ferrovias possui computadores distribuídos com a função específica de administrar o acionamento de um comutador dos trilhos nos entroncamentos. Este servidor recebe comandos de uma central informando a seqüência de comutações a serem efetuadas em função dos trens que partem das diferentes estações em distintos horários. Deseja-se fazer a substituição da versão do software instalado neste servidor devido, por exemplo, à construção e liberação de mais uma linha para o tráfego o de trens. O sistema servidor não pode ser interrompido para que ocorra um período de validação do novo software. Pode ser admitida uma pequena interrupção entre a passagem de um trem e outro para a instalação da nova versão. Mas não ocorre interrupção do sistema central que coordena e controla o fluxo em uma escala maior. Em caso de falha do novo sistema, o sistema antigo precisa ser reativado. Figura 5.5.

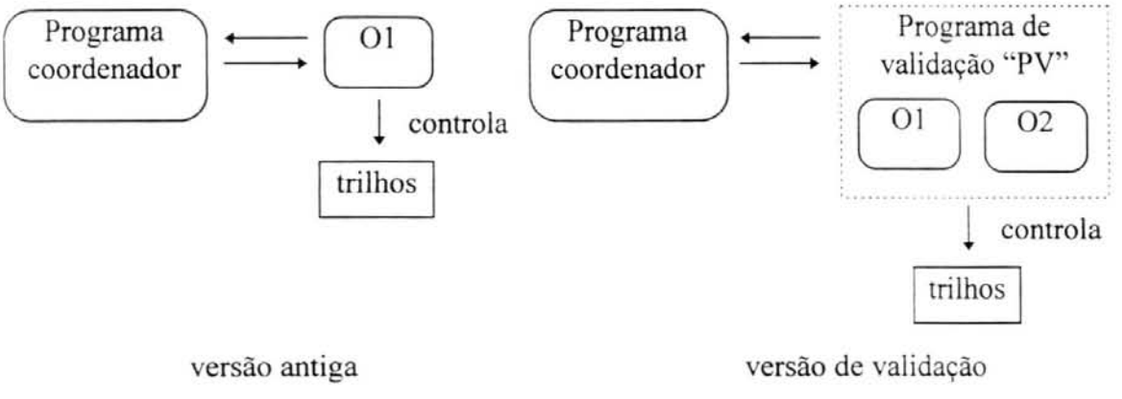

FIGURA 5.5 - Cenário de um sistema de ferrovias

• o sistema de controle de drenagem de uma mina [BUR96], possui um sensor de nível de água e um motor que aciona a bomba controlados por um sistema de computador. O sensor é controlado por um objeto servidor, o mesmo ocorrendo com o motor. Um programa coordenador central gerencia a operação da mina, fazendo suas solicitações de serviço aos objetos servidores. Desejo substituir a versão do objeto sensor para operar com um novo sensor cuja escala de leituras é mais ampla. Não é possível desativar o objeto servidor durante um tempo longo de validação da nova versão. Somente é possível cancelar o objeto servidor durante um curto intervalo de tempo em que estiver ocioso entre uma leitura e outra. O programa coordenador que gerencia a drenagem da mina não é interrompido para efetuar a substituição da versão do objeto servidor. Ambos comunicam-se através da troca de mensagens. Figura 5.6.

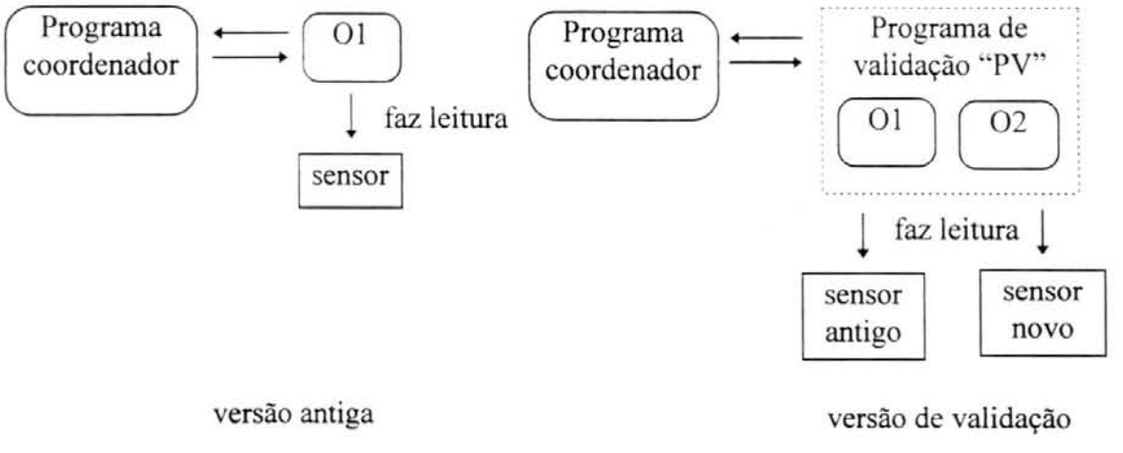

FIGURA 5.6 - Cenário de um sistema de drenagem

Todos este exemplos mostram a necessidade de um sistema de substituição dinâmica que seja aplicado aos programas servidores de tempo-real. O programa central não é interrompido, garantindo assim a transparência ao sistema; apenas ocorre substituição da versão do software nos programas servidores, que não podem deixar de fornecer seus serviços com alta disponibilidade e de forma correta e confiável. Uma rápida interrupção na execução do servidor é feita num momento de ociosidade, para proceder à instalação de uma nova versão do seu software, colocando-o em atividade logo em seguida.

O objeto antigo tem um funcionamento bem conhecido pelo sistema e o objeto novo pode apresentar algum comportamento inesperado como reação a alguma mensagem. Caso seja encontrada alguma divergência entre os resultados, o sistema de substituição faz o registro em um arquivo de log e retorna ao cliente o resultado da versão antiga.

Os objetos servidores podem ser vistos (figura 5.7) como uma unidade de software associada a uma unidade de hardware. Uma classe é usada para abrigar abstrações. De uma maneira geral um sensor abstrato é um objeto que modela um sensor e relaciona-se diretamente com um sensor físico instalado no meio ambiente.

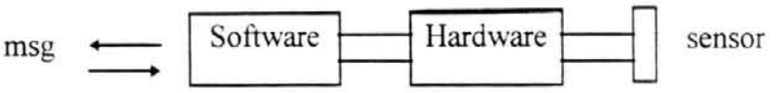

FIGURA 5.7 - Esquema de um objeto servidor

As classes permitem um dinamismo que não existe em componentes de hardware, pois os objetos podem ter sua estrutura e comportamento alterados mais facilmente. Estas características podem ser positivamente exploradas para a substituição dinâmica de componentes em programas O-O, visto que um objeto é considerado um componente atômico de uma aplicação, possui uma funcionalidade bem definida e encapsulada. Assim, admite facilmente o uso de versões na forma de objetos distintos de igual funcionalidade, mas que se distinguem pelo seu estado.

A decisão de fazer a substituição de um determinado componente (objeto) é tomada pelo administrador do sistema. A carga de uma nova versão do objeto é feita manualmente, pela ação de um programador via console. O computador principal ou cliente não tomará conhecimento de que está havendo a carga de outro objeto em lugar do anterior. Os programas clientes dos serviços não têm ciência de que houve de fato uma troca de versão e que está havendo uma etapa de validação do novo software instalado no objeto servidor. Os resultados do monitoramento feito durante o período de testabilidade também ficam armazenados no próprio nodo remoto.

O nodo servidor continua prestando seus serviços "ininterruptamente", mesmo durante a fase de testes e até em caso de falha na nova versão instalada. Apenas ocorre uma interrupção mínima no servidor que corresponde ao tempo para fazer a carga do novo programa. O programa principal (cliente) que utiliza os serviços do objeto servidor não é cancelado para a substituição.

5.1 O Processo de Substituição

O computador servidor executa a versão antiga de um programa P1 cujo código contém essencialmente o objeto O1 que executa as suas funções de serviço corretamente e de forma confiável. Quer se substituir este programa P1 por outro programa P2 que contém o código do objeto O2. Mas temporariamente (durante o processo de validação), vai de fato ser executado um programa de validação "PV" que contém simultaneamente ambas as versões O1 e O2 e que vai passar a gerenciar estes dois objetos: O1 e O2, além de executar as tarefas de validação e TF.

Quando o programa P1 é cancelado o objeto O1 tem sua execução descontinuada, mas logo em seguida é carregado o programa de validação PV que possui uma réplica de O1. O objeto O1, agora instanciado por PV, é uma cópia idêntica do objeto que foi instanciado por P1 e que estava em execução até o momento da substituição. Assim O1 volta à atividade, mas desta vez fazendo parte do programa de validação PV. O objeto O2 é a nova versão que irá mais tarde substituir o objeto O1 em definitivo quando for realizada a substituição do programa PV pelo programa P2. Figura 5.8.

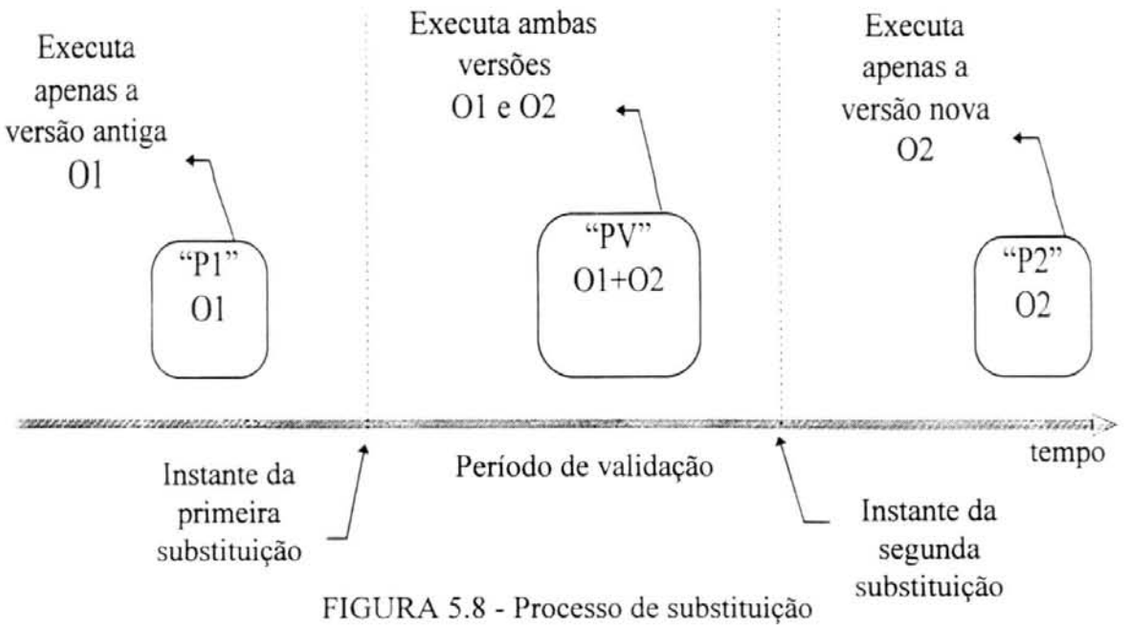

FIGURA 5.8 - Processo de substituição

A interrupção do programa P1 e a carga do programa PV é feita de forma rápida num instante em que O1 não esteja efetivamente executando. Isto é, entre a prestação de um serviço e outro.

O objeto O1 não recebe uma solicitação de serviço quando está descontinuado porque as solicitações não são enviadas a ele diretamente, mas são centralizadas num objeto de caixa-postal que armazena as mensagens. Quando o objeto O1 volta a operar, agora dentro do programa PV, ele acessa sua caixa-postal e retira suas mensagens de solicitação de serviço. De fato o serviço não é prestado por O1 durante um curto intervalo de tempo, mas deve ser levado em conta que a aplicação tem a característica de tempo-real brando e admite atrasos eventuais nos resultados. Ou ainda a aplicação pode ser um sistema de tempo-real rígido, mas que tem um tempo de resposta suficientemente grande para comportar e suportar a operação de instalação da nova versão.

O programa PV contém meta-objetos associados a O1 e a O2 e que fazem o papel de árbitro da substituição. O meta-objeto assume as funções de TF, fazendo a detecção de falhas, o confinamento de danos, registro de erros num arquivo de log, etc. São possíveis duas abordagens para a estruturação do programa de validação PV:

1) usar um meta-objeto associado apenas à nova versão O2. Neste caso o objeto O1 não possui meta-objeto associado a ele e não há necessidade em utilizar reflexão computacional para mapear o seu estado interno. O1 é executado normalmente e são importantes apenas os resultados fornecidos por ele. Esta implementação é mais simples. Figura 5.9

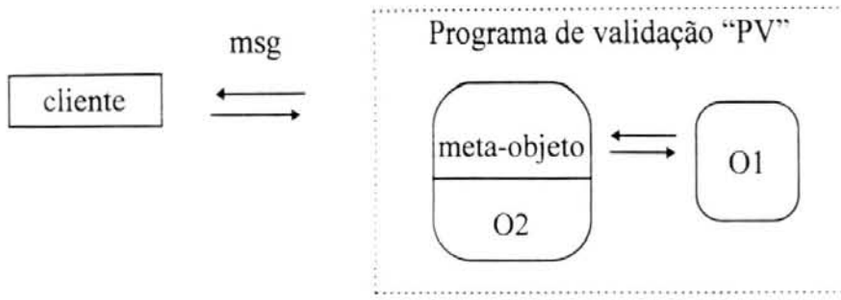

FIGURA 5.9 - Versão antiga O1 não possui meta-objeto

2) Empregar um meta-objeto para cada um dos componentes: para a versão antiga e para a nova. Neste caso, além de fazer introspecção no objeto novo O2, é possível fazer também no objeto antigo O1. Figura 5.10 A introspecção no objeto antigo é importante quando se quer comparar ou mapear o estado interno de ambos objetos O1 e O2, e não apenas comparar os resultados por eles fornecidos.

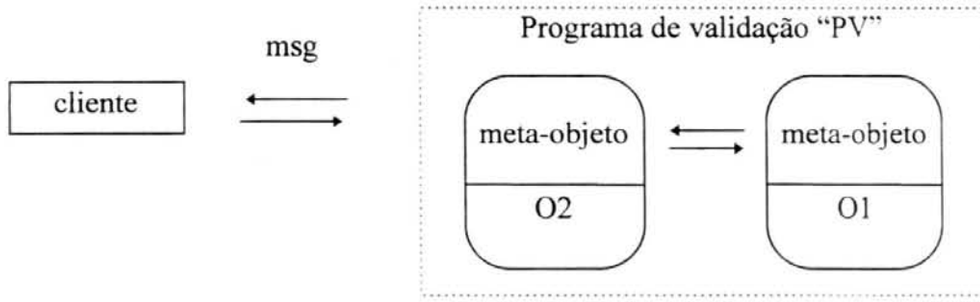

FIGURA 5.10 - Cada versão com um meta-objeto

Há duas abordagens para coordenar a execução de ambas as versões no programa PV.

1) As versões O1 e O2 são executadas seqüencialmente uma após a outra. No final os resultados são comparados. Se o resultado fornecido por O2 for incorreto, será passado ao cliente o resultado de O1. Esta solução necessita um tempo Δt maior até que seja fornecido o resultado definitivo ao cliente, mas permite que os meta-objetos tenham menores preocupações com sincronismo e proteção de dados.

2) Ambas as versões O1 e O2 são executadas simultaneamente e realizam seu processamento em paralelo para ganhar tempo. São comparados ambos os resultados. Se o resultado fornecido pela nova versão O2 for errôneo, o resultado correto é fornecido pela versão antiga O1 será repassada ao cliente. Esta operação ocorrerá de forma mais rápida, após um tempo $\Delta t'$, figura 5.11.

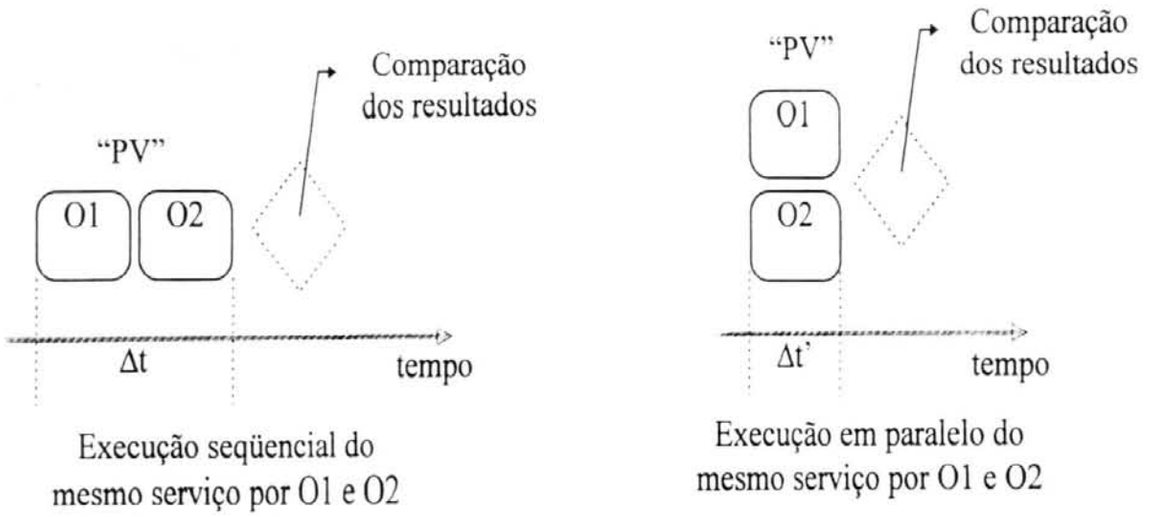

FIGURA 5.11 - Coordenação temporal da execução de O1 e O2

5.2 Salvamento do Estado da Computação

Para efetivar a instalação de outra versão do software, o programa P1 que executa o objeto O1 é descontinuado (cancelado) temporariamente durante um período em que estiver ocioso. Imediatamente após é instalada a nova versão PV que passa a executar as mesmas funcionalidades executadas antes pelo objeto O1.

O objeto O1 possui um estado interno [BOO94] que se modifica a cada ativação. Antes de ser cancelado o objeto O1 possuía um estado interno, o qual deve ser mantido ou preservado para assegurar que seu estado, após a substituição, seja igual ao que ele teria caso não fosse substituído e seu funcionamento permaneça compatível com o ambiente real existente antes da substituição. Este estado interno deve ser salvo antes do cancelamento de P1 e restaurado na volta da execução do objeto O1, agora instanciado dentro do programa de validação PV.

Quando o objeto O1 tiver seu estado interno restaurado ele possuirá um estado idêntico ao que possuía no momento imediatamente antes de ser cancelado [GUP96] e continuará executando as mesmas ações que executaria caso não fosse interrompido. Figura 5.12

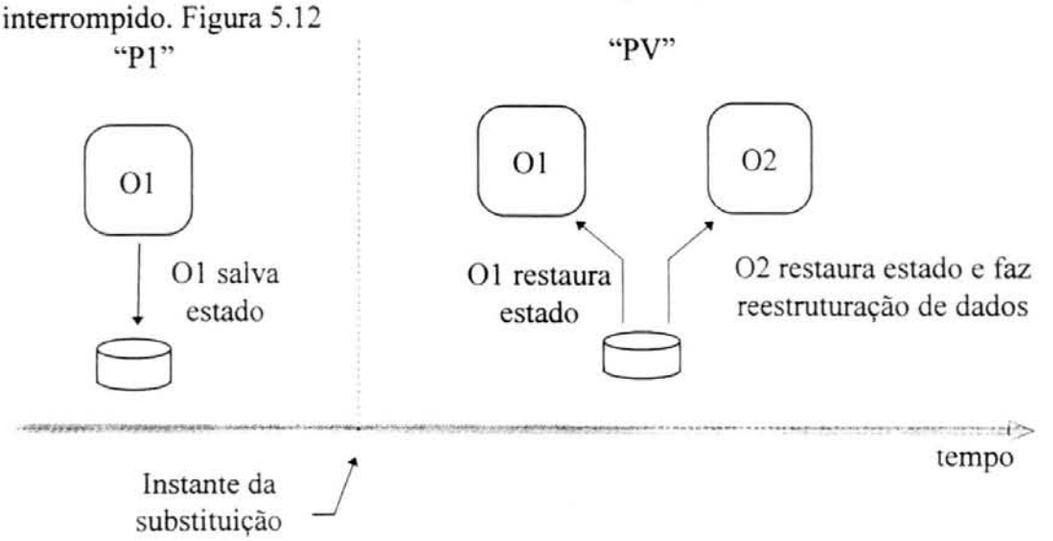

FIGURA 5.12 - Salvamento e restauração do estado

O salvamento do estado pode ser programado no método destrutor do objeto O1 e que é invocado antes de o objeto ser cancelado. Cabe lembrar que se o programa P1 não possui meta-objeto associado a O1, o próprio objeto deve prover um método para salvamento e outro para restauração do estado.

Neste ponto o objeto O1 (instanciado dentro do programa PV) já tem o seu estado restaurado. Mas o programa PV instancia também o objeto O2 que é a nova versão de software que deve ser testada. Este objeto O2 também precisa partir de um estado equivalente àquele encontrado em O1. Então é necessário fazer a transferência de estado de O1 para O2. Duas restaurações de estado estão envolvidas nesta etapa do processo de substituição onde entra em atividade o programa PV: uma de O1 para O1 (restauração para si próprio) e outra de O1 para O2. O objeto O2 tem seu estado inicial produzido ou formado a partir da mesma base de dados em que foi salvo o estado de O1.

Se o objeto O2, além de possuir uma implementação diferente do objeto O1 possui ainda uma estrutura de dados interna diferente de O1, então o estado interno de O2 não é exatamente o mesmo que o de O1. Torna-se necessário fazer uma reestruturação de dados [LIS94]. A reestruturação de dados é necessária, por exemplo, quando existe uma mudança no tipo dos dados armazenados ou quando ocorre mudança na estrutura de dados com que é implementado um algoritmo: um conjunto de sensores tinha seu acesso organizado na forma de um 'array' na versão 1, e que vai passar a ser organizado na forma de uma lista encadeada na versão 2. Para isto a lista encadeada precisa ser criada, cada nodo apontado para o seguinte, o último nodo deve ter ponteiro com o valor nulo 'null', além de ser necessário uma variável "cabeça" que aponta para o início da lista.

Uma vez que o programa de validação PV está carregado e ambos objetos com seu estado interno restaurado, inicia-se o processo de monitoramento da execução do objeto O2 feito pelo meta-objeto associado a ele, e que faz uso das técnicas de TF para garantir a continuidade dos serviços prestados.

5.3 A validação: Testabilidade e Técnicas de Tolerância a Falhas

Aqui será descrito o papel da validação, da testabilidade e das principais técnicas de TF para este projeto.

5.3.1 Validação e Testabilidade

A validação da substituição é o elemento inovador mais importante desta proposta, e provavelmente vai ser a etapa mais prolongada de todo o processo que envolve a substituição.

A validação exerce um papel similar ao da testabilidade de um programa. As técnicas de testabilidade de software que podem ser aplicadas são: (I) o teste da caixa preta, onde são verificados os resultados encontrados na saída do objeto em função dos valores aplicados na entrada. Para implementar esta verificação são indicadas técnicas de Tolerância a Falhas (TF) como o emprego de testes de razoabilidade. (II) o teste da caixa branca, onde é estudado o estado interno do objeto e são monitorados os caminhos por onde passa a execução. Os mecanismos de introspecção da reflexão

computacional são bastante adequados para dar suporte a testes da caixa branca, mas estes testes precisam ser estudados e implementados caso a caso.

A abordagem mais simples é aplicar o teste da caixa preta sobre o novo componente durante a validação. Isto é feito através da comparação entre os resultados fornecidos pela versão antiga que executa juntamente com a versão nova. Caso os resultados sejam divergentes é assumido que a nova versão apresenta falha. O teste da caixa preta é genérico pois é aplicado apenas sobre os valores dos resultados além de ser de mais fácil implementação do que os testes da caixa branca.

5.3.2 Técnicas de Tolerância a Falhas

As técnicas de TF agregam significativo conhecimento sobre gerenciamento de componentes. Através destas técnicas o comportamento de ambos objetos pode ser monitorado: o objeto antigo que tem um funcionamento bem conhecido pelo sistema e o objeto novo que pode apresentar algum comportamento inesperado como reação a alguma mensagem. Caso seja encontrada alguma divergência o sistema de substituição faz o registro em um arquivo de log e fornece o resultado da versão antiga. O tempo de coexistência dos componentes (antigo e novo) é determinado pela análise estatística dos resultados fornecidos pelo objeto novo.

Quando a confiabilidade do componente antigo não é assegurada, a identificação de resultados distintos dos dois componentes não permite identificar qual das duas versões apresentou comportamento incorreto.

5.3.3 Blocos de Recuperação

A técnica de TF que está presente de forma mais marcante é blocos de recuperação [AND81, SOM97] pois ela resume a idéia central da abordagem proposta.

Blocos de recuperação é uma técnica de redundância de software que visa a detectar falhas de projeto na construção do software [RAN78]. São desenvolvidas diversas versões de um programa, mas somente são executados os programas-extra, caso seja detectado um erro. Ao término da execução de um programa é feito um teste de aceitação do resultado que verifica a existência de erros. Em caso afirmativo, é executado um programa alternativo, que também tem seu resultado verificado pelo teste de aceitação, e assim sucessivamente. A vantagem desta técnica é que pode ser utilizada uma versão mais eficiente para executar em primeiro lugar, e caso esta apresente erro, outra versão de menor eficiência, porém mais robusta, pode ser executada, aumentando assim a eficiência e a confiabilidade do sistema como um todo.

No caso de substituição de versões a técnica de blocos de recuperação emprega apenas dois programas: a versão antiga e a versão nova. Cada vez que a nova versão O2 apresentar um funcionamento ou resultado incorreto, é fornecido para o cliente o resultado da versão antiga O1.

5.3.4 Mascaramento de Falhas

A técnica de mascaramento de falhas e confinamento de danos é empregada neste trabalho todas as vezes em que o objeto O2 apresentar um resultado incorreto. O erro ocorrido não é propagado para o cliente, que recebe sempre o resultado correto fornecido pelo objeto O1. O erro de O2 é então tratado pelo meta-objeto que providencia a sua recuperação.

5.3.5 Pontos de Recuperação

Um dos serviços mais simples de TF é restaurar o sistema a um estado consistente [JAL94]. Baseada nesta afirmação está a estratégia de recuperação adotada neste modelo.

Como a nova versão (objeto O2) está em testes não é uma surpresa que ele venha a apresentar erros como decorrência de falhas de projeto. O erro é caracterizado pelo fornecimento de valores incorretos em suas saídas. Um valor errado na saída é decorrente de mau funcionamento do algoritmo executando operações incorretas e que pode levar o objeto a um estado interno inválido.

É desejável que após a manifestação de uma falha o objeto O2 seja recuperado e volte a ter um estado consistente para poder prosseguir com suas atividades normalmente. A nova versão O2 precisa mostrar um diagnóstico amplo e claro ao projetista daquelas situações em que apresenta operações corretas e das que apresenta falha, então é interessante que O2 não se torne inativo e inoperante após uma falha.

O objeto O2 adquire um estado inconsistente como decorrência de uma falha, mas O1 ainda apresenta um estado interno válido e correto. Assim O2 pode restabelecer novamente seu estado a partir do estado de O1, da mesma maneira que O2 utilizou quando entrou em operação pela primeira vez, logo após a carga do programa PV.

O meta-objeto (MO) associado a O2 toma conhecimento da falha de O2 através da comparação dos seus resultados com os resultados corretos fornecidos por O1. O MO faz o registro da falha no arquivo de log e envia uma mensagem para O1 salvar novamente o seu estado para um arquivo temporário e, a partir da leitura deste arquivo, O2 consegue restaurar-se e novamente apresentar um estado correto e consistente, podendo retornar à operação normal, ser submetido a novas ativações e continuar sendo validado sob as próximas circunstâncias. O estado incorreto de O2 é substituído pelo estado correto de O1, figura 5.13.

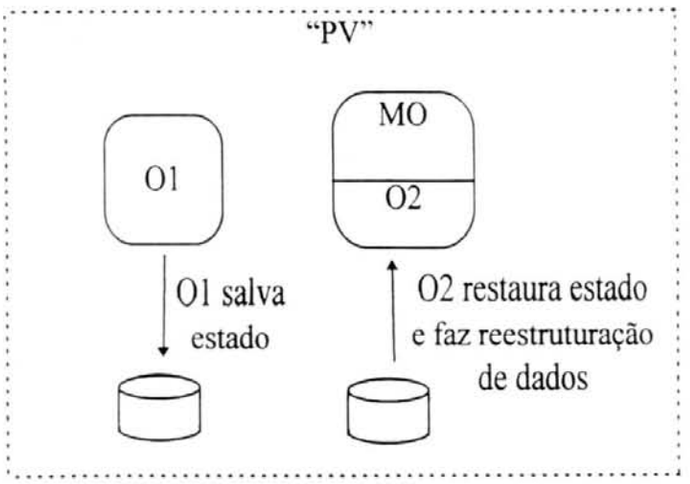

FIGURA 5.13 - Recuperação após uma falha

De fato, a cada ativação de O1 é produzido um ponto de recuperação, que compreende o estado interno de O1 e que está apto a ser utilizado por O2 após uma falha, figura 5.14. Este ponto de recuperação não é transferido para O2 em condições ou circunstâncias normais de operação, apenas nos casos de falha. O processo de salvamento e restauração deve ser realizado num instante em que os objetos não estão executando.

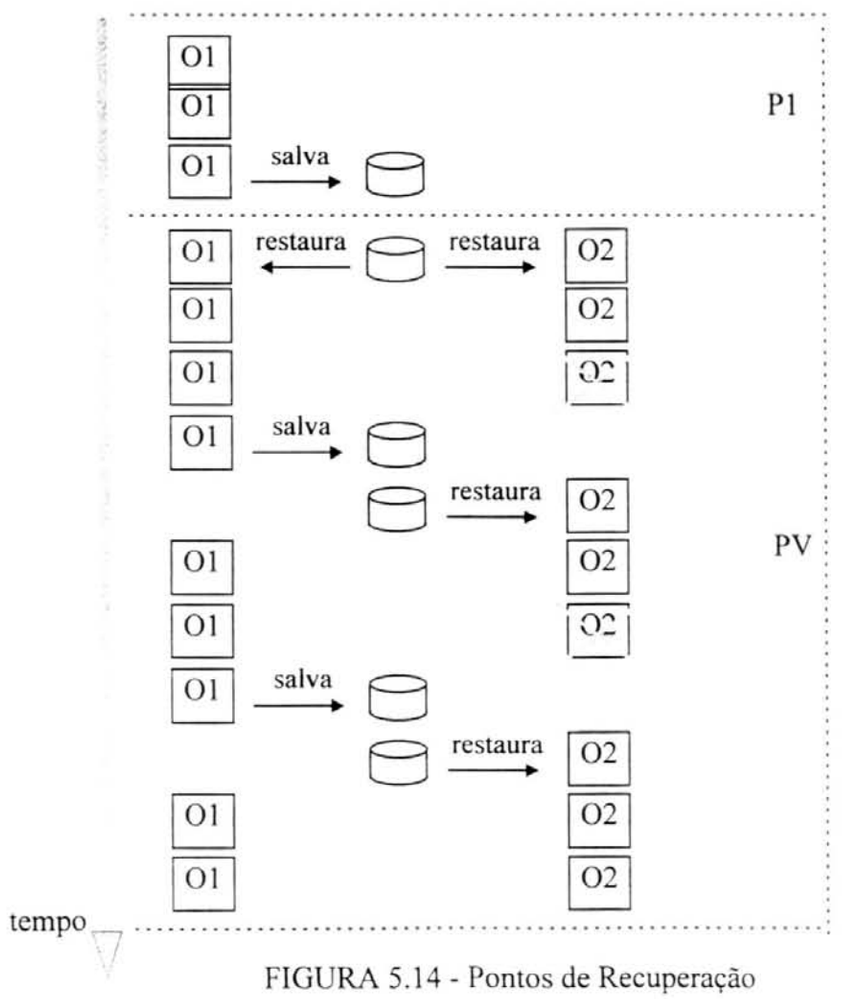

FIGURA 5.14 - Pontos de Recuperação

5.3.6 Pontos de Verificação

Os pontos de verificação são aqueles instantes em que é feita a comparação entre os resultados fornecidos por O1 e por O2. A comparação ocorre a cada ativação destes objetos. Além de fazer uma simples comparação entre os resultados fornecidos por ambos, que é a abordagem mais simples, pode ser feita a comparação do estado interno de ambos através dos mecanismos de introspeção proporcionados pela reflexão computacional. Para permitir esta introspeção, ambos objetos precisam ter um meta-objeto associado.

A comparação dos valores internos pode significar um ponto de verificação importante, mas no caso de as estruturas internas de ambos serem diferentes, esta comparação pode se tornar mais complexa.

O fato de ser mantido um arquivo de log associado ao programa de validação viabiliza que seja armazenado também o estado de O2 por inúmeras vezes 'checkpoints', por exemplo a cada ativação deste objeto. Registrar log do estado do objeto para cada ativação é um caso extremo. Com isso o programador/projetista do objeto poderá, através da análise deste arquivo de log, observar o estado de O2 em cada uma das suas ativações e deter sua atenção principalmente naquelas ativações que produziram erro nas saídas do objeto, e com isso fazer uma depuração bastante detalhada do erro.

5.4 Idempotência

Idempotência é a propriedade através da qual realizar a execução de uma ação por uma única vez tem o mesmo efeito que realizar a sua execução por diversas vezes [MUL93]. Muitas operações sobre arquivos são idempotentes, por exemplo, fazer a leitura de um bloco uma vez tem o mesmo efeito que fazer a leitura do bloco várias vezes. O mesmo vale para gravação. Criação e deleção de arquivos são sempre idempotentes.

Existem operações que não gozam desta propriedade. Por exemplo fazer a operação de incremento sobre um dado em um arquivo por uma vez terá um resultado diferente do que realizar esta operação por diversas vezes. As operações que não são idempotentes podem causar inconsistência de dados se forem executadas por mais de uma vez.

No âmbito deste trabalho a idempotência é importante durante a fase de validação da nova versão, que compreende o período em que ambas as versões, nova e antiga, coexistem e executam simultaneamente. Todo o serviço que é solicitado pelo cliente é executado por ambos objetos servidores O1 e O2 para que o meta-objeto consiga fazer a comparação entre as execuções e funcionalidades de ambos objetos.

Disso decorre que algumas operações serão executadas duas vezes: uma vez pelo objeto O1 e outra vez pelo objeto O2, podendo resultar em inconsistência nos dados caso estas operações não sejam idempotentes. Como solução para estes casos é necessário fazer a criação de arquivos e dispositivos de teste ou 'dummy' para serem acessados pelo objeto O2. Arquivos de teste são arquivos gerados a partir da base de

dados original ou de produção e que contém os mesmos dados existentes no ambiente real.

Com isso o objeto antigo O1 executa suas operações sobre o ambiente e os dados reais, enquanto o objeto novo O2 executa suas operações sobre a base de testes. Assim, sobre os dados do ambiente real é realizada apenas uma operação, feita pelo objeto O1. O objeto O2 não está de fato atuando sobre o ambiente real de produção, mas o resultado e a validade das suas operações pode ser igualmente analisada através da observação do arquivo de teste e com a vantagem de que fica garantida a segurança de que não será causada uma inconsistência na base de dados real.

Depois de concluída a fase de validação o objeto O2 é instalado em definitivo no sistema passando a operar diretamente sobre as bases de dados reais.

5.5 Outras Técnicas

Outras técnicas de TF também são sugeridas, tais como:

- testes de limite de tempo: no caso de o objeto O2 não fornecer sua resposta dentro do tempo esperado ela é descartada e registrado em log. É fornecida ao cliente a resposta do objeto O1.

- recuperação por avanço: tomando o exemplo da substituição da versão de um sensor, cuja etapa de substituição apresente uma falha de maneira tal que a última leitura do sensor esteja perdida. O sistema de tempo-real necessita de um valor para prosseguir com seu processamento. Um tipo de leitura feita por um sensor sobre um meio físico vai registrar as variações físicas (nível de água, temperatura, etc) ocorridas neste meio desde a última leitura realizada. Um meio físico sofre variações graduais. Quando são feitas muitas leituras em curto período, a perda de um valor de uma leitura não implica em grande diferença com relação às leituras anteriores. Neste caso é viável aplicar um algoritmo que leva em conta a média ou ainda a tendência das últimas N leituras realizadas e produz como resultado um valor estimado para o próximo valor. Este resultado pode ser fornecido como resposta ao cliente e ainda estar dentro de uma boa margem de acerto.

- repetição da computação: é aplicável no caso de resultados diferentes serem apresentados por O1 e por O2 e havendo possibilidade da manifestação de falhas transitórias em O2 e havendo tempo suficiente para uma segunda execução de O2.

5.6 Segurança

Autenticação é importante para evitar que a versão de um objeto ou programa seja substituída indevidamente por um programa intruso com vistas a prejudicar o funcionamento do sistema. Os aspectos de segurança de um sistema de substituição 'on-line' envolvem evitar a carga de um código de origem duvidosa quando é feita a substituição de algum programa ou objeto do sistema. O método de segurança deve ser baseado num mecanismo de autenticação.

Quer-se garantir que as versões O1, OV e O2 quando forem substituídas no sistema, sejam de fato os componentes autênticos do sistema. A técnica mais simples

é usar um mecanismo de autenticação que seja externo aos componentes, para que não ocupe espaço interno destes componentes e que faça o controle de autenticação através do uso de senhas. A técnica de senhas não reutilizáveis, obtidas por função hash, oferece maior segurança do que o uso de senha única. Algoritmos como o Message Digest (MD4 ou MD5) [PFL97] são próprios para esta finalidade. Os componentes são autenticados antes de ser realizada a sua carga. Figura 5.15.

FIGURA 5.15 - Gerenciamento de segurança

5.7 Conclusões

Partindo de diversos cenários de sistemas, várias facetas do problema de substituição dinâmica de componentes foram objeto de estudo e registradas neste capítulo.

Com base nesse estudo constatou-se que as técnicas tradicionais do domínio de tolerância a falhas também encontram aplicabilidade no tratamento de problemas de manutenção de software. Mais especificamente, contribuem para validar o processo de substituição de componentes.

A reflexão computacional, aqui usada como técnica de programação, possibilita a introspecção nos objetos que participam da substituição e a separação das atividades de gerenciamento da substituição.

6 Estudo de Caso

Neste capítulo será descrita a implementação e os experimentos realizados. O protótipo de implementação foi adaptado de [BUR96], que consiste de um sistema de drenagem para uma mina. A linguagem de programação utilizada para a implementação foi Java e a ferramenta empregada para reflexão foi MetaJava [GOL97].

6.1 O Cenário

O cenário básico que foi escolhido para desenvolver a aplicação com a qual foram feitos os testes de substituição de versão, comumente aparece na literatura e compreende um sistema de tempo-real que controla o nível de água de um reservatório através de um sensor. O sistema possui as características tipicamente embutidas nos sistemas de tempo-real e pode ser implantado em uma arquitetura distribuída ou centralizada.

No exemplo apresentado em [BUR96] o sistema é usado para bombear água de uma mina. Uma bomba faz a coleta de água de um recipiente no fundo de um poço e a transporta para a superfície.

No exemplo adotado neste trabalho a relação entre o sistema de controle e os dispositivos externos e os requisitos de funcionamento são que o nível de água do reservatório deve ser mantido em um valor baixo. A taxa com que a água entra no poço não é constante. O motor que aciona a bomba tem uma potência variável e que pode ser ajustada de acordo com o nível de água existente no reservatório. O sensor 1, que verifica o nível de água existente no reservatório, possui os seus pontos de detecção em alturas diferentes e assim fornece uma leitura da quantidade de água numa escala com vários níveis de valores (por exemplo 10 níveis).

Quer-se instalar outro sensor no sistema, em substituição ao sensor antigo. O sensor 2 é um sensor mais moderno e preciso, possui outra tecnologia de fabricação e apresenta seus valores em uma escala de leituras mais ampla. A tecnologia usada pelo sensor 2 para determinar a quantidade de água é baseada na determinação da pressão exercida pela coluna d'água. Quer-se substituir o sensor 1 pelo sensor 2, bem como o software associado a ambos, sem descontinuar o sistema . Figura 6.1.

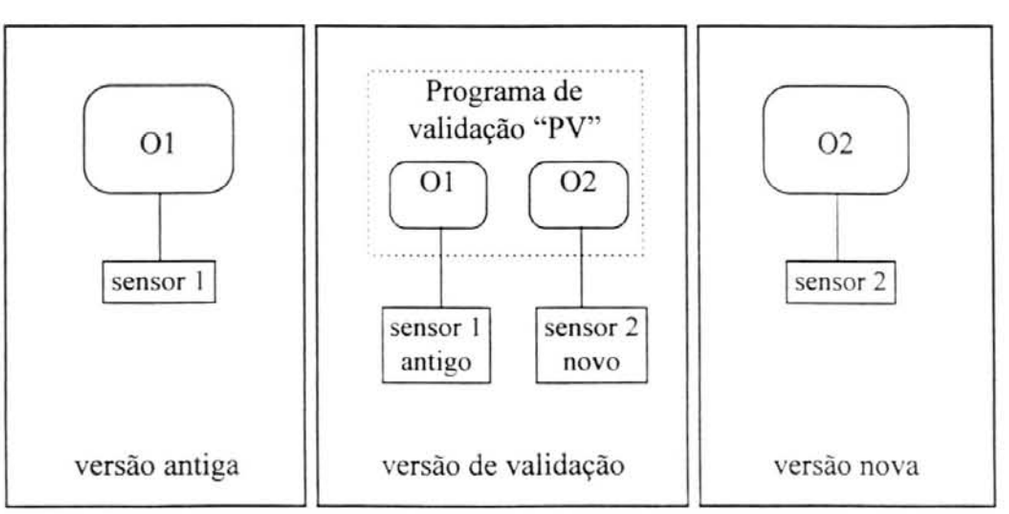

FIGURA 6.1 - Cenário de Transição

Cada sensor físico está associado a um componente de software. O objeto O1 está associado ao sensor 1 e o objeto O2 está associado ao sensor 2. O objeto comanda a leitura dos dados, aplica um tratamento computacional sobre os valores fornecidos pelo sensor e os passa ao cliente. É suposto que o objeto O1 não apresenta falha por se tratar de um objeto que já está em operação no sistema há muito tempo, demonstrando que está amplamente testado e aprovado. Assim os resultados fornecidos por O1 são confiáveis e podem ser usados como parâmetro de comparação com os resultados fornecidos por O2 e assim validar o funcionamento do objeto O2.

Inicialmente o sistema opera apenas com o sensor 1 e o objeto O1. Então é instalada uma versão de transição OV (Objeto de Validação ou Programa de Validação) que manipula ambos sensores e ambos objetos e faz a validação de O2. Depois de realizada a validação a versão OV é substituída em definitivo pela versão O2.

Aqui será inicialmente descrito o funcionamento básico do objeto O1, figura 6.2. Os valores fornecidos pelo sensor 1 passam pelo seguinte tratamento realizado pelo objeto O1:

- o objeto O1 lê um valor inteiro vindo do sensor que corresponde ao nível de água existente no reservatório
- o objeto O1 acrescenta a hora em que a leitura foi realizada
- as últimas N leituras são armazenadas em uma estrutura de dados privada do objeto O1. Esta estrutura de dados irá caracterizar o estado interno de O1
- em seguida é calculada a média dos últimos N valores
- por fim os dados são enviados ao cliente: o valor do nível fornecido pelo sensor, a hora em que foi obtida esta leitura, e a média das últimas N leituras

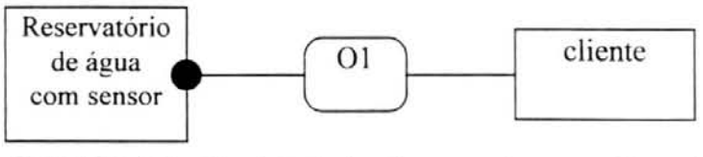

FIGURA 6.2 - Cenário da implementação com objeto O1

O programa cliente é um programa de tempo-real que não pode ser descontinuado. O objeto O1 possui um estado interno que compreende os últimos valores armazenados e é provido de um método que salva o seu estado para um arquivo. O método *salvaEstado()* é chamado quando o objeto O1 termina a sua execução.

```
public void salvaEstado()
  { try
     { f1 = new FileOutputStream ("O1estado.txt");
       o1 = new PrintWriter(f1,true);
       for (int i=0;i<10;i++) o1.println(vetLeituras[i]);
     }
     catch {...}
  }
```

O objeto de validação OV é a versão de transição entre O1 e O2. Ao ser iniciado este objeto, é executada primeiramente a carga do estado interno de O1 e assim o objeto O1 passa a funcionar novamente da mesma forma que vinha fazendo antes da substituição. Em seguida é atribuído o estado inicial para O2, a partir do mesmo arquivo que foi usado para iniciar O1. Neste momento ambos objetos apresentam um estado interno equivalente e podem realizar suas operações também de forma equivalente. Toda vez que é invocado um método de O2, é feito um desvio e ativado o método equivalente em O1. O objeto OV através da reflexão computacional começa a fazer análises e realizar comparações sobre os valores fornecidos por ambos objetos com intenção de detectar alguma falha no novo componente. A execução de O1 e de O2 é feita seqüencialmente.

Os objetos O1 e O2 também possuem estruturas de dados diferentes. Em O1 o armazenamento dos valores lidos do sensor 1 é implementado com um vetor de valores inteiros com capacidade para armazenar e calcular a média entre 10 valores. Em O2 a capacidade do vetor é de 100 valores inteiros. Também o cálculo da média é diferente em O2, onde é feita a média dos últimos 100 valores.

O sensor 1 é um componente de hardware antigo com a limitação de que sua escala de leituras fornece os valores múltiplos de 10: (10, 20, 30, ..., 90 e 100) para o nível de água. O sensor 2 é mais acurado e sua escala fornece valores múltiplos de 1: (1,2,3, ..., 11, 12, ..., 98, 99, 100) para o nível de água. Figura 6.3.

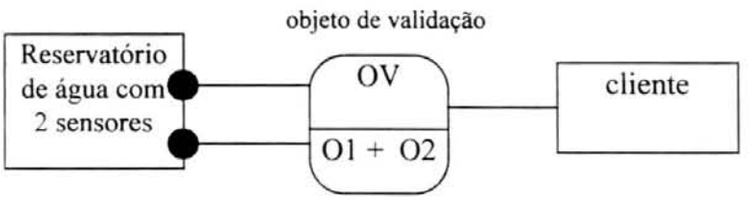

FIGURA 6.3 - Cenário da implementação com objeto OV

O software do sistema de tempo-real já é adaptado para operar com a escala mais ampla de valores, mas o sistema apresentava uma limitação devido à presença do sensor 1 antigo, disso resulta que o sistema não proporciona bom desempenho devido aos valores pouco precisos fornecidos pelo sensor 1.

Assim a modificação que se faz necessária no sistema é a substituição do sensor 1, mais limitado, pelo sensor 2, mais preciso. Também o cálculo da média dos últimos valores lidos passará a ser mais acurada. Para isso é indicada uma substituição do objeto servidor O1 que faz acesso ao sensor do reservatório e faz os cálculos, enviando o valor lido e a média ao programa cliente de tempo-real, pelo objeto O2 que realiza as mesmas funcionalidades, porém com maior exatidão.

Esta substituição precisa ser realizada sem descontinuar o sistema como um todo e sem interromper o processamento de tempo-real, uma vez que o reservatório de água precisa ser monitorado constantemente. Além disso é difícil que sejam feitos testes com o novo sensor 2 para analisar o seu funcionamento dentro do sistema, pois não há possibilidade de interromper o sistema para esta finalidade.

Para este caso torna-se indicada uma substituição dinâmica 'on-line' com tolerância a falhas. A substituição é feita apenas no objeto servidor, e num curto instante de tempo em que o mesmo está ocioso. O objeto servidor também pode ser interrompido após uma execução completa de um serviço. As mensagens destinadas ao servidor são enfileiradas em um objeto de caixa postal para o seu processamento imediatamente posterior à instalação da nova versão.

A substituição é tolerante a falhas porque qualquer anormalidade na leitura dos valores fornecidos por O2 é detectada pelo software de validação e imediatamente é restabelecido o valor antigo, fornecido por O1, que é então repassado ao cliente. A média fornecida ao cliente também é a média calculada por O1 neste caso.

Com esta técnica de substituição tolerante a falhas, o objeto de validação OV irá realizar os testes com o novo sensor 2 e com o software O2, sem comprometer o funcionamento do sistema de tempo-real. Caso não fosse empregada uma substituição tolerante a falhas os riscos de falha seriam bem maiores comprometendo todo o sistema e podendo gerar prejuízos.

6.2 Testes com a Ferramenta

Como ferramenta para embutir reflexão computacional à implementação foram feitos experimentos com MetaJava. Esta ferramenta é um protótipo desenvolvido por [GOL97] e possui as classes para efetuar a reflexão comportamental na linguagem Java. A classe *MetaTraceMethCall* é usada para interceptar todas as invocações de métodos a um objeto.

Uma instância da classe *MetaTrace* precisa ser associada ao objeto através da passagem dos parâmetros *obj* e *methods*, onde *obj* é o objeto-base e *methods* é um 'array' contendo nomes de métodos do objeto-base. Todas as invocações destes métodos passam a ser interceptadas pelo meta-objeto. A seguir é listado um exemplo da codificação em MetaJava mostrando a interceptação de chamada dos métodos *recebeDado()* e *calculaMedia()*.

```
public class MO
 { public static void main(String argv[])
    { O2a objO2 = new O2a();
      String methods [] = new String[2];
      methods[0] = "recebeDado()V";
      methods[1] = "calculaMedia()V";
      new MetaTraceMethCall(objO2,methods);
      objO2.recebeDado();
      objO2.calculaMedia();
  } }
```

MO é a classe que faz a validação da substituição através da comparação dos resultados fornecidos pelos métodos e pela introspecção das variáveis do objeto O2.

Algumas deficiências ou limitações foram constatadas na ferramenta MetaJava dentre elas pode ser citada a parte de comunicação por 'sockets' e a gravação de arquivos em disco que não encontra suporte na ferramenta.

Foram feitos testes com os segmentos dos programas O1, OV e O2 em MetaJava, porém sem fazer acesso aos 'sockets' e nem fazer a gravação em arquivo do estado dos objetos nem gravar o arquivo de log. Mas a ferramenta permite realizar e testar as operações de reflexão computacional.

Os programas O1, OV e O2 foram efetivamente desenvolvidos em linguagem Java padrão, que não possui as bibliotecas ou ferramentas para reflexão comportamental, mas que realiza o acesso aos 'sockets' e faz a gravação de dados em arquivos. Assim para realizar a reflexão foram simulados os desvios para os métodos reflexivos através de chamadas simples a estes métodos.

```
class O1a extends Frame
  { ... }
class O2a extends Frame
  { ... }
class MO extends Frame
{ O1a objO1 = new O1a();
  O2a objO2 = new O2a();
public int recebeDado()
  { valorLido1 = objO1.recebeDado(R12);
    valorLido2 = objO2.recebeDado(R12);
    ... }
public float calculaMedia()
  { media1=objO1.calculaMedia();
    media2=objO2.calculaMedia();
```

Quando é invocado o método *recebeDado()* do objeto O2, é feito um desvio e é invocado o método *recebeDado()* do objeto-base O1 e após é executada a nova versão *recebeDado()* do objeto-base O2.

6.3 A Comunicação

A aplicação é do tipo duas camadas 'two tiers'. O cliente e o servidor comunicam-se através de 'sockets', usando as classes Socket e ServerSocket importadas da biblioteca java.net. 'Sockets' fornecem uma interface de baixo nível usada para enviar fluxos de bytes através da rede. O 'socket' faz uma ligação virtual entre o servidor e o cliente. O servidor ao ser executado instancia um objeto 'socket' que fica em 'loop' aguardando alguma mensagem do cliente.

```
class ConexBasica extends Thread
  int pt; ServerSocket ssocket; Conexsec csec;
  public ConexBasica(int port) {pt = port;}
  public void run()
   { try {ssocket = new ServerSocket(pt);} catch {...}
     while (true)
     { try
        { Socket clisocket = ssocket.accept();
          csec = new Conexsec(clisocket);
          csec.start();
        }
       catch {...}
     }   }
```

```
class Conexsec extends Thread
  Socket csk; PrintStream soutput; DataInputStream sinput;
  public Conexsec(Socket s) {csk = s;}
  public void run()
  { try
     { soutput = new PrintStream(csk.getOutputStream());
       sinput  = new DataInputStream(csk.getInputStream());
       while(true)
        { String str = sinput.readLine();
          // faz acesso a O1
          // responde ao cliente
          soutput.println("...");
        }
     }
    catch {...}
```

Ao ser efetuada uma substituição do objeto servidor antigo pelo novo, a conexão entre o cliente e o objeto servidor antigo é temporariamente desfeita e em seguida restabelecida pelo objeto que executa a nova versão. A troca de versão de O1 para OV e de OV para O2 é feita através de comandos dados por um programador ou operador. O programa cliente não é interrompido durante este processo de substituição.

6.4 Tolerância a Falhas

Quando é constatada uma divergência entre o valor fornecido por O1 e o valor fornecido por O2, o meta-objeto registra ambos os resultados no arquivo de log e passa ao cliente o resultado de O1. Os resultados obtidos também são exibidos na tela apresentada pelo programa servidor.

```
class O1a extends Frame
  { ... }
class O2a extends Frame
  { ... }
class MO extends Frame
{ O1a objO1 = new O1a();
  O2a objO2 = new O2a();
public int recebeDado()
  { valorLido1 = objO1.recebeDado(R12);
    valorLido2 = objO2.recebeDado(R12);
    if (Divergencia)
       { registraNoLog();
         objO1.salvaEstado();
         objO2.restauraEstado();
         valorLido12=valorLido1; }
    else valorLido12=valorLido2;
    return valorLido12; }
public float calculaMedia()
  { media1=objO1.calculaMedia();
    media2=objO2.calculaMedia();
    if (Divergencia)
       { registraNoLog();
         media12=media1; }
```

Os mecanismos de injeção de falhas são muito usados na validação de sistemas. A injeção de falhas é uma técnica usada para acelerar a ocorrência de falhas num sistema durante a sua verificação dinâmica, ou seja, durante o seu tempo de execução. Consiste em introduzir falhas de forma controlada visando auxiliando no processo de tratamento de erros num sistema tolerante a falhas.

Neste trabalho foi usada a injeção de falhas por software, que consiste em introduzir erros através de alterações no conteúdo de registros e variáveis e fazer alterações na seqüência de instruções do software com o objetivo de emular a conseqüência de falhas físicas ou de projeto no sistema.

Quando é feita a validação de um software, a injeção de falhas deve ser analisada caso a caso, por ser muito dependente da aplicação e do tipo do software. Para testar a implementação foram simuladas as seguintes situações de falha: (a) falha na leitura fornecida pelo sensor2; (b) falha no algoritmo que realiza o cálculo da média das leituras.

(a) Um ponto para injeção de falhas foi adotado na classe Reservatório que fornece os valores lidos pelos sensores. A chamada de uma função que gera um número randômico introduz um valor incorreto para a leitura do sensor feita por O2 em momentos aleatórios.

```
class Reservatorio12
{ int nivel=50;
  public Reservatorio12()    {  }
  private void leNivel ()
   { int temp = (int)(Math.random()*20);
     if (temp > 15)    nivel+=10;  // fazem o nivel variar
     if (temp < 4)     nivel-=10;
     if (nivel > 100) nivel=100;
     if (nivel < 0)    nivel=0;    }
  public int leSensor1 ()  // fornece valores multiplos de 10
   { leNivel();
     return nivel;    }
  public int leSensor2 ()  // fornece valores multiplos de 1
   { int temp = nivel;
     temp = temp-4 + (int)(Math.random()*10);
     if (temp > 100) temp=100;
     if (temp < 0)   temp=0;
     int probabilidadeDeFalha=15, // indica uma chance em 15 de ser
         amplitudeMaxDaFalha=80;  //                injetada a falha
     int injetaFalha=(int)(Math.random()*probabilidadeDeFalha);
     if (injetaFalha==1)
        temp = (temp-amplitudeMaxDaFalha/2+1
               + (int)(Math.random()*amplitudeMaxDaFalha));
```

(b) Outro ponto de injeção de falhas foi introduzido na função que calcula a média dos valores armazenados no vetor privado do objeto O2. Esta falha equivale a um erro de projeto de programa introduzido na programação da fórmula de cálculo desta média para o objeto O2.

Os valores fornecidos por O1 e por O2 são comparados pelo meta-objeto e os erros são detectados forçando a gravação da divergência entre O1 e O2 no arquivo de log e fazendo a gravação do estado interno de O2 no mesmo arquivo. Em seguida é invocado o método que restaura o estado de O2 a partir do estado de O1. Fazendo a posterior análise do arquivo de log, o programador pode fazer constatações sobre falhas existentes no objeto O2.

O meta-objeto tem acesso ao conteúdo das variáveis do objeto O2 através do mecanismo de introspecção, que é implementado em MetaJava através da classe *MetaTraceFieldAccess*.

```
public class MO
 { public static void main(String argv[])
    { O2a objO2 = new O2a();
      String fields [] = new String[2];
      fields[0] = "valorLido(I)";
      fields[1] = "ind(I)";
      new MetaTraceFieldAccess (objO2,fields);
      objO2.calculaMedia();
  } }
```

6.5 Conclusões

Os testes realizados mostraram a viabilidade da solução proposta. O objeto servidor teve sua versão substituída sem descontinuar o cliente. Foi realizada a substituição 'on-line' da versão do objeto servidor, utilizando reflexão computacional para introduzir técnicas de tolerância a falhas. Esta é a contribuição mais importante deste trabalho. As divergências entre os resultados fornecidos por O1 e O2 foram registrados no arquivo de log e o cliente sempre recebeu a resposta correta fornecida por O1, mesmo na presença de falhas em O2. Assim o objeto O2 foi testado e validado sem propagar erro para o sistema como um todo.

Através da reflexão computacional é possível separar os algoritmos de validação da substituição, dos algoritmos da aplicação em si. A ferramenta utilizada para incorporar reflexão computacional apresentou limitações que inviabilizaram a realização de todas as operações requeridas pelo sistema. As operações de interceptação de mensagens foram simuladas na linguagem Java padrão, não invalidando o conteúdo da idéia proposta.

Uma limitação da solução é não resolver o problema da substituição da versão do programa principal que é o gerenciador das tarefas de tempo-real. Esta solução permite apenas trocar a versão dos objetos servidores, sem interromper a aplicação principal. Mas este problema também existe nos modelos procedurais que também não permitem a substituição da função principal 'main()' do programa.

7 Conclusões

O objetivo da dissertação foi de estudar e propor uma estratégia para substituição dinâmica de versões de componentes de software tolerante a falhas orientado a objetos. Este objetivo foi atingido e foi demonstrada a viabilidade de construção de um sistema com esta finalidade. O trabalho apresenta uma solução genérica para o problema da substituição sendo aplicável, em sua essência, a sistemas centralizados, distribuídos, batch ou tempo-real.

O problema apresentado é relevante visto que a manutenção de software é uma realidade e causa interrupções inconvenientes aos usuários dos sistemas. A solução proposta é inovadora no sentido de ser voltada para o paradigma da orientação a objetos, utiliza técnicas de tolerância a falhas e adota uma arquitetura reflexiva. O uso de meta-objetos facilita a programação das rotinas de substituição e validação. Os objetos substituídos podem ser monitorados por um período de tempo, antes de entrar em operação em definitivo dispensando o uso de ambientes de simulação de software. As técnicas de tolerância a falhas são usadas para validar a substituição, fazendo o monitoramento de ambas as versões através da comparação dos resultados por elas fornecidos.

A solução proposta também permite suporte à realização de testes de componentes de software. Em muitos sistemas é difícil fazer simples testes com novas versões de módulos de software executando nos sistemas reais de produção para os quais eles foram desenvolvidos, porque o ambiente de produção não pode ser comprometido e colocado em risco de falha. Geralmente os testes são feitos em ambientes de simulação, que são caros e muitas vezes exigem hardware específico. Com o sistema proposto, e adotando a mesma estratégia, uma nova versão de um objeto pode ser testada em seu ambiente real de execução, tendo o seu comportamento monitorado pelo meta-objeto. Em caso de ocorrer alguma falha no objeto novo sob teste, esta falha será interceptada pelo meta-objeto, que fará o tratamento apropriado, liberando ao cliente o resultado fornecido pela versão antiga. Com isso fica dispensado o emprego de um ambiente especialmente desenvolvido para realização de testes.

O modelo proposto neste trabalho também estende sua aplicabilidade e versatilidade a sistemas administrativos sem característica de tempo-real (programas batch). Estes sistemas também necessitam sofrer modificações e quer-se evitar que uma falha na nova versão comprometa a correção dos resultados fornecidos pelo sistema. Ainda que a aplicação administrativa seja executada com intervalos longos, a solução proposta pode ser empregada para dar suporte ao período de validação da nova versão.

A idéia principal deste trabalho reside de fato no programa de validação "PV", que gerencia ambas as versões (antiga e nova) e utiliza reflexão computacional associada a técnicas de TF para detectar e confinar erros na nova versão do componente de software. Os sistemas de substituição dinâmica estudados não implementam esta etapa de validação da nova versão, mas simplesmente adotam a hipótese de que a nova versão é correta e que executará corretamente as suas funções depois de substituída no sistema. Nenhuma das abordagens pesquisadas apresenta

execução conjunta de ambas versões (a antiga e a nova) durante um período de validação, todas elas descartam a versão antiga imediatamente após a substituição.

O emprego da reflexão computacional permite separar o código funcional e não-funcional, o que torna mais fácil compreender, manter e depurar o programa em si, permitindo que o programador se dedique mais ao domínio da aplicação. A separação do algoritmo reflexivo do algoritmo base torna possível o reúso das políticas desenvolvidas para fazer a validação da substituição.

Como projetos futuros é proposta a construção de uma biblioteca de classes ou 'framework' contendo meta-componentes que visam suprir as funcionalidades adicionais a esta técnica e fornecer suporte, através de estruturas e ações básicas, para desenvolver outros sistemas de substituição dinâmica de software. As meta-classes desenvolvidas podem ser reusadas por outros sistemas através do mecanismo de herança.

Esta solução de substituição dinâmica de versões de software tolerante a falhas pode se tornar um pouco pesada, por isso ela está restrita para ser utilizada em um tipo de sistema de tempo-real em que as restrições temporais não sejam rígidas. Contudo com o crescente aumento do poder de processamento é uma questão de aguardar o surgimento de máquinas mais potentes para estender esta solução para sistemas de tempo-real cada vez mais velozes.

Bibliografia

[ANA97] ANAXÁGORAS, M. Girão; CASTRO, H. de Souza, T.; LEE, P. A. **Fault Tolerance**: Principles and Pratice. Englewood Cliffs: Prentice-Hall, 1997.

[AND81] ANDERSON, T.; LEE, P. A. **Fault Tolerance**: Principles and Pratice. Englewood Cliffs: Prentice-Hall, 1981.

[BIR91a] BIRMAN, Kenneth; COOPER, Robert, The ISIS Project: Real experience with a fault tolerant programming system. **Operating Systems Review**, New York, v. 25, n. 2, p.103-107, Apr 1991.

[BIR91b] BIRMAN, Kenneth; SCHIPER, André; STEPHENSON, Pat. Lightweight Causal and Atomic Group Multicast. **ACM Transactions on Computer Systems**, New York, v. 9, n. 3, p.272-314, Aug 1991.

[BIR93] BIRMAN, P. Kenneth, The Process Group Approach to Reliable Distributed Computing. **Communications of The ACM**, New York, v. 36 n. 12, p.37-53, Dec 1993.

[BIR9?] BIRMAN, P. Kenneth, **Building Secure and Reliable Network Applications**. Ainda não publicado, 199?

[BOO94] BOOCH, G., **Object-Oriented Analysis and Design With Applications**. 2. ed. Redwood City-California: The Benjamin/Cummings, 1994.

[BOR95a] BORLAND INTERNATIONAL. **Borland C++:** Programmer's Guide: version 4.5. Scotts Valley: Borland International, 1995.

[BOR95b] BORLAND INTERNATIONAL. **Borland C++:** Reference Manual: version 4.5. Scotts Valley: Borland International, 1995.

[BUR96] BURNS, Alan; WELLINGS Andy, **Real-Time Systems and Programming Languages**. 2. ed. Edinburgh Gate-Harlow: Addison Wesley, 1996.

[BUZ95] BUZATO, L. E. Tolerância a Falhas, Orientação a Objetos e Ações Atômicas. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6, 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p. 33-48.

[CAV94] CAVALHEIRO, Gerson Geraldo Homrich. **Um Modelo para Linguagens Orientadas a Objetos Distribuído.** Porto Alegre: CPGCC da UFRGS, 1994. Dissertação de Mestrado.

[CHI93] CHIBA, Shigeru. **OpenC++ Programmer's Guide.** Tokyo: Dept. of Information Science, University of Tokyo, 1993, 21p. (Technical Report n. 93-3).

[CHI95] CHIBA, Shigeru. A Metaobject Protocol for C++. **ACM SIGPLAN Notices**. New York, v.30, n.10, p. 285-299, Oct. 1995. Trabalho apresentado na Conference on Object-Oriented Programming Systems, Languages and Applications, 10., 1995, Austin, US.

[ELL91] ELLIS, C.A.; GIBBS, S. J.; REIN, G. L. Groupware some issues and experiences. **Communications of The ACM**, New York, v. 34, n. 1, Jan 1991.

[FER95] FERNANDES, Aguinaldo Aragon. **Gerência de Software Através de Métricas.** São Paulo: Atlas, 1995.

[FOS95] FOSTER, Ian T. **Designing and Building Parallel Programs.** [S.l.]: Addison-Wesley, 1995.

[FRA96a] FRAGA, Joni et al. Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 23., 1996, Recife-PE. **Anais...** 1996. p. 249-260.

[FRA96b] FRAGA, Joni et al. Implementação de Serviços Replicados em um Ambiente Aberto Usando uma Abordagem Reflexiva. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 16., 1996. **Anais...** [S.l.:s.n.], 1996. p. 261-272.

[FRA97] FRANZ, Michael. Dynamic Linking os Software Components. **Computer**, New York, p. 74-81, Mar 1997.

[GHE91] GHEZZI, C.; JAZAYERI, M.; MANDRIOLI, D. **Fundamentals of Software Engineering.** New Jersey: Prentice-Hall, 1991.

[GOL97] GOLM, Michael. **Design and Implementation of a Meta Architecture for Java.** Nürnberg: Institut für Mathematische Maschinen und Datenverarbeitung der Friedrich-Alexander-Universität, 1997.

[GRE75] GREIBACH, A. Sheila. **Theory of Program Structure: Schemes, Semanties, Verification.** Heidelberg: Springer-Verlag, 1975. (Lecture Notes in Computer Science)

[GUP93] GUPTA, Deepak; JALOTE, Pankaj. Dynamic Software Version Change Using State Transfer Between Processes. **Software Practice and Experience**, [S.l.], v. 23, p. 949-964, Sept 1993.

[GUP96] GUPTA, Deepak; JALOTE, Pankaj; BARUA, Gautam. A Formal Framework for Dynamic Software Version Change. **IEEE Transactions on Software Engineering**, New York, v. 22, n. 2, p. 120-131, Feb 1996.

[HAE96] HAETINGER, Werner. Um Exemplo de Técnica de Tolerância a Falhas Implementada com Grupos de Objetos. In SIMPÓSIO REGIONAL DE TOLERÂNCIA A FALHAS, 1., 1996, Porto Alegre. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1996. p.115-122.

[JAL94] JALOTE, Pankaj. **Fault-Tolerance in Distributed Systems.** New Jersey: Prentice-Hall, 1994.

[LAD96] LADD, Robert Scott. **C++: Components and Algorithms.** 3. ed. New York: M&T, 1996.

[LIS94] LISBÔA, Maria L. Blanck; AZEREDO, P.A. Paradigmas de programação: o enfoque de tolerância a falhas. In: JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA, 13., 1994, Caxambú-MG. [S.l.:s.n.], 1994.

[LIS95a] LISBÔA, Maria L. Blanck; CAVALHEIRO, Gerson, G. Homrich. Reflexão Computacional sobre Técnicas de Tolerância a Falhas em Software. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.405-416.

[LIS95b] LISBÔA, Maria L. Blanck; VENDRUSCOLO, G. Controle de Sensores Através de Meta-objetos com Comportamento Inteligente. In: SIMPÓSIO BRASILEIRO DE AUTOMAÇÃO INTELIGENTE, 2., 1995, Curitiba, PR. **Anais...** Curitiba: CEFET, 1995. p. 219-224.

[LIS95c] LISBÔA, Maria L. Blanck. **MOTF: Meta-objetos para Tolerância a Falhas.** Porto Alegre: CPGCC da UFRGS, 1995. Tese de Doutorado.

[LIS97a] LISBÔA, M.L. Reflexão Computacional no Modelo de Objetos. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 2., 1997, Campinas-SP. **Anais...** Campinas:[s.n], 1997. p. 5.

[LIS97b] LISBÔA, M.L.; HAETINGER, W. Troca Dinâmica de Componentes: problemas e soluções no modelo OO. In: ARGENTINE SIMPOSIUM ON OBJECT-ORIENTATION, 1997, Buenos Aires. **Anales...** Buenos Aires:[s.n.], 1997. p. 67-75.

[MAC96] MACEDO, Marcos F. F. **Um Framework Orientado a Objetos para Construção de Compiladores para Linguagens de Programação com Tipagem Estática Orientadas a Objetos**. Porto Alegre: CPGCC da UFRGS, 1996.

[MAE87] MAES, P. Concepts and Experiments in Computational Reflection. **ACM SIGPLAN Notices**, New York, v. 22, n. 12, p.147-155, Dec. 1987. Trabalho apresentado na Conference on Object-Oriented Programming Systems, Languages and Applications, 1987.

[MAN93] MANFREDINI, Ricardo Augusto. **Um Estudo Sobre Objetos Inteligentes:** Trabalho Individual. Porto Alegre: CPGCC da UFRGS, 1993. (TI-322).

[MUL90] MULLENDER, S. J. et al. Amoeba - A Distributed Operation System for the 1990's. **Computers**, Los Alamitos, v. 23, n.5, p. 44-53, May 1990.

[MUL93] MULLENDER, Sape J. **Distributed Systems**. 2. ed. New York: Addison-Wesley, 1993.

[OLI97] OLIVA, Alexandre. dissertação de mestrado em fase final de implantação ainda não publicada. Disponível por http em URL:http//www.dcc.unicamp.br/~oliva, (set. 1997).

[PFL97] PFLEEGER, Charles P. **Security in Computing.** New Jersey: Prentice-Hall, 1997.

[PRE88] PRESSMAN, Roger S. **Making Software Engineering Happen.** Englewood Cliffs: Prentice-Hall, 1988.

[RAN78] RANDELL, B.; LEE, P. A.; TRELEAVEN, P. C. Reliability Issues in Computing System Design. **ACM Computing Surveys**, New York, v.10, n. 2, p. 123-166, June 1978.

[RIP93] RIPPS, David, L. **Guia de Implementação para Programação em Tempo Real.** Rio de Janeiro: Campus, 1993.

[SEG93] SEGAL, M., Mark E., Frieder O. On-the-fly Program Modification: Systems for Dynamic Updating. **IEEE Software**, New York, v. 10, n. 3, p. 53-65, Mar 1993.

[SHR95] SHRIVASTAVA, Santosh, K. **Lessons Learned from Building and Using the Arjuna Distributed Programming System.** [S.l.:s.n.], 1995. (Lecture Notes in Computer Science, n. 938).

[SIL97] SILVA, R. et al. Reflexão Computacional em Linguagens de Programação: Um estudo Comparativo. SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 2., 1997, Campinas-SP. **Anais...** Campinas, [s.n.], 1997. p. 245-266.

[SOM97] SOMANI, Arun, K. Understanding Fault Tolerance and Reliability. **Computer**, New York, v. 30, n. 4, p. 45-50, Apr 1997.

[STR95] STROUD, R. J.; WU, Z. **Using Metaobject Protocols to Implement Atomic Data Types.** In: ENCOOP, EUROPEAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, 9., 1995, Aarhus, Denmark. **Proceedings...** [S.l.:s.n.], 1995. p. 168-189. (Lecture Notes in Computer Science, n. 952).

[VEN97] VENDRUSCOLO, Geane. M. **Um Estudo sobre Reflexão Computacional por meio de Meta-objetos com Comportamento Inteligente**. Porto Alegre: CPGCC da UFRGS, 1997. Dissertação de Mestrado.

[VER96] VERÍSSIMO, Paulo; RODRIGUES, Luís; CASIMIRO, António. **CesiumSpray: a Precise and Accurate Global Clock Service for Large-scale Systems.** Lisboa: Techical Univ. of Lisboa, 1996.

[WEB96] WEBER, T.; PÔRTO, I.; WEBER, R. **Tolerância a Falhas**. Porto Alegre: CPGCC da UFGRS, 1996. Apostila usada na disciplina de Fundamentos de Tolerância a Falhas.

[YAN96] YANG, Zhonghua; DUTTY, Keith. CORBA: A Platform for Distributed Object Computing: (A State-of-the-Art Report on OMG/CORBA). **Operating Systems Review**, [S.l.], v. 30, n. 2, p. 4-31, Apr 1996.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Troca Dinâmica de Versões de Componentes de Programas no Modelo de Objetos

por

Werner Haetinger

Dissertação apresentada aos Senhores:

Prof. Dr. Olinto José Varela Furtado (UFSC)

Prof. Dr. Cláudio Fernando Resin Geyer

Prof. Dr. Paulo Alberto de Azeredo

Vista e permitida a impressão.
Porto Alegre, 28/07/98.

Profa. Dra. Maria Lúcia Blanck Lisbôa,
Orientadora.