Configuração Dinâmica como Ferramenta para Construção de Sistemas Distribuídos Flexíveis e Robustos

George Roger Ribeiro Justo
Paulo Roberto Freire Cunha
Departamento de Informática
Universidade Federal de Pernambuco
Recife, PE

Resumo

A programação baseada em configuração dinâmica permite a construção de programas eficientes e bem estruturados. Neste trabalho, o enfoque é a aplicação da configuração dinâmica de processos para aumentar a flexibilidade e a robustez dos sistemas distribuídos. Nós apresentamos um modelo de programação baseado em modularidade, objetos e configuração dinâmica e uma metodologia de programação que permite construir sistemas distribuídos bem estruturados, flexíveis e robustos.

1 Introdução

A necessidade de utilizar sistemas que atendam às novas demandas de aplicações dos usuários tem estimulado os projetistas a desenvolverem modelos de máquinas alternativos. Um destes modelos é a máquina distribuída cuja principal característica é a descentralização do processamento. Esta propriedade, além de refletir o comportamente de muitas aplicações, aumenta o desempenho de processamento.

Os projetistas podem fazer várias escolhas quando programando tais sistemas. Um método é estender as linguagens de programação seqüencial com primitivas para comunicação e paralelismo. Modelos de linguagens lógicas e funcionais, também, permitem a execução distribuída, pois o paralelismo pode ser invisível para o programador, sendo colocado implicitamente na implementação. Outras linguagens oferecem meios explícitos para o particionamento dos programas combinados com os mecanismos para comunicação e sincronização.

Uma preocupação importante é que os sistemas distribuídos potencialmente oferecem um ambiente flexível para modificações e extensões [1], onde computadores podem ser adicionados quando necessário. Além disso, nos sistemas distibuídos uma falha pode afetar

um dos elementos de processamento sem contudo parar todo o sistema. 702
Neste sentido, uma flexibilidade de configuração similar é necessária para os componentes de software do sistema.

A idéia de estruturação multi-processos [2] define um programa como um conjunto de processos e suas interações, particularmente aquelas via passagem de mensagens. Esta estruturação impõe uma ordenação das funções do programa associando-as a processos que as executam, permitindo que a estrutura possa mudar dinamicamante, durante a execução, pela criação e destruição dos processos. Este tipo de estruturação, à princípio, pode ser aplicado na programação distribuída a fim de oferecer a dinâmica necessária para que os programas se adaptem a mudanças de configuração. Contudo, devemos considerar dois aspectos: o primeiro é que se deve ter um cuidado a fim de que a estrutura do sistema possa ser claramente identificada, pois a complexidade decorrente da criação e destruição de processos pode tornar os programas difíceis de gerenciar tanto no desenvolvimento quanto na verificação. O segundo aspecto é a decomposição do sistema, pois o critério baseado na criação e destruiçao de processos pode dificultar a abstração ou especificação, pois esta propriedade nem sempre faz parte da realidade sendo modelada, estando relacionada principalmente com a eficiência (operacionalidade).

Neste trabalho, apresentamos um modelo para programação distribuída que utiliza a estruturação dinâmica, contudo apresenta solução tanto para o problema de decomposição quanto para a falta de clareza. Discutiremos os princípios que caracterizam o modelo e apresentaremos uma metodologia para desenvolvimento de programas baseados no modelo. Finalmente, é feito um estudo de caso a fim de avaliar a aplicabilidade tanto do modelo quanto da metodologia.

1. Modelo Baseado em Objeto e Configuração Dinâmica

O objetivo ao se criar um modelo de programação é oferecer critérios para se conhecer a realizada sendo estudada e facilitar o desenvolvimento da solução. A seguir descrevemos as principais propriedades do nosso modelo.

A. MODULARIDADE

O principal conceito envolvido no desenvolvimento de um sistema é a modularidade [3], cujas vantagens são bem conhecidas. A

modularidade se baseia na premissa de que para desenvolver uma grande tarefa a melhor solução é dividí-la em partes menores e mais fáceis de gerenciar. Deve-se, no entanto, definir critérios para particionar o sistema em módulos e determinar propriedades para que uma parte de um sistema possa ser considerada como um módulo.

Myers [4] propõe dois critérios para serem usados na decomposição de um sistema em módulos: a *Concentração do Módulo* e o *Casamento* (acoplamento) *dos Módulos*. O objetivo é aumentar a concentração do módulo e diminuir o acoplamento dos módulos, pois módulos de alta concentração aumentam a independência maximizando as relações dentro do módulo e, por outro lado, tentando diminuir a dependência entre os módulos, mininizando o acoplamento desnecessário entre alguns módulos. A proposta é utilizar a *Concentração Funcional* (onde cada módulo deve representar funções no sentido matemático) e a *Concentração Informacional* (onde o módulo esconde a implementação de algum conceito ou estrutura de dado). Este critério de particionamento se concentra no programa abstrato, logo para garantir a modularidade na implementação é preciso impor algumas propriedades.

Um segmento de programa é chamado de módulo se ele satisfaz as seguintes propriedades: *Não-Interferência Sintática*, *Independência de Contexto Semântico* e *Generalidade de Dados* [3]. A não-interferência sintática mede a possibilidade de combinar segmentos de programas sem ter que realizar mudanças sintáticas em quaisquer dos segmentos. A independência de contexto assegura que um dado segmento não pode causar efeitos colaterais nem pode ser afetado por efeitos colaterais. A propriedade de generalidade de dados requer que os módulos sejam capazes de se comunicarem via estruturas de dados. A generalidade de dados permite esconder a implementação do módulo sendo necessário saber apenas a sua especificação (serviços) para utilizá-lo.

No nosso modelo, a decomposição do sistema é feita em componente que podem ser programados, compilados e testados separadamente. O sistema é, então, construido como a configuração deste componentes de software. A separação das atividades dos componentes de programação e da configuração do sistema tem sido referida como *programming-in-the-small* e *programming-in-the-large*, respectivamente [5].

A Linguagem de Programação de Módulos [6] permite definir um módulo tarefa como um processo seqüencial e auto-contido independente de como será configurado. Esta caracteristica obriga que todas referências sejam feitas a objetos locais e não possa haver chamada

direta a outros módulos. Na verdade, os módulos contêm uma interface bem definida através de portas. A comunicação é feita enviando ou recebendo dados através das portas.

A Linguagem de Configuração [8] especifica a estrutura do sistema (configuração) identificando os módulos e as suas inteconexões pela ligação das portas. É fácil observar que utilizamos esta linguagem para definirmos o aspecto dinâmico através da reestruturação do sistema.

Como pretendemos desenvolver sistemas modulares, o primeiro critério a considerar é se o conceito de módulo tarefa utilizado preenche os requisitos rigorosos para definição de um módulo, ou seja, a não-interferência sintática, a independência de contexto e a generalidade de dados. Os módulos tarefas são unidades de programas compiladas, independentemente, e com interações bem definidas através das portas de comunicação, logo não existe qualquer amarração sintática entre os módulos, nem dependências de contexto, pois os módulos só podem trocar informações via passagem de mensagens através das portas não implicando em efeitos colaterais. A propriedade mais difícil de garantir é a generalidade de dados, contudo o conceito de portas de comunicação satisfaz, naturalmente, esta propriedade, pois os módulos se relacionam num nível puramente funcional, onde cada módulo só conhece o aspecto funcional (serviços oferecidos) do outro módulo definido pela sua interface sem o conhecimento do aspecto operacional (como o serviço é realizado). Concluimos, então, que um módulo tarefa pode ser considerado, conceitualmente, como um módulo.

B. ABSTRAÇÃO DE DADOS

É possível observar que tanto no nível abstrato, através do conceito de concentração informacional, como no nível de implementação, através do conceito de generalidade de dados, a estrutura de dados assume um papel importante no desenvolvimento de sistemas. A sistematização de programas envolvendo estruturas de dados é um fator muito importante nas metodologias de programação, pois pode-se dizer que qualquer programa necessita de dados organizados convenientemente para o seu funcionamento eficiente. Neste sentido, o conceito de dados passou a ser utilizado não apenas no estágio final de desenvolvimento do projeto, mas passou a influenciar nas decisões do projeto através do conceito de abstração de dados onde decisões que não forem relevantes são deixadas para o final.

A programação utilizando Tipos Abstratos de Dados (T.A.D.) foi introduzida por Liskov e Zilles [7]. Nela fornece-se um mecanismo para o desenvolvimento dos programas que são expressos em função dos tipos de dados mais convenientes para o problema em mão e a implementação destes tipos pode ser realizada numa fase posterior de acordo com a representação mais conveniente. A partir deste conceito, novas abordagens de tipos de dados foram aplicadas na programação como o conceito de recurso definido em [8], que representa uma generalização do conceito de tipo abstrato de dado utilizado em linguagens seqüenciais. Neste caso, as operações realizadas sobre o tipo de dado podem ser executadas paralelamente sem violar as suas propriedades. Desta forma é possível passar, naturalmente, do conceito de T.A.D. aplicado a um ambiente seqüencial para um ambiente distribuído incluindo a parte de sincronização de suas operações.

Outra abordagem de tipos de dados que facilita o desenvolvimento de sistemas é o paradigma de programação orientada a objetos [9]. Os objetos podem ser vistos como unidades conceituais distintas que têm alguma significação abstrata no contexto de um sistema. Eles são expressos através de sua estrutura de dados como se tivessem memória particular e seu comportamento é descrito por um conjunto de procedimentos, ou seja, um objeto é um T.A.D. A propriedade essencial é que um objeto não é inerte, e sim ativo e vivo, ou seja, para que as computações sejam efetuadas requer-se que os objetos sejam criados, sendo a execução das operações feita enviando mensagens ao objeto. Note que existe uma semelhança entre um objeto e um recurso, sendo este último mais genérico.

O Modelo de Objetos [10], como é chamado o sistema de programa obtido pela programação orientada a objetos, facilita a compreensão de um modelo de sistema único em que as entidades tratadas pertencem tanto ao mundo da aplicação (especificação) como ao mundo computacional (implementação), uma vez que não temos mais conceitos como programas e arquivos a serem representados, sendo o processo de desenvolvimento de um sistema visto inteiramento como o processo de criação e manipulação de representação, onde informações são obtidas e apresentadas nestas representações. A vantagem desta abordagem é evitar o problema de perda de clareza decorrente da criação de níveis abstratos no desenvolvimento de um sistema, pois a idéia de decompor um sistema em módulos abstratos intermediários pode levar a uma solução onde a estrutura final do sistema não apresenta uma correspondência clara com a estrutura inicial, dificultando a

compreensão e tornando a validação mais complicada, desde que os requisitos do sistema não podem ser facilmente compreendidos na solução final [11].

Abstrair e esconder informação formam o fundamento de desenvolvimento orientado a objeto. Uma abstração é uma descrição simplificada de um sistema que enfatiza algumas de suas propriedades e suprime outras [10]. Esconder informação sugere que deveriamos decompor o sistema baseando-se no princípio de esconder decisões de projeto na abstação.

Como vimos o conceito de módulo tarefa garante a generalidade de dados, logo podemos satisfazer as propriedades de abstrair e esconder informação requeridas para o desenvolvimento orientado a objetos. Neste sentido, consideraremos os módulos como objetos ativos que apresentam concentração funcional (módulos que implementam funções do sistema de forma semelhante a uma função matemática, onde a entrada são mensagens e a saída são outras mensagens contendo o resultado da operação) e concentração informacional, já que um objeto correspondem a um T.A.D. Logo o modelo de programação sugere a definição de módulos fortemente funcionais e informacionais, garantindo a independência dos módulos, a extensibilidade e a modificabilidade.

C. FLEXIBILIDADE

Outra característica do nosso modelo de programação é que a flexibilidade pode ser obtida em dois níveis primeiramente, através da definição de um sistema modular desenvolvido utilizando a separação entre a programação dos módulos e da estrutura, logo facilitando a alteração de um sistema, pois, geralmente, a modificação se concentra na estrutura (configuração), inserindo ou retirando módulos para que o sistema atenda a mudanças do ambiente. O outro nível de flexibilidade ocorre na definição do sistema com estruturação dinâmica onde a sua estrutura depende de condições testadas durante a execução. Neste trabalho nos concentraremos no segundo nível de flexibilidade, pois eia permite melhor exploração do paralelismo oferecido pelo ambiente distribuído resultando em soluções mais eficientes. Como discutimos, a estruturação multi-processos [2] define um programa como um conjunto de processos e suas interações, sendo a estrutura alterada dinamicamente pela criação e destruição deste processos. A nossa idéia se baseia na estruturação multi-processos, definindo critérios para o desenvolvimento de uma estrutura de módulos que explore a criação e

destruição de instâncias.

Devemos avaliar a possibilidade de utilizar a estruturação dinâmica para aumentar a eficiência do sistema, sendo considerado o possível particionamento de certos módulos em sub-unidades, porém para garantir a clareza não será permitida a modificação da estrutura inicial de modo que seja possível ver claramente o particionamento.

Uma forma de refinamento da estrutura é que se um módulo (entidade ou objeto) responde a eventos que ocorrem assincronamente em relação a sua execução, vários sub-módulos devem ser criados para alcançar a resposta assíncrona, de modo que para cada evento assíncrono existe um módulo que fica suspenso esperando que o evento ocorra e processe o evento. Assim, o evento é síncrono em relação ao módulo, mas é assíncrono em relação à estrutura geral do sistema. A programação é disciplinada desde que cada módulo responde apenas a eventos síncronos com a sua execução.

Outro aspecto da estruturação dinâmica é a criação e destruição de instâncias de módulos que é uma importante ferramenta de programação, pois podemos desenvolver soluções elegantes e eficientes para muitos problemas. Exemplos de estruturação dinâmica podem ser vistos no desenvolvimento dos sistemas operacionais THOTH [2] e V [12]. O princípio básico utilizado é a separação das partes que podem ser destruídas daquelas que não podem.

Deve-se observar que a estrutura inicial definida para o sistema, em alguns casos, pode ser utilizada diretamente sem ser necessário definir outros módulos. Contudo, é importante ver que o aspecto dinâmico de um sistema deve ser utilizado para aumentar a eficiência, sendo portanto aplicado num nível menos abstrato e mais operacional, logo não é aconselhável considerá-lo na fase de especificação do sistema, mas como um refinamento.

D. CONFIGURAÇÃO DINÂMICA

Os componentes de um modelo de configuração requerem que algumas propriedades sejam satisfeitas a fim de que a configuração dinâmica se torne mais simples e mais eficiente [13]. A seguir mostraremos como o nosso modelo satisfaz estas propriedades:

- A Linguagem de Programação de Módulos

  Esta linguagem deve fornecer módulos que possam ser escritos e compilados independentemente da configuração (*modularidade e independência de contexto*).

A chamada a outros módulos restringe a flexibilidade da configuração, pois mudanças envolveriam modificações destes nomes no texto do programa e requereriam recompilação, logo um módulo só deve se comunicar com o ambiente através da sua interface. Assim, a conexão dos módulos é especificada e verificada no nível de configuração.

Devemos observar, também, que desde que os módulos podem ser alocados numa mesma estação ou em estações distintas, eles deveriam ter o mesmo comportamento para permitir flexibilidade de configuração (*Transparência de Comunicação*). Neste sentido, não deve haver diferença lógica (sintaxe e semântica) nas primitivas para comunicação intra-estação ou inter-estação.

Como vimos, um módulo tarefa é conceitualmente um módulo, logo são garantidas as propriedades de *modularidade* e *independência de contexto*. As primitivas de comunicação não se baseam em variáveis compartilhadas garantindo à *transparência de comunicação*, como pode ser visto em [6].

- A Linguagem de Configuração

Esta linguagem deve estruturar a configuração estabelecendo funções separadas para cada etapa da configuração, a saber, especificar o conjunto de módulos a partir do qual o sistema é construido (*Definição de Contexto*), especificar, também, as instâncias criadas a partir de cada módulo (*Instanciação*) e descrever o modo pelo qual as instâncias estão interconectadas (*Interconexão*).

A reconfiguração deve ser expressa por funções inversas as de configuração, ou seja, a desconexão de instâncias, a destruição de instâncias e a remoção de módulos do contexto do sistema.

A Linguagem de Configuração que propomos em [6] apresenta comandos independentes para realizar cada uma das etapas tanto de configuração quanto de reconfiguração.

III. Desenvolvendo Programas com Configuração Dinâmica

É sabido que a construção por refinamentos sucessivos através de abstrações pode facilitar o desenvolvimento de sistemas, porém pode dificultar a clareza ao passarmos da abstração para a implementação. Neste sentido, é interessante que possamos utilizar um modelo de representação aplicável tanto na especificação como na implementação. Talvez, o mais importante benefício do desenvolvimento de sistemas

utilizando técnicas orientadas a objetos seja que esta abordagem nos fornece mecanismo para formalizar o nosso modelo de realidade, tornando possível uma correspondência direta e natural entre o mundo e este modelo [10].

Como o conceito de módulo tarefa oferece abstração e permite esconder informação, é possível utilizar um modelo semelhante ao modelo de objetos no nosso modelo para ambiente distribuído, de modo que possamos ter uma notação que permita a representação nos dois níveis. A idéia é tentar utilizar, diretamente, o modelo de objetos para representar o sistema em estudo utilizando, basicamente, a mesma representação até o final do desenvolvimento. O princípio desta abordagem é modelar a aplicação em termos de entidades, como no modelo entidade-relacionamento, e de ações que elas podem sofrer ou realizar. Neste sentido, são *identificadas as entidades e seus atributos* que são o resultado da análise de todas as pessoas, coisas e organizações pertencentes à aplicação e as ações que elas sofrem ou requisitam a outras entidades. Para tornar a representação mais ordenada podemos fazer uma separação entre o que seja um objeto e uma entidade (externa). As entidades utilizam os objetos e são identificadas, às vezes, apenas para indicar o relacionamento entre o sistema e o mundo externo, enquanto que os objetos são as entidades que representam um T.A.D. do sistema.

A segunda etapa é *identificar as operações sofridas ou requisitas a cada objeto*. Isto serve para caracterizar o comportamento de cada objeto. Aqui, nós estabelecemos a semática de cada objeto determinando as operações que são realizadas no objeto ou pelo objeto. Nesta etapa, podemos determinar as caractrísticas dinâmicas de cada objeto pela identificação das restrições de tempo ou espaço que devem ser observadas. Por exemplo, devemos especificar se existe uma ordem que as operações devem seguir.

A etapa seguinte é a *definição da interface de cada objeto*. Ela serve como um contrato entre os clientes de cada um dos objetos e o objeto em si e determina o limite da visão externa e interna de um objeto.

A representação do sistema se completa com a representação dos relacionamentos e o fluxo de dados identificados entre os objetos e as entidades, sendo o fluxo de dados utilizado para definir as interfaces dos objetos e das entidades. A validação entre o modelo e os requisitos do sistema pode ser feita facilmente desde que as entidades pertencem ao mundo da aplicação. Como a visualização é

importante para o entendimento de uma representação é interessante desenhar o esquema do modelo como um grafo indicando o relacionamento e o fluxo de dados entre os objetos e as entidades.

Depois deste primeiro refinamento, determinamos quais são os objetos do modelo, ou seja, associamos as entidades e as ações realizadas e sofridas. Conhecendo as entidades e os objetos é possível descrever o fluxo de relacionamento, possivelmente utilizando uma representação gráfica (semelhante à utilizada no modelo de objetos [10]) para facilitar o entendimento[1]. O princípio básico que deve ser considerado na definição das entidades é o critério de concentração funcional, enquanto que os objetos são caracterizados pela concentração informacional a fim de satisfazerem as condições de modularidade.

Consideramos que a checagem entre a solução do problema e os requisitos do sistema devem ser feitas, nesta etapa, pois já podemos identificar desvios de projeto e as medidas corretivas não implicarão, ainda, em grande perda do trabalho realizado.

Só depois da primeira validação devemos continuar o detalhamento do modelo a fim de evitar grandes mudanças. Nesta etapa, descreveremos o fluxo de dados, ou seja, as mensagens trocadas pelos objetos e entidades (*Definição das Interface*), lembrando que somente as interfaces são conhecidas externamente aos módulos. Então, neste ponto já contamos com uma estrutura do sistema onde temos os módulos, que correspodem às entidades e aos objetos, e a definição das relações entre os módulos (interface). Para facilitar o estudo é sugerido alterar o gráfico representativo do sistema substituindo as ações pelo fluxo de dados. Com base nesta estrutura analisaremos os aspectos dinâmicos envolvidos ou a possibilidade de sua utilização para melhorar a eficiência.

Nesta etapa, refinaremos a estrutura definida anteriormente dividindo as funções de um objeto ou entidade a fim de utilizar a criação ou destruição dinâmica (*Refinamento da Configuração*).

Outro ponto a analisar quanto a eficiência é o grau de paralelismo obtido pela alocação dos módulos nas diversas estações (processadores), já que estamos num ambiente distribuído. A definição de um critério para determinar a alocação dos módulos (funções) aos

---

[1] Na nossa representação gráfica, as entidades são consideradas como quadrados com linhas simples, os objetos como quadrados com linha duplas e as interações são linhas direcionadas da entidade que realizou o pedido para a que sofreu.

processadores é muito difícil, pois depende das características da aplicação. No entanto, achamos que um bom critério que pode ser utilizado é avaliar o número de mensagens trocadas entre os módulos para determinar onde eles serão alocados. O objetivo é diminuir o fluxo de mensagens no meio físico de comunicação agrupando os módulos que apresentam um alto fluxo de mensagens numa mesma estação.

Como resultado desta etapa temos a estrutura final do sistema. Devemos observar que na etapa anterior definimos os objetos, entidades e o fluxo de dados, porém com este detalhamento é provável que novas mensagens sejam identificadas para definir o relacionamento entre os sub-módulos resultantes.

Considerando que cada objeto realiza as ações definidas de acordo com a sua função, bem como que cada entidade realiza o pedido de ação ao objeto indicado, é possível definir a configuração do sistema utilizando a Linguagem de Configuração, no entanto não será possível, ainda, realizar a implementação, pois os módulos não foram desenvolvidos. Isto, no entanto, sugere a definição de um protótipo do sistema para verificar o seu comportamento global (*Prototipagem e Simulação*). Isto pode ser feito criando módulos testes com a função apenas de enviar ou receber as mensagens definidas pela sua interface. A vantagem é facilitar a verificação de grandes sistemas, principalmente, aqueles com estruturas fortemente dinâmicas a fim de encontrar erros de projetos em fases, ainda, iniciais. Além disso, podemos escolher representações altenativas dos objetos a fim de experimentar o comportamento do sistema diante de várias implementações.

A última etapa é o desenvolvimento dos módulos (entidades e objetos) através de programação seqüencial (*Estruturação Interna dos Módulos*). Como resultado desta etapa, pretendemos obter o modelo de objetos e entidades a serem implementados. Partindo do princípio que a metodologia garante a modularidade, geralmente, o desenvolvimento dos módulos se torna bem mais fácil, pois eles são unidades pequenas, independentes e com funções claramente definidas. A implementação de um módulo é, na verdade, o resultado natural da expansão da estrutura definida, como por exemplo, a implementação das operações realizadas sobre o tipo de dado.

A interação do módulo com o ambiente (outros módulos) ocorre via passagem de mensagens, onde mensagens são enviadas pedindo a realização de alguma operação ou mensagens são recebidas passando o resultado ou realizando a atividade requerida. Neste sentido, a

seqüência de passagem de mensagens é de vital importância para a definição da estrutura interna de cada módulo. É possível caracterizar a chegada de uma mensagem como um evento que ocorre e requer que alguma providência seja tomada, certamente ativando um conjunto de ações que consomem a informação recebida e causam, provavelmente, a produção de outra informação. Então, podemos definir o comportamento do módulo como uma função matemática que opera os valores de entrada (mensagens recebidas) e devolve os resultados (mensagens de resposta), logo a estrutura de um módulo é determinada pelos eventos e as ações realizadas, como em Queiroz [14]. No modelo de objetos de um sistema, podemos identificar as mensagens trocadas pelos objetos e entidades (os eventos) e as ações realizadas, logo resta apenas ordenar os pares (eventos, ações) para os diversos eventos que determinam o comportamento do módulo. A implementação consistirá, basicamente, no desenvolvimento do algoritmo seqüencial que realiza as ações correspondentes aos módulos (desde que a estrutura da comunicação já foi definida).

IV. Estudo de Caso

Os sistemas distribuídos oferecem muitas vantagens em relação aos sistemas centralizados. O paralelismo inerente a eles permite aumentar o desempenho, facilitar as modificações e extensões, etc. Um dos principais benefícios da distribuição de um sistema é o potencial para maior tolerância a falha, pois como eles são baseados em componentes independentes, uma falha parcial não leva a uma falha total como ocorre em sistemas centralizados.

Muitas aplicações necessitam de diferentes graus de tolerância a falha. Uma classificação de sistemas de acordo com os requisitos de confiabilidade define a relação entre a falha de qualquer um dos módulos constituintes de um sistema e a confiabilidade geral requerida pelo sistema. Identificamos dois tipos, a saber, os sistemas fracamente dependentes de falha e os fortemente dependentes [18]. Os sistema do 1º tipo são projetados de tal forma que os requisitos de segurança não são violados se um dos constituintes falhar, i.e, o serviço fornecido é degradado, mas este é ainda capaz de satisfazer os requisitos. Já os sistemas do 2º tipo têm os requisitos violados se um dos módulos falhar, i.e, uma falha num módulo leva a uma falha do sistema.

Um exemplo típico de sistema de controle de processos é o chamado sistema supervisor que avalia o funcionamento de um conjunto de equipamentos, emitindo relatórios de desempenho e, em alguns casos, controlando o próprio funcionamento dos equipamentos. O nosso exemplo de sistema supervisor é formado por n estações, onde cada estação contém um conjunto de equipamentos sob seu controle. Cada estação, chamada de local, deve periodicamente realizar uma varredura dos seus equipamentos a fim de executar os controles, atualizar o seu banco de dados, emitir relatórios, etc. Uma destas estações assume, também, um papel de estação mestre, cuja função é se comunicar com cada estação local e recolher os dados dos seus equipamentos a fim de diagnosticar o comportamento de todo o sistema. Qualquer estação pode assumir o papel de mestre, sendo esta escolhida de acordo com o comportamento (carga) das estações no início da execução do sistema. Ocorrendo uma falha na estação mestre, outra estação deve automaticamente assumir esta função. A nova estação é escolhida de acordo com o mesmo critério anterior.

Para formalizar o nosso modelo de realidade devemos, inicialmente, analisar quais as entidades que compõem o sistema (*Identificação das Entidades e Objetos e seus atributos*) Numa primeira abstração, encontramos as seguintes entidades:

- *Supervisor Local* que realiza a monitoração dos equipamentos subordinados, passando informações para o Supervisor Mestre e recebe pedidos de operações dos usuários;
- *Supervisor Mestre* que monitora todo o sistema através do recebimento de informações dos Supervisores Locais e, também, realiza operações a pedido dos usuários;
- *Usuário* que interage com os supervisores a fim de obter informações sobre o estado do sistema ou para pedir a realização de alguma operação;
- *Equipamento* que realiza a monitoração e o controle de um equipamento físico, por exemplo uma caldeira ou uma comporta. O Supervisor Local, periodicamente, pede informação sobre o estado dos equipamentos e pode pedir a realização de uma determinada operação de acordo com a ocorrência de alguma situação.

Poderíamos definir uma entidade que correspondesse ao banco de dados local e outra para o banco de dados mestre, no entanto, consideraremos na nossa solução que os bancos de dados são apenas tabelas que o supervisor mantém, logo não os definiremos como uma entidade. As ações realizadas neste ambiente são: ler dados do banco

de dados local ou mestre, emitir relatório com estes dados, ligar um equipamento, etc (*Identificação da Operações Realizadas ou Sofridas*). A fig. 4.1 ilustra o diagrama do sistema.

No nosso exemplo, consideraremos o sistema como fortemente dependente em relação ao Supervisor Mestre. A solução é manter uma cópia do Supervisor Mestre noutra estação. No entanto, para que a cópia possa realizar as funções de mestre a partir do ponto anterior a falha, deverá existir troca de informações entre estes dois módulos. Assim temos um novo objeto no nosso modelo, um "Supervisor Mestre Auxiliar". Chamaremos estes dois módulos de ativo e passivo respectivamente.

Podemos observar que, embora os módulos troquem diversos tipos de mensagens, é possivel definir uma única estrutura sintática que represente as diversas mensagens (eventos), simplificando a interface. Representaremos apenas a chegada de um pedido e o envio da resposta, sendo internamente analisado o significado da mensagem (*Definição das Interfaces*)

FIG. 4.1 MODELO DE REPRESENTAÇÃO E/R E FLUXO DE DADOS

Note que o sistema se baseia no funcionamento do Supervisor Local que deve ser alocado em cada estação e do Supervisor Mestre que é colocado em uma das estações. Os Supervisores Locais correspondem simplesmente a instâncias do módulo Supervisor Local, enquanto que para o Supervisor Mestre devemos utilizar uma função que determina onde este módulo deve ser alocado. O principal problema a tratar na estrutura é a ocorrência de falhas na estação onde o Supervisor Mestre está executando, pois o sistema não deve ficar sem a supervisão geral. A estrutura do sistema será definida de modo que seja constantemente testado se existe um Supervisor Mestre em execução. Quando não existir, o módulo passivo é ativado. Para aumentar a segurança do sistema é interessante que sempre exista um módulo passivo, logo quando um módulo passivo for transformado em ativo uma outra cópia

deve ser criada em outra estação.

Como discutimos, depois de criada a estrutura do sistema, deve-se avaliar a possibilidade de particionamente de módulos para aumentar a eficiência (*Refinamento da Configuração*). Neste exemplo, podemos notar que a troca de informações entre os módulos ativo e passivo, que estão em estações distintas, pode degradar a realização das funções normais do sistema. A fim de amenizar este problema, dividiremos as funções do Supervisor Mestre criando um módulo gerente de segurança que é local. A função deste gerente é receber as informações do Supervisor Mestre e se comunicar com o gerente remoto onde está o módulo passivo, evitando que o supervisor fique muito tempo bloqueado. O diagrama final da estrutura do sistema é mostrado na fig. 4.2.

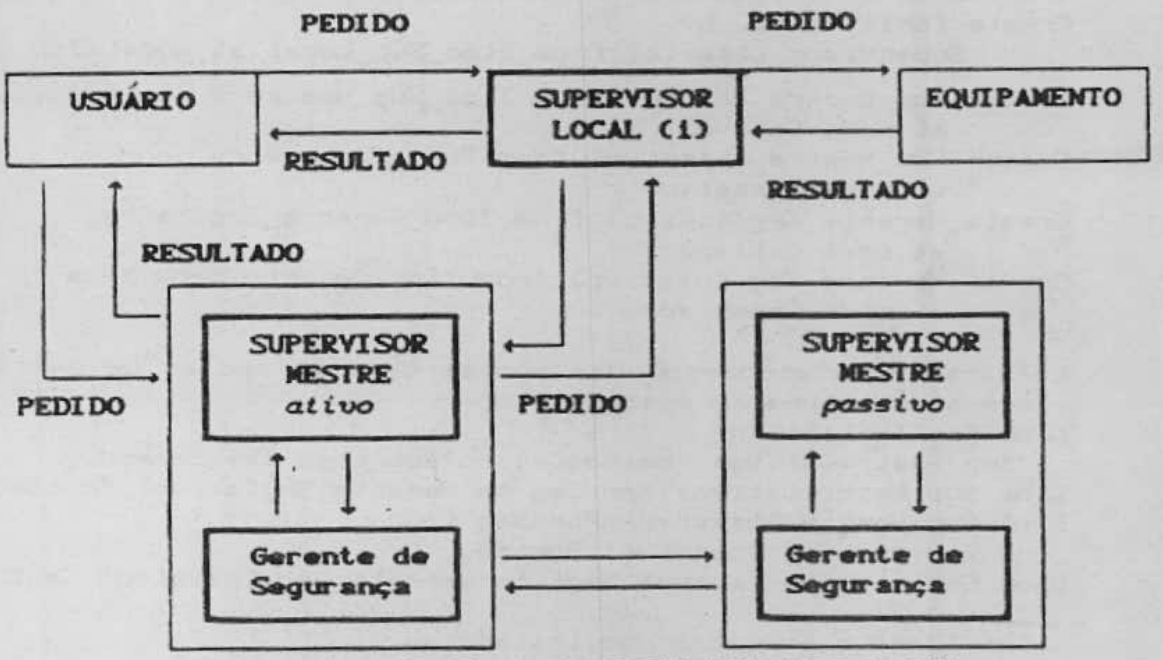

FIG. 4.2 - DIAGRAMA DA ESTRUTURA DO SISTEMA

Veremos, a seguir, um programa de configuração que define a estrutura do sistema. A Linguagem de Configuração que definimos é semelhante à de Conic [17], sendo mais poderosa. Será fácil observar que o programa de configuração está bem estruturado tornando fácil o entendimento, os detalhes da linguagem podem ser vistos em [18].

```
CONFIGURAÇÃO DO SISTEMA
System Supervisor;
  Const Fim_do_Mundo = false;
        n = 10; { No. de estacoes }
  Var
     ativo, passivo {indicam quais os nos que assumem o papel de}
     k : integer; {mestre ativo e passivo respectivamente }
```

```
    estacoes : array [1..n] of integer; { armazena o estado das
               estacoes, indicando quais estao funcionando e
               a carga de trabalho de cada uma delas }
Begin
    ativo   := 0;
    passivo := 0;
    Inicia_Tabela (estacoes);

    { Este procedimento escolhe os nos onde serao executados os
      modulos Supervisores Mestres ativo e passivo}
    Escolhe_Estacao (ativo, Passivo);

    { Define-se o Contexto, i.e, indicam-se os modulos que formam
      a estrutura do sistema}
    Use Tipo_Sup_Local, Tipo_Sup_Mestre; {
    Use Tipo_Gerente_Seguranca;

    { Criam-se as instancias dos modulos}
    Create family k:[1..n]
          Supervisor_Local[k] from Tipo_Sup_Local at node(k);
    Create Sup_Mestre (ativo) from Tipo_Sup_Mestre
           at node (ativo);
    Create Sup_Mestre (passivo) from Tipo_Sup_Mestre
           at node (passivo);
    Create Gerente_Seg (ativo) from Tipo_Gerente_Seguranca
           at node (ativo);
    Create Gerente_Seg (passivo) from Tipo_Gerente_Seguranca
           at node (passivo);

    { Faz-se a interconexao das portas das instancias de acordo
      com a estrutura do sistema}
    Link family k:[1..n]
       Sup_Mestre(ativo).comando[k] to Sup_Local[k].comando;
    Link Sup_Mestre(ativo).PortSeg to Gerente_Seg(ativo).PortSeg;
    Link Sup_Mestre (passivo).PortSeg to
         Gerente_Seg (passivo).PortSeg;
    Link Gerente_Seg (ativo).Seg1 to Gerente_Seg (passivo).Seg2;

    { Inicia-se a execucao das instancias }
    Activate family k:[1..n]
             Supervisor_Local[k];
    Activate Sup_Mestre (ativo); { Inicia a execucao Sup. Mestre }
    Activate Gerente_Seg (ativo), Gerente_Seg (passivo);

  Repeat
    { Este comando bloqueia o programa enquanto a instancia
      indicada estiver em execucao. Neste caso isto significa
      esperar ate ocorrer uma falha na estacao mestre
      interrompendo a execucao do Supervisor Mestre ativo }
    Wait Sup_Mestre (ativo)
    => { Desconecta as instancias }
       Unlink family k:[1..n] (Mestre)
         Sup_Mestre(ativo).comando[k] from
         Supervisor_Local[k].comando;
       Unlink Sup_Mestre(ativo).PortSeg from
              Gerente_Seg(ativo).PortSeg;
       Unlink Gerente_Seg (ativo).Seg1 from
              Gerente_Seg (passivo).Seg2;
```

```
            ( Destroi as instancias Sup.Mestre(ativo) e Gerente Seg.)
            Delete Sup_Mestre (ativo), Gerente_Seg (ativo);

            ( Liga os Sup. Locais ao Sup. Mestre passivo e ativa a sua
              execucao, transformando-o em Supervisor Mestre ativo )
            Link family k:[1..n] Sup_Mestre(passivo).comando[k] to
                 Supervisor_Local[k].comando;
            Activate Sup_Mestre (passivo);

            ( Escolhe uma nova estacao passiva e recria um novo
              Supervisor Mestre passivo )
            ativo   := passivo;
            passivo := 0;
            Calcula_Estacoes (ativo, passivo);
            Create Sup_Mestre (passivo) from Tipo_Sup_Mestre
                   at node (passivo);
            Link Sup_Mestre (ativo).PortSeg to
                 Gerente_Seg (ativo).PortSeg;
            Activate Gerente_Seg (ativo);
            Create Gerente_Seg (passivo) from Tipo_Gerente_de_Seguranca
                   at node (passivo);
            Link Sup_Mestre (passivo).PortSeg to
                 Gerente_Seg.PortSeg;
            Link Gerente_Seg (ativo).Seg1 to Gerente_Seg
                 (passivo).Seg2;
            Activate Gerente_Seg (ativo), Gerente_Seg (passivo);
        End
      Until Fim_do-Mundo
    End Supervisor;
```

Como o problema principal desta aplicação é a questão de gerenciamento da configuração, não vamos detalhar o comportamento dos módulos individualmente, pois as suas funções não interferem no comportamento do programa de configuração. Vejamos, apenas, a definição das interfaces para que se possa desenvolver um protótipo para testar a configuração.

MÓDULO TIPO-SUP-LOCAL

```
    Task Module Tipo_Sup_Local;
       Entryport comando: Tipo_Comando reply Tipo_Dado;
    End Tipo_Sup_Local.
```

MÓDULO TIPO-SUP-MESTRE

```
    Task Module Tipo_Sup_Mestre;
       Exitport comando[1..n]: Tipo_Comando reply Tipo_Dado;
       Exitport PortSeg: MsgSeg Reply MsgSeg;
    End Tipo_Sup_Mestre;
```

MÓDULO TIPO-GERENTE-SEGURANCA

```
    Task Module Tipo_Gerente_Seguranca;
       Entryport PortSeg: MsgSeg Reply MsgSeg;
       Exitport  Seg1: MsgSeg Reply MsgSeg;
       Entryport Seg2: MsgSeg Reply MsgSeg;
    End Tipo_Gerente_Seguranca;
```

V. Conclusão

O grande poder da programação distribuída com configuração dinâmica é permitir utilizar a flexibilidade e a robustez dos sistemas distribuídos no nível lógico (*programação*). A principal preocupação é que a criação e destruição de instâncias não provoque um grande *overhead* que possa comprometer o desenvolvimento de algumas aplicações. No entanto, devemos lembrar que como as instâncias em execução são, na verdade, processos o importante é avaliar se o sistema operacional que dará suporte à implementação oferece recursos para a criação e destruição de processos eficientemente.

Quanto ao modelo de programação, consideramos que foram garantidas propriedades importantes para o desenvolvimento de programas distribuídos que fossem fáceis de usar e fáceis de entender como a exploração da flexibilidade e robustez, inclusive em sistemas dinâmicos.

O exemplo visto nesta seção permitiu avaliar o bom desempenho da metodologia de programação. Por uma questão de simplicidade não detalhamos todas as etapas da metodologia enfatizando, basicamente, as questões referentes a definição da estrutura do sistema. No entanto, acreditamos que foi possível demonstrar de uma forma simples que ela oferece critérios para o desenvolvimento de programas bem estruturados e eficientes.

Notamos que, embora o objetivo principal do modelo seja a programação distribuída, a propriedade de transparência de comunicação garantida pela Linguagem de Programação de Módulos permite utilizá-lo num ambiente multi-programável para aumentar a eficiência das aplicações, bem como para melhorar a estruturação de muitos programas.

Em conclusão, o uso de configuração dinâmica em programas distribuídos implica em muitas questões de projeto e de implementação. Pretendemos aplicar o modelo em diversas situações a fim de garantir os princípios propostas. Além disso, um passo seguinte do nosso projeto é o desenvolvimento de ferramentas automáticas que facilitem a construção de programas.

REFERÊNCIAS

[1] Kleinrock, L.: "Distributed Systems"; Communications of The ACM, Vol. 28, no. 11, Novembro 1985, (pp. 1200-1212).

[2] Cheriton, D.R.: "Multi-Process Structuring and the THOTH Operating Systems"; Ph.D Thesis, University of Waterloo, Ontario, Canada, 1978.

[3] Parnas, D.J.: "On The Criteria To Be Used in Decomposing Systems Into Modules"; CACM, Dezembro 1972, (pp. 1053-1058).

[4] Meyers, G.J.: "Composite/Structured Design"; Van Nostrand Reinhold Co.; 1978.

[5] DeRemen, F.; Kron, H.: "Programming-in-the-large Versus Programming-in-the-small"; Proc. Conf. Reliable Software, 1975, pp 114-121.

[6] Justo, G.R.R.; Cunha, P.R.F.: "Desenvolvimento de Programas Distribuídos Através de Configuração Dinâmica de Processos"; Anais do 6o. SBRC, Belo Horizonte, Março 1988.

[7] Liskov, B.H.; Zilles, S.: "Programming With Abstract Data Types"; Proc. Conference on Very High Level Languages, SIGPLAN, Vol. 9, Abril 1974.

[8] Cunha, P.R.F.: "Design and Analysis of Message Oriented Programs"; Ph.D Thesis, University of Waterloo, Ontario, Canadá, 1981, (199 pp.).

[9] Rotenberg, H.B.: "Programação Orientada A Objetos: Um Enfoque de Engenharia de Software"; Dissertação de Mestrado, Departamento de Informática PUC/RJ, 1984.

[10] Booch, G.: "Object-Oriented Development"; IEEE Trans. on Software Engineering, Vol Sb-12, No. 2, Fevereiro 1986.

[11] Adrion, W.R.; Bronstand, ; Chernavsky, J.C.: "Validation, Verification and Testing of Computer Software"; Computer Surveys, Vol. 14, No. 2, Junho 1982.

[12] Cheriton, D.R.: "The V Distributed System"; Communications of the ACM, Vol. 31, No. 3, Março 1988.

[13] Kramer, J.; Magee, J.: "Dynamic Configuration for Distributed Systems"; IEEE Transsactions on Software Engineering, Vol. SE-11, No. 4, Abril 1985, (pp. 424-436).

[14] Queiroz, R.J.G.B.: "Uma Metodologia de Programação Para Implementação de Protocolos"; Tese de Mestrado, Depto. de Informática, Universidade Federal de Pernambuco, Recife, PE, 1984, 171pp.

[15] Cameron, J.R.: "An Overview of JSD"; IEEE Trans. on Software Engineering, Vol SE-12, No.2, fevereiro 1986.

[16] Loques, O.G.; Kramer, J.: "Flexible Fault-Tolerance for Distributed Computer Systems"; IEEE Proceedings, Vol 133, No. 6, Novembro 1986, (pp. 319-331).

[17] Sloman, M.; Kramer, J.; Magee, J.: "The CONIC Toolkit For Distributed Systems"; Proceedings of The 6th IFAC Distributed Computer Control System Workshop, Monterey, California, USA, Maio 1985.

[18] Justo, G.R.R.: "Ambiente de Programação Distribuída Com Configuração Dinâmica de Processos"; Dissertação de Mestrado, Depto. de Informática, UFPE/CCEN, Setembro 1988, 180pp.