Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial
Universidade Estadual de Campinas

Classificação e Seleção de Componentes de Software Concorrentes

Rogério de Carlo

Fevereiro de 1999

Banca Examinadora:

- Profa. Dra. Ana Cristina Vieira de Melo (Orientadora)
- Prof. Dr. Rafael Santos Mendes (Co-orientador)
- Prof. Dr. Mario Jino (DCA - FEEC - UNICAMP)
- Prof. Dr. Flávio Soares Corrêa da Silva (MAC - IME - USP)
- Profa. Dra. Beatriz Mascia Daltrini (DCA - FEEC - UNICAMP) (Suplente)

Este exemplar corresponde a redação final da tese defendida por Rogério de Carlo e aprovada pela Comissão Julgada em 26/02/99

Orientador

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

C195c

Carlo, Rogério de
Classificação e seleção de componentes de software concorrentes. / Rogério de Carlo.--Campinas, SP: [s.n.], 1999.

Orientadores: Ana Cristina Vieira de Melo, Rafael Santos Mendes.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Software - Reutilização. 2. Software – Classificação. 3. LOTOS (Linguagem de programação de computador). I. Melo, Ana Cristina Vieira de. II. Mendes, Rafael Santos. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Classificação e Seleção de Componentes de Software Concorrentes

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rogério de Carlo e aprovada pela Banca Examinadora.

Campinas, 26 de Fevereiro de 1999.

Profa. Dra. Ana Cristina Vieira de Melo
(Orientadora)

Prof. Dr. Rafael Santos Mendes
(Co-orientador)

Dissertação apresentada a Faculdade de Engenharia Elétrica e de Computação - Departamento de Engenharia de Computação e Automação Industrial, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica e Computação.

Resumo

Reutilizar componentes de software pode trazer uma série de benefícios, tais como, a redução no custo e tempo de desenvolvimento dos sistemas, e o aumento na qualidade e confiabilidade do produto final. Entretanto, para que a reutilização se torne uma prática comum dentro do ambiente de desenvolvimento de software, uma série de questões e problemas ainda precisam ser solucionados.

Este trabalho visa tratar alguns problemas presentes na primeira fase do processo de reutilização: a seleção dos componentes de software. Nesta etapa, é fundamental que o conjunto de componentes recuperados seja relativamente pequeno e composto apenas por componentes que desempenham a função procurada, ou alguma outra função similar. Caso contrário, a tarefa de analisar cada componente separadamente torna-se inviável, prejudicando todas as demais fases do processo de reutilização.

Assim sendo, o objetivo básico deste trabalho é fornecer uma ferramenta simples, que seja capaz de responder as seguintes indagações:

- Como organizar e armazenar componentes tanto sequenciais quanto concorrentes, de maneira clara e estruturada?

- Como recuperar os componentes que executam uma determinada função, em meio a uma grande biblioteca de software?

Para tanto, apresentamos uma ferramenta composta por dois filtros: 1. classificação por atributos e 2. casamento de interfaces. No primeiro filtro os componentes são classificados e selecionados através de atributos, os quais descrevem sua funcionalidade e forma de processamento. Já no segundo, os componentes são selecionados com base nos tipos dos parâmetros presentes em suas interfaces, formalmente especificadas através da linguagem LOTOS. Em resumo, procuramos mesclar técnicas formais e informais em uma mesma ferramenta, com o intuito de facilitar a seleção de componentes de software concorrentes ou sequenciais.

Abstract

Reusing software components can bring a lot of benefits, such as the reduction in systems development cost and time and the improvement of the final product quality and reliability. However, in order to turn reuse a common place into the software development environment, some questions and problems must be answered.

This work aims to deal with some problems present in the first stage of the reuse process: the retrieving of software components. It's important that the set of retrieved components be small and composed only by components that do the searched function, or another similar one. Otherwise, the work of understanding each component becomes infeasible, affecting the others stages of the reuse process.

So, the goal of this work is to develop a simple tool, that can answer these questions:

- How to organize and store as sequential as concurrent components, in a precise and structured way?
- How to find the component that do a specific function, into a large software library?

In this way, we present a tool composed by two filters: 1. attributes classification and 2. signature matching. In the first filter the components are classified and retrieved by attributes that describe their functionality and processing. In the second one, the components are retrieved by their interface parameter types, formally specified through the LOTOS language. So, we joined formal and informal techniques into the same tool to help the user in retrieving concurrent or sequential software components.

Agradecimentos

Esta dissertação de Mestrado, com certeza, não foi um trabalho realizado por uma única pessoa. Ao contrário, sem a ajuda das dezenas de pessoas que passaram em minha vida nestes últimos anos, provavelmente, ainda estaria escrevendo o primeiro parágrafo.

Assim, em primeiro lugar agradeço a Deus, por estar sempre ao meu lado em todas as etapas da minha vida. Sua Força e Luz me guiam e me socorrem nos momentos mais importantes e difíceis...

Depois, tenho que agradecer a minha família e as pessoas que mais adoro na vida: meu pai, mãe e irmã. Tudo o que SOU e SEI, devo única e exclusivamente a estas três pessoas. Apesar da distância, estiveram comigo a cada minuto, me apoiando, incentivando e acreditando em minha capacidade.

Também devo fazer um agradecimento especial a minha orientadora e amiga Ana Cristina. Ela me mostrou o verdadeiro papel de um professor e pesquisador. Suas idéias, conversas e apoio foram minha inspiração e base de sustentação desta dissertação.

Agora vem a parte mais complicada: agradecer a todos os meus amigos. O apoio, a alegria, as inúmeras festas, as viagens e a amizade de todos que conheci neste Mestrado tornaram estes últimos anos simplesmente maravilhosos e inesquecíveis. Assim, agradeço a todo o pessoal da turma do Mestrado de 96 (IC) e agregados, em especial: André, Gandhi, Freud, Marcelo Marcos, Pavão, Alessandra, Marília, Janne, Gláucia, Guilherme Pimentel, Guilherme Albuquerque, Gisele, Marcelo de Jesus, Roberto Façanha, Gutemberg, Luciano, Delano, Marcos André, Cleidson, Mário e Laura, Francisco e Jucele, Victor e Cristina, Christiane Campos e Cláudio, Karina, Patricia Ropelatto (pela montanha de besteiras e por ter sido a única a ter assistido minha defesa), Vaguinho (por ter me ensinado a lavar e a passar roupas), Márcio-Brow e Ralph-Brau (dois grandes amigos que vão ter que me aguentar no mesmo apartamento por muito mais tempo - República BBB) e o Edi (por ter me suportado durante dois anos, me ensinado a cozinhar e mais uma infinidade de lições). E, por fim, dentre meus amigos tenho que ressaltar três pessoas que foram mais que fundamentais em minha vida de Mestrado... foram minhas irmãs e a quem

devo um agradecimento eterno: Amanda, Cristina Toyota e Selma Bássiga.

Agradeço também ao CNPq, pelo apoio financeiro durante os dois primeiros anos do Mestrado.

Não posso deixar de citar o inventor do Miojo e do Hamburguer pelo apoio alimentar.

E, para terminar, agradeço a todo mundo que já conheci ou ainda vou conhecer na minha vida... e a você que gastou o maior tempo lendo estes agradecimentos!!!

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A demanda por novos sistemas computacionais cresce a cada dia, na mesma proporção em que aumentam seus tamanhos e complexidades. Ao mesmo tempo, o mercado consumidor exige produtos com menor preço e tempo de entrega, e com maior confiabilidade e qualidade. Uma das tecnologias atualmente disponíveis que apresenta grande potencial para suprir tais exigências são as chamadas 'bibliotecas de componentes de software reutilizáveis'.

Reutilizar componentes pré-existentes pode reduzir o tempo e o custo de desenvolvimento de um novo sistema, uma vez que menos código precisa ser escrito. Além disso, como os componentes presentes em uma biblioteca já foram anteriormente testados, utilizados e de preferência formalmente especificados, diminui-se o tempo gasto com depurações e aumenta-se a confiabilidade do produto final.

Entretanto, para conseguir utilizar tais bibliotecas é imprescindível a disponibilidade de ferramentas que auxiliem na localização dos componentes procurados. Deve ser obrigatoriamente mais simples e rápido selecionar um determinado componente do que desenvolvê-lo por completo. Caso contrário, a reutilização perde o sentido, e uma biblioteca repleta de componentes torna-se totalmente inútil.

1.1 Motivação

Para que a reutilização se torne uma abordagem realmente eficiente, o esforço necessário para se reutilizar um componente deve ser consideravelmente menor do que o esforço gasto durante a criação de um novo componente. O processo de reutilização de software é composto basicamente por quatro passos: selecionar o componente, compreender sua estrutura e código, modificá-lo para se adequar aos novos requisitos, e integrá-lo ao sistema. O seguinte algoritmo ilustra, num

alto nível de abstração, o esquema geral da reutilização [Pri85]:

```
Dada a especificação do componente desejado
begin
    Pesquise a biblioteca de componentes
    if Encontrou exatamente o componente desejado then Termine a busca
    else begin
        Procure componentes similares
        for each Componente similar encontrado begin
                Compreenda sua estrutura e código
                Compare suas características à especificação fornecida
        end
        Selecione o melhor componente
        Modifique o componente selecionado
        Integre o componente modificado ao sistema
    end
end
```

Nesta dissertação de Mestrado, estamos interessados em fornecer uma ferramenta que auxilie o trabalho do usuário durante a primeira fase da reutilização: a classificação e seleção de componentes sequenciais ou concorrentes. A importância desta etapa reside no fato de que todo e qualquer processo de reutilização deve inicialmente começar com a seleção do componente desejado pelo usuário, ou seja, para reutilizar é preciso antes classificar e selecionar. De outra forma, a biblioteca de software se torna um grande repositório de componentes desordenados e difíceis de serem localizados.

Uma biblioteca de componentes de software reutilizáveis pode ser derivada a partir de fontes variadas: reengenharia de projetos passados, coletando componentes passíveis de serem reutilizados; desenvolvimento de novos componentes específicos para reutilização; ou compra de componentes e pacotes de software de outras empresas organizacionais. A questão é que uma biblioteca para ser realmente útil deve ser composta por uma grande quantidade de componentes, uma vez que é difícil prever, com precisão, quais funções serão necessárias em projetos futuros. O problema se agrava ainda mais quando se leva em consideração que, dentre estes componentes, vários podem desempenhar funções semelhantes e ser classificados da mesma forma. Assim sendo, torna-se inviável ao usuário classificar e selecionar um componente em uma biblioteca deste porte sem a utilização de uma ferramenta de auxílio. Fica claro então, a necessidade de se

fornecer sistemas computacionais que, nesta primeira fase do processo de reutilização, facilitem o trabalho do usuário na execução das seguintes tarefas:

- **Classificação** dos componentes dentro da biblioteca, de forma clara e precisa;
- **Seleção** dos componentes a partir de uma determinada especificação. Esta seleção deve retornar tanto componentes que executem *exatamente* a função desejada (algo difícil de acontecer) quanto componentes que executem funções *similares* e que sejam facilmente adaptáveis.

A maior parte dos trabalhos atualmente disponíveis na área concentra-se principalmente na seleção de componentes sequenciais [MMM97]. Nestes casos, geralmente não há um tratamento especial para os componentes concorrentes, sendo estes manipulados sem uma distinção real entre as variadas formas de processamento. Em nosso trabalho, o foco de atenção foi direcionado à classificação e seleção de componentes de software concorrentes, formalmente especificados através da linguagem LOTOS [ISO87]. Tais componentes são imprescindíveis em sistemas de tempo-real ou distribuídos, por exemplo, onde o tempo de resposta e o desempenho são fatores críticos. Assim, devido à complexidade e importância de tais componentes, torna-se importante a especificação formal dos mesmos, de modo que possam ser interpretados de maneira clara e sem ambiguidades. Para tanto, adotamos a linguagem de especificação formal LOTOS como base para nossos componentes, uma vez que tal linguagem foi criada basicamente com elementos que facilitam a especificação de sistemas concorrentes [Tur96].

1.2 Uma Proposta de Solução

Levando em consideração os dois pontos anteriormente especificados, apresentamos nesta dissertação a implementação de um sistema baseado em dois filtros: 1. classificação por atributos e 2. casamento de interfaces LOTOS.

O primeiro filtro é responsável pela classificação dos componentes (sequenciais ou concorrentes) dentro da biblioteca de software. Esta classificação é baseada em atributos que descrevem as características funcionais e de processamento de cada componente, sendo independente de qualquer linguagem de programação. Uma vez classificados, os componentes podem ser futuramente selecionados através da descrição de seus atributos básicos. Esta seleção pode acontecer através de um 'casamento exato' ou através de 'casamentos parciais', onde os componentes selecionados são similares à função procurada. Neste caso, o custo de modificar um componente semelhante geralmente é menor do que desenvolvê-lo por completo.

O segundo filtro trabalha com as interfaces dos componentes especificadas em LOTOS. A linguagem de especificação LOTOS é particularmente importante para o sistema, pois através dela somos capazes de especificar tanto componentes sequenciais quanto concorrentes. Para este filtro, todos os componentes precisam ter suas especificações formalmente definidas em LOTOS, o que acaba aumentando o nível de confiabilidade nos mesmos. Neste caso, a seleção baseia-se no 'casamento' dos tipos dos parâmetros de entrada e saída de cada componente. Uma vez que isso ocorra, o conjunto final selecionado será composto apenas por componentes que manipulam os tipos de variáveis que o usuário espera trabalhar.

Dessa forma, nosso sistema apresenta uma solução aos problemas da primeira etapa do processo de reutilização da seguinte forma:

1. Classifica os componentes através de atributos que descrevem sua funcionalidade e processamento. Estes componentes podem ser tanto sequenciais quanto concorrentes.

2. Seleciona os componentes através de filtros baseados em atributos e interfaces formalmente especificadas em LOTOS, que sucessivamente reduzem o conjunto final retornado ao usuário. O primeiro filtro garante que os componentes selecionados apresentam a funcionalidade desejada (ou uma função similar), enquanto o segundo filtro garante que os componentes selecionados trabalham apenas com os tipos básicos exigidos pelo usuário.

1.3 Organização da Dissertação

Este trabalho está apresentado neste Capítulo inicial e mais cinco Capítulos restantes. O primeiro descreve conceitos básicos sobre reutilização de software, os dois seguintes apresentam o trabalho tema, o quarto Capítulo descreve a implementação do sistema, e o último, considerações sobre o desenvolvimento do trabalho e temas relacionados:

Capítulo 2 Define reutilização de software, bem como suas características, objetivos e níveis de abstração. Explica todas as etapas do processo de reutilização, e insere os métodos formais dentro deste contexto. Este capítulo é importante para definir o escopo da dissertação dentro da abordagem de reutilização.

Capítulo 3 Definição do primeiro filtro do sistema: a classificação por atributos. Explica as características de um sistema de classificação, principalmente no que diz respeito à classificação de componentes de software. Apresenta uma extensão ao trabalho de Prieto-Díaz [Pri85], no sentido de possibilitar a classificação de componentes concorrentes. E,

por fim, define novas modificações ao dicionário de termos sinônimos e à rede de termos semânticos, responsáveis pela seleção de componentes similares.

Capítulo 4 Definição do segundo filtro do sistema: o casamento de interfaces em LOTOS. Explica as características principais de LOTOS, dando ênfase ao seu sistema de tipos básicos. Apresenta o casamento de interfaces e seus variados níveis de seleção. Define um novo esquema simplificado de casamento de interfaces empregando especificações formais desenvolvidas em LOTOS.

Capítulo 5 Este capítulo corresponde à implementação do sistema de classificação e seleção de componentes concorrentes, baseado em classificação por atributos e casamento de interfaces LOTOS. Apresenta as estruturas principais do sistema e um exemplo completo de utilização do mesmo.

Capítulo 6 Neste capítulo são relatadas as considerações finais sobre o trabalho. Estas considerações estão relacionadas ao contexto geral onde o trabalho está inserido e à experiência adquirida com o desenvolvimento do mesmo.

Capítulo 2

Reutilização de Software

Neste capítulo inicial, serão abordadas as principais questões relacionadas à area de reutilização de software. O objetivo será apresentar seus conceitos mais importantes, pontos positivos e negativos, bem como um esquema geral introduzindo o desenvolvimento de software *para* e *com* reutilização. Dessa forma, determinaremos o escopo desta dissertação dentro do contexto deste esquema de reutilização. E, por fim, serão apresentadas as características das principais tecnologias envolvidas, bem como a utilização de métodos formais no desenvolvimento de software sob o princípio de reutilização.

2.1 Introdução

A produção de sistemas de computação é um processo caro, que envolve grandes quantidades de recursos financeiros, materiais e humanos. A cada dia novos requisitos são impostos pelo mercado consumidor, exigindo produtos cada vez mais eficientes, de boa qualidade e confiabilidade. Os sistemas tendem a tornar-se maiores, tanto em tamanho de linhas de código, quanto em complexidade.

Apesar do surgimento das linguagens de programação de alto nível, das metodologias de desenvolvimento, e de outras ferramentas auxiliares, ainda há uma necessidade contínua por novos processos que sejam capazes de minimizar os custos e melhorar os resultados do desenvolvimento de sistemas computacionais [BR98]. Entre as várias áreas de pesquisa atuais, uma das que apresenta grande potencial para alcançar tais objetivos é a chamada 'Reutilização de Software'. Há muito tempo é reconhecido que um dos problemas fundamentais na criação de software é o fato de que os novos sistemas geralmente são construídos a partir do 'zero' [BR89, CDG94]. Todas as suas rotinas, funções e procedimentos são totalmente analisados e codificados a cada

novo projeto. Isto acaba se tornando uma situação inaceitável, visto que, como alguns estudos apontam [HM89], uma grande parte da estrutura e código de qualquer sistema apresenta similaridades com aplicações anteriormente desenvolvidas. Dessa forma, o objetivo básico da reutilização é aumentar a produtividade e a qualidade do processo de produção de software, identificando e reutilizando características comuns de conjuntos de tarefas similares.

O restante deste capítulo é organizado da seguinte maneira: na Seção 2.2 são explicados os conceitos, objetivos e problemas que envolvem a reutilização de software. Em seguida, na Seção 2.3 é detalhado um esquema geral: o processo de desenvolvimento de componentes 'para' e 'com' reutilização. Na Seção 2.4 são analisadas as principais características das tecnologias de reutilização. Na Seção 2.5 é analisada a importância da aplicação de métodos formais dentro desta área de reutilização. E, por fim, na Seção 2.6 é definido o escopo da dissertação, explicando o modelo empregado.

2.2 O que é Reutilização de Software?

A noção de reutilização é uma idéia antiga, que se faz presente na cultura humana desde que ela começou a ficar envolvida com a solução de problemas. Conforme as soluções para determinados problemas são descobertas, elas passam a ser aplicadas em outros problemas similares. Quanto mais esta solução é utilizada em problemas parecidos, mais ela se torna aceita, generalizada e padronizada. Em geral, quando nos deparamos com um problema, intuitivamente verificamos em nossa memória se já não resolvemos o mesmo problema anteriormente. Quando isto falha, iniciamos a procura por problemas similares, que poderiam fornecer soluções adaptáveis à situação atual. E, finalmente, se esta alternativa também falha, começamos a desenvolver uma solução original a partir de nosso conhecimento e habilidades [MMM95].

De fato, há inúmeras definições para o que seja reutilização de software. A maioria destas se concentra em definir apenas reutilização de componentes de código, ou seja, o nível mais baixo e específico de abstração de software. No entanto, no seu sentido mais amplo, reutilização de software pode ser descrita como a reaplicação de uma variedade de tipos de conhecimento sobre um sistema anteriormente desenvolvido a outro sistema similar, com o intuito de reduzir o esforço na produção e manutenção deste último. Este conhecimento reutilizável pode incluir todos os objetos produzidos e consumidos durante o processo de software, tais como: experiência no desenvolvimento, conhecimento do domínio, decisões de projeto, requisitos do sistema, estruturas arquiteturais, código, casos de teste e documentação [BP89a, McC97].

Do ponto de vista do desenvolvimento de software bem como nesta dissertação, a reutilização está relacionada somente ao processo de *construção* de sistemas computacionais [Kru92]. Utilizar

o código-fonte ou a estrutura de uma função durante a *construção* de um programa é um exemplo de reutilização; entretanto, as repetidas chamadas a uma mesma função durante a *execução* do programa não é considerado reutilização. Da mesma forma, a duplicação de um programa com o intuito de distribuição, também não pode ser chamado de reutilização. Assim, limitamos nossa visão em relação ao que seja reutilização somente à criação de novos sistemas a partir de estruturas desenvolvidas anteriormente.

Ao contrário das várias definições, os objetivos básicos da reutilização são bem definidos e aceitos pela maioria dos autores [BP89a, BP89b, SPM94]. Entre estes objetivos, podem ser citados:

- *Aumento da produtividade:* o esforço no desenvolvimento e manutenção dos produtos é reduzido quando componentes reutilizáveis são empregados. Ou seja, utilizam-se menos objetos para criar um sistema, mas alcança-se uma funcionalidade igual ou superior à de um sistema gerado sem reutilização.

- *Aumento da qualidade:* a qualidade do software melhora consideravelmente ao reutilizar-se componentes anteriormente testados. Com o uso frequente de tais componentes, aumenta-se a probabilidade de que os possíveis erros sejam encontrados e corrigidos. Além disso, o custo de se investir mais em depurações e testes do código reutilizável é minimizado através dos repetidos usos.

- *Redução dos custos:* o custo inicial da criação de componentes reutilizáveis é gradualmente amortizado à medida em que estes vão sendo utilizados. Por sua vez, a redução substancial do esforço necessário para o desenvolvimento e manutenção de um software leva naturalmente à redução nos custos de todo o processo.

- *Redução no tempo de entrega:* altos níveis de reutilização podem reduzir o tempo de desenvolvimento de uma aplicação e, consequentemente, de entrega ao mercado consumidor.

- *Padronização:* ao reutilizar os mesmos componentes, os quais seguem determinados padrões técnicos, o código e os sistemas produzidos tornam-se naturalmente padronizados.

- *Interoperabilidade/compatibilidade:* utilizar componentes padronizados garante que os sistemas apresentem comportamento comum, aumentando a interoperabilidade entre as aplicações.

- *Outros benefícios:* aumento da previsibilidade, confiabilidade e funcionalidade dos sistemas e redução de riscos.

Assim sendo, a aplicação da reutilização no processo de desenvolvimento de software traz uma série de benefícios potenciais. Porém, como toda nova tecnologia há vários problemas que ainda não apresentam uma solução definitiva, impedindo que a reutilização torne-se realmente eficiente e amplamente utilizada. Na próxima seção apresentamos seus principais problemas, sob a forma de dilemas da reutilização.

2.2.1 Dilemas da Reutilização

A reutilização apresenta alguns dilemas, como especificados em [BR89]. A forma geral de um dilema é que uma mudança positiva em um de seus parâmetros, geralmente leva a uma mudança negativa em outro parâmetro.

Generalidade da Aplicação Versus Poder Computacional

Tecnologias que são muito gerais, isto é, que podem ser aplicadas a uma grande variedade de domínios de aplicação, tendem a ter um poder computacional muito menor do que sistemas que são direcionados para atuarem sobre domínios de aplicação específicos. Em outras palavras, ferramentas específicas geralmente são mais fáceis de ser utilizadas, amplificando as capacidades do usuário e gerando um melhor retorno sobre o investimento. Por outro lado, apresentam uma menor capacidade de reutilização.

Para tornar a reutilização um processo realmente produtivo, é necessário que se encontre um ponto médio, de maneira a desenvolver ferramentas flexíveis, capazes de atuar sobre domínios de aplicação variados, mas que, ainda assim, mantenham um alto poder computacional. Na verdade, um sistema projetado para ser utilizado sem mudanças, em muitas situações apresenta-se extremamente ineficiente; o ideal é projetá-lo de forma flexível, sendo fácil de adaptá-lo a novas situações.

Tamanho do Componente Versus Potencial de Reutilização

Conforme um componente cresce em tamanho, os benefícios provenientes da sua reutilização também aumentam, uma vez que menos código precisará ser escrito e depurado. Entretanto, um componente grande se torna mais específico, o que reduz a possibilidade de reutilizá-lo em domínios variados, ao mesmo tempo que aumenta o custo de reutilizá-lo quando modificações são necessárias.

Este dilema é amplificado quando se tenta trabalhar com níveis mais baixos de abstração (código) visto que, por sua própria natureza, estes tendem a apresentar uma grande especificidade. Utilizar componentes pequenos para construir um sistema acaba gerando muitos problemas

na fase de integrar todas as partes num único bloco. Por sua vez, componentes grandes apresentam um baixo potencial de reutilização — sua especificidade reduz a probabilidade de que exatamente o mesmo conjunto de requisitos será novamente necessário — e aumentam o esforço para compreendê-los e adaptá-los a um novo sistema.

Quanto mais específicos os componentes se tornam, mais difícil fica encontrar um candidato com características que 'casem' exatamente com a especificação desejada. Assim, deve-se criar bibliotecas de software com uma grande quantidade de componentes para tornar a reutilização realmente viável. O objetivo, então, é fatorar esta especificidade, criando componentes suficientemente abstratos, onde cada um representa apenas um aspecto ou princípio de um problema.

O Custo da Criação

Antes que o processo de reutilização comece a trazer lucros de maneira significativa, deve-se investir muito tempo, capital intelectual e financeiro. O problema é que as organizações querem ver seus investimentos gerando lucros a curto-prazo. E isto dificilmente acontece quando se cria um projeto de reutilização; neste caso, seus benefícios — aumento da qualidade e produtividade — só podem ser percebidos a longo-prazo. Em outras palavras, o projeto que financiou o desenvolvimento de componentes para a reutilização, raramente é o mesmo projeto que usufruirá da disponibilidade de tais artefatos.

Faz-se necessário, por exemplo, a criação de grandes bibliotecas de componentes, o treinamento de pessoal especializado e uma reestruturação organizacional. Tudo isso leva tempo e necessita de acordo e cooperação em todos os níveis hierárquicos, o que geralmente é difícil de ser alcançado, visto que a maioria das pessoas é avessa a grandes mudanças. Neste sentido, um dos argumentos mais utilizados é de que a reutilização inibe a 'criatividade' [BR89]. Porém, na verdade, a reutilização permite que os analistas de software tenham mais tempo para atacar problemas desafiantes e rapidamente explorar soluções alternativas. Para que um projeto de reutilização realmente funcione, todos precisam saber da existência dos componentes, como encontrá-los e utilizá-los e, principalmente, têm de estar motivados a trabalhar com reutilização.

2.2.2 Outras Questões e Problemas...

Além dos dilemas anteriormente mencionados, várias questões de ordem técnica ainda precisam ser respondidas, antes que se possa desenvolver um ambiente para reutilização eficiente. Algumas destas questões podem ser encontradas em [Nei89]:

- *Classificação:* qual é a linguagem ou esquema apropriado para especificar e encontrar descrições sobre 'o que' cada componente faz? A reutilização perde o sentido caso haja

a necessidade de se examinar todo o código-fonte de um componente para verificar sua verdadeira funcionalidade.

- *Seleção:* como desenvolver esquemas de seleção eficientes, de maneira a simplificar o trabalho de vasculhar a biblioteca de componentes, à procura da função desejada? Para o usuário, deve ser mais fácil e rápido encontrar um componente e compreender seu funcionamento, do que construí-lo por completo.

- *Armazenamento:* como os componentes serão catalogados, para que possam ser facilmente localizados? Como criar bibliotecas de software fáceis de usar, precisamente caracterizadas e altamente confiáveis?

- *Especificação estrutural:* qual é a linguagem ou esquema apropriado para especificar descrições de 'como' o componente funciona? Em outras palavras, como descrever um componente, de forma que outras pessoas possam compreendê-lo de maneira clara e precisa?

- *Flexibilidade:* quais decisões de projeto e implementação devem ser mantidas 'fixas' ou então 'livres' (parametrizadas), em cada componente?

- *Integração:* este ponto pode ser dividido em duas linhas: (1) muitas vezes um único componente não é capaz de desempenhar a funcionalidade desejada [Gog86]; sendo assim, como combiná-lo com outros componentes de forma que juntos, alcançem o resultado esperado?; (2) como inserir o componente selecionado dentro de um sistema, de maneira simples e sem que afete a funcionalidade dos demais módulos?

Cada um destes pontos, por si só, representa um desafio que precisa ser amplamente pesquisado. Esta dissertação se concentrará basicamente nas 3 primeiras questões. Todo o esforço será direcionado aos problemas envolvidos com as fases iniciais de qualquer processo de reutilização: a classificação e a seleção de componentes de software. Neste caso, estamos interessados particularmente na reutilização formal de componentes concorrentes. A seguir, apresentaremos todo o processo de reutilização, dividindo-o em dois módulos básicos: o desenvolvimento *para* reutilização e o desenvolvimento *com* reutilização. Dentro de cada módulo, detalharemos suas fases principais.

2.3 O Processo de Reutilização

Quando se fala em reutilização, deve-se considerar dois aspectos do processo: o desenvolvimento *para* reutilização e o desenvolvimento *com* reutilização [RS94].

2.3.1 O Desenvolvimento *para* Reutilização

O desenvolvimento de software para reutilização está relacionado à reunião de componentes de software potencialmente reutilizáveis numa determinada biblioteca. Estes objetos podem ser obtidos dentro da própria organização, provenientes de projetos passados (reengenharia) ou desenvolvidos especificamente para serem reutilizados; ou então, podem ser oriundos de bibliotecas utilizadas por outras organizações. O importante é que esta seleção seja feita sob critérios rigorosos, que garantam a eficiência e a qualidade de cada componente. Assim, o desenvolvimento para reutilização envolve mecanismos para:

- Estabelecer critérios para a seleção dos componentes.
- Desenvolver novos componentes ou extraí-los de projetos passados, de acordo com os critérios estabelecidos.
- Qualificar e modificar os componentes para que possam fazer parte da biblioteca de software.
- Classificar e armazenar os componentes qualificados dentro da biblioteca.
- Formalizar, personalizar ou generalizar os componentes dentro da biblioteca, fornecendo um *feedback* de forma a melhorar os critérios existentes.

A seguir cada uma destas fases será explicada em maiores detalhes.

Estabelecimento de Critérios de Seleção

Ao final de cada projeto, cria-se uma noção dos tipos de objetos que podem ser efetivamente reutilizáveis e quais as suas características. Dessa forma, analisando suas características comuns e funcionamento, consegue-se estabelecer os critérios para determinar quais componentes deveriam tornar-se disponíveis na biblioteca de software. À medida que estes componentes vão sendo reutilizados e um *feedback* é fornecido, os critérios são refinados e personalizados à estrutura dos projetos da organização.

Da mesma forma, pode-se determinar quais componentes deveriam ser retirados da biblioteca. Componentes pouco (re)utilizados apenas 'incham' a biblioteca, tornando o processo de busca e seleção mais difícil e demorado.

Extração de Componentes

O objetivo da extração é obter os componentes a partir de projetos passados e, assim, identificar candidatos que possam aumentar a biblioteca de software. Para isto, faz-se necessário uma

descrição precisa das necessidades do usuário, baseada em modelos bem definidos que descrevam claramente a função de um componente; repositórios de projetos acessíveis e bem documentados onde possam ser feitas as pesquisas; e mecanismos eficientes para a extração dos componentes a partir de projetos antigos, de acordo com os critérios de seleção estabelecidos.

Qualificação/Modificação de Componentes

O objetivo da qualificação é garantir que os componentes se comportem conforme o desejado empregando, por exemplo, casos de teste ou métodos formais. Se necessário, alguma modificação pode ser feita nos mesmos, desde que o custo desta modificação seja compensado pelos retornos esperados com as reutilizações futuras. Neste caso, é necessária uma caracterização precisa dos candidatos extraídos, de forma que possam ser facilmente qualificados; uma especificação dos critérios de qualidade adotados pela organização; e mecanismos para a certificação e modificação dos componentes candidatos.

Classificação/Armazenamento de Componentes

O objetivo da classificação é criar uma biblioteca de software bem especificada e organizada de modo a facilitar o processo futuro de reutilização. Para isto, necessita-se de: uma caracterização precisa de cada componente candidato, descrevendo exatamente 'o que' ele faz; o projeto e a implementação de uma biblioteca eficiente; e mecanismos para o armazenamento, classificação e recuperação de componentes.

Manutenção de Componentes e *Feedback*

O objetivo da manutenção é aumentar o potencial de reutilização de cada componente, conforme mudam as necessidades do usuário ou surgem novas visões e perspectivas do que incentiva ou inibe a reutilização (critérios de seleção). Neste caso, necessita-se de meios para a identificação das necessidades do usuário e mecanismos para personalizar, generalizar e formalizar as características de cada componente. Personalizar significa direcionar o componente para executar uma tarefa específica ou para exibir atributos especiais, tais como tamanho e desempenho. Generalizar consiste em aumentar os domínios de aplicação associados a um componente. E, por fim, formalizar significa aumentar o potencial de reutilização de um componente, armazenando-o de maneira precisa e melhor compreendida. Neste ponto, devem ser levados em consideração os dilemas apresentados na Seção 2.2.1.

O *feedback*, por sua vez, tem como objetivo melhorar os critérios existentes para a reutilização, baseado em experiências passadas nas atividades de qualificação e manutenção.

2.3.2 O Desenvolvimento *com* Reutilização

O desenvolvimento de software com reutilização assume a existência de um número suficiente de componentes potencialmente reutilizáveis dentro de uma biblioteca. Dado as necessidades específicas de um usuário, este deve considerar a existência de componentes já existentes ao invés de começar a criá-los do 'zero'.

A questão a ser considerada é que a maioria da pessoas envolvidas com o desenvolvimento de software já pratica alguma forma de reutilização. Mesmo que inconscientemente, ao criar um novo sistema acabam reutilizando o conhecimento adquirido quando desenvolveram outros sistemas similares. Conhecimento este que engloba decisões de projeto, ferramentas, rotinas, procedimentos ou funções. O problema é que este tipo de reutilização é feito de maneira totalmente informal, limitada e ineficiente. Busca-se então, uma maneira mais formal e sistemática de reutilização, com definições precisas e, principalmente, padrões a serem seguidos. A idéia seria parecida com o esquema utilizado pelos produtores de componentes de hardware, ou seja, aqueles componentes que compartilham padrões arquiteturais similares (interfaces e sinais padrões) podem ser compostos de forma a criar uma estrutura maior de alto-nível. Basta procurar em um catálogo onde estão discriminados todos os componentes disponíveis, escolher o mais adequado à função procurada, e integrá-lo ao projeto em desenvolvimento [BR98]. Caso haja necessidade, os requisitos iniciais do projeto ou função podem ser modificados, para usufruir das vantagens oferecidas pelo componente. Este é o objetivo principal que está por trás do desenvolvimento de software com reutilização: criar componentes de software padronizados, que possam ser selecionados em uma biblioteca ou catálogo e integrados a um novo projeto. A diferença é que, devido à própria natureza do software que é mais simples de modificar do que um componente de hardware, provavelmente a estrutura do componente escolhido é que será alterada, ao invés dos requisitos iniciais do projeto. Sendo assim, o processo de desenvolvimento *com* reutilização deve envolver mecanismos para:

- Especificar as necessidades do usuário.
- Recuperar um conjunto de componentes candidatos de uma biblioteca de software, baseado no casamento de certas características relevantes às necessidades do usuário.
- Avaliar o potencial de cada candidato recuperado de acordo com o grau de satisfação das características relevantes.
- Selecionar o melhor componente candidato.
- Se necessário, modificar o componente selecionado.

- Integrar o componente modificado dentro do projeto em desenvolvimento.

A seguir cada uma destas fases será explicada em maiores detalhes.

Especificação das Necessidades do Usuário

O objetivo é determinar o tipo de componente que o usuário necessita e, assim, procurar na biblioteca de software, os componentes que 'casam' com a especificação fornecida. Em ambientes de desenvolvimento baseados em reutilização, a ação natural do usuário deve ser primeiramente especificar suas necessidades e, então, procurar por objetos anteriormente desenvolvidos e testados, ao invés de criá-los novamente.

Recuperação de Componentes Candidatos

O objetivo é encontrar um conjunto de candidatos com potencial para satisfazer as necessidades do usuário. A idéia não é apenas localizar componentes que 'casem' exatamente com a especificação (algo difícil de acontecer), mas também encontrar componentes candidatos similares ao desejado. Assim, mesmo que um candidato precise ser parcialmente reprojetado, o esforço gasto nesta tarefa geralmente é menor do que projetar o componente por completo. Como pode-se perceber, esta atividade está intimamente relacionada ao dilema descrito na Seção 2.2.1.

Para a recuperação de tais componentes, faz-se necessário uma clara especificação das necessidades do usuário; esquemas para organizar e classificar os componentes dentro da biblioteca; e mecanismos eficientes que executem o casamento exato ou parcial de padrões entre a especificação fornecida e os componentes candidatos.

Avaliação dos Componentes

Os objetivos são: avaliar o grau de discrepância entre a especificação do usuário e os componentes candidatos identificados, e determinar o custo necessário para modificar tais componentes de forma a satisfazer por completo a especificação. O primeiro objetivo pode ser alcançado através da obtenção de informações detalhadas como, por exemplo, especificações comportamentais dos componentes candidatos e da especificação fornecida. O segundo objetivo requer a disposição de informações que caracterizem o próprio processo de reutilização sendo utilizado e dados sobre experiências passadas em atividades similares.

Esta atividade é de fundamental importância dentro do processo de reutilização. Com ela, pode-se determinar o valor real de reutilizar um determinado componente ou então, desenvolvê-lo por inteiro, dependendo do custo e esforço necessários.

Modificação dos Componentes

O objetivo é modificar um dado componente candidato de modo a satisfazer a especificação fornecida pelo usuário, caso o custo desta modificação seja menor do que o de desenvolvê-lo a partir 'zero' (avaliado no item anterior). Assim, faz-se necessário uma caracterização precisa das necessidades do usuário bem como ferramentas que auxiliem na modificação.

O processo de modificação pode ser considerado a essência de um ambiente de reutilização. É através dele que uma biblioteca estática de componentes de software se transforma em um sistema dinâmico, repleto de componentes que geram novos sistemas à medida que os requisitos do ambiente se alteram.

Integração dos Componentes

O objetivo é integrar os componentes finais dentro do contexto do projeto em desenvolvimento. Isto requer a presença de uma estrutura básica que torne a integração possível, mas sem a necessidade de muitas modificações nos módulos já existentes.

2.3.3 Fábrica de Componentes

A Figura 2.1 mostra o esquema geral do desenvolvimento *para* reutilização e do desenvolvimento *com* reutilização, bem como suas interações através da biblioteca de componentes. Esta estrutura geral é denominada de 'Fábrica de Componentes' [CB91].

Um ponto a ser destacado é que a produção de componentes pode ser feita tanto de maneira síncrona quanto assíncrona. No primeiro caso (atividade síncrona), é criado um novo componente diante do pedido do usuário; isto deverá ocorrer sempre que o componente desejado não for encontrado na biblioteca ou quando o custo de modificar um componente similar é praticamente proibitivo. No segundo caso (atividade assíncrona), a fábrica fica constantemente produzindo novos componentes para reutilização, os quais considera que serão úteis em pedidos futuros. Quanto maior for a qualidade das atividades assíncronas, ou seja, quanto melhor for a capacidade de prever com precisão quais funções serão desejadas no futuro, maior será a eficiência da fábrica de componentes como um todo.

2.4 Níveis de Abstração

Até agora, procuramos definir o que é reutilização, alguns dos seus problemas e vantagens, e explicar as fases do seu processo. Nesta seção, ficaremos concentrados em questões mais técnicas,

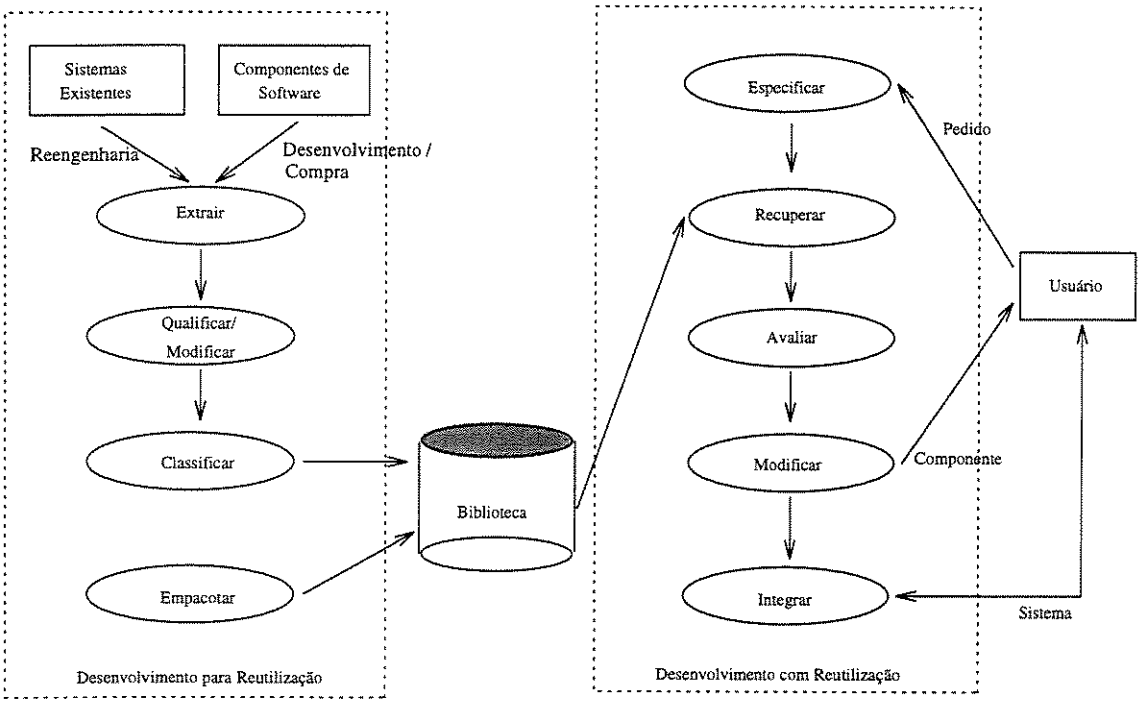

Figura 2.1: Estrutura geral de reutilização: fábrica de componentes

tais como, quais as tecnologias disponíveis, como funcionam, até que ponto diferem umas das outras e, principalmente, qual o nível de abstração empregado.

As tecnologias disponíveis atualmente para reutilização podem ser divididas em dois grandes grupos, de acordo com a natureza dos componentes sendo reutilizados: tecnologias de composição e tecnologias de geração [Kru92, MMM95, McC97]. A Tabela 2.1 mostra uma forma de classificação das tecnologias disponíveis [BR89].

Com as tecnologias baseadas em composição, o caso ideal seria que os componentes a serem reutilizados fossem atômicos, imutáveis e passivos, sendo compostos e integrados por agentes externos. Entretanto, na prática, os componentes geralmente sofrem algum tipo de modificação, seja para melhorar seu desempenho ou para alcançar a funcionalidade desejada. Após as devidas modificações, os componentes passam a se comportar como blocos de construção. O desenvolvimento de um novo programa torna-se então um problema de composição: como combinar/integrar os componentes de modo a criar um sistema funcionalmente eficiente? Exemplos desta estrutura incluem as bibliotecas formadas por componentes em código-fonte e os esquemas de software.

Características	*Abordagens*				
Componente reutilizado	Blocos de construção		Padrões		
Natureza do componente	Atômico e passivo		Difuso e ativo		
Princípio da reutilização	Composição		Geração		
Ênfase	Componentes em código-fonte	Parametrização de componentes	Geradores baseados em linguagens	Geradores de aplicação	Sistemas de transformação
Sistemas típicos	Bibliotecas de funções e procedimentos	Esquemas de software	VHLLs	Construtores de interface e processamento de informação	Transformadores de linguagens

Tabela 2.1: Classificação das tecnologias de reutilização

Componentes em Código-Fonte Este pode ser considerado o nível mais baixo de abstração e também o mais utilizado na prática [MMM97]. Exemplos de sucesso incluem as bibliotecas de funções matemáticas e estatísticas. Uma vez criados e catalogados, o custo inicial de tais componentes é amortizado a cada vez que são reutilizados.

Componentes de código precisam de abstrações bem definidas e ao mesmo tempo simples, explicitando suas funcionalidades ('o que' eles fazem). Também são necessárias ferramentas poderosas para classificá-los, selecioná-los, modificá-los e integrá-los de forma a criar novos sistemas.

Nesta dissertação, ficaremos concentrados neste tipo de tecnologia: a classificação e seleção do código-fonte de componentes de software (concorrentes ou sequenciais), devido a sua simplicidade e praticidade para o usuário final.

Esquemas de Software Esquemas são similares aos componentes em código-fonte, exceto que os objetos reutilizáveis são definidos num nível maior de abstração, permitindo uma gama maior de instanciações e composições. Além disso, o alto nível de parametrização torna possível a construção de estruturas mais complexas e úteis [MMM95].

A especificação abstrata de um esquema procura definir de maneira formal e parametrizada as estruturas de dados e algoritmos. Ao selecionar, instanciar — substituir as partes parametrizadas por construções de linguagem, fragmentos de código ou outros esquemas — e integrar tais especificações, acaba-se gerando o código-fonte desejado.

Os esquemas de software não são capazes de cobrir todas as necessidades de um domínio de aplicação. Além disso, suas especificações podem ser um tanto grandes e complexas dificultando o trabalho de localizá-los, compreendê-los e utilizá-los. Um exemplo que emprega esta abordagem é o sistema PARIS [KRT89].

As tecnologias baseadas em geração, por sua vez, são mais difíceis de caracterizar, visto que os componentes reutilizados não são realmente entidades concretas. Tais componentes são padrões e estruturas embutidas em um programa gerador. No caso dos geradores de aplicação, são reutilizados padrões de código e estruturas. Já com os sistemas de transformação os padrões reutilizados assumem a forma de regras de transformação. Assim, os sistemas resultantes se tornam altamente individuais, sendo pouco semelhantes entre si e com os programas que os geraram. Além dos geradores de aplicação e dos sistemas de transformação, são incluídas nesta categoria as VHLLs (*Very High-Level Languages*).

Geradores de Aplicação Os geradores funcionam como os compiladores de linguagens de alto-nível: recebem especificações de entrada produzidas pelo desenvolvedor, que são automaticamente mapeadas em programas executáveis. Eles diferem dos compiladores comuns no sentido de que suas especificações de entrada se apresentam num nível maior de abstração, provenientes de domínios de aplicação muito pequenos. Enquanto os esquemas de software reutilizam apenas projetos de algoritmos e estruturas de dados, os geradores de aplicação vão um passo além e reutilizam projetos completos de sistemas. Os algoritmos e estruturas de dados são automaticamente selecionados, fazendo uma clara separação entre a especificação de um sistema e sua implementação. Como resultado, o gerador é capaz de produzir um sistema quase completo e executável, enquanto os esquemas de software produzem apenas unidades menores, tais como um módulo ou subsistema [Kru92].

É muito complicado construir geradores com funcionalidade e desempenho apropriados para uma grande gama de aplicações. Assim, eles ficam limitados a pequenos domínios de aplicação, tornando difícil ou até mesmo impossível encontrar um gerador capaz de solucionar um determinado problema de desenvolvimento. Exemplos típicos são os sistemas Lex e Yacc do Unix, os quais produzem parsers.

VHLLs Como o próprio nome sugere, as VHLLs (*Very High-Level Languages*) são uma tentativa de aumentar o sucesso das linguagens de alto-nível convencionais (HLLs). Elas permitem que os desenvolvedores criem sistemas executáveis utilizando construções que são consideradas especificações semânticas de alto-nível com relação às HLLs. São parecidas com os geradores de

aplicação, no sentido de que especificações abstratas de entrada são automaticamente traduzidas em sistemas executáveis. A grande diferença é que enquanto os geradores utilizam abstrações específicas a um domínio de aplicação, as VHLLs empregam abstrações matemáticas de propósito geral e independentes de aplicação: construções de linguagem parametrizadas são especializadas através de substituição recursiva por outras construções [Kru92].

O desempenho dos sistemas em tempo de execução escritos utilizando-se uma VHLL geralmente é muito baixo. Isto porque quanto mais geral é uma tecnologia de reutilização, menor é seu desempenho computacional (Seção 2.2.1). Neste caso, um alto nível de automação é possível, ao custo de uma baixa eficiência de código e qualidade de projeto. Além disso, abstrações matemáticas são um tanto difíceis de serem entendidas e escritas por usuários que não são especializados no assunto.

Sistemas de Transformação Com os sistemas de transformação, o desenvolvimento de software ocorre em duas etapas distintas [Kru92]:

1. Os usuários descrevem o comportamento semântico do software, utilizando uma linguagem de especificação de alto-nível.

2. Os usuários então aplicam regras de transformação a estas especificações.

A primeira fase é equivalente a utilizar uma VHLL, porém num nível ainda mais alto de abstração. Já a segunda fase é mais parecida com uma compilação interativa e direcionada pelo usuário. O objetivo é criar um sistema executável que satisfaça a especificação e também apresente um desempenho comparável a uma implementação em uma linguagem de alto-nível convencional (HLL).

Da mesma forma que as VHLLs, um dos problemas é a dificuldade de se entender e escrever especificações abstratas. Em muitos casos, as especificações são mais complicadas de se ler do que os próprios programas resultantes [HM89]. Além disso, aplicar tais transformações é um trabalho difícil, que requer muito tempo e esforço por parte do usuário.

Uma classificação possível destas duas abordagens diz respeito ao nível de automação, ou seja, o esforço necessário ao usuário para gerar um sistema executável, partindo de uma especificação abstrata. De acordo com esta idéia, as tecnologias podem ser assim ordenadas, onde a primeira é aquela a apresentar o maior poder computacional:

1. Geradores de aplicação;

2. VHLLs;

3. Sistemas de transformação;

4. Esquemas de software;

5. Componentes em código-fonte.

É interessante notar que a classificação das tecnologias mais gerais, ou seja, aquelas que podem ser utilizadas em domínios de aplicação variados, é praticamente o contrário da anteriormente apresentada. Isto se encaixa perfeitamente com o dilema apresentado na Seção 2.2.1:

1. Componentes em código-fonte;

2. Esquemas de software;

3. VHLLs;

4. Sistemas de transformação;

5. Geradores de aplicação.

Obviamente o ideal seria encontrar uma tecnologia que tivesse um alto poder computacional e nível de abstração e, ainda assim, pudesse ser utilizada em todos os domínios de aplicação. Porém, esta tecnologia perfeita ainda é um sonho que, por enquanto, está distante de se tornar realidade. Com este intuito, várias pesquisas vêm sendo desenvolvidas, dentre as quais aquelas que procuram agregar as características dos métodos formais ao processo de reutilização. Apesar dos problemas relacionados ao formalismo, alguns dos quais já mencionados em várias das tecnologias anteriores, esta é uma das abordagens que apresenta grande potencial para alcançar o nível de abstração, qualidade e confiabilidade necessários à reutilização. Assim sendo, no desenvolvimento desta tese procuramos empregar um certo nível formal, que será detalhado no Capítulo 4. Antes, porém, serão apresentados os problemas e as vantagens da utilização de tais métodos em reutilização, principalmente para a especificação e seleção de componentes, bem como alguns trabalhos nesta área.

2.5 Métodos Formais no Processo de Reutilização

Como já foi citado no início da Seção 2.3.2, as pessoas envolvidas com o desenvolvimento de software já praticam há muito tempo, de maneira informal, alguma forma de reutilização. O problema é que este processo é ineficiente e pouco produtivo, visto que se baseia principalmente

na procura por soluções anteriores armazenadas em suas memórias. Dessa forma, surge uma grande necessidade de se desenvolver metodologias capazes de tornar a reutilização de software uma prática sistemática e formal. Assim, o emprego de métodos formais se apresenta como uma alternativa importante, servindo de base à reutilização de todo artefato que é gerado dentro do processo de desenvolvimento de software.

A partir do momento em que os métodos formais estiverem plenamente inseridos no processo de reutilização, uma série de vantagens poderão ser observadas [CDG94]. A primeira vantagem é que tal representação formal do conhecimento evita a ambiguidade das representações que não são baseadas em uma semântica formal. A funcionalidade de cada componente é especificada de maneira clara e precisa, sendo bem compreendida por todos os usuários. O mais importante é indicar o comportamento geral do componente ('o que ele faz') e abstrair os vários detalhes de implementação ('como ele faz'). A segunda vantagem é que a semântica formal permite o desenvolvimento de um raciocínio matemático sobre a corretude dos objetos especificados, sendo que este raciocínio pode ser feito de forma manual, ou preferivelmente, por ferramentas automáticas. Assim, as atividades de selecionar, analisar, modificar e integrar os componentes são executadas de maneira mais confiável, havendo a possibilidade de se desenvolver refinamentos e provas formais de suas corretudes. E uma terceira vantagem é que, dependendo do método utilizado, a especificação formal de um sistema permite uma prototipação rápida da sua funcionalidade, reduzindo ainda mais o tempo de desenvolvimento gasto com componentes reutilizáveis.

Por um lado, o processo de reutilização fornece uma base econômica para a aplicação dos métodos formais, o que acaba derrubando a afirmação de que o uso de tais formalismos encarece o custo do processo de desenvolvimento de software. Isto porque, o uso repetitivo dos componentes reduz os altos custos iniciais do desenvolvimento de uma descrição formal, ou seja, o retorno acaba surgindo conforme a reutilização vai sendo praticada sistematicamente. Por outro lado, os métodos formais fornecem a base para o aumento da qualidade no processo de desenvolvimento e nos sistemas gerados, o que representa um dos principais ganhos esperados com a reutilização. O fato é que a representação formal de componentes permite o raciocínio sobre sua corretude e, no caso ideal, é livre de falhas. Como consequência, a reutilização de tais componentes também não deverá produzir outros erros. Além disso, devido à esta semântica inambígua é possível determinar se o componente reutilizável se enquadra em outros contextos ampliando, assim, os domínios de aplicação associados a uma representação. Em resumo, os métodos formais aumentarão a qualidade e a produtividade do processo de reutilização, representando-o de maneira mais sucinta e precisa.

Até agora, apresentamos apenas as vantagens de se reutilizar componentes formalmente especificados. Entretanto, vários problemas também existem e ainda estão um pouco distantes de uma solução eficiente. Primeiro, há a dificuldade em se trabalhar com especificações formais. Como este é um paradigma novo para muitas pessoas e baseado principalmente em conceitos matemáticos, há sempre uma certa resistência natural em se trabalhar com esta forma de desenvolvimento. Para resolver esta situação, necessita-se aplicar em treinamento, mostrando que todos já possuem o embasamento matemático inicial, o qual precisa apenas ser adaptado ao novo contexto. Outro ponto muito questionado é que as especificações formais são difíceis de entender e escrever. Novamente aqui há a resistência dos usuários que precisa ser ultrapassada e, além disso, a necessidade do desenvolvimento de ferramentas integradas que facilitem o trabalho de criar as especificações. Dependendo da abordagem utilizada, o processo de busca e seleção de componentes pode tornar-se excessivamente lento. Como na questão anterior, há a necessidade da criação de algoritmos e ferramentas mais poderosas, principalmente no que diz respeito aos provadores de teorema. E, por fim, há sempre o problema de que existem poucas aplicações realmente práticas. A maioria das pesquisas se concentra em apresentar apenas exemplos acadêmicos, tais como a utilização de bibliotecas muito pequenas ou especificações de componentes muito simples. Neste caso, deve haver um maior investimento em pesquisas variadas, que aumentarão a gama de aplicações tanto de ordem teórica quanto prática. Estas são apenas algumas das questões mais relacionadas com as fases de especificação e seleção de componentes, que são o alvo desta dissertação. Várias outras questões, abordando o emprego de métodos formais em geral, podem ser encontradas no trabalho de Bowen e Hinchey [BH94].

No meio acadêmico, existe uma grande quantidade de trabalhos já desenvolvidos na área de especificação e seleção formal de componentes de software. Com relação à aplicação de métodos formais para a construção de bibliotecas de software, Jeng e Cheng [JC93, JC95] utilizam a lógica de predicados para especificar os componentes, avaliar o nível de similaridade entre suas funções e criar relações de hierarquia. No que diz respeito ao processo de seleção, Rollins e Wing [RW91] utilizam especificações formais escritas em λProlog como chaves de busca em bibliotecas de software. A seleção é feita é duas etapas, com um casamento de interface e depois, o casamento de especificação, propriamente dito. Zaremski e Wing [ZW95b, ZW95a] empregam uma idéia semelhante fazendo a seleção também através de dois filtros: casamento de interface e casamento de especificações, escritas utilizando-se a linguagem Larch/ML. O interessante deste trabalho é que apresentam vários níveis de casamento, recuperando funções simples ou módulos compostos, de maneira exata ou em variados estados de 'relaxamento'. Outro trabalho semelhante é desenvolvido por Fischer *et.all* [FKS94], que faz uma primeira filtragem através

de interfaces e depois utiliza o provador de teoremas OTTER para fazer o casamento das especificações escritas em VDM. Penix e Alexander [PA97, PA95] utilizam uma idéia um pouco diferente das anteriores, criando uma classificação baseada em atributos, a partir de especificações formais. Estes atributos são automaticamente associados a cada componente, baseado em condições necessárias que são implicadas por suas especificações. Uma vez classificados, os componentes são recuperados através do casamento sintático dos conjuntos de atributos.

Todos estes trabalhos ainda estão em fase de desenvolvimento e testes. Dessa forma, ainda sofrem muito dos problemas anteriormente citados. Porém, cada um a seu modo é um grande passo em direção à popularização da reutilização dentro do ambiente de desenvolvimento de software.

2.6 Modelo Proposto

Nesta dissertação trabalharemos em ambas as estruturas de reutilização que foram apresentadas na Seção 2.3. Com relação ao desenvolvimento *para* reutilização, o objetivo será fornecer um mecanismo para classificar e armazenar os componentes de software. Já no desenvolvimento *com* reutilização, apresentaremos um modelo para especificar as necessidades do usuário e recuperar componentes-candidatos. Assim, ficaremos concentrados nas fases de classificação e seleção de componentes. Um ponto importante é que neste nosso modelo os componentes classificados poderão apresentar tanto processamento sequencial quanto concorrente.

Os componentes envolvidos serão representados no nível mais baixo de abstração, ou seja, em código-fonte (Seção 2.4). No entanto, a idéia não é formar uma biblioteca de fragmentos de código-fonte isolados. Ao contrário, neste modelo nossa biblioteca será composta apenas por ponteiros especificando as características funcionais de cada componente. Estes ponteiros então indicarão a localização física dos componentes, juntamente com outras informações relevantes como suas especificações, projeto, testes e documentação.

Tais componentes em código-fonte apresentam alguns problemas (Seção 2.2.1), principalmente devido à ambiguidade e rigidez inerentes ao software. No entanto, sua semântica e em geral sua sintaxe, são bem compreendidas por qualquer usuário, ao contrário dos métodos puramente formais. Além disso, o custo inicial de desenvolvê-los e classificá-los é reduzido gradativamente, conforme vão sendo reutilizados em novos sistemas. Assim, o retorno esperado em se trabalhar com este nível de abstração acaba compensando os custos das fases iniciais de desenvolvimento e especificação.

Porém, as vantagens de se utilizar uma abordagem formal também não podem ser ignoradas. Uma das maneiras de se desenvolver sistemas computacionais de qualidade e confiabilidade é

através do emprego de métodos formais (Seção 2.5). Assim sendo, neste modelo procuramos desenvolver uma abordagem mista, empregando fragmentos de código e também especificações formais. Criamos um sistema para a seleção de componentes de software concorrentes, baseado em dois filtros: (1) classificação por atributos (Capítulo 3) e (2) casamento de interfaces (Capítulo 4). O primeiro filtro é baseado em atributos e o segundo em especificações formais escritas em LOTOS.

2.7 Conclusão

A reutilização de software se apresenta como uma área de pesquisa que possui grande potencial para alcançar um alto nível de qualidade, produtividade e confiabilidade no processo de desenvolvimento de sistemas computacionais. A idéia principal é criar novos sistemas a partir da utilização de componentes anteriormente desenvolvidos e testados, reduzindo consideravelmente o tempo e o esforço nesta atividade.

Neste capítulo inicial, procuramos fornecer uma visão geral e introdutória da área de reutilização, bem como definir o escopo da dissertação dentro deste contexto.

Foi apresentada uma definição concreta do que seja reutilização, juntamente com seus objetivos, vantagens e problemas. Definimos um esquema geral de reutilização, dividindo-o em duas partes distintas: desenvolvimento *para* reutilização e desenvolvimento *com* reutilização. Para cada parte, explicamos as várias fases envolvidas no processo. Apresentamos seus variados níveis de abstração, representados nas duas abordagens atualmente disponíveis: tecnologias de composição e tecnologias de geração. Apresentamos os pontos positivos e negativos de se utilizar uma abordagem formal no processo de reutilização, principalmente no que diz respeito à especificação e seleção de componentes. Mostramos que o emprego do formalismo é uma necessidade e que há soluções para seus principais problemas, comprovadas através das várias aplicações e metodologias já desenvolvidas. E, finalmente, propomos um novo modelo para a reutilização, no qual o restante da dissertação está baseado.

No capítulo seguinte, explicaremos o primeiro filtro de nosso sistema de classificação e seleção de componentes. Este filtro é baseado em um esquema que classifica os componentes de acordo com atributos que descrevem sua funcionalidade e seu processamento.

Capítulo 3

Classificação de componentes

Selecionar componentes de software de uma grande biblioteca, sem o apoio de ferramentas computacionais, é uma tarefa praticamente inviável. Organizar os componentes através de um esquema de classificação é muito útil como um primeiro filtro no processo de seleção.

Neste capítulo apresentaremos uma extensão ao sistema de classificação por atributos de Prieto-Díaz [Pri85]. O objetivo será adaptar o esquema de modo a dar suporte à classificação de componentes concorrentes.

Para tanto, descreveremos as principais características de um sistema de classificação; detalharemos o esquema de classificação por atributos e como estendê-lo de maneira a englobar o domínio dos componentes de software concorrentes; e apresentaremos o primeiro filtro de nossa ferramenta de apoio ao processo de seleção, baseado em um dicionário de termos e uma rede semântica. O trabalho será concentrado na modificação da rede semântica, de modo a torná-la uma ferramenta mais simples e eficiente para a seleção de componentes parcialmente similares aos desejados pelo usuário.

3.1 Introdução

Um dos grandes problemas na área de reutilização de software é conseguir organizar coleções de componentes reutilizáveis, de modo que sejam facilmente localizados e recuperados. Para tornar a reutilização uma abordagem prática que auxilie no desenvolvimento de software, é essencial que se desenvolvam maneiras eficientes de classificar grandes coleções de procedimentos, rotinas e outras unidades funcionais. Caso contrário, o significado da reutilização perde o sentido, pois os processos de seleção, compreensão e modificação ficam tão lentos e ineficientes que se torna mais simples desenvolver um componente do 'zero' do que localizá-lo em uma biblioteca.

Ao se projetar uma estrutura de classificação bem definida, reduz-se o esforço necessário para 'vasculhar' uma biblioteca de componentes. Com o emprego de um esquema para discriminar cada componente, o espaço de busca torna-se menor, o que acaba naturalmente aumentando a velocidade dos processos de busca e recuperação.

Nesta dissertação utilizamos um método de classificação baseado no trabalho desenvolvido por Prieto-Díaz [Pri85, PF87, Pri91]. Este método, aqui denominado de classificação por atributos[1], apresenta duas restrições principais:

1. Classifica apenas componentes sequenciais;

2. Emprega um método de análise da proximidade entre termos extremamente complexo.

Com base nos aspectos acima citados, estendemos o método de forma a conseguir também classificar componentes concorrentes e analisá-los de maneira mais simples. Para tanto, removemos alguns atributos e criamos outros novos, os quais consideramos mais adequados ao novo contexto de concorrência, e modificamos consideravelmente o grafo de proximidade[2] entre termos, de forma a torná-lo mais simples e intuitivo. Apesar de existirem outros esquemas de classificação, este método empregando atributos apresenta-se mais simples e flexível, sendo útil como um primeiro filtro para reduzir o espaço de busca — número de componentes a serem analisados — em bibliotecas extensas.

O restante do capítulo é organizado como segue. A Seção 3.2 explica alguns princípios de classificação, e as características e diferenças entre os métodos de classificação por atributos e enumerativo. A Seção 3.3 descreve o processo e os atributos selecionados para a classificação de componentes de software concorrentes. A Seção 3.4 descreve o primeiro filtro de nossa ferramenta de classificação, juntamente com os dois conceitos sobre os quais é baseada: o dicionário de termos sinônimos e a rede de proximidade semântica entre termos. A Seção 3.5 explica o processo de recuperação e fornece uma visão geral de todo o esquema. E, finalmente, as conclusões obtidas são apresentadas na Seção 3.6.

3.2 Princípios de Classificação

O conceito de classificação carrega implicitamente a noção de agrupar entidades similares. Todos os membros de um grupo (ou classe) produzidos por um esquema de classificação compartilham, no mínimo, uma característica que todos os membros das demais classes não possuem. Dessa

[1] *Faceted classification* no original.
[2] *Weighted conceptual graph* no original.

forma, a classificação reflete os relacionamentos entre os objetos e entre as classes de objetos, tendo como resultado a formação de uma rede ou estrutura de relacionamentos [PF87].

Um método de classificação é basicamente um mecanismo empregado para produzir uma determinada ordem entre os objetos de uma coleção, baseado em um vocabulário controlado e estruturado. Este vocabulário é constituído por um conjunto de termos, representando conceitos ou classes, apresentados de forma a mostrar explicitamente os relacionamentos entre as classes. Estes relacionamentos podem ser divididos em dois tipos básicos: hierárquicos ou sintáticos [PF87]. Os relacionamentos hierárquicos são baseados no princípio de subordinação ou inclusão, onde o conjunto de termos formam um grafo composto por escalas hierárquicas. Já os relacionamentos sintáticos são aqueles criados para relacionar dois ou mais conceitos pertencentes a diferentes hierarquias, formando classes compostas de termos. Um exemplo é a classe composta 'multiplicação de inteiros', que relaciona o termo 'multiplicação' pertencente à classe 'Função' com o termo 'inteiros' da classe 'Objeto'. Existem ainda outros métodos de classificação, baseados em um vocabulário não-controlado, ou seja, que não impõem nenhum tipo de restrição sobre os termos ou a sintaxe que podem ser utilizados para descrever um objeto. Um exemplo desta abordagem é o sistema GURU [MBK91], onde os componentes são automaticamente indexados, extraindo-se informações de afinidade léxica e de frequência de termos diretamente da documentação do software em linguagem natural. No entanto, este tipo de classificação é ineficiente devido a problemas de ambiguidade semântica e, por isso, pouco utilizado. Mais informações sobre estes métodos alternativos podem ser encontradas em [FG90, FP94].

Os esquemas de classificação que empregam um vocabulário controlado e baseados em classes, podem ser subdivididos em dois grupos: enumerativos ou por atributos. A seguir, explicaremos cada esquema, juntamente com suas diferenças.

3.2.1 Classificação Enumerativa Versus Classificação por Atributos

O método enumerativo pressupõe a divisão dos termos em classes sucessivamente menores, que incluem todas as possíveis classes compostas. Estas classes são então ordenadas de modo a mostrar seus relacionamentos hierárquicos. Um sistema de classificação enumerativo deve satisfazer quatro critérios [FG90]:

1. Todas as possíveis classes devem ser previamente definidas.
2. As classes devem ser mutuamente exclusivas;
3. As classes não podem ser combinadas para formar novas classes;
4. As classes devem possuir ao menos uma ordem hierárquica parcial;

A principal vantagem deste sistema é sua estrutura hierárquica. Uma estrutura bem definida torna mais fácil aos usuários identificar e interpretar os relacionamentos entre os termos. Os usuários podem modificar suas consultas, tornando-as mais gerais ou específicas, simplesmente movendo-se através da hierarquia. Entretanto, a força deste sistema — sua estrutura rígida — também é sua principal desvantagem. Criar uma estrutura de classificação bem definida, exige muita experiência e análise exaustiva do domínio de aplicação. Da mesma forma, discriminar um novo objeto exige muito conhecimento da área, para assim conseguir decidir qual seria a classe mais apropriada para representar cada objeto. Além disso, modificações neste sistema são difíceis e consomem muito tempo, uma vez que as mudanças não podem ser feitas sem uma reestruturação geral do sistema. Em aplicações de software, onde os domínios e a terminologia mudam constantemente, esta torna-se uma limitação considerável.

O método por atributos, por sua vez, inicialmente identifica vários termos básicos que melhor descrevem o universo de discurso. Em seguida, estes termos são conceitualmente agrupados na forma de conjuntos de atributos, os quais devem ser ordenados de acordo com sua importância para os usuários. Os objetos são então classificados através da escolha de termos pré-definidos provenientes das listas de atributos. Assim, as classes compostas são expressas a partir da identificação de suas classes elementares, sendo derivadas através da síntese de termos básicos provenientes de atributos diferentes. Dessa forma, os atributos podem ser considerados como perspectivas, pontos-de-vista ou dimensões dentro de uma área de interesse. Um exemplo de classificação utilizando o esquema por atributos é apresentado em [PF87]:

(Atributo Processo)
- Fisiologia
  - Respiração
  - Reprodução

(Atributo Habitat)
- Animais Marinhos
- Animais Terrestres

(Atributo Taxonomia)
- Invertebrados
  - Insetos
- Vertebrados
  - Répteis

O mesmo exemplo, porém empregando o método de classificação enumerativo, pode ser descrito parcialmente por [PF87]:

Fisiologia
- Respiração
- Reprodução

Animais Marinhos
- Fisiologia dos Animais Marinhos
  - Respiração dos animais marinhos
  - Reprodução dos animais marinhos

Animais Terrestres
- Fisiologia dos Animais Terrestres
  - Respiração dos animais terrestres
  - Reprodução dos animais terrestres

Invertebrados
- Fisiologia dos invertebrados
- *etc...*

Os dois esquemas de classificação podem representar o mesmo número de classes. A grande diferença é que com o método enumerativo deve-se encontrar entre as classes pré-definidas, aquela que melhor descreve o objeto a ser classificado. Isto acaba se tornando uma tarefa muito subjetiva e propensa a erros, principalmente quando se trabalha com um grande número de classes. Por sua vez, no método por atributos o objeto é relacionado a alguns termos básicos escolhidos dentro de cada atributo que, ao serem compostos, criam uma classe personalizada para descrevê-lo. Esta capacidade de compor classes a partir da síntese de termos básicos é a principal característica que distingue os dois métodos.

Continuando com o exemplo anterior, iremos expandir a classificação, através da inclusão de novos termos. Após incluir o termo 'nutrição' no atributo 'Processo' e os termos 'mamíferos', 'ursos' e 'pardos' no atributo 'Taxonomia', a estrutura de classificação por atributos apresenta-se da seguinte forma [PF87]:

(Atributo Processo)

Fisiologia

Respiração

Reprodução

Nutrição

(Atributo Habitat)

Animais Marinhos

Animais Terrestres

(Atributo Taxonomia)

Invertebrados

Insetos

Vertebrados

Répteis

Mamíferos

Ursos

Pardos

Para classificar o título 'Necessidades nutricionais dos ursos pardos', por exemplo, deve-se primeiro escolher uma ordem de citação entre os atributos, de acordo com a importância aos usuários da coleção. Esta ordem definirá o formato das tuplas de identificação de cada título, ou em outras palavras, a sequência dos termos dentro das tuplas. Neste exemplo, a ordem escolhida foi Processo → Habitat → Taxonomia.

A seguir, verifica-se em cada atributo quais os termos mais relacionados ou que melhor descrevem o título a ser classificado. Neste caso, selecionamos o termo 'nutrição' do atributo 'Processo' e o termo 'pardos' do atributo 'Taxonomia'; com relação ao atributo 'Habitat', não há nenhum termo que possa ser relacionado ao título. Dessa forma, a classificação resultante será 'nutrição/pardos'. Ao contrário, num esquema enumerativo, a menos que uma classe composta tal como 'Nutrição dos Ursos' esteja especificada, haverá a necessidade de se fazer uma escolha subjetiva entre as classes 'nutrição' ou 'ursos', de modo a realizar a classificação.

Assim, devido à facilidade para atualizar e modificar os atributos de maneira independente uns dos outros, este esquema apresenta-se mais flexível e adequado para dar suporte a coleções em contínua expansão. Isso representa uma grande vantagem, significando que nem todas as

classes precisam ser especificadas ao se criar o esquema inicial. No caso da estrutura mais rígida de um esquema enumerativo, as mudanças não podem ser realizadas sem uma completa reestruturação no sistema, o que exige que todas as classes sejam definidas previamente.

No entanto, como todo método de classificação, o esquema por atributos também apresenta alguns problemas. Um sistema de classificação com um grande número de atributos e termos pode ser difícil de utilizar de maneira eficiente. Os usuários podem ter dificuldades para, no meio de uma grande coleção de atributos, encontrar a combinação correta de termos básicos que melhor descrevem o objeto de interesse [FG90]. Da mesma forma que no método enumerativo, um especialista precisa criar o vocabulário controlado, procurando pelos atributos e termos que melhor descrevem a área de interesse. Isto inevitavelmente acaba exigindo um grande gasto de tempo e esforço manual.

Além desses problemas, há algumas limitações devido ao emprego de um conjunto finito de termos. Como alguns estudos apontam, ao se utilizar um vocabulário controlado cria-se uma ruptura natural entre precisão[3] e *recall*[4] [Sal86]. Precisão é a proporção de componentes relevantes ao usuário que são recuperados dentre o total de componentes recuperados; *recall* indica a proporção de componentes relevantes que são recuperados dentre o total de componentes relevantes presentes na biblioteca. De maneira geral:

$$Precisão = \frac{|Rel \cap Rec|}{|Rec|} \quad e \quad Recall = \frac{|Rel \cap Rec|}{|Rel|}$$

onde Rel corresponde ao conjunto de componentes relevantes presentes na biblioteca e Rec é o conjunto de componentes recuperados pelo usuário. No caso ideal, $Rec = Rel$. Entretanto, na maioria dos casos uma precisão alta significa um *recall* baixo e vice-versa. Isto ocorre porque as pessoas utilizam uma grande variedade de termos para identificar o mesmo objeto, criando a necessidade do emprego massivo de sinônimos para obter um *recall* considerável o que, por sua vez, diminui a precisão [FLGD87]. Esta imprecisão pode ser reduzida trabalhando-se com esquemas de classificação específicos a um domínio, como observado por Prieto-Díaz [Pri91], o que acaba obviamente excluindo a idéia de bibliotecas heterogêneas e de uso geral.

Apesar destes problemas, a flexibilidade e a expressividade de um esquema por atributos torna-o um dos sistemas de classificação mais adequados para dar suporte à reutilização de componentes de software. Neste domínio, onde as necessidades mudam rapidamente e os conceitos não são bem definidos, o mais importante é a facilidade de se modificar o que foi previamente definido. Portanto, esta é a razão principal pela qual o primeiro filtro de nosso sistema baseia-se

[3] *Precision* em inglês.

[4] Mantivemos a palavra em inglês, pois não encontramos nenhuma tradução em português que tivesse um significado semelhante.

no esquema de classificação por atributos de Prieto-Díaz.

3.3 Classificação de Componentes de Software

Como foi descrito na Seção 3.2.1, os atuais esquemas de classificação enumerativos apresentam uma série de problemas, tais como:

- Falta de flexibilidade;
- Dificuldade para manipular e atualizar;
- Falta de precisão;
- Necessidade de um grande conhecimento sobre a estrutura de classificação;
- Necessidade de um grande conhecimento sobre o domínio do problema.

Dessa forma, não satisfazem alguns dos critérios básicos que um esquema de classificação para bibliotecas de componentes de software reutilizáveis, realmente útil, deveria apresentar [Pri91]:

1. Suporte a coleções em contínua expansão.
2. Capacidade para encontrar componentes 'altamente similares', e não apenas casamentos exatos.
3. Capacidade para encontrar componentes funcionalmente equivalentes.
4. Precisão e alto poder descritivo.
5. Facilidade de manutenção, isto é, capacidade para adicionar, eliminar e atualizar a estrutura de classes e o vocabulário de termos, sem a necessidade de gastar muito esforço nestas atividades.
6. Fácil utilização tanto pela pessoa que vai realizar a classificação (bibliotecário), quanto pelo usuário final.
7. Fácil automação.

O esquema de classificação por atributos satisfaz com certa eficiência a maioria destas exigências [Pri91]. A expansão e a manutenção (1) são facilmente executadas pela adição e atualização das listas de atributos. Precisão e poder descritivo (4) são consequências diretas do esquema de classificação utilizado: classes são compostas simplesmente selecionando-se alguns

termos de listas de atributos. Uma lista consistente de termos para cada atributo fornece um vocabulário padrão, que é comum ao bibliotecário e ao usuário-final (6), enquanto que seu formato simples e tabular, facilita a implementação, como por exemplo, empregando um banco de dados relacional (5,7). E, por fim, um dicionário e uma rede semântica de termos, os quais serão explicados nas Seções 3.4.1 e 3.4.2, respectivamente, ajudam a medir o grau de similaridade entre os termos (2,3) e facilitam a formulação de consultas e a manutenção (2,5,6).

Dessa forma, o esquema utilizado descreve os componentes de software através de um vocabulário padrão; impõe uma ordem de citação para os atributos; e fornece uma métrica conceitual de distâncias (rede) entre os termos que formam cada atributo, auxiliando na seleção de componentes relacionados ou similares. A seguir, este esquema será apresentado de uma forma mais geral.

3.3.1 Generalização de Identificadores

Definimos cada atributo A_n como sendo um conjunto de termos $\{t_{n1}, t_{n2}, \ldots, t_{nj}\}$, onde j é finito. Todo componente de software α tem um identificador particular i_α que é definido como uma tupla [Pri85]:

$$i_\alpha = < v_1, v_2, \ldots, v_n >$$

tal que α é uma instância de i_α; cada v_n é um termo do atributo A_n; e todo $v_n = t_{nj}$.

Nosso universo de identificadores é definido como o conjunto I resultante do produto cartesiano dos n atributos:

$$i_\alpha \in I = A_1 X A_2 X \ldots X A_n$$

Conforme o número n de atributos aumenta, também cresce o poder descritivo dos identificadores, ou seja, aumenta-se o nível de detalhes na descrição dos componentes. Quando n é muito pequeno, a precisão diminui.

Seja U o universo de todos os componentes disponíveis numa biblioteca; uma função de seleção $X : i_\alpha \rightarrow 2^U$ de um identificador $i_\alpha = < v_1, v_2, \ldots, v_m >$ é definida como [Pri85]:

$$X(i_\alpha) = \{x \mid x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots \wedge x_m = v_m\}$$

onde $X \subseteq U$; α é uma instância de algum membro de X; e U e I são independentes. Nós chamamos X de *função de casamento exato.* Neste tipo de função, só são selecionados aqueles componentes cujos atributos de identificação são exatamente iguais aos atributos procurados. Em geral, como X é uma função muito específica, terá como retorno apenas o conjunto vazio.

Para aumentar a probabilidade de recuperar um conjunto não-vazio de X, dois artifícios podem ser aplicados: (1) utilizar um símbolo chamado 'qualquer', escrito como $*$; ou (2) expandir a consulta através da rede semântica de termos (Seção 3.4.2) .

No primeiro caso, o símbolo $*$ casa com qualquer termo t_{nj} num identificador. Este símbolo tem o efeito de reduzir o número de elementos em um identificador. Se, por exemplo, v_1 é substituído por $*$ na função X acima, então esta nova função torna-se

$$X'(i_\alpha) = \{x \mid x_2 = v_2 \wedge \ldots \wedge x_m = v_m\}$$

Neste caso, o critério de casamento é modificado para

$$\forall t; x_t = v_t \;\; ou \;\; x_t = *$$

o que resulta na redução da especificidade do identificador. Geralmente, quanto mais $*$ forem introduzidos numa consulta, maior será o tamanho de X. No caso extremo, quando todos $v_t = *$, $X = U$.

Com o segundo artifício, a rede semântica expande a função X, tornando-a uma *função de casamento parcial* X_p. Neste caso, todos os demais termos que têm alguma relação de proximidade com o termo t_{nj}, são utilizados na consulta. Assim, aqueles componentes cujos identificadores têm alguma proximidade com a consulta original são selecionados. Definimos esta função parcial como

$$X_p(i_\alpha) = \{x \mid x_1 = v_{e1} \wedge x_2 = v_{e2} \wedge \ldots \wedge x_n = v_{en}\}$$

onde

$$v_{en} = \{y \mid y \text{ é uma expansão do termo original } v_n\}$$

e $X_p \subseteq U$; α é uma instância de algum membro de X_p; e U e I são independentes.

Quando U é muito grande, o tamanho dos conjuntos X que casam com alguma consulta também se torna proporcionalmente maior, dificultando a tarefa de discriminar entre componentes relevantes e irrelevantes.

Um modo de obter grupos maiores de componentes relevantes é impor alguma ordem arbitrária nos atributos, de acordo com sua importância aos usuários. Esta ordem parcial é definida como [Pri85]:

$$A_i \succ A_j \;\; sempre \;\; que \;\; i < j$$

O símbolo '$\succ$' significa 'mais relevante que'.

Se, por exemplo, $n = 6$ e o identificador procurado $i_p = < v_1, v_2, v_3, v_4, v_5, v_6 >$ retornar $X(i_p) = \{\}$, há necessidade de se modificar a consulta, aplicando um dos dois artifícios anteriormente comentados.

A rede semântica de termos será detalhada na Seção 3.4.2. Com o segundo artifício, podemos criar vários outros identificadores, introduzindo o símbolo $*$, tais como:

$$i_1 = < v_1, v_2, v_3, v_4, v_5, * > \quad e \quad i_2 = < v_1, *, v_3, v_4, v_5, v_6 >$$

Neste caso, i_1 é uma *generalização mais relevante de* i_p do que i_2, uma vez que preserva os outros termos mais importantes do identificador. Como resultado, o conjunto $X(i_1)$ é mais relevante do que $X(i_2)$.

O posicionamento do símbolo $*$ fornece um mecanismo para controlar a ordem de relevância desejada durante o processo de seleção. Quando mais do que um símbolo * é introduzido, a relevância da generalização é determinada pela posição do símbolo mais à esquerda. Se $i_3 = < v_1, v_2, v_3, v_4, *, * >$, a ordem de relevância resultante nos três casos será $i_1 \succ i_3 \succ i_2$. De um modo geral, a relevância de um identificador pode ser calculado por [Pri85]:

$$R(i_\alpha) = min\{n \mid x_n = *\}$$

Aplicando esta equação aos três identificadores, nós obtemos

$$R(i_1) = 6, \quad R(i_2) = 2 \quad e \quad R(i_3) = 5$$

Assim, $R(i_1) > R(i_3) > R(i_2)$.

Nesta seção apresentamos os aspectos relacionados ao esquema de classificação por atributos de uma maneira generalizada. No restante deste capítulo, estas características serão direcionadas especificamente à classificação de componentes de software, sejam sequenciais ou, principalmente, concorrentes.

3.3.2 Por que Reutilizar Componentes Concorrentes?

Como já foi comentado no início do capítulo, um dos objetivos desta dissertação é estender o trabalho de Prieto-Díaz, de forma a classificar e selecionar também componentes concorrentes, além dos sequenciais. Criar uma biblioteca de componentes de software concorrentes pode trazer uma série de benefícios potenciais, que serão apresentados nesta seção.

Atualmente os sistemas concorrentes estão sendo utilizados nas mais diversas áreas. Concorrência, no seu sentido mais amplo, significa 'ao mesmo tempo'. Em outras palavras, um programa concorrente é um conjunto de processos sequenciais que são executados em paralelo. Este paralelismo pode ser real, onde os processos são executados em processadores fisicamente separados, ou abstrato, segundo o qual todos os processos são executados de maneira intercalada em um mesmo processador. A vantagem mais evidente é um possível aumento na velocidade

de um programa, uma vez que as várias tarefas pelas quais é composto, são executadas todas ao mesmo tempo. Alguns outros benefícios são uma melhor compreensão do programa pela sua decomposição em processos básicos; abstração dos detalhes, facilitando a modelagem e o estudo de sistemas complexos; e um melhor tratamento matemático para sistemas com multi-processadores [Ben90].

Como exemplos de sistemas concorrentes mais conhecidos podemos citar: sistemas de tempo-real, sistemas de processamento de transações e gerenciamento de banco de dados, e sistemas operacionais. Todos estes têm como característica comum a necessidade de gerenciar ou responder a atividades simultâneas em seus ambientes externos de maneira rápida e eficiente [Bac92]. Os sistemas de tempo-real trabalham sob rigorosas restrições de tempo, monitorando e controlando estações de força, sistemas hospitalares, robôs industriais, sistemas de tráfego aéreo, etc. Os sistemas de banco de dados devem interagir simultaneamente com diversos usuários, organizando operações de leitura e escrita de informações. E os sistemas operacionais, por sua vez, também devem interagir com vários usuários, no caso de sistemas multi-usuários, ao mesmo tempo em que controlam recursos e operações diversas.

Desta forma, a necessidade de novos métodos que dêem suporte ao desenvolvimento de sistemas concorrentes torna-se evidente. Assim como com os sistemas sequenciais, a reutilização de componentes anteriormente produzidos e testados aumentaria a qualidade e a confiabilidade do produto final. Além disso, o tempo de manutenção, custo e desenvolvimento de tais sistemas poderia ser bastante reduzido. Como os sistemas concorrentes, em geral, são extremamente mais complexos do que os sequenciais, estas facilidades tornam-se vantagens significativas dentro de um ambiente de desenvolvimento de software.

Entretanto, apesar de todas essas vantagens evidentes, pouco tem-se desenvolvido na área de reutilização de componentes concorrentes. Isto porque a grande maioria se concentra apenas na reutilização de componentes puramente sequenciais. Assim sendo, esta dissertação tem como objetivo pesquisar esta nova área, propondo um esquema que possa usufruir de todos os benefícios oferecidos pela aplicação da concorrência.

3.3.3 Classificação de Componentes Concorrentes

Até agora mostramos as vantagens de se criar bibliotecas de componentes concorrentes e empregar o esquema por atributos para classificá-los. A seguir, mostraremos como classificar e selecionar componentes concorrentes visando a reutilização dos mesmos.

No trabalho de Prieto-Díaz [Pri85], a descrição de um componente é formada por duas partes: uma descrevendo sua funcionalidade e outra detalhando seu ambiente. Como nesta dissertação

estamos preocupados com sistemas concorrentes, estendemos esta descrição de modo a satisfazer as características do novo contexto.

Em termos práticos, a descrição de nossos componentes também é constituída por duas partes básicas: uma parte composta por atributos que explicam sua funcionalidade e outra descrevendo características relacionadas à sua forma de processamento. Neste caso, retiramos os atributos com informações relacionadas ao seu ambiente. No contexto de concorrência que está inserida esta dissertação, é irrelevante o ambiente físico (área funcional ou departamento) onde o componente poderá ser utilizado. O mais importante é determinar sua funcionalidade e características internas.

Funcionalidade

Geralmente os componentes que executam a mesma função em domínios de aplicação distintos, apresentam detalhes de implementação, tais como as estruturas de dados utilizadas, totalmente diferentes. Por exemplo, dois programas de seleção distintos poderiam ser funcionalmente similares; ambos até poderiam ser implementações do mesmo algoritmo. Entretanto, um dos programas poderia ser específico para a seleção de números inteiros e o outro para a seleção de caracteres. Ou então, poderiam trabalhar com estruturas de dados diferentes, como por exemplo com uma tabela de símbolos no caso de um compilador, ou uma árvore-B para um banco de dados. Apesar de funcionalmente equivalentes, suas implementações — estruturas e objetos manipulados — podem ser muito diferentes.

Dessa maneira, para a especificação de componentes funcionalmente equivalentes precisamos, além da informação sobre suas funções propriamente ditas, da especificação dos objetos e das estruturas por eles manipuladas. Utilizaremos então três atributos para a descrição funcional de um componente:

- Função;
- Objeto;
- Meio.

Neste caso, 'Função' é o nome da ação primária executada pelo componente; sua função principal. Algumas funções típicas são: adicionar, transferir, comparar, criar, excluir, modificar, etc. 'Objeto' refere-se ao tipo dos objetos manipulados — produzidos ou consumidos — pelo componente, tais como, caracter, string, inteiro, diretório, expressão, real, valor booleano, etc. E, por fim, 'Meio' indica a estrutura utilizada ou o local onde a ação é executada, tal como, fila, pilha, lista, árvore, arquivo, tela, impressora, etc. No caso do exemplo anterior, apesar dos dois

programas executarem a mesma função de seleção, seriam classificados de maneiras distintas: o programa utilizado no compilador manipula símbolos (Objeto) através de uma tabela (Meio), enquanto que o programa para o banco de dados trabalha com registros (Objeto) através de uma árvore-B (Meio). Outros exemplos de classificação de componentes empregando estes atributos são:

<transferir, caracter, buffer>
<comparar, real, matriz>
<criar, inteiro, arquivo>
<listar, diretório, tela>
<adicionar, inteiro, fila>

Forma de Processamento

Os atributos que descrevem a funcionalidade de um componente são insuficientes para discriminar sistemas concorrentes de sistemas puramente sequenciais. Até este ponto, ambos os sistemas seriam classificados da mesma maneira, sem nenhuma distinção aparente. Assim, sentimos a necessidade de criar novos atributos, que pudessem indicar as diferenças entre os sistemas.

Procuramos então identificar as características essenciais relacionadas a um processo concorrente e que são importantes no momento de fazer a seleção. A partir desta análise, chegamos às seguintes conclusões:

- A funcionalidade dos sistemas concorrentes e sequenciais pode ser descrita pelos mesmos atributos (Seção 3.3.3);
- Existem poucos termos que são característicos dos sistemas concorrentes e que poderiam ser utilizados para classificá-los;
- A característica mais importante de tais sistemas é sua forma de comunicação.

Considerando estas observações, identificamos três atributos capazes de fazer a distinção entre sistemas sequenciais e concorrentes:

- Processamento;
- Comunicação;
- Composição.

Processamento Este atributo descreve a forma de processamento utilizada pelo componente: sequencial ou concorrente. A necessidade da especificação de tal atributo é conseguir distinguir e identificar componentes que apresentam a mesma funcionalidade, mas que são processados de maneiras diferentes. Um exemplo claro deste problema é o programa de ordenação Merge-sort [Ben90]:

```
procedure Merge_Sort is
      A : array(1..N) of Integer;
      procedure Sort(Low, High : Integer);
      procedure Merge;
begin
      Sort(1, N/2);
      Sort(N/2+1, N);
      Merge;
end.
```

Este programa pode apresentar duas versões diferentes: uma processando de forma concorrente e outra sequencial. Como uma versão sequencial, classificamos cada metade do vetor (*array*) separadamente e então agregamos (*merge*) os resultados. Já na versão concorrente, a classificação das duas metades do vetor podem ser feitas em paralelo, melhorando o desempenho. Por exemplo, se a sequência de entrada for (4,2,7,6,1,8,5,0,3,9), os dois procedimentos Sort podem ser executados em paralelo sobre os vetores (4,2,7,6,1) e (8,5,0,3,9), produzindo (1,2,4,6,7) e (0,3,5,8,9), respectivamente. Então, o procedimento Merge combina estes dois vetores, gerando o resultado final: (0,1,2,3,4,5,6,7,8,9).

As duas versões do programa Merge apesar de executarem a mesma função, são dois componentes totalmente diferentes, com desempenho e velocidades variadas. Sendo assim, precisam ser classificados de maneira distinta, o que pode ser feito através do atributo 'Processamento'.

Comunicação Para descrever o tipo de comunicação entre os componentes, utilizamos o atributo 'Comunicação'. Este pode ser considerado o atributo mais importante quando se trabalha com processos concorrentes, visto que uma de suas principais características é a forma de se comunicar — interagir e interferir — com o ambiente e demais processos. Assim sendo, este atributo relaciona-se exclusivamente à descrição de componentes concorrentes, pois são os únicos a apresentarem esquemas de interação ou comunicação entre si (inter-processos). Aos demais componentes (sequenciais), este atributo é indiferente ou nulo.

Dois processos iniciam uma comunicação por dois motivos básicos: para *cooperar* na execução de alguma tarefa ou para *competir* pelo uso exclusivo de serviços ou recursos. Independentemente do motivo, a comunicação entre os processos é realizada sob a forma de *troca de mensagens* ou *memória compartilhada* [Ben90]. Assim, criamos os seguintes termos para descrever ambas as formas de comunicação: síncrona, assíncrona, memória compartilhada, ou nulo, sendo este último relacionado aos componentes não-concorrentes.

No esquema de troca de mensagens, a comunicação requer a presença de dois processos: um para enviar a mensagem (emissor); e outro para recebê-la (receptor). Uma mensagem típica é mostrada na Figura 3.1. Outros campos adicionais podem estar presentes na mensagem, a qual contém: o identificador do emissor; o identificador do receptor; o tipo da mensagem, que poderia indicar por exemplo, se a mensagem é um pedido inicial para algum serviço ou uma resposta a algum pedido anterior, ou ainda qual a sua prioridade; e, por fim, o conteúdo da mensagem propriamente dito. O sistema faz uso do cabeçalho da mensagem para entregá-la ao destino correto. Seu conteúdo (corpo da mensagem) é irrelevante ao sistema de transporte de mensagens, sendo importante apenas aos dois processos se comunicando. Ambos devem estar de acordo com seu tamanho, conteúdo e estrutura, estabelecidos através de um protocolo de comunicação.

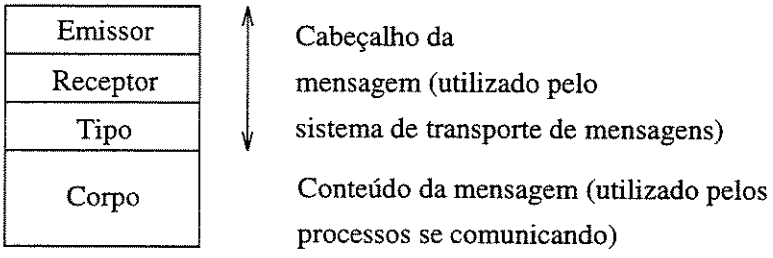

Figura 3.1: Estrutura típica de uma mensagem[Bac92]

Essa troca de mensagens, como especificado pelos termos do atributo 'Comunicação', pode ser feita de duas maneiras distintas:

- *Síncrona*: se o emissor está pronto para enviar a mensagem, mas o receptor não está pronto para recebê-la, o emissor é bloqueado. Da mesma forma, se o receptor é o primeiro processo pronto para se comunicar, ele é bloqueado até receber alguma mensagem do emissor. A ação de comunicação sincroniza as sequências de execução dos dois processos. Uma característica importante é que não existe a necessidade de armazenamento da mensagem, uma vez que esta somente é transferida se o receptor estiver pronto.

- *Assíncrona*: o emissor pode enviar uma mensagem e continuar sua execução normal, sem

ficar bloqueado, mesmo que o receptor ainda não esteja preparado para recebê-la. Não há uma conexão no tempo (sincronizada) entre as sequências de execução dos dois processos. Neste tipo de comunicação, ao contrário da síncrona, as mensagens devem ficar armazenadas em um *buffer*, gerando filas de mensagens; a grande vantagem é que o paralelismo na execução dos processos é maximizado devido à ausência de bloqueio das tarefas.

O esquema de comunicação por troca de mensagens é tipicamente utilizado em sistemas distribuídos. Já os processos presentes em computadores isolados geralmente se comunicam através do esquema de memória compartilhada. Neste caso, é assumida a existência de uma memória comum acessível a todos os processos. Esta memória comum pode ser utilizada de duas maneiras, que diferem apenas nos objetos que são manipulados pelos processos [Ben90]:

- Dados globais que podem ser lidos ou escritos por mais de um processo;
- Código objeto das rotinas do sistema operacional (*system-calls*) que podem ser chamadas por mais de um processo [Dub94].

Quando vários processos compartilham uma memória, são capazes de utilizar estruturas de dados comuns para cooperarem e se comunicarem. O problema é que a utilização de tais estruturas, as quais acessam regiões compartilhadas denominadas de regiões críticas, deve ser rigidamente controlado, pois todos os processos ficam competindo para conseguir realizar operações de leitura ou escrita. Assim, estruturas auxiliares de controle precisam ser implementadas, tais como os conhecidos semáforos e monitores [Bac92].

Composição Finalmente, o atributo 'Composição' descreve a forma de concorrência empregada por um componente, podendo ser concorrência real ou *interleaving*[5]. Há ainda um terceiro termo 'nulo', o qual se relaciona aos demais componentes não-concorrentes (sequenciais). No esquema de concorrência real, os processos concorrentes são executados, simultaneamente, em diferentes processadores. Neste caso, não há disputa pelo processador, já que o código de cada componente 'detém' um processador próprio. No caso de *interleaving*, há uma disputa pela posse do processador. Os componentes são vistos como conjuntos de sequências de instruções atômicas, que têm suas execuções intercaladas em um mesmo processador, criando uma espécie de paralelismo virtual.

Um exemplo simples da diferença entre os dois esquemas é apresentado na Figura 3.2. Nesta, dois processos procuram utilizar os recursos disponíveis em uma fila. Na Figura 3.2 (a) dois

[5]Mantivemos a palavra em inglês, pois não encontramos nenhuma tradução em português que tivesse um significado semelhante.

processos estão sendo executados em processadores separados, num esquema de concorrência real. Ambos verificam o número de recursos disponíveis e então alocam um recurso para si próprio. Na Figura 3.2 (b) dois processos estão rodando em um mesmo processador através de um esquema de *interleaving*. O Processo A verifica o número de recursos disponíveis e em seguida 'perde' o processador em favor do Processo B. O estado do Processo A é salvo, sendo posteriormente restaurado ao ser escalonado novamente.

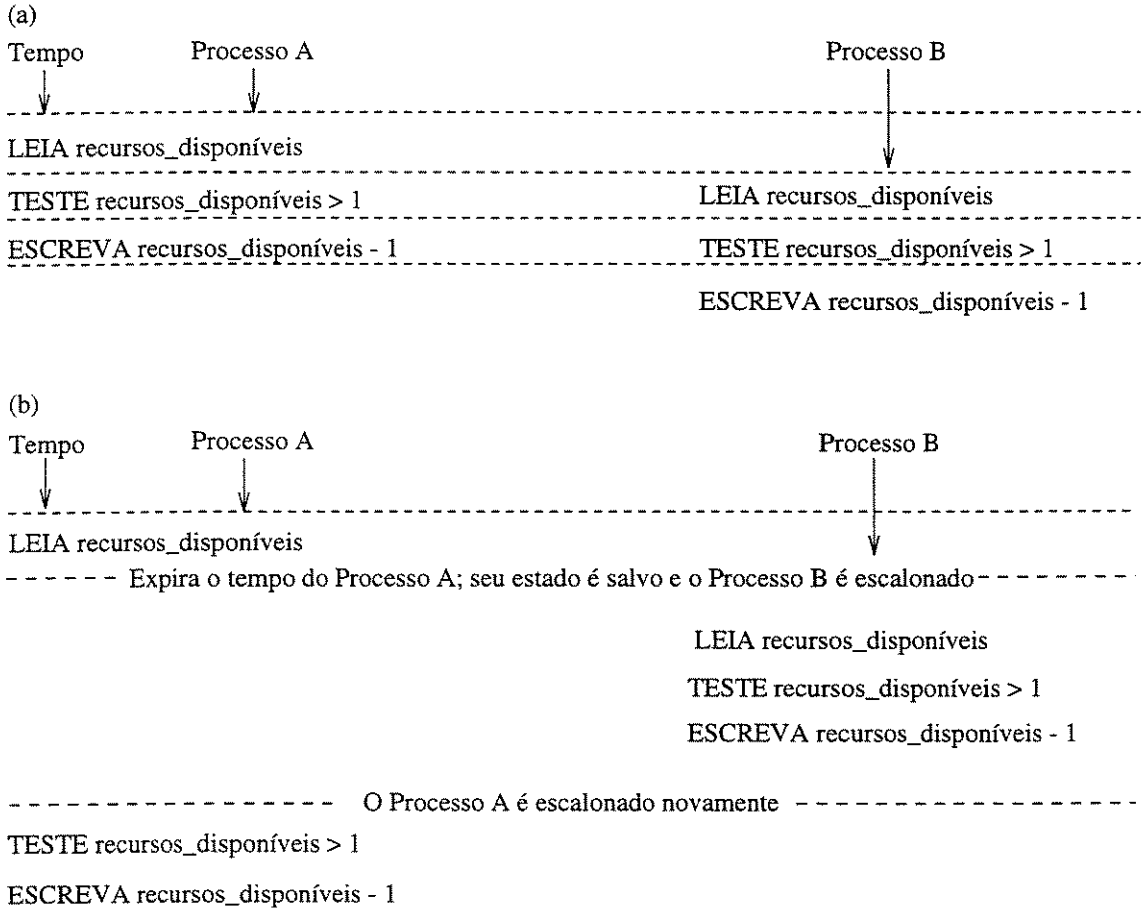

Figura 3.2: Esquemas de concorrência real (a) e por interleaving (b)

Ambos os processos poderiam ser classificados com os seguintes termos: *alocar* no atributo 'Função', *recurso* no atributo 'Objeto' e *fila* no atributo 'Meio'. Dessa forma, suas descrições seriam muito parecidas, sendo diferenciadas apenas pelos termos associados ao atributo 'Composição'.

3.3.4 Exemplos de Classificação

Nesta seção, apresentamos alguns exemplos de classificação utilizando o esquema anteriormente definido. Antes, porém, precisamos especificar uma ordem de citação entre os atributos, de acordo com sua relevância aos usuários do esquema (Seção 3.3.1). Assumindo que os usuários típicos deste esquema são engenheiros de software projetando e construindo novos sistemas a partir de bibliotecas de componentes, a ordem de citação escolhida será: Função $\succ$ Objeto $\succ$ Meio $\succ$ Processamento $\succ$ Comunicação $\succ$ Composição. A Tabela 3.1 mostra uma listagem parcial do esquema.

Domínio: Componentes de Software					
Função	*Objeto*	*Meio*	*Processamento*	*Comunicação*	*Composição*
adicionar	inteiro	pilha	sequencial	síncrona	real
atualizar	argumento	fila	concorrente	assíncrona	interleaving
criar	booleano	tela		memória-compartilhada	nulo
suspender	caracter	árvore		nulo	
comparar	string	arquivo			
excluir	diretório	impressora			
modificar	expressão	matriz			
inserir	linha	mouse			
fechar	macro	lista			
dividir	evento	teclado			
transferir	texto	buffer			
formatar	real	sensor			
associar	instrução	array			
substituir	variável	disco			
mapear	ponteiro	tabela			
⋮	⋮	⋮			

Tabela 3.1: Listagem parcial do esquema de classificação por atributos

Classificar um componente consiste em selecionar a sêxtupla que melhor o descreve. Exemplos de identificadores de componentes são:

<copiar, caracter, buffer, concorrente, síncrono, interleaving>

<listar, diretório, tela, sequencial, nulo, nulo>

<adicionar, inteiro, fila, concorrente, memória compartilhada, real>

<criar, inteiro, arquivo, sequencial, nulo, nulo>

<comparar, real, matriz, concorrente, síncrono, interleaving>

<modificar, booleano, array, concorrente, assíncrono, real>

No Apêndice A, vários outros exemplos de identificadores são apresentados: componentes e funções simples obtidos a partir de outros trabalhos são classificados utilizando-se este esquema de seis atributos.

Quando o número de termos é pequeno, a escolha dos termos corretos para classificar e selecionar os componentes não apresenta dificuldades. Entretanto, conforme o número de termos e de componentes aumenta, estas tarefas tendem a tornar-se cada vez mais complicadas. Para controlar estes problemas, descreveremos alguns mecanismos que facilitam a elaboração de consultas e auxiliam na recuperação dos componentes em uma biblioteca.

3.4 Ferramenta de Apoio à Classificação e Seleção

Quando o número de termos é muito grande, classificar e selecionar componentes torna-se uma tarefa difícil. Dentro de uma grande coleção, surgem vários termos semelhantes ou até mesmo sinônimos. A procura pelo termo mais adequado torna-se um exercício braçal e acaba desestimulando o usuário.

Para manter estes problemas sob controle, descreveremos o primeiro filtro de uma ferramenta de auxílio ao processo de classificação e seleção de componentes concorrentes. Nosso objetivo foi criar uma ferramenta prática, que pudesse ser facilmente utilizada pelo usuário final. O mais importante é facilitar seu trabalho, fornecendo uma interface simples, sobre uma base de conceitos teóricos. Dentre esses conceitos, os mais importantes são: o dicionário de termos sinônimos (Seção 3.4.1) e a rede de termos semanticamente relacionados (Seção 3.4.2).

3.4.1 Dicionário de Termos

Descrever o código de um componente através de uma tupla formada por alguns termos apresenta um pequeno problema: a presença de termos sinônimos. Isto porque os sinônimos podem produzir descrições diferentes para um mesmo componente, gerando certa confusão no momento de classificá-lo. Por exemplo, no caso da funcionalidade, as tuplas <copiar, string, arquivo> e <transferir, caracter, arquivo> podem ser duas descrições diferentes do mesmo componente. Com o intuito de evitar duplicações e classificações ambíguas, especificamos um dicionário de termos sinônimos baseado em um vocabulário controlado.

Um dicionário de termos [Pri85] é necessário para agrupar todos os sinônimos sob um único conceito. O termo que melhor descreve o conceito é selecionado como o termo representativo e a este é associado um conjunto de sinônimos. Dessa forma, reduzimos a ambiguidade da classificação e melhoramos consideravelmente a qualidade dos componentes recuperados ao se realizar

uma busca numa biblioteca. Outra vantagem do emprego de um dicionário é a possibilidade de se controlar a quantidade de termos presentes na coleção. Aumentando-se o número de termos associados a um grupo particular, ou quebrando-se os grupos em conjuntos menores, conseguimos variar a quantidade de termos disponíveis para classificação [PF87]. Alguns exemplos de entradas num dicionário de sinônimos, relacionados ao atributo 'Função', são mostrados na Tabela 3.2.

Atributo Função	
Termo Representativo	*Sinônimos*
adicionar	incrementar/somar/totalizar
comparar	testar/casar/checar/verificar
complementar	negar/inverter
enumerar	contar/medir/listar
abrir	conectar/anexar
associar	unir/juntar
mostrar	imprimir/exibir/listar
criar	produzir/gerar/construir/formar
retirar	eliminar/remover
substituir	trocar/converter/atualizar

Tabela 3.2: Exemplo de entradas no dicionário de termos

Utilizando tal dicionário, conseguimos eliminar a ambiguidade de alguns termos, selecionando um contexto específico através da lista dos termos mais representativos. A ambiguidade do termo 'listar' na Tabela 3.2, por exemplo, é resolvida selecionando-se um dos dois contextos: enumerar (criar uma lista de elementos) ou mostrar (apresentar os elementos num dispositivo de saída, como o monitor).

Na fase de seleção, o dicionário é utilizado durante a escolha dos termos que descrevem o componente desejado pelo usuário. Este deve necessariamente montar a descrição do componente a partir do conjunto de termos representativos evitando, assim, o emprego de sinônimos. Com isso, conseguimos criar identificadores mais claros e livres de ambiguidade.

Apesar de todas as vantagens, a construção de um dicionário de termos é uma tarefa que exige muito esforço manual por parte do usuário. É necessário procurar por termos relevantes, fazendo-se uma vasta pesquisa em fontes variadas relacionadas ao assunto. No caso de componentes de software, estas fontes podem ser descrições de programas, comentários no código-fonte, livros e artigos científicos, revistas, etc. No entanto, deve ser considerado que todo este esforço inicial será exigido uma única vez, sendo posteriormente recompensado pelas muitas aplicações do dicionário.

3.4.2 Rede Semântica

Dentro de uma grande coleção vários termos apresentam alguma relação de proximidade. Por exemplo, os termos *substituir* e *remover* do atributo 'Função' são semanticamente próximos, uma vez que ambos denotam uma noção de troca. Aproveitando esta relação natural, derivamos uma rede semântica de termos, com o intuito de determinar a proximidade ou grau de similaridade entre os termos de um atributo. Através desta rede podemos recuperar componentes *similares* ao desejado, os quais possuem em seus identificadores termos semanticamente próximos ao identificador procurado.

Esta rede é uma modificação do grafo conceitual de pesos elaborado por Prieto-Díaz [Pri85], de forma a torná-lo mais simples e compreensível. Eliminamos sua idéia de supertipos, especificando a distância entre os termos através da associação direta de pesos.

Na verdade, a rede é um grafo direcionado [Pri85] cujos nós correspondem a termos específicos, e seus arcos possuem pesos que determinam o grau de relacionamento entre os termos. Para cada termo em um atributo, identificamos os demais termos que apresentam algum relacionamento semântico ou conceitual com o mesmo. A esta ligação associamos um valor não-negativo (peso) indicando o grau de relacionamento entre os dois termos: quanto menor o peso, maior o nível de proximidade entre ambos. É importante observar que estes pesos devem ser derivados de maneira subjetiva de acordo com a experiência e a intuição do usuário. Ou seja, a rede semântica é construída de forma manual, ao contrário de outros sistemas que empregam esquemas de indexação automáticos [MBK91]. Os valores correspondem a estimativas que são utilizadas para medir proximidade e não possuem nenhuma analogia com o 'mundo real'. Por exemplo, um peso 5 indica um grau de relacionamento mais próximo do que um peso 10 mas, na verdade, 5 e 10 não refletem diretamente quantidades mensuráveis, tais como linhas de código ou horas-homem.

Alguns pares de nós em uma rede semântica podem não estar conectados diretamente, indicando que não há uma relação imediata entre ambos. Ainda assim é possível estimar a distância ou grau de similaridade entre tais pares. A distância entre um termo t_1 e um termo t_2 é definida como o peso do menor caminho entre t_1 e t_2 dentro da rede [OHPB92]. Se nenhum caminho existe, a distância é dita infinita. Obviamente, se t_1 e t_2 são o mesmo termo, a distância entre ambos é zero.

Dessa forma, um componente C_d é similar a um componente C_c se a distância geral entre seus termos correspondentes é pequena, indicando o esforço necessário para obter o componente desejado C_d dado o componente candidato C_c. Esta distância geral é calculada através da soma das distâncias individuais entre os termos equivalentes dos dois componentes, sendo representada

pela seguinte expressão:

$$D_p(C_d, C_c) = \sum_{at \in \Gamma} D_{at}(C_d.at, C_c.at)$$

onde Γ representa o conjunto de atributos; $C_d.at$ representa o termo t_d associado ao atributo at do componente desejado C_d; $C_c.at$ representa o termo t_c associado ao atributo at do componente candidato C_c; e, finalmente, $D_{at}(C_d.at, C_c.at)$ é a distância (peso) do menor caminho entre o termo t_d e o termo t_c, como indicado pela rede semântica do atributo at. A função D_p é chamada de *comparador de proximidade* e as funções D_{at} são chamadas *comparadores de atributo* [OHPB92].

Por exemplo, considerando os descritores funcionais de dois componentes diferentes:

C_1 = <remover, inteiro, matriz> e
C_2 = <inserir, real, vetor>

A distância de proximidade entre C_1 e C_2 é dada pela seguinte expressão que calcula a soma das distâncias entre seus termos correspondentes para cada atributo:

$$\begin{aligned} D_p(C_1, C_2) \;=\; & D_{Função}(remover, inserir) + \\ & D_{Objeto}(inteiro, real) + \\ & D_{Meio}(matriz, vetor) \end{aligned}$$

onde $D_{Função}$, D_{Objeto} e D_{Meio} são as funções de comparação de atributos para os atributos Função, Objeto e Meio, respectivamente. Estas funções são calculadas utilizando-se suas redes de proximidade semântica correspondentes.

Construção de uma Rede Semântica

Redes semânticas são projetadas inicialmente baseadas na intuição e no conhecimento de um especialista sobre o domínio particular sendo modelado. A cada termo t de um atributo at são associadas distâncias a um pequeno conjunto de termos em at que apresentam alguma similaridade com t. Estas associações definem grupos de termos intimamente relacionados. Por exemplo, a Figura 3.3 mostra os grupos associados aos termos Substituir, Modificar e Adicionar do atributo Função[6].

Quando estes grupos são integrados, eles definem uma rede semântica combinada, a qual permite a comparação de termos que não estão relacionados diretamente em um grupo. A Figura 3.4 mostra uma rede semântica parcial definida pelos três grupos da Figura 3.3. Baseado

[6]Os valores foram selecionados utilizando-se uma escala no intervalo de 0 (sinônimo) a 100 (distante).

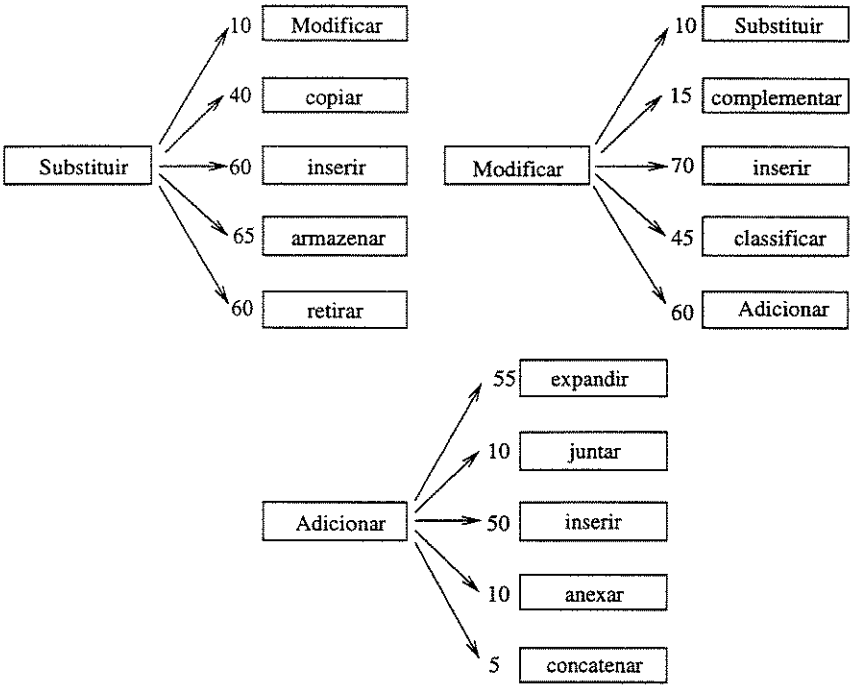

Figura 3.3: Grupos de termos relacionados do atributo Função

nesta rede podemos, por exemplo, estimar o grau de similaridade entre os termos 'substituir' e 'concatenar': o peso do caminho substituir → modificar → adicionar → concatenar, que tem 75 como resultado [7].

Uma rede semântica deste tipo e suas medidas de proximidade são fundamentais durante o processo de recuperação de um componente. Caso um termo em uma consulta não 'case' exatamente com nenhuma descrição disponível na coleção, o usuário tem a possibilidade de tentar encontrar na rede o termo seguinte mais próximo ao procurado. Dessa forma, consegue recuperar descrições de componentes muito similares ao desejado, o que é uma característica fundamental em reutilização de software, visto que geralmente é muito mais eficiente e simples modificar algumas partes de um componente do que construí-lo por completo.

Um exemplo seria o termo 'modificar', definido na Figura 3.4. Caso o usuário elabore uma consulta utilizando o termo 'modificar' e o sistema verifique que não existe nenhum componente na biblioteca classificado com este termo, ele poderá procurar por outros componentes similares, utilizando os termos semanticamente relacionados, na seguinte ordem: substituir → complementar → classificar → copiar → adicionar → concatenar → inserir → ... É interessante observar que o sistema sempre procura pelo menor peso, independentemente se ele faz parte do grupo

[7] Os arcos com peso zero representam termos sinônimos (linhas pontilhadas).

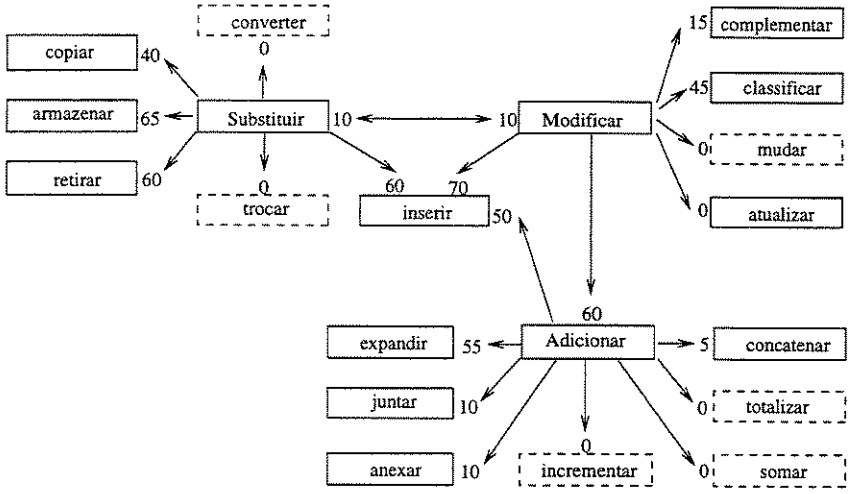

Figura 3.4: Rede semântica parcial do atributo Função

sendo expandido ou não. Por exemplo, no caso da expansão anterior do termo 'modificar', o termo 'copiar' pertencente ao grupo 'substituir' seria utilizado antes do termo 'adicionar'. Isto porque o peso do caminho até 'copiar' (modificar → substituir → copiar) é 50, enquanto 'adicionar' tem peso 60.

Os arcos com peso zero na Figura 3.4 representam termos sinônimos (linhas pontilhadas). Para manipular termos sinônimos, à distância entre um termo e seu sinônimo dentro de uma rede semântica é associado o valor zero [OHPB92]. Isto tem o efeito de que a distância entre um componente candidato C_c, que é descrito utilizando-se termos que são sinônimos aos termos empregados para descrever um componente desejado C_d, será zero, indicando que C_c é o melhor candidato para C_d.

Da mesma forma como ocorre com o dicionário de sinônimos (Seção 3.4.1), a construção de uma rede semântica exige um determinado esforço inicial. Um especialista na área deve usar de experiência e intuição para relacionar e associar pesos aos termos. Como esta é uma tarefa subjetiva e manual, também é propensa a erros. Porém, conforme a rede vai sendo utilizada e as modificações necessárias vão sendo executadas, a rede tende a tornar-se gradualmente melhor adaptada ao domínio da aplicação.

Exemplo de Utilização de uma Rede Semântica

Para ilustrar como uma rede pode ser utilizada, vamos criar um exemplo baseado em apenas dois atributos: Função e Objeto. Os termos relacionados a estes atributos são especificados na Tabela 3.3.

Função		*Objeto*	
adicionar	inserir	arquivo	página
anexar	juntar	árvore	palavra
armazenar	modificar	buffer	ponteiro
classificar	copiar	caracter	tabela
complementar	retirar	diretório	token
concatenar	substituir	linha	vetor
expandir		lista	

Tabela 3.3: Conjunto de termos

A ordem de citação dos atributos será *Função* $\succ$ *Objeto*. A rede semântica parcial do atributo Função já foi mostrada na Figura 3.4. Os grupos de termos e a rede semântica parcial do atributo Objeto são apresentados nas Figuras 3.5 e 3.6, respectivamente. Ambas as redes foram criadas de acordo com os critérios do autor.

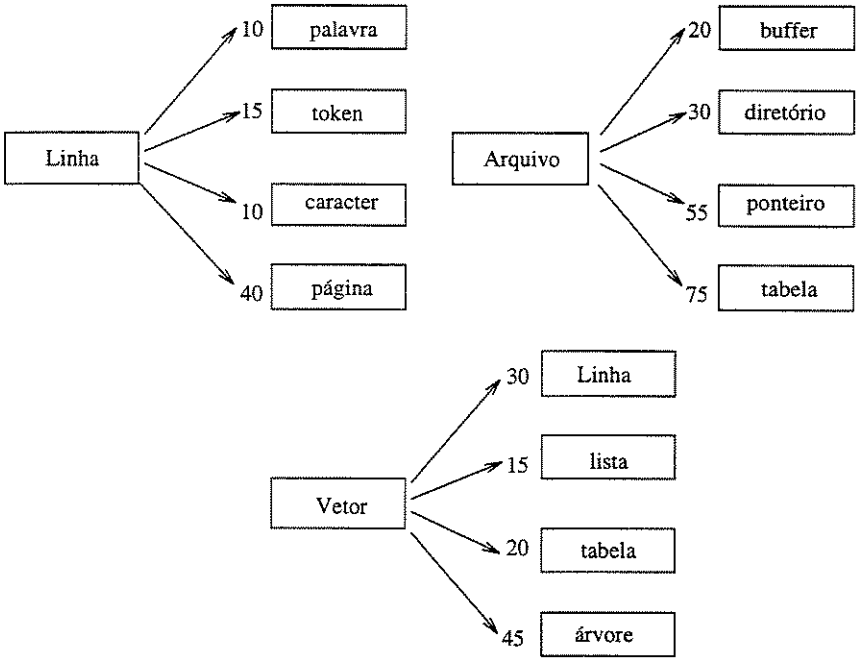

Figura 3.5: Grupos de termos relacionados do atributo Objeto

Utilizando o seguinte identificador

$$< modificar, vetor >$$

o objetivo será listar o conjunto de identificadores conceitualmente próximos a *modificar* e *vetor*. Uma vez que *Função* $\succ$ *Objeto*, um identificador que possui um objeto diferente de *vetor* é mais relevante do que um identificador com uma função diferente de *modificar*. Por exemplo,

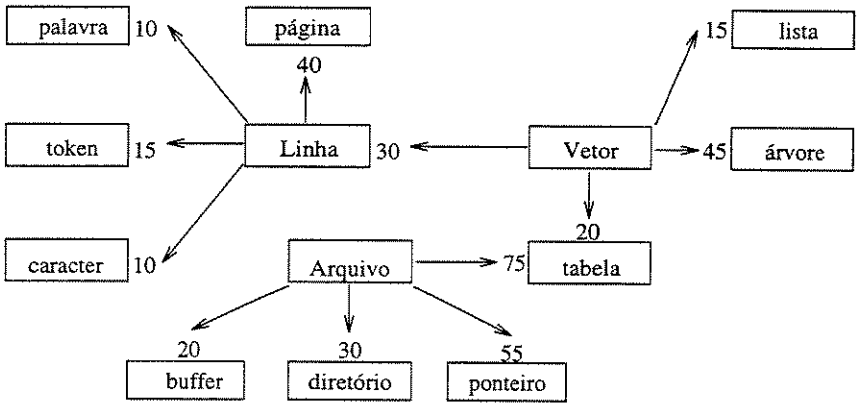

Figura 3.6: Rede semântica parcial do atributo Objeto

$< modificar, linha >$ é um identificador mais relevante do que $< adicionar, vetor >$, pois consideramos a funcionalidade do componente mais importante do que o objeto por ele manipulado.

Calculamos então os valores da distância semântica (peso) entre *modificar* e os demais termos do atributo Função, a partir da rede semântica da Figura 3.4. Estas distâncias são os resultados das funções D_{at} (comparadores de atributo), definidas na Seção 3.4.2. De maneira geral, a função de comparação pode ser descrita como: $D_{Função}(modificar, termo)$. Os valores são listados na Tabela 3.4, juntamente com o nível de proximidade com *modificar*.

Nível	*Distância*	*Termo*	*Nível*	*Distância*	*Termo*
1	10	substituir	7	70	inserir
2	15	complementar	7	70	retirar
3	45	classificar	7	70	anexar
4	50	copiar	7	70	juntar
5	60	adicionar	8	75	armazenar
6	65	concatenar	9	115	expandir

Tabela 3.4: Distâncias do termo *modificar*

O mesmo procedimento, utilizando a rede semântica da Figura 3.6, produz os termos relacionados a *vetor*: $D_{Objeto}(vetor, termo)$, listados na Tabela 3.5.

Cruzando os valores das Tabelas 3.4 e 3.5, finalmente obtemos o conjunto completo dos identificadores semanticamente próximos a $< modificar, vetor >$. Este conjunto é listado na Tabela 3.6, ordenado de acordo com o nível de proximidade de cada identificador. As distâncias totais correspondem aos resultados das funções D_p (comparador de proximidade), descritos na Seção 3.4.2. De maneira geral:

$$D_p(ident1, ident2) = D_{Função}(modificar, termo_1) + D_{Objeto}(vetor, termo_2)$$

Nível	*Distância*	*Termo*	*Nível*	*Distância*	*Termo*
1	15	lista	5	45	token
2	20	tabela	6	70	página
3	30	linha	∞	∞	arquivo
4	40	palavra	∞	∞	buffer
4	40	caracter	∞	∞	diretório
5	45	árvore	∞	∞	ponteiro

Tabela 3.5: Distâncias do termo *vetor*

onde *ident*1 corresponde a $< modificar, vetor >$ e *ident*2 é o identificador que nos interessa saber a distância semântica, ou seja, $< termo_1, termo_2 >$. É importante observar que estas distâncias são ordenadas dentro de cada nível, ou seja, as distâncias dentro do nível $1.x$ não têm relação com as do nível $2.x$, e assim por diante. Dessa forma, apesar da distância do nível 2.1 ser inferior à do nível 1.3, o identificador $< substituir, linha >$ apresenta uma maior proximidade semântica a $< modificar, vetor >$ do que o indentificador $< complementar, lista >$. Isto é claro, uma vez que *Função* $\succ$ *Objeto* e *substituir* é mais próximo de *modificar* do que *complementar*.

Nível	*Distância*	*Identificador*
1.1	25	substituir/lista
1.2	30	substituir/tabela
1.3	40	substituir/linha
1.4	50	substituir/palavra
1.4	50	substituir/caracter
1.5	55	substituir/árvore
1.5	55	substituir/token
1.6	80	substituir/página
2.1	30	complementar/lista
2.2	35	complementar/tabela
2.3	45	complementar/linha
⋮	⋮	⋮
9.5	160	expandir/árvore
9.5	160	expandir/token
9.6	185	expandir/página

Tabela 3.6: Lista ordenada de identificadores

3.5 O Processo de Recuperação

Depois que os componentes de software tiverem sido classificados, tem-se então uma biblioteca de componentes. Com esta, podem ser feitas consultas variadas, recuperando-se componentes

específicos de acordo com as necessidades do usuário. As consultas são realizadas da mesma forma que foi feita a classificação: selecionando-se a sêxtupla de termos que melhor descreve o componente desejado. A partir daí, os componentes candidatos são recuperados, baseados no grau de similaridade entre a descrição do componente desejado e a descrição dos componentes presentes na biblioteca, calculada através da rede de proximidade semântica descrita na Seção 3.4.2. E, por fim, para facilitar a elaboração de consultas é empregado um dicionário de termos sinônimos, detalhado na Seção 3.4.1.

A Figura 3.7 apresenta uma visão abstrata de todo o esquema [Pri91]. Todo componente α possui um identificador i_α, que é um conjunto ordenado de termos provenientes de cada atributo A_n. Os termos de cada atributo são relacionados a alguns outros termos, com os quais apresentam algum relacionamento conceitual, criando uma rede semântica para cada atributo. Da mesma forma é criado um dicionário de termos, especificando-se um conjunto de sinônimos e escolhendo-se o termo mais representativo para defini-los.

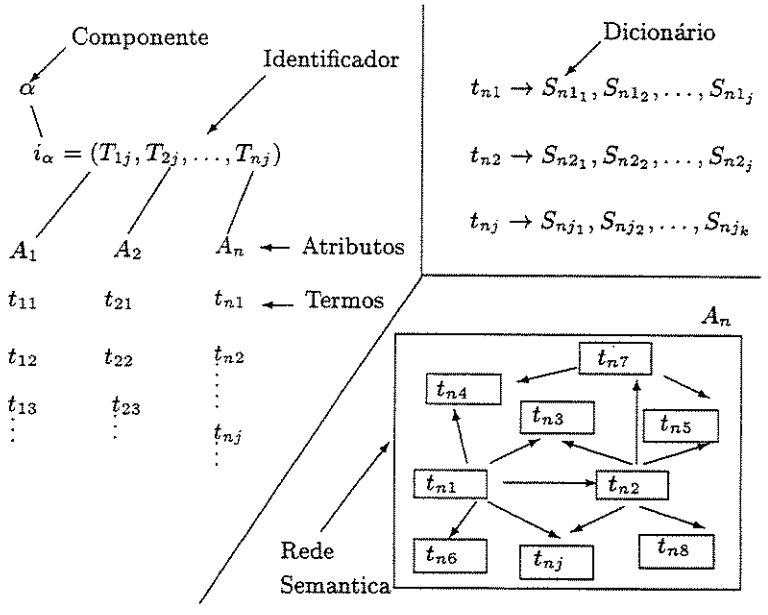

Figura 3.7: Visão abstrata do esquema de classificação por atributos

Durante a recuperação de componentes, uma consulta é formada por um identificador $i_{\alpha c}$ de termos selecionados dentro de cada atributo. Caso não haja um 'casamento exato' na coleção para $i_{\alpha c}$, termos semanticamente próximos são selecionados calculando-se as distâncias na rede semântica correspondente, produzindo assim, novos identificadores $i_{\alpha n}$. 'Casamentos' com os identificadores $i_{\alpha n}$ subsequentes, recuperarão componentes similares ao componente especificado por $i_{\alpha c}$.

3.6 Conclusão

A flexibilidade do esquema de classificação por atributos torna-o adequado à organização de componentes de software. A facilidade para a expansão dos termos e atributos faz com que este esquema se adapte bem a domínios dinâmicos e em constante alteração.

Expandimos o trabalho de Prieto-Díaz [Pri85], de maneira a dar suporte à classificação de componentes de software concorrentes. Como descrevemos na Seção 3.3.2, tais componentes são fundamentais em aplicações críticas que necessitam de eficiência e velocidade. Assim sendo, dentro deste contexto criamos três novos atributos de classificação: Processamento, Comunicação e Composição. A validade destes novos atributos foi demonstrada em vários exemplos de classificação (Apêndice A).

Quando se trabalha com esquemas baseados em vocabulários controlados, naturalmente surgem vários termos semelhantes ou até mesmo sinônimos. Para manter este problema sob controle, apresentamos uma ferramenta de apoio ao usuário baseada em um dicionário de sinônimos e uma rede semântica de termos. Procuramos dar maior enfoque na definição da rede semântica, principal instrumento para realizar casamentos parciais de componentes. Criamos uma rede simples e eficiente, baseada na distância semântica (peso) entre termos. Sua utilização foi demonstrada em um exemplo final, englobando todos os conceitos apresentados no capítulo.

Este esquema é utilizado como um primeiro filtro do processo de seleção de componentes concorrentes. O objetivo é classificar os componentes, de modo a reduzir o espaço de busca para o filtro subsequente: o casamento de interfaces. Este segundo filtro trabalha com informações sobre os tipos dos objetos manipulados pelos componentes. O sistema de tipos utilizado é aquele da linguagem de especificação formal LOTOS, como será detalhado no capítulo seguinte.

Capítulo 4

Casamento de Interface

Neste capítulo apresentamos o segundo filtro de nosso processo de seleção: o casamento de interface empregando o sistema de tipos da linguagem de especificação formal LOTOS.

Neste sentido, descrevemos um histórico básico da linguagem LOTOS, bem como suas características e operadores mais significativos; explicaremos o que é o casamento de interface; e apresentaremos seus variados níveis de precisão. E, por fim, mostraremos um novo nível, baseado em tipos genéricos, bem como um método simplificado de casamento de interface utilizando os tipos de LOTOS.

4.1 Introdução

A seleção de componentes, como já visto, tem como objetivo principal entregar ao usuário o menor conjunto possível de componentes que seja capaz de satisfazer suas necessidades. Caso a quantidade de componentes recuperados seja muito grande, a atividade de analisar cada um separadamente exigirá muito esforço, inviabilizando qualquer processo de reutilização. Quanto mais refinada for a fase de seleção, mais específico e eficiente será o conjunto final de componentes recuperados.

Nosso sistema de classificação e seleção é composto por dois filtros básicos: uma classificação por atributos e um casamento sintático. A classificação organiza os componentes de maneira hierárquica, selecionando-os através de atributos que especificam sua funcionalidade e processamento (Capítulo 3). A característica deste filtro é trabalhar num alto nível de abstração, independentemente de qualquer linguagem de programação específica.

Neste capítulo, apresentamos o segundo filtro de seleção: o casamento de interfaces. Este filtro tem como característica principal a análise de informações relacionadas aos tipos dos

objetos manipulados pelos componentes. A seleção é feita através de um casamento sintático entre os tipos de uma determinada consulta e as especificações dos componentes presentes na biblioteca. O nível de abstração passa a ser mais baixo, próximo ao código-fonte, sendo portanto necessária a escolha de uma linguagem-alvo. Como nesta dissertação estamos interessados em aproveitar as vantagens de uma abordagem formal (Seção 2.5), optamos por trabalhar com o sistema de tipos da linguagem de especificação formal LOTOS. Na Seção 4.2.2 explicamos quais os motivos que nos levaram a tal escolha. Dessa forma, componentes formalmente especificados em LOTOS podem ser selecionados com maior facilidade, através do casamento dos tipos básicos que compõem sua função principal.

O restante do capítulo fica assim organizado: na Seção 4.2 são definidas as características da linguagem LOTOS, cujo sistema de tipos é utilizado como base para nosso método de seleção. Apresentamos seus tipos e operadores básicos, juntamente com um exemplo de especificação. Na Seção 4.3 são detalhadas as características e vantagens do emprego do casamento de interfaces na fase de seleção de componentes. A Seção 4.4 descreve os variados níveis de casamento e suas possíveis combinações. Em seguida, na Seção 4.5, propomos um novo esquema de casamento de interface simplificado, baseado apenas na quantidade e tipo dos parâmetros. Nesta mesma seção, explicamos um novo nível de casamento, o qual utiliza tipos genéricos. E, por fim, na Seção 4.6 são apresentadas as conclusões obtidas neste capítulo.

4.2 LOTOS: *Language of Temporal Ordering Specification*

LOTOS nasceu do esforço da ISO (*International Organization for Standardization*) para produzir padrões para o OSI (*Open Systems Interconnection*), em resposta à demanda dos fabricantes e usuários por esquemas de interconexão compatíveis para equipamentos heterogêneos [Tur96]. Conforme este trabalho teve início, no final da década de 70, a ISO percebeu que para os padrões tornarem-se realmente eficientes, eles teriam que ser descritos de maneira clara e precisa. Criaram-se então grupos para o desenvolvimento de Técnicas de Descrição Formal[1] (FDTs), cujo objetivo era criar técnicas para a especificação exata de protocolos e serviços de comunicação.

Algumas das características desejáveis de uma FDT foram descritas como [LFH92]:

- Alto nível de abstração;
- Independência de qualquer tipo de implementação;
- Semântica formal;

[1] *Formal Description Techniques*, do inglês.

- Suporte de métodos de teste e verificação.

Como na época ainda não havia nenhuma FDT que preenchesse todas estas características, foi criado o comitê SC21/WG1 (Subcomitê para a padronização das camadas superiores do OSI / Grupo de trabalho para o desenvolvimento dos conceitos estruturais básicos do OSI) para definir uma FDT adequada ao OSI. Inicialmente, em torno de 20 técnicas diferentes foram propostas, divididas em três subgrupos básicos [Tur96]:

- Subgrupo A: aspectos estruturais das FDTs;
- Subgrupo B: técnicas com máquinas de estado finito, as quais levaram à criação de ESTELLE (Extended Finite State-Machine Language);
- Subgrupo C: técnicas algébricas com ordem temporal, que propiciaram o surgimento de LOTOS [ISO87].

O subgrupo C iniciou seus trabalhos em 1983, adotando o CCS (*Calculus of Communicating Systems*) [Mil80] como a base para a nova linguagem de especificação. Em 1984 a linguagem foi expandida, adquirindo algumas características de CSP (*Communicating Sequential Processes*) [Hoa78], principalmente no que diz respeito aos aspectos de sincronização. Neste mesmo ano, a linguagem ACT ONE foi adicionada, para assim permitir a especificação formal e abstrata de tipos de dados.

LOTOS tornou-se uma linguagem completa em 1987. Entretanto, teve de esperar até 1988 para ser aceita como um padrão internacional.

4.2.1 Princípios de LOTOS

Alguns dos princípios básicos que inspiraram o projeto de LOTOS podem ser definidos como [LFH92]:

- *Formalismo complementar para 'dados' e 'controle'.* LOTOS foi concebida como a união de dois formalismos: ACT ONE para a parte de dados, e CCS/CSP para a parte de controle.
- *Definição formal.* Sintaxe e semântica formalmente definidas, através de gramáticas de atributos e regras de inferência.
- *Álgebra de processos.* Seguindo as idéias de Milner [Mil80], a semântica é definida de maneira que seja possível provar um grande conjunto de propriedades algébricas de equivalência, baseado em vários tipos de relações de equivalência. Estas propriedades podem

ser utilizadas para provar a equivalência ou a corretude das especificações, bem como para transformar a estrutura de uma especificação.

- *Concorrência por interleaving.* Os eventos em LOTOS são considerados atômicos, cujas execuções em paralelo se dão sob a forma de intervalos intercalados.

- *Especificação executável.* Como a semântica de LOTOS é definida por regras de inferência, é possível implementar esta semântica em um interpretador. Isto significa que as especificações podem ser escritas, interpretadas e traduzidas em um programa executável.

- *Modularidade.* LOTOS estimula a decomposição dos processos. Utilizando parametrização, estes processos podem tornar-se reutilizáveis.

Dois objetos são fundamentais em LOTOS: processos e eventos. Um processo [2] é um componente de uma especificação, equivalente a uma atividade em uma implementação. Por exemplo, `'transferência-de-informações'`, `'classificação-de-termos'`, `'ativação-de-alarme'`, etc. Os processos se comunicam através de um mecanismo denominado de *ponto de interação*, que equivale à abstração de uma interface em uma implementação. No caso das linguagens de especificação, este ponto de interação é chamado de *gate*. A especificação do estado de um processo é feita através de uma *expressão de comportamento*[3]. Esta expressão define o comportamento de um componente, visível externamente em termos das sequências possíveis de eventos das quais ele pode participar. Existem duas expressões pré-definidas em LOTOS: 'Stop' que representa *deadlock*, e indica que nenhum outro evento pode mais ocorrer; e 'Exit' que indica o término com sucesso de uma sequência de eventos.

Os eventos representam um ponto de sincronização entre os processos. São ações atômicas, que numa implementação real podem representar uma sequência de passos elementares. Exemplos de eventos são: 'tempo-expirado', 'pronto', 'alarme-ativado' [4], etc. Na verdade, todos os processos que participam de uma sincronização (evento) devem cooperar igualmente, ou seja, não existe o conceito de um processo iniciar a sincronização e os demais apenas responderem. Em LOTOS, três tipos de eventos são possíveis [Tur96]:

- *Sincronização pura:* nenhum valor é trocado entre os processos;

- *Disposição de valores:* um ou mais processos fornecem um valor específico, o qual deve pertencer a um conjunto aceitável aos outros processos;

[2] Utilizaremos os termos 'processo' e 'componente' de maneira intercambiável.

[3] *Behaviour expression*, do inglês.

[4] Escreveremos os nomes dos eventos em letras minúsculas; processos em estilo de `máquina`; e expressões de comportamento apenas com a primeira letra maiúscula.

- *Negociação de valores:* dois ou mais processos concordam com um determinado conjunto de valores.

As especificações escritas em LOTOS determinam a *ordem temporal* dos eventos, isto é, a ordem relativa dos eventos no tempo. Esta é a característica que torna LOTOS uma técnica algébrica com ordem temporal.

4.2.2 A Utilização de LOTOS

Como foi comentado na Seção 4.2, LOTOS foi inicialmente projetada para a especificação de padrões de telecomunicação relacionados à interconexão de sistemas heterogêneos (OSI). Neste sentido, as seguintes especificações da OSI já foram desenvolvidas [Tur96]:

- Camada de aplicação;
- Camada de apresentação;
- Camada de sessão;
- Camada de transporte;
- Camada de rede;
- Camada de dados.

Entretanto, LOTOS tem uma abrangência que vai muito além da especificação de protocolos de comunicação. Ela pode ser utilizada para a descrição tanto de sistemas sequenciais, quanto concorrentes ou distribuídos. Variados tipos de sistemas já foram especificados com LOTOS [LFH92], dentre os quais podemos citar: processamento de chamadas telefônicas, mecanismos de segurança (protocolos de autenticação), sistemas operacionais distribuídos e algoritmos distribuídos.

Nesta dissertação, em particular, estamos interessados principalmente na reutilização de componentes concorrentes, além dos sequenciais. Além disso, queremos aproveitar as vantagens oferecidas pelos métodos formais neste processo (Seção 2.5), uma vez que uma biblioteca composta por componentes formalmente definidos oferece maior qualidade e confiabilidade ao usuário. Assim sendo, com o intuito de suprir estas necessidades, adotamos a linguagem LOTOS como base para a especificação formal de nossos componentes concorrentes.

Como LOTOS oferece variados mecanismos de concorrência e sincronização, acaba facilitando a especificação dos componentes. Além disso, devido a sua modularidade, influencia o desenvolvimento de componentes coesos e parametrizados, auxiliando assim todo o processo de

reutilização. E, por fim, como é explicado nos próximos capítulos, uma extensão desejável a este trabalho é a criação de um terceiro filtro, baseado na análise do comportamento dos componentes. Para isto, o terceiro filtro deverá trabalhar com o casamento de especificações formais dos componentes. Neste caso, as propriedades algébricas de LOTOS serão importantes no processo de demonstrar a equivalência e a corretude das especificações. Além disso, seu conjunto de regras de inferência permitirá que as especificações dos componentes possam ser escritas, interpretadas e traduzidas diretamente em programas executáveis.

4.2.3 Operadores Básicos de LOTOS

Nesta seção apresentamos os tipos básicos, a sintaxe da interface de um processo, e alguns operadores de LOTOS. Fornecemos uma breve descrição daqueles operadores indispensáveis à compreensão e especificação de processos concorrentes simples em LOTOS. Uma descrição completa da linguagem pode ser encontrada em [EVD89, LFH92, Tur96]. Em seguida especificamos três processos concorrentes, exemplificando com o problema do produtor-consumidor. Nossa ênfase fica restrita aos tipos básicos, sem considerar os tipos abstratos de dados (descritos através da linguagem ACT ONE). Trabalhamos com o chamado *LOTOS básico* ou *LOTOS puro*, onde a sincronização entre os processos é obtida sem a utilização de tipos abstratos de dados [LFH92]. Como em nosso sistema os tipos de LOTOS são empregados apenas na fase de casamento de interface, a utilização dos tipos básicos já é suficiente para demonstrar a funcionalidade deste filtro. Em trabalhos futuros, este filtro poderá ser expandido, incorporando também os tipos abstratos de dados.

Em LOTOS as variáveis se comportam como variáveis matemáticas: elas não assumem o papel de um local de armazenamento, o qual recebe um determinado valor. Ao contrário, as variáveis representam simplesmente um nome o qual fica limitado a um determinado conjunto de valores [Tur96]. Os valores aos quais uma variável pode ser associada são limitados de acordo com os valores do *sort* ao qual pertence. Em outras palavras, o *sort* é o 'tipo' de uma variável, representando o conjunto de valores que a ela podem ser associados. Os tipos [5] básicos de LOTOS e que empregamos em nosso sistema são:

- Bool-Sort: os valores booleanos (*true, false*);
- Int-Sort: os números inteiros (...,-1,0,+1,...);
- Nat0-Sort: os números inteiros não-negativos (0,1,...);

[5] Utilizaremos os termos 'sort' e 'tipo' de maneira intercambiável.

- Nat-Sort: os números inteiros positivos (1,2,...).

A descrição da interface de um processo apresenta a seguinte sintaxe:

process *Nome* [*Eventos*] (*Parâmetros-de-Entrada : Tipo*) : **exit**(*Tipo*)

onde os tipos que seguem a palavra **exit** representam os tipos dos parâmetros de saída do processo. Empregando esta sintaxe, um processo que realiza a adição de dois números inteiros poderia ser descrito pelas interfaces:

1) **process** `adição` (num1, num2 : Int-Sort) : **exit**(Int-Sort)
2) **process** `adição` [num2] (num1 : Int-Sort) : **exit**(Int-Sort)

No primeiro caso, o processo não participa de nenhum evento, e a adição é feita diretamente com os dois parâmetros de entrada. No segundo caso, o processo recebe apenas um dos números como parâmetro de entrada. O outro número é informado ao processo através de um evento denominado de 'num2'.

Os operadores de LOTOS são utilizados para combinar eventos e expressões de comportamento para produzir outras expressões de comportamento (Seção 4.2.1). Entre estes operadores podemos destacar: composição sequencial, seleção, paralelismo (*interleaving* e sincronização), habilitação[6] e desabilitação[7].

Composição Sequencial Este operador, identificado através de um ';', é utilizado para ordenar sequencialmente as ações no tempo. Expressa basicamente a composição de um evento e uma expressão de comportamento. Por exemplo,

$$confirmado; Copiar - arquivo$$

denota um comportamento onde o evento 'confirmado' deve ser disparado antes da expressão 'Copiar-arquivo'.

Seleção Este operador representa a escolha entre dois ou mais comportamentos alternativos. É identificado pelo símbolo '[ ]'. Um exemplo da escolha entre dois caminhos diferentes é:

(componente-selecionado;Informar-ao-usuário)
[ ]
(componente-não-encontrado;Procurar-similar)

[6] *Enabling*, do inglês.
[7] *Disabling*, do inglês.

Paralelismo Há dois operadores de paralelismo: o operador de composição paralela por *interleaving* e o operador de composição paralela sincronizado. No primeiro caso, representado pelo símbolo '|||', o conceito de paralelismo é expresso sem a necessidade de sincronização. Os eventos de cada processo são executados de maneira intercalada e completamente independente uns dos outros. Por exemplo:

(escrever;desenhar;Imprimir-arquivo)
|||
(abrir;selecionar;indexar;Criar-banco-de-dados)

inclui, entre vários outros, os seguintes comportamentos:

escrever;abrir;selecionar;desenhar;...
abrir;escrever;selecionar;indexar;...

No caso do operador de composição sincronizado, identificado por '||', os processos devem obrigatoriamente estar sincronizados em todos os eventos observáveis. Assim, o exemplo:

(escrever;desenhar;salvar;Imprimir-arquivo)
||
(escrever;desenhar;salvar;Editar-arquivo)

pode gerar a seguinte sequência de eventos:

escrever;desenhar;salvar;...

Se apenas certos eventos precisam estar sincronizados entre os processos, pode-se empregar o operador '|[ ]|'. Assim, todos os eventos nos quais os processos devem sincronizar são explicitamente listados:

(produzir;pronto;enviar;Produtor)
|[*pronto*]|
(analisar;consumir;pronto;calcular;Consumidor)

Neste caso, o produtor fica aguardando até que o consumidor alcance o mesmo evento 'pronto', ou seja, tenha consumido o que foi produzido. A partir deste ponto, ambos os processos continuam a sequência de seus eventos de maneira independente.

Habilitação Este operador, representado pelo símbolo '>>', é empregado para combinar duas expressões de comportamento em sequência. Por exemplo, em $E1 >> E2$, a expressão E2 será executada apenas se E1 terminar com sucesso (Exit). Caso contrário, E2 nunca será inicializada.

Desabilitação O operador '[>' modela a desabilitação de um processo por outro. Por exemplo,

(enviar-dado;iniciar-tempo;receber-confirmação;Exit)
[>
(tempo-terminado;disparar-alarme;Stop)

pode terminar com sucesso, caso uma confirmação seja recebida, ou então, disparar um sinal de alarme e 'travar', caso nenhuma confirmação tenha sido recebida dentro de um determinado período de tempo.

O Exemplo do Produtor-Consumidor

Utilizando os elementos anteriormente descritos: tipos básicos, interface, e operadores, podemos apresentar um exemplo composto por três processos concorrentes. Estes processos representarão o problema do produtor-consumidor [LFH92]: o processo `produtor` gera um código que será utilizado pelo processo `consumidor` e uma indicação se o processamento teve êxito. Por sua vez, o `consumidor` recebe o código do `produtor`, processa-o e então retorna a situação da operação executada. Há ainda um terceiro processo `canal`, responsável pela sincronização entre os dois processos anteriores (Figura 4.1).

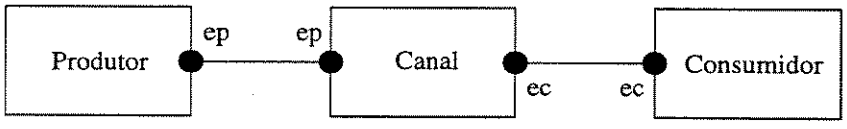

Figura 4.1: Exemplo do Produtor-Consumidor

A especificação completa de tal sistema pode ser assim definida:

specification Produtor-Consumidor [ep,ec] : **exit**

behaviour

(
`produtor` [ep]
|||
`consumidor` [ec]
)
||
`canal` [ep,ec]

where

process `produtor` [ep] : **exit**(código : Nat-Sort, estado : Bool-Sort) :=
ep; Produzir-código; **exit**(código, *true*)
[>
Problemas-na-produção; **exit**(erro, *false*)
endproc

process `consumidor` [ec](código : Nat-Sort) : **exit**(estado : Bool-Sort) :=
ec; Consumir-código; **exit**(*true*)
[>
Problemas-no-consumo; **exit**(*false*)
endproc

process `canal` [ep,ec] : **exit** :=
ep; ec; Exit
endproc

endspec

Neste exemplo, os processos `produtor` e `consumidor` têm seus eventos executados em paralelo (*interleaving*). Ao mesmo tempo, ambos estão sincronizados com o processo `canal`, através dos eventos 'ep' e 'ec'. No caso de haver algum problema durante os processamentos (representado pelo operador de desabilitação), ambos os processos retornam uma indição de erro.

Uma outra maneira de escrever uma especificação com comportamento semelhante, porém utilizando apenas o paralelismo com sincronização por listagem de eventos seria:

specification Produtor-Consumidor [ep,ec] : **exit**

behaviour
(
`produtor` [ep]
|[*ep*]|
`canal` [ep,ec]
|[*ec*]|
`consumidor` [ec]

where...

Até este ponto, definimos todas as características de LOTOS que permitem a especificação de componentes concorrentes. Nas próximas seções detalharemos o funcionamento do segundo filtro de nosso sistema, baseado no casamento de interfaces descritas em LOTOS.

4.3 O que é Casamento de Interface?

O casamento de interface é um processo para se determinar quais componentes em uma biblioteca possuem em sua função principal, a interface que o usuário procura. Por interface, denotamos o conjunto de tipos dos parâmetros de entrada e parâmetros de saída de uma função. Por exemplo, a função que realiza a adição de dois números inteiros, descrita na Seção 4.2.3, possui a seguinte especificação de interface:

(Int-Sort, Int-Sort) $\rightarrow$ (Int-Sort)

Neste caso, a função recebe como parâmetros de entrada dois números do tipo inteiro e retorna outro número inteiro como parâmetro de saída. De forma mais geral, o casamento de interface pode ser definido como [Zar96]:

$$Casamento(C, P, B) = \{c \in B : P(c, C)\}$$

Em outras palavras, dada uma consulta C, um predicado de casamento (exato ou parcial) P, e uma biblioteca de componentes B, o casamento de interface retorna o conjunto de componentes c que satisfazem o referido predicado.

A grande vantagem deste método reside essencialmente na simplicidade de sua estrutura. Definir os parâmetros de uma função é um trabalho relativamente simples, que não impõe grandes problemas a um usuário/programador. Em muitos casos, dependendo da linguagem em que o componente foi desenvolvido, é possível determinar seus parâmetros através de uma análise direta em seu código-fonte. Por outro lado, utilizar descrições de interfaces para selecionar componentes, significa que os usuários trabalharão com informações e consultas em uma linguagem que eles já dominam [Zar96]. A semântica do esquema (sistema de tipos de uma linguagem de programação) já é conhecida, o que facilita o aprendizado e a utilização da ferramenta de seleção.

Neste esquema de seleção, a questão central é determinar a quantidade e o tipo dos parâmetros da função desejada pelo usuário. Com esta informação como referência, são identificados na biblioteca todos os componentes que apresentam uma interface sintaticamente semelhante à procurada. De maneira geral, são selecionados apenas aqueles que possuem a mesma quantidade e tipo dos parâmetros fornecidos pelo usuário.

De acordo com as necessidades do usuário (quantidade e precisão dos componentes), o predicado que realiza o casamento sintático pode ser alterado. Em trabalhos disponíveis na área [FKS94, Zar96], exemplos de níveis de casamento vão desde o exato, passando pelos parciais com reordenação de tipos e eliminação de 'parênteses', até uma combinação de vários tipos de casamentos parciais. Na próxima seção, explicaremos em detalhes os conceitos envolvidos nestes diferentes níveis de casamento.

4.4 Níveis de Casamento de Interface

O casamento de interface depende basicamente do predicado utilizado para estabelecer a relação sintática entre uma consulta e os componentes na biblioteca. De acordo com o nível de precisão desejado, o casamento pode variar de exato a parcial.

No casamento exato, a quantidade de componentes recuperados pode ser extremamente pequena. Entretanto, tais componentes apresentam uma interface sintaticamente idêntica à procurada. Por outro lado, no casamento parcial o número de componentes selecionados aumenta consideravelmente. Em contrapartida, a precisão da consulta é reduzida, uma vez que as interfaces dos componentes podem ser um tanto diferentes do que o usuário realmente está procurando.

4.4.1 Casamento Exato

Um componente candidato C_c presente na biblioteca casa exatamente com um componente C_p procurado pelo usuário caso eles possuam o mesmo conjunto de tipos em seus parâmetros [OHPB92]. Por exemplo, todos os componentes que possuírem em suas interfaces exatamente dois números inteiros como entrada e um inteiro como saída, serão selecionados (casarão) com a especificação da interface da função de adição anteriormente apresentada (Seção 4.3).

De maneira geral, o casamento exato é definido por:

$$Casamento_E(C_c, C_p) = \{i_c =_L i_p\}$$

onde i_c corresponde à interface do componente candidado C_c e i_p é a interface especificada pelo usuário, do componente procurado C_p. O símbolo $=_L$ indica uma igualdade léxica entre os tipos

descritos nas duas interfaces. Neste caso, a ordem ou posição dos tipos dentro da interface é levado em consideração. Assim, $(Int-Sort, Bool-Sort)$ e $(Bool-Sort, Int-Sort)$ representam duas interfaces distintas.

O emprego do casamento exato como único método de seleção torna-se extremamente restritivo. Podem existir muitos componentes na biblioteca que não 'casam' exatamente com a consulta, mas que são de alguma forma similares ao procurado. Como exemplo, as seguintes definições, apesar de representarem a mesma função, seriam consideradas completamente diferentes:

(Int-Sort, Int-Sort, Bool-Sort) → (Bool-Sort)
(Bool-Sort, Int-Sort, Int-Sort) → (Bool-Sort)
((Int-Sort, Int-Sort), Bool-Sort) → (Bool-Sort)
(Bool-Sort, (Int-Sort, Int-Sort)) → (Bool-Sort)

No casamento exato, a posição dos tipos e dos 'parênteses' dentro da especificação de uma interface é levada em consideração. Em outras palavras, tal nível de precisão acaba eliminando do conjunto final componentes também úteis que poderiam ser modificados e passariam a exercer exatamente a função desejada. O usuário não precisaria começar a escrever uma função do 'zero'; ao contrário, iria selecionar componentes similares, analisar suas diferenças e, caso fossem pequenas, modificar suas estruturas.

Para evitar este tipo de problema, além do casamento exato, consideramos também os casos de casamentos parciais de interface. Nestes, os parâmetros de uma consulta podem ser ligeiramente modificados, recuperando uma quantidade maior de componentes.

4.4.2 Casamento Parcial

No casamento parcial são reduzidas as imposições exigidas pelo casamento exato. Neste caso, podem ser aplicados dois novos graus de casamento: um casamento com reordenação de tipos e um casamento com eliminação de parênteses.

Reordenação de Tipos

É muito improvável que o usuário defina corretamente a posição exata dos tipos na descrição de uma interface. Além disso, é irrelevante do ponto de vista da seleção, a ordem específica mantida dentro da interface. Sendo assim, podemos utilizar uma alternativa ao casamento exato através de uma reordenação dos tipos [ZW95a, FKS94]:

$$Casamento_R(C_c, C_p) = \{\exists \textit{ uma sequência de permutações } M \textit{ na posição dos tipos, tal que } Casamento_E(M(i_c) = i_p)\}$$

onde i_c corresponde à interface do componente candidado C_c; $M(i_c)$ à permutação dos tipos dentro da interface do componente candidato; e i_p é a interface especificada pelo usuário, do componente procurado C_p. Neste caso, por exemplo, suponha a seguinte descrição de uma interface:

(Nat-Sort, Nat0-Sort) → (Nat-Sort, Bool-Sort)

a qual representa uma função que recebe como parâmetros de entrada dois números naturais e retorna o resultado da operação com os mesmos, juntamente com um valor booleano indicando algum erro. Aplicando o casamento com reordenação de tipos, esta interface casaria com todas as outras, cuja mudança na ordem dos tipos as tornaria exatamente equivalentes:

(Nat-Sort, Nat0-Sort) → (Nat-Sort, Bool-Sort)
(Nat0-Sort, Nat-Sort) → (Nat-Sort, Bool-Sort)
(Nat-Sort, Nat0-Sort) → (Bool-Sort, Nat-Sort)
(Nat0-Sort, Nat-Sort) → (Bool-Sort, Nat-Sort)

Eliminação de 'parênteses'

Uma interface, assim como qualquer definição matemática, poderia apresentar em sua estrutura, uma determinada quantidade de 'parênteses'. No casamento exato, este detalhe também poderia dificultar a seleção de componentes com interfaces similares. Neste caso, as seguintes descrições do mesmo componente seriam consideradas interfaces completamente diferentes:

(Int-Sort, Int-Sort, Bool-Sort) → (Bool-Sort)
((Int-Sort, Int-Sort), Bool-Sort) → (Bool-Sort)
((Int-Sort, Int-Sort), Bool-Sort) → Bool-Sort

Para resolver este problema o mais simples a ser feito é, no momento do casamento, eliminar de todas as interfaces seus respectivos 'parênteses'. A interface passa a ser analisada apenas pelo seu conteúdo e não por sua definição léxica. Em geral, este tipo de casamento é assim representado:

$$Casamento_{EP}(C_c, C_p) = \{\exists \textit{ uma sequência de eliminação de parênteses } P, \textit{ tal que } Casamento_E(P(i_c) = i_p)\}$$

onde i_c corresponde à interface do componente candidato C_c; $P(i_c)$ representa a interface do componente candidato sem a presença de parênteses; e i_p é a interface especificada pelo usuário, do componente procurado C_p. Dessa forma, todas as descrições anteriores poderiam ser agrupadas em uma única representação:

Int-Sort, Int-Sort, Bool-Sort → Bool-Sort

É importante salientar que esta fase de 'eliminação de parênteses' só se aplica pelo fato de estarmos manipulando apenas os tipos básicos de LOTOS. Em outros trabalhos na área [OHPB92, FKS94, ZW95a], os quais tratam também os tipos abstratos de dados, o emprego de uma eliminação como esta, poderia modificar sensivelmente a definição de um tipo e, consequentemente, os resultados finais.

Combinação de Casamentos Parciais

Cada casamento parcial é, individualmente, um mecanismo útil à seleção de componentes a partir de uma dada interface. Entretanto, a combinação dos dois níveis de casamento parcial pode ampliar o conjunto de componentes recuperados [ZW95a]. Voltando aos exemplos da Seção 4.4.1:

1. (Int-Sort, Int-Sort, Bool-Sort) → (Bool-Sort)
2. (Bool-Sort, Int-Sort, Int-Sort) → (Bool-Sort)
3. ((Int-Sort, Int-Sort), Bool-Sort) → (Bool-Sort)
4. (Bool-Sort, (Int-Sort, Int-Sort)) → (Bool-Sort)

O casamento por reordenação de tipos permitiria que uma consulta do tipo (1) ou (2) casasse com funções dos tipos (1) ou (2), mas não com os tipos (3) ou (4). O casamento com eliminação de parênteses, por sua vez, permitiria que uma consulta do tipo (3) casasse com funções do tipo (1); e uma consulta do tipo (4) casasse com o tipo (2). Entretanto, nenhuma consulta seria capaz de casar com todos os tipos de (1) a (4). Para obter tal resultado, podemos fazer uma combinação dos dois tipos de casamentos parciais, obtendo o casamento parcial combinado:

$$Casamento_P(C_c, C_p) = \{Casamento_R(Casamento_{EP}(i_c, i_p))\}$$

Assim, após ser realizada a eliminação dos 'parênteses' dentro da interface, é possível fazer uma mudança na ordem destes tipos. Aplicando este casamento combinado, uma consulta

empregando a sintaxe de uma das interfaces anteriores, seria capaz de selecionar as quatro funções ao mesmo tempo.

O esquema de casamento de interface que explicamos até agora baseia-se no emprego de uma rigorosa estrutura sintática. Como apresentamos, duas interfaces são consideradas similares apenas se possuírem exatamente a mesma sintaxe, ou então, se for possível mudar a posição de seus tipos e eliminar seus 'parênteses' de maneira a torná-las idênticas. No intuito de reduzir estas restrições, apresentamos um novo esquema simplificado de casamento, que leva em consideração apenas as informações relacionadas à quantidade e tipo dos parâmetros. É sobre este esquema que estaremos discutindo na próxima seção.

4.5 Casamento de Interface Simplificado

O método que aqui iremos apresentar, basicamente é uma simplificação de outros trabalhos na área de casamento de interface (*signature matching*) [FKS94, Zar96]. Nestes, a interface de uma função deve ter rigorosamente a mesma estrutura sintática da função descrita pelo usuário, para que possa haver um 'casamento'. Isto significa que na descrição das interfaces (consulta e componente), todos os tipos e outras informações (tais como os 'parênteses') devem estar exatamente na mesma posição.

Como é muito claro, este tipo de raciocínio acaba tornando-se extremamente limitado: o usuário deve conhecer muito bem a sintaxe dos componentes presentes na biblioteca, ou então, ter muita 'sorte' na definição correta da interface procurada. Caso contrário (mais provável de ocorrer), o número de componentes recuperados é muito pequeno. Para resolver este problema, pode-se recorrer a formas de casamento mais flexíveis, incluindo a reordenação dos tipos dentro de uma interface, bem como a eliminação dos 'parênteses' (Seção 4.4.2) durante o processo de seleção. Isto acaba exigindo o desenvolvimento e a aplicação de algoritmos relativamente complexos de unificação e transformação de tipos [OHPB92, ZW95a].

Com o intuito de abstrair estas restrições e tornar o processamento do filtro de seleção mais simples e direto, eliminamos a necessidade da definição léxica da estrutura da interface. Como estamos trabalhando apenas com os tipos básicos de LOTOS, o mais importante não é a ordem, mas sim a estrutura sintática representada pelos mesmos. Em outras palavras, ao sistema cabe definir apenas a quantidade e o tipo dos parâmetros (entrada/saída) da função procurada, sem se importar com a disposição dos mesmos dentro da interface (o que é irrelevante do ponto de vista da funcionalidade do componente). Desta forma, ao invés de considerar cada uma das seguintes descrições:

1. (Int-Sort, Int-Sort, Bool-Sort) → (Bool-Sort)
2. (Bool-Sort, Int-Sort, Int-Sort) → (Bool-Sort)
3. ((Int-Sort, Int-Sort), Bool-Sort) → (Bool-Sort)
4. (Bool-Sort, (Int-Sort, Int-Sort)) → (Bool-Sort)

em nosso método, todas seriam agrupadas em uma única interface:

> Parâmetros de entrada: 2 Int-Sort e 1 Bool-Sort;
> Parâmetro de saída: 1 Bool-Sort

Assim, todos os componentes que possuírem os tipos anteriores são selecionados, independentemente da interface específica de cada um. Para obter este resultado, aplicamos um predicado que compara apenas a quantidade e o tipo dos parâmetros em cada interface, sem considerar sua descrição léxica. Basicamente aplicamos a idéia de 'intersecção' da Teoria de Conjuntos. O primeiro conjunto é composto pelos tipos da própria interface procurada, enquanto o segundo conjunto são os tipos de cada interface presente na biblioteca de componentes. Após a intersecção entre estes conjuntos, recuperamos todos aqueles componentes que apresentam os mesmos tipos da interface procurada.

Através desse esquema, os casamentos parciais com reordenação de tipos e com eliminação de 'parênteses' (Seção 4.4.2) tornam-se naturalmente desnecessários. Criamos então um terceiro tipo de casamento parcial, o qual faz uso de tipos genéricos nas interfaces. Quando o usuário não sabe exatamente os tipos específicos que procura, este nível de casamento parcial torna-se extremamente útil, como descreveremos na próxima seção.

4.5.1 Tipos Genéricos

Algumas vezes o usuário não conhece exatamente quais os tipos que poderiam estar presentes em uma interface. Para estes casos, criamos dois novos tipos genéricos: um tipo X para os parâmetros de entrada e um tipo Y para os parâmetros de saída. Estes dois tipos assumem o papel de variáveis, casando com qualquer outro tipo listado na interface. Por exemplo, supondo que o usuário especifique a seguinte interface genérica:

> **process** `exemplo-1` (num1, num2 : X) : **exit**(Y)

ou de maneira equivalente:

Parâmetros de entrada: 2 X;
Parâmetro de saída: 1 Y

Esta interface irá casar com qualquer outra que possua dois parâmetros de entrada e um de saída, independentemente de quais sejam seus tipos específicos. Desse modo, todas as interfaces apresentadas na Tabela 4.1 seriam selecionadas, além de várias outras semelhantes.

Função	*Parâmetros de Entrada*	*Parâmetros de Saída*
1	2 Int-Sort	1 Int-Sort
2	2 Nat0-Sort	1 Nat0-Sort
3	1 Int-Sort; 1 Nat-Sort	1 Bool-Sort
4	1 Nat-Sort; 1 Bool-Sort	1 Bool-Sort
5	2 Nat0-Sort	1 Bool-Sort
⋮	⋮	⋮

Tabela 4.1: Exemplos de interfaces

De maneira geral, o casamento baseado em tipos genéricos é equivalente a:

$$Casamento_G(C_c, C_p) = \{\exists \textit{ uma sequência de substituição de variáveis } V, \textit{ tal que } Casamento_E(V(i_p) = i_c)\}$$

onde V corresponde à substituição das variáveis X e Y da interface procurada i_p, pelos tipos específicos de cada interface candidata i_c.

A vantagem deste casamento é que o usuário não precisa conhecer quais são os tipos da função procurada, mas apenas a quantidade dos parâmetros, o que corresponde a um valor relativamente mais simples de se determinar. Entretanto, como podemos perceber pelo exemplo anterior, quanto mais genericamente for especificada a interface procurada, maior será a quantidade de componentes irrelevantes recuperados. Ou seja, quando tentamos aumentar o *recall* de uma seleção, naturalmente reduzimos sua precisão (Seção 3.2.1). Para evitar este problema, o usuário deve procurar utilizar a menor quantidade possível de tipos genéricos. Por exemplo, a seguinte interface teria um nível de precisão consideravelmente maior do que a anterior:

process `exemplo-2` (num1 : Int-Sort, num2 : X) : **exit**(Bool-Sort)

ou

Parâmetros de entrada: 1 Int-Sort e 1 X;
Parâmetro de saída: 1 Int-Sort

Com esta interface, os componentes recuperados seriam todos aqueles que tivessem como parâmetros de entrada um número inteiro juntamente com outro tipo qualquer, e como saída outro valor inteiro.

De certa forma, o esquema que empregamos em nosso sistema nada mais é do que a combinação dos três tipos de casamentos parciais que já definimos:

$$Casamento_C(C_c, C_p) = \{Casamento_R(Casamento_{EP}((Casamento_G(i_c, i_p)))\}$$

Assim, é realizada uma sequência de substituição das variáveis genéricas por tipos específicos; seguida pela eliminação de 'parênteses'; e, finalmente, uma mudança na ordem dos tipos.

4.6 Conclusão

O método de seleção de componentes baseado em um casamento sintático de tipos se apresenta bastante simples e eficiente. Empregar conceitos com que o usuário já está familiarizado (sistema de tipos) facilita e, ao mesmo tempo, reduz o impacto da utilização de uma ferramenta deste tipo na prática.

O esquema que apresentamos é uma tentativa de tornar mais simples o casamento de interface desenvolvido em outras pesquisas na área [FKS94, Zar96]. Ao contrário de tais pesquisas, nosso intuito foi trabalhar apenas com tipos básicos e, assim, conseguir abstrair os detalhes de uma interface, tais como a ordem dos termos e a presença de 'parênteses'. Procuramos criar uma maneira de trabalhar apenas com suas características mais importantes: o tipo e a quantidade dos parâmetros de entrada e saída de uma função.

Dessa forma, o número de componentes recuperados (*recall*) aumenta sensivelmente. Entretanto, esta maior flexibilidade também tem seu lado negativo: caso o método de seleção simplificado baseado em interfaces seja utilizado como o único filtro de um processo de reutilização, a quantidade final de componentes selecionados pode tornar-se tão grande que acaba inviabilizando uma possível análise dos mesmos. Assim sendo, em nosso sistema este método deve ser empregado como o *segundo* filtro no processo de seleção. O objetivo é reduzir o número de componentes selecionados pelo primeiro filtro (classificação por atributos) e entregar ao usuário somente aqueles componentes que manipulam os tipos adequados. Assim, conseguimos selecionar componentes especificados formalmente em LOTOS, utilizando apenas seu sistema de tipos básicos, o qual é simples e bastante estruturado.

Neste capítulo detalhamos variados níveis de casamento: exato, parcial com reordenação de tipos, parcial com 'eliminação de parênteses' e parcial com o emprego de tipos genéricos. Fornecemos também uma visão geral das características da linguagem LOTOS, cujo sistema de

tipos empregamos para implementar o segundo filtro de nosso sistema de classificação e seleção de componentes. Tal sistema, juntamente com seus dois filtros de seleção será explicado em detalhes no próximo capítulo.

Capítulo 5

Implementação do Sistema

Neste capítulo descreveremos as características relacionadas à implementação de nosso sistema de classificação e seleção de componentes de software concorrentes. Para tanto, explicaremos suas estruturas de dados, bem como as ferramentas auxiliares utilizadas no seu desenvolvimento.

Para ilustrar as facilidades da utilização desse sistema, apresentaremos um exemplo completo de um processo de seleção, juntamente com todas as etapas necessárias para a concretização de tal tarefa.

5.1 Introdução

A reutilização de software, na prática, ainda é pouco utilizada [McC97]. Uma das razões é a falta de ferramentas simples, que realmente facilitem o trabalho do usuário durante o desenvolvimento de um novo sistema. Os esquemas de representação de software disponíveis exigem que o usuário possua um conhecimento prévio substancial sobre o método em questão [Nei89, JC93, FKS94, ZW95b]. Por outro lado, os componentes geralmente são dedicados a um domínio específico ou a uma única linguagem de programação [KRT89, PP92]. Além disso, tais métodos são desenvolvidos com o objetivo de manipular apenas componentes cuja execução seja sequencial, ignorando os componentes concorrentes. Assim sendo, como alguns estudos apontam [FP94, MMM97], tais pesquisas não conseguem alcançar um ponto médio entre os esquemas formais (mais confiáveis) e os esquemas mais simples e informais (mais utilizados na prática).

Com o intuito de superar estas limitações, criamos uma ferramenta simples e eficiente, direcionada especificamente ao usuário dotado de um conhecimento básico sobre as estruturas de uma linguagem de programação e de especificação formal. Este sistema, como já foi comentado nos capítulos anteriores, está centrado em dois filtros principais: 1. classificação por atributos

(Capítulo 3) e 2. casamento de interfaces (Capítulo 4).

O primeiro filtro emprega termos e atributos, sendo independente de qualquer linguagem de programação. Dessa forma, o usuário é capaz de selecionar um conjunto inicial de componentes candidatos (concorrentes ou sequenciais), fornecendo apenas as características básicas da função que deseja recuperar. O segundo filtro, casamento de interfaces, baseia-se na linguagem de especificação formal LOTOS. Neste caso, o usuário deve possuir um conhecimento mínimo da linguagem, direcionando sua atenção ao sistema de tipos básicos de LOTOS.

Uma característica importante é que o sistema foi projetado de maneira a tornar os dois filtros totalmente independentes um do outro, podendo ser empregados em conjunto ou separadamente. O objetivo foi desenvolver um novo sistema que pudesse ser utilizado de forma natural, como uma extensão ao ambiente de trabalho do usuário.

O restante deste capítulo é dividido da seguinte maneira: na Seção 5.2 são apresentadas as características do filtro de classificação e seleção por atributos. Detalhamos as estruturas de dados utilizadas para armazenar as informações, enfatizando a importância da rede de termos semânticos, responsável pela seleção de componentes similares ao desejado pelo usuário. Na Seção 5.3 são descritos os detalhes do filtro por casamento de interfaces LOTOS. Em seguida, a Seção 5.4 explica as ferramentas utilizadas no desenvolvimento do sistema: a linguagem de programação, o banco de dados e a interface gráfica. Na Seção 5.5 é apresentado um exemplo completo de utilização do sistema, juntamente com todos os passos necessários para a seleção de um determinado componente. E, finalmente, as conclusões obtidas com este capítulo são descritas na Seção 5.6.

5.2 Primeiro Filtro: Classificação por Atributos

Este primeiro filtro incorpora as características descritas no Capítulo 3. Ele é composto basicamente por seis atributos: função, objeto, meio, processamento, comunicação e composição; juntamente com seus respectivos termos descritivos.

O usuário seleciona um componente na biblioteca, simplesmente escolhendo seis termos que melhor descrevem a função procurada. Caso esta consulta não retorne nenhum componente, ao usuário é fornecida a opção de expandir a consulta, recuperando componentes parcialmente semelhantes ao procurado. A característica mais importante deste esquema é que a classificação e a seleção são feitas de maneira independente de qualquer linguagem de programação. O usuário precisa apenas descrever a função que procura através de alguns termos, enquanto o sistema será responsável por 'vasculhar' a biblioteca, à procura daqueles componentes que possuem tais termos em suas descrições.

Entretanto, antes de ser possível a execução de uma consulta propriamente dita, faz-se necessária a definição de um conjunto de termos para cada atributo, juntamente com seus termos sinônimos e termos semanticamente relacionados.

5.2.1 A Criação do Conjunto de Termos Descritivos

Para cada um dos seis atributos deve ser relacionado um conjunto de termos significativos, capaz de representá-lo e descrevê-lo de maneira abrangente. Assim sendo, para tornar-se realmente representativo, cada um destes conjuntos pode ser composto por um número extremamente alto de termos. Com o intuito de tornar o processo de seleção rápido, apesar desta grande quantidade de termos, decidimos armazená-los em uma estrutura de dados denominada de 'árvore-AVL' [Wir89]. Através desta estrutura, os termos são ordenados sintaticamente, criando uma árvore binária balanceada, onde todos os seus 'ramos' possuem a mesma altura. No pior dos casos, o número de termos percorridos até se alcançar aquele procurado é no máximo $\log n$, onde n é a quantidade total de termos presentes na árvore. Dessa forma, o emprego de tal estrutura torna uma consulta a qualquer um dos termos armazenados extremamente eficiente e rápida.

Durante a definição do conjunto de termos de cada atributo, podem surgir vários termos que possuem significados semelhantes e que poderiam ocasionalmente confundir o usuário no momento da seleção. Para resolver este problema, criamos um dicionário de termos sinônimos, no qual à cada termo é associado um conjunto de outros termos que possuem significados semelhantes (Seção 3.4.1). Assim, dentro de um determinado conjunto de termos sinônimos deve ser escolhido um único termo representativo, o qual será o termo que poderá ser utilizado na definição de uma consulta. Caso o usuário especifique um termo sinônimo numa operação de classificação/seleção, o sistema imediatamente sugere sua substituição pelo termo representativo do conjunto do qual faz parte. Como o número de sinônimos para cada termo geralmente é pequeno, escolhemos armazená-los em uma estrutura mais simples, na forma de uma 'lista encadeada'.

Aplicando a mesma idéia anterior, para cada termo é criado um conjunto de termos semanticamente semelhantes (Seção 3.4.2). Estes termos semânticos são fundamentais no momento de selecionar componentes que não possuem 'exatamente' a funcionalidade desejada pelo usuário mas que, no entanto, são semelhantes e que através de modificações podem passar a se comportar da maneira esperada. Assim, a cada termo é relacionada uma 'lista encadeada', contendo os termos semânticos e suas respectivas distâncias (pesos) ao termo original (dentro da lista, os termos são ordenados de acordo com suas distâncias). Quanto menor for este número, mais parecidos são os dois termos (termos sinônimos possuem peso 0).

Um esquema geral das estruturas utilizadas na implementação deste filtro por atributos é apresentado na Figura 5.1.

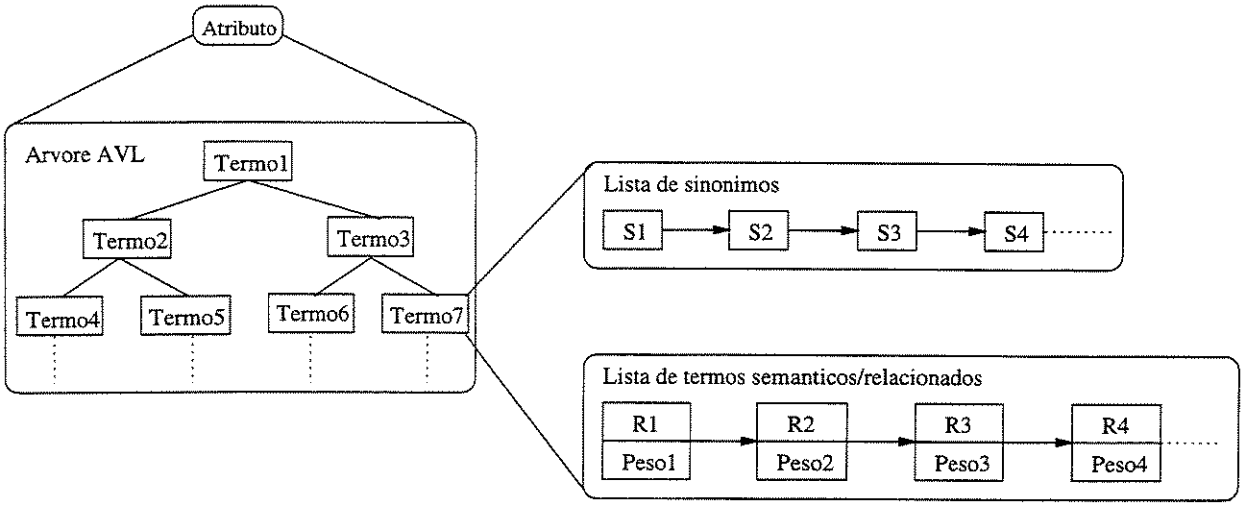

Figura 5.1: Esquema geral das estruturas do primeiro filtro

5.2.2 Classificação e Seleção através de Atributos

Uma vez estabelecido o conjunto de termos para cada um dos seis atributos, o usuário pode começar a classificar seus componentes criando, então, uma biblioteca. Para cada componente são escolhidos os termos que melhor descrevem sua funcionalidade e comportamento gerando, assim, seu identificador. Também são armazenados seu nome ou código, juntamente com uma pequena descrição do componente.

Depois de formada a biblioteca, esta será utilizada sempre que o usuário necessitar de uma determinada função, durante o desenvolvimento de um novo sistema. Ao invés de criar a referida função a partir do 'zero', o usuário terá a opção de procurá-la dentre os componentes anteriormente desenvolvidos e classificados. Para isto, deve descrever a função procurada, utilizando os termos disponíveis ou então o símbolo $*$. Neste caso, o símbolo $*$ significa 'qualquer' e indica que, para o respectivo atributo, o usuário não tem interesse em nenhum termo em particular. Assim, 'qualquer' termo que os componentes possuam em seus identificadores para este atributo em particular poderá 'casar' com a consulta. No caso do usuário escolher os seis termos como $*$, o sistema obviamente retornará todos os componentes presentes na biblioteca.

Caso esta consulta inicial não consiga recuperar nenhum componente da biblioteca, o usuário pode então expandi-la. Através da indicação da distância máxima de expansão para cada termo,

o sistema monta uma rede semântica para recuperar componentes similares ao desejado. O funcionamento desta rede de proximidade será explicado em detalhes na próxima seção.

5.2.3 Implementação da Rede Semântica de Termos

Nesta seção daremos um enfoque especial à estrutura mais importante do filtro de classificação: a rede semântica. Através desta estrutura, o sistema é capaz de recuperar componentes similares ao desejado pelo usuário aumentando, assim, o *recall* (Seção 3.2.1) de uma determinada consulta.

Como foi anteriormente comentado, a cada um dos seis atributos é relacionada uma árvore-AVL, contendo todos os seus termos ordenados. Para cada termo, são criadas duas listas-encadeadas: uma responsável pelo armazenamento de sinônimos e a outra pelos termos semanticamente próximos. É nessa segunda lista onde ficam discriminadas as distâncias entre os termos, sendo utilizada no momento da expansão de uma consulta.

A atividade de expandir uma consulta procede da seguinte maneira: após ser estabelecida pelo usuário a distância máxima de expansão de um determinado termo, o sistema começa a criar uma rede de termos semânticos. Na verdade, o sistema monta uma estrutura semelhante a uma árvore, onde a raiz é o termo a ser expandido e suas folhas, num primeiro nível, são compostas por seus termos semânticos, ordenados por suas respectivas distâncias. Da mesma forma, para cada folha é criada uma sub-árvore, composta por seus termos semânticos, e assim por diante, de maneira recursiva. Em seguida, esta árvore é percorrida em 'profundidade', somando-se as distâncias até que se alcance o nível de similaridade pretendido. Neste ponto a busca termina e são recuperados todos os componentes que possuem algum dos termos expandidos em seus identificadores.

Um exemplo simples deste processo é apresentado na Figura 5.2. Nesta são descritos alguns termos, juntamente com suas respectivas listas encadeadas de termos semanticamente relacionados. Para cada termo semântico é indicada sua distância ao termo principal. Já na Figura 5.3 é criada a árvore completa de expansão do termo 'concatenar', a partir destes conjuntos de listas. Para cada termo são apresentados dois números: um indicando sua distância ao termo 'pai' (ou raíz de sua sub-árvore) e o outro (à direita) sua distância total ao termo 'concatenar'.

Um outro exemplo mais completo deste procedimento, porém num nível mais alto de abstração pode ser encontrado na Seção 3.4.2.

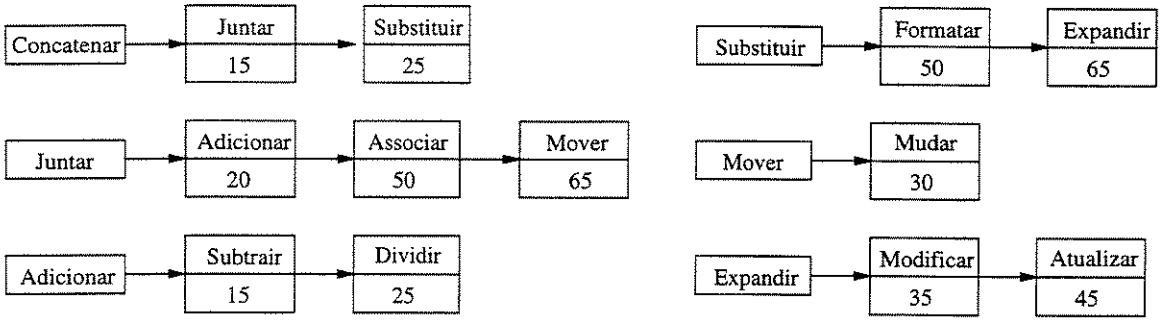

Figura 5.2: Conjunto de termos semânticos

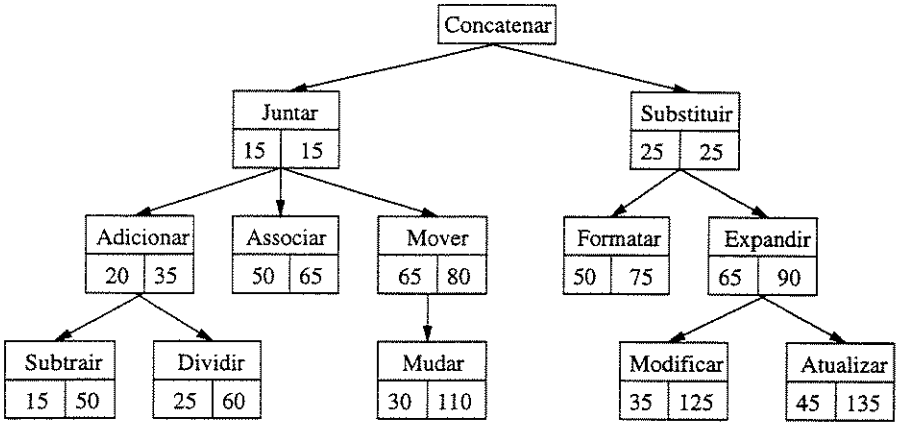

Figura 5.3: Rede/Árvore de termos semânticos

5.3 Segundo Filtro: Casamento de Interfaces

Este filtro é a realização dos detalhes descritos no Capítulo 4. Sua característica principal é estar situado em um nível mais formal do que o filtro anterior, pois aqui o objetivo é trabalhar com interfaces de especificações formais escritas em LOTOS. Dessa forma, o usuário deve possuir um conhecimento básico da estrutura de uma linguagem de especificação formal, principalmente no que diz respeito ao seu sistema de tipos. Como em nosso sistema decidimos trabalhar apenas com os tipos básicos de LOTOS (pelo fato de se tratar de um protótipo inicial), este conhecimento pode ser adquirido sem maiores dificuldades.

Após ser fornecida a especificação em LOTOS da interface do componente desejado, o sistema modifica esta interface, colocando-a em um formato interno simplificado. Neste formato final, são armazenadas apenas suas características mais importantes: o número e o tipo dos parâmetros de entrada e saída da função. A partir daí, o sistema realiza uma pesquisa na biblioteca, identificando aqueles componentes que possuem interfaces semelhantes em suas descrições.

É importante salientar que em nosso sistema este filtro deve ser utilizado sobre o subconjunto de componentes recuperados através do filtro de classificação por atributos e não sobre a biblioteca de componentes como um todo. Isto porque o casamento de interfaces trabalha com tipos, sendo que vários componentes podem ser especificados com a mesma quantidade e tipo de parâmetros. Em outras palavras, conseguimos deliberadamente aumentar o *recall*, em detrimento da precisão. Para contornar esta situação podemos utilizar inicialmente a classificação por atributos, reduzindo, assim, o espaço de busca para o filtro subsequente. Dessa forma, ao utilizar os dois filtros em conjunto, obtemos um grupo final de componentes onde todos executam a função desejada (ou alguma função semelhante), ao mesmo tempo em que possuem apenas a quantidade e tipo dos parâmetros exigidos pelo usuário.

5.3.1 A Especificação das Interfaces

A interface de cada componente é armazenada na biblioteca, no momento em que o mesmo é classificado no primeiro filtro. Depois de selecionados os seis atributos que o descrevem, o usuário pode fornecer um 'sétimo atributo', representando a especificação formal de sua interface em LOTOS.

É importante esclarecer que este é o único ponto onde os dois filtros se tornam interdependentes. A partir daí, o processo de seleção pode tomar direções diferentes, de acordo com as necessidades do usuário. Caso queira fazer uma seleção mais refinada, o usuário pode utilizar inicialmente o filtro por classificação e, em seguida, empregando o conjunto recuperado no primeiro passo, realizar a seleção com base em suas interfaces. De outra forma, tem a opção de utilizar ambos os filtros de maneira independente, recuperando conjuntos distintos de componentes em uma única etapa. O objetivo foi oferecer a maior flexibilidade possível ao usuário, fornecendo uma ferramenta na qual possa decidir o caminho aparentemente mais adequado para selecionar um certo componente.

Além dos tipos básicos de LOTOS (Bool-Sort, Int-Sort, Nat0-Sort, Nat-Sort), criamos dois novos tipos genéricos, denominados de 'X' e 'Y'. Estes fornecem ao usuário a possibilidade de abstrair os tipos específicos procurados em uma interface. Neste caso, é preciso fornecer apenas a quantidade de parâmetros de entrada ou saída, sem se importar com os tipos dos mesmos. Esta funcionalidade pode ser útil numa atividade de, por exemplo, analisar quantos componentes na biblioteca possuem dois parâmetros como saída, independentemente de quais sejam seus tipos específicos [Zar96].

5.4 A Implementação do Sistema

Nosso sistema de classificação e seleção foi desenvolvido utilizando-se a linguagem de programação C, dentro do ambiente X-Windows.

Procuramos criar uma interface simples, que pudesse ser utilizada em um ambiente de desenvolvimento de software sem causar grande impacto ao usuário final. Assim sendo, criamos uma interface gráfica, composta em grande parte por listas nas quais o usuário simplesmente escolhe os termos ou componentes que deseja. Essa interface gráfica foi desenvolvida empregando-se a ferramenta XForms.

Com relação à base utilizada para armazenar os componentes, procuramos selecionar um banco de dados simples, que possuísse uma ampla integração com a linguagem C. Nossa principal preocupação não foi com seu desempenho, mas sim com sua capacidade de realizar consultas simples, armazenamentos eficientes e comunicação direta com C. Escolhemos então o banco de dados relacional MSql (MiniSql), o qual como o próprio nome já diz, implementa uma parte da funcionalidade da linguagem SQL. Ele é um banco de dados relativamente fácil de ser manipulado, pequeno, mas que possui todas as funções de que precisamos para implementar nosso sistema.

Ambas as ferramentas empregadas – XForms [1] e MSql [2] – estão disponíveis gratuitamente na Internet, juntamente com todas as suas documentações.

5.5 Exemplo de Utilização do Sistema

Nesta seção, forneceremos um exemplo completo da utilização de nosso sistema. Descreveremos, passo-a-passo, juntamente com as próprias telas do sistema, o processo de seleção de um determinado componente da biblioteca.

O objetivo deste exemplo será encontrar um componente cuja função principal seja indicar em qual linha de um arquivo-texto se encontra um determinado identificador numérico. O detalhe é que este processo deve ser executado concorrentemente com o processo que o disparou. Para isto o componente deve receber um arquivo-texto e um número como parâmetros de entrada e retornar um número inteiro como parâmetro de saída. Este problema seria semelhante ao Exemplo 9 do Apêndice A.

[1] http://bragg.phys.uwm.edu/xforms
[2] http://www.Hughes.com.au

5.5.1 Primeiro Filtro

O usuário inicialmente deve escolher os seis termos que melhor definem a função procurada. Como apresentado na Figura 5.4, os seis termos selecionados para a função de busca são:

<search,integer,file,concurrent,synchronous,interleaving>

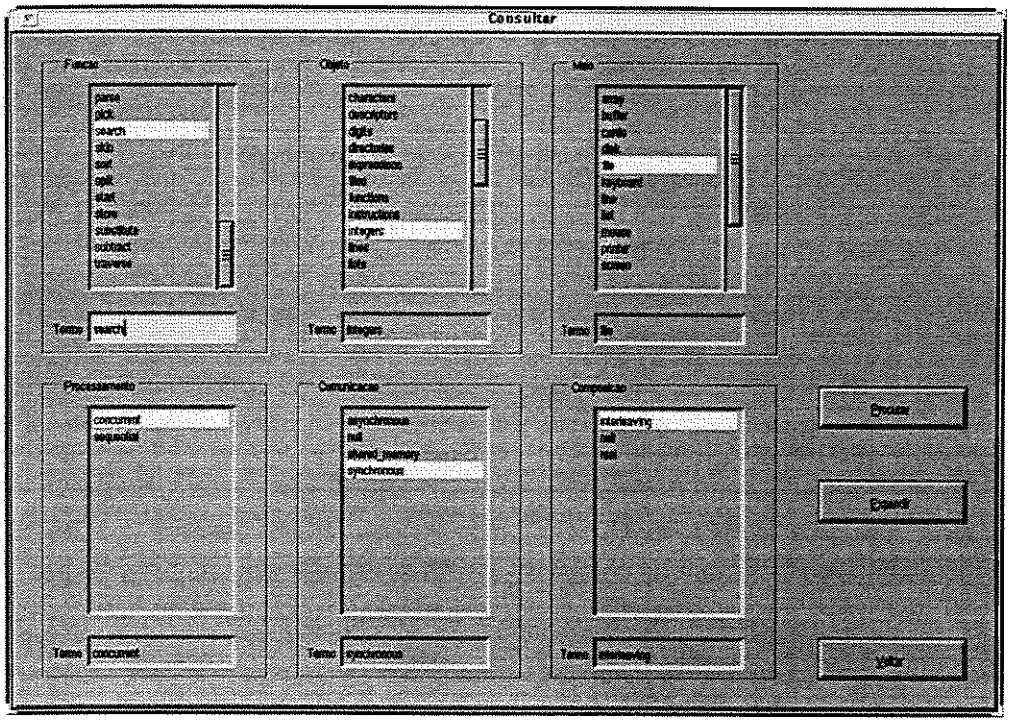

Figura 5.4: Seleção dos termos da consulta

Após ser realizada uma pesquisa na biblioteca de componentes, o sistema verifica que esta função não está classificada. Surge então a possibilidade de expandir a consulta, numa tentativa de encontrar componentes que executem funções semelhantes. O usuário deve decidir quais termos gostaria de expandir e até qual nível de similaridade esta expansão deve ocorrer. Como uma função de busca pode manipular diversos tipos de objetos em diferentes estruturas, são expandidos os termos 'integer' e 'file' até um nível relativamente próximo de similaridade (Figura 5.5).

Através da consulta expandida, são recuperados vários componentes, como mostrados na Figura 5.6. Dessa forma aumenta-se consideravelmente o *recall*, em detrimento da precisão (Seção 3.2.1). Para contornar este problema, podemos melhorar a precisão refinando a seleção através da utilização do segundo filtro do sistema: o casamento de interfaces.

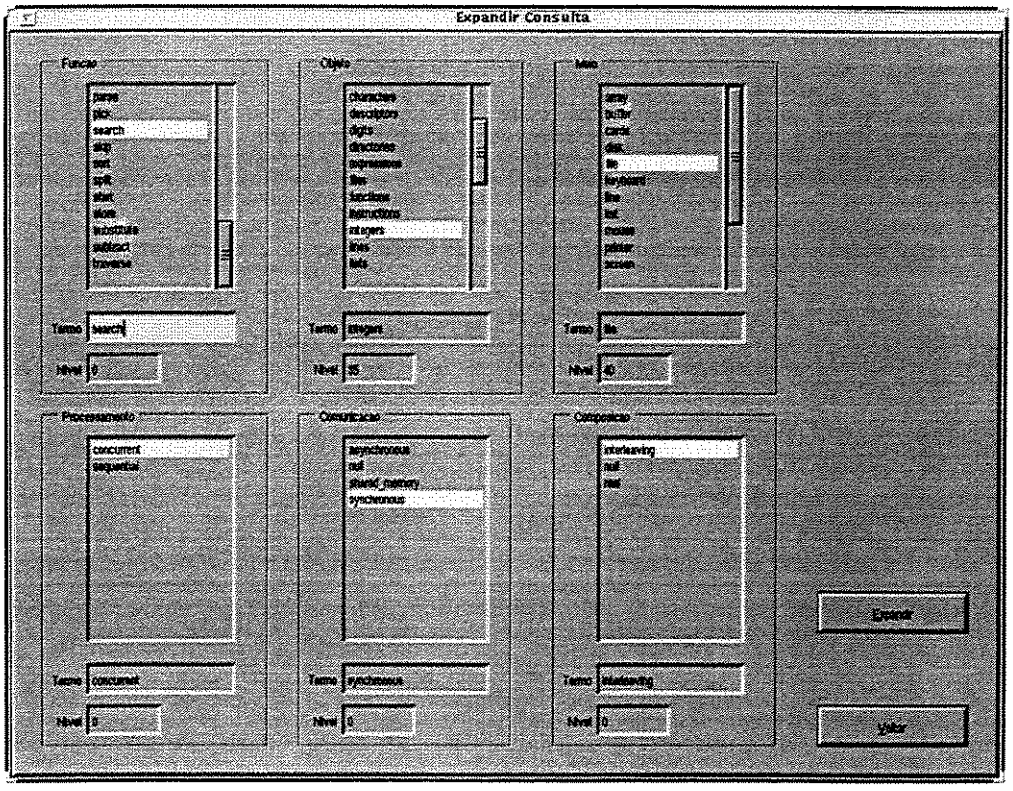

Figura 5.5: Expansão da consulta

5.5.2 Segundo Filtro

Ao entrar no segundo filtro do sistema, o usuário tem disponível automaticamente, como conjunto inicial de busca, os componentes recuperados no primeiro filtro. A partir daí, pode-se reduzir este conjunto, selecionando-se apenas aqueles componentes que possuem em suas definições, a quantidade e os tipos dos parâmetros procurados pelo usuário.

Neste exemplo em particular, a interface do componente procurado deveria apresentar como parâmetros de entrada, pelo menos um número inteiro e como saída outro número inteiro. O outro parâmetro de entrada (arquivo) é definido como um tipo genérico, uma vez que este não faz parte do conjunto de tipos básicos de LOTOS. Estas definições são apresentadas na Figura 5.7.

Com esta interface, o sistema reduz o conjunto de componentes, fornecendo ao usuário um pequeno número de funções a serem analisadas (Figura 5.8) . Caso deseje, o usuário pode tentar refinar a consulta, fornecendo maiores detalhes na descrição da interface. Caso contrário, pode encerrar a pesquisa e passar a analisar o código de cada componente recuperado.

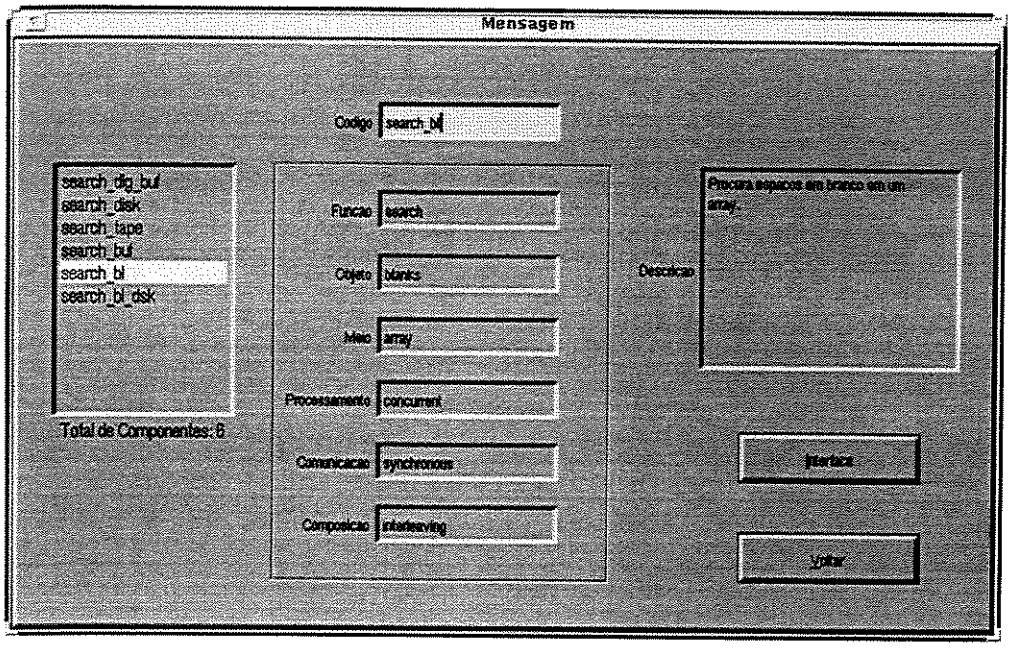

Figura 5.6: Componentes recuperados no primeiro filtro

5.6 Conclusão

Para que a reutilização de software torne-se um princípio realmente aplicável dentro de um ambiente de desenvolvimento de sistemas, surge a necessidade da produção de ferramentas ao mesmo tempo simples, práticas e eficientes. Entretanto, a maioria dos recursos atualmente disponíveis são direcionados a usuários experientes, com amplos conhecimentos em ambientes, sistemas, ou linguagens específicas [MMM97].

Com o intuito de suprir esta necessidade, apresentamos uma ferramenta de auxílio à classificação e seleção de componentes de software concorrentes. Nosso objetivo foi desenvolver uma ferramenta simples e facilmente utilizável por um usuário com um conhecimento básico sobre linguagens de programação e de especificação formal. Neste sentido, criamos um sistema centrado em dois filtros: 1. classificação por atributos e 2. casamento de interfaces.

Neste capítulo ficamos concentrados em apresentar os detalhes de implementação de cada um destes filtros. Descrevemos as estruturas de dados utilizadas, dando ênfase à rede semântica do primeiro filtro, responsável pela seleção de componentes semanticamente próximos ao desejado pelo usuário. Apresentamos algumas das características do sistema, tais como a linguagem de programação, o banco de dados e a interface gráfica utilizado no seu desenvolvimento. E, por fim, demonstramos sua utilização através de um exemplo prático, ilustrando todos os passos, juntamente com as próprias telas do sistema, de um processo de seleção.

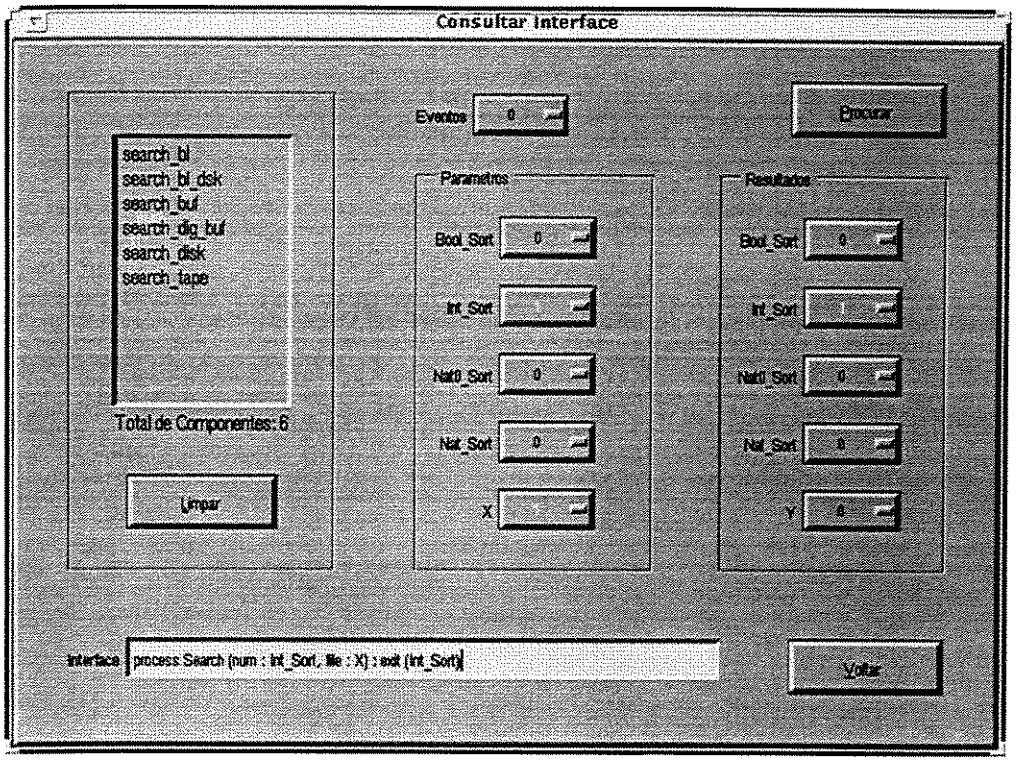

Figura 5.7: Descrição da interface

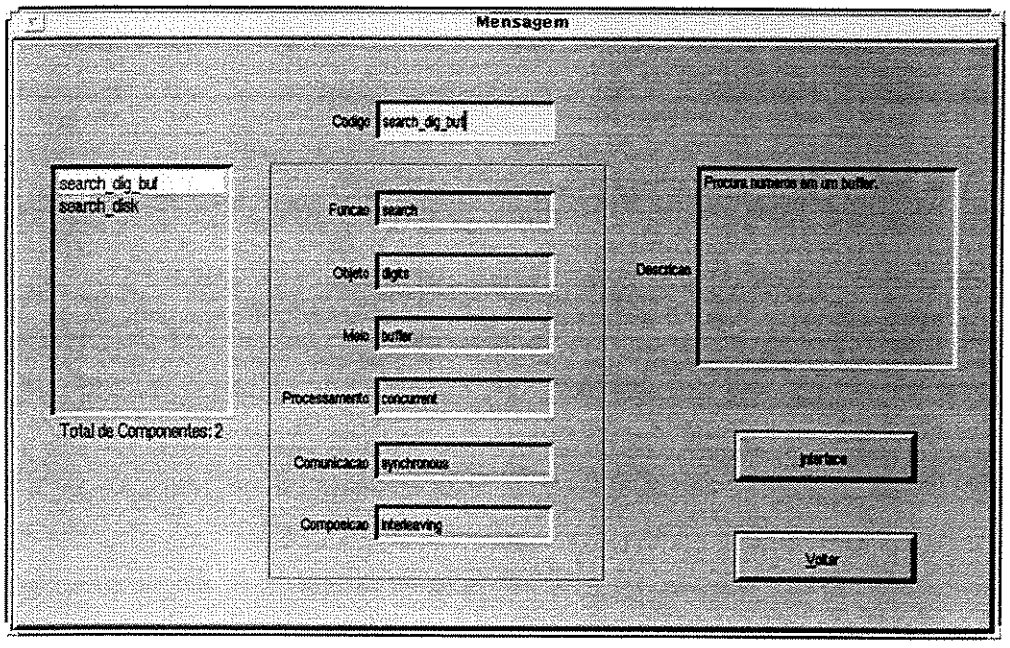

Figura 5.8: Conjunto final de componentes

Capítulo 6

Conclusões e Trabalhos Futuros

Neste capítulo descrevemos a experiência adquirida e as conclusões alcançadas no desenvolvimento do trabalho. Também apresentamos algumas vantagens e limitações do sistema criado, e direções para trabalhos futuros.

6.1 Considerações Gerais

A reutilização de software deveria ser parte integrante de qualquer ambiente de desenvolvimento de sistemas computacionais. Suas vantagens evidentes já foram descritas ao longo deste trabalho, principalmente no Capítulo 2. Entretanto, ainda faltam no mercado consumidor ferramentas realmente práticas, que tornem o processo de reutilização um hábito comum para qualquer analista/desenvolvedor.

Como alguns estudos apontam [BP89b, McC97], os gastos despendidos na fase inicial de um projeto de reutilização — montagem da biblioteca de software, implementação de ferramentas de apoio, treinamento de pessoal — são rapidamente amortizados ao longo de seu período de uso real. Neste ponto, os problemas deixam o nível técnico e recaem sobre questões culturais: como tornar um hábito natural, ao usuário, tentar reutilizar componentes prontos, ao invés de desenvolvê-los por completo? Uma vez que os usuários já estão cientes das facilidades proporcionadas pela reutilização — sistemas mais seguros, confiáveis e desenvolvidos em menor tempo e custo — a questão passa a ser o fornecimento de ferramentas auxiliares que tornem seu trabalho mais simples. Como já foi anteriormente comentado, deve ser mais fácil e rápido encontrar o componente que desempenha uma determinada função, em uma biblioteca, do que implementá-lo a partir do 'zero'. Caso contrário, qualquer pessoa fica desestimulada e, assim, todo o projeto de reutilização perde o sentido e logo cai, provavelmente, no esquecimento.

6.2 Considerações Específicas

A reutilização de software pode ser dividida em quatro etapas básicas: seleção, análise, modificação e integração dos componentes. Cada uma das etapas, por si só, é bastante complexa e exige uma profunda análise e estudo. Assim sendo, neste trabalho decidimos nos concentrar apenas na fase de seleção de componentes de software. Como esta é a fase inicial de todo o processo, torna-se extremamente crítica e importante para as demais de análise, modificação e integração. Por exemplo, caso a quantidade de componentes selecionados seja muito grande, ou os componentes sejam muito diferentes daquele desejado pelo usuário, a análise e modificação dos mesmos pode tornar-se uma tarefa extremamente cansativa ou até mesmo inútil.

Apresentamos então uma ferramenta específica à primeira fase do processo de reutilização: a classificação e seleção de componentes de software. Nosso objetivo foi criar um sistema baseado em filtros que, sucessivamente, vão reduzindo o número de componentes selecionados. O primeiro filtro — classificação por atributos (Capítulo 3) — é responsável por selecionar apenas os componentes que apresentam a funcionalidade e o processamento semelhante ao procurado. O segundo — casamento de interfaces (Capítulo 4) — por sua vez, seleciona apenas aqueles componentes que possuem os tipos corretos, em seus parâmetros de entrada e saída.

A classificação por atributos é bastante simples, sendo independente de qualquer linguagem de programação. Após ser criado o conjunto inicial de termos, juntamente com seus termos semânticos e sinônimos, o processo de seleção se torna rápido e direto. A classificação dos componentes, propriamente dita, também não apresenta grandes dificuldades, caso a pessoa responsável por tal tarefa tenha um bom conhecimento dos detalhes de cada componente. Neste caso, deve-se apenas escolher os termos que mais se adequam ao componente em questão formando, assim, uma descrição composta por seis termos básicos: função, objeto, meio, processamento, comunicação e composição. Da mesma forma, aumentar o número de termos cadastrados é apenas uma questão de definir seus termos sinônimos e semânticos. Para o sistema, desenvolvemos uma interface gráfica baseada principalmente em listas, onde são apresentados todos os termos cadastrados. O usuário apenas seleciona um termo da lista, ou se preferir, pode escrevê-lo diretamente. O sistema é responsável por localizar o termo, verificando se o mesmo é um sinônimo, ou um termo realmente cadastrado na base de dados. Caso a consulta inicial não retorne nenhum componente, o usuário tem a opção de expandi-la, recuperando componentes similares ao desejado. A rede semântica, responsável pela seleção destes componentes semelhantes, é montada de forma automática e dinâmica pelo sistema, apenas no momento da seleção. A grande vantagem deste esquema é que a expansão pode ser controlada através da variação dos pesos de cada termo. Dessa forma, consegue-se recuperar conjuntos diferentes de componentes, de acordo

com o nível de proximidade esperado.

Caso a quantidade de componentes recuperados no primeiro filtro ainda seja grande, o usuário tem a opção de refinar a consulta, aplicando sobre este conjunto, o filtro por casamento de interfaces. Neste caso, o nível de abstração é menor, exigindo do usuário um conhecimento básico sobre o sistema de tipos de uma linguagem de programação e especificação, em particular, da linguagem de especificação formal LOTOS. Apesar de trabalhar com componentes formalmente definidos, a seleção de tais componentes ainda se mantém relativamente simples, uma vez que o usuário só precisa definir quais os tipos e a quantidade de parâmetros de entrada e saída da função procurada. Em outras palavras, a ferramenta facilita a seleção de componentes formalmente definidos através da utilização de suas interfaces, ao invés da especificação LOTOS como um todo. Somente nas fases seguintes, de análise e modificação, é que esta especificação, propriamente dita, deverá ser utilizada 'integralmente', ou então, num possível terceiro filtro da ferramenta (Seção 6.4), baseado nas características semânticas de cada componente. Assim sendo, o sistema recebe uma interface, definida com a sintaxe e os tipos de LOTOS — Int-Sort, Bool-Sort, Nat-Sort, Nat0-Sort — e a partir daí, extrai suas informações essenciais: a quantidade e o tipo dos parâmetros. Além destes quatro tipos básicos, o usuário pode utilizar mais dois tipos genéricos, empregados quando não se tem certeza sobre os tipos procurados. Com tais informações, a ferramenta passa então a percorrer o subconjunto de componentes recuperados pelo primeiro filtro, selecionando apenas aqueles que possuem os tipos adequados em suas especificações.

6.3 Vantagens e Limitações do Trabalho

A ferramenta foi desenvolvida com uma interface gráfica — padrão X-Windows — simples, composta basicamente por listas de termos e componentes. Desta forma, a maioria das operações torna-se bastante intuitiva, bastando ao usuário escolher determinados termos ou descrever pequenas estruturas sintáticas.

A partir de uma grande biblioteca de componentes de software, o sistema seleciona um subconjunto formado por componentes que apresentam as seguintes características:

1. Executam a função especificada pelo usuário, ou senão, alguma função similar, de acordo com as distâncias entre seus atributos;

2. Possuem como parâmetros de entrada e saída, apenas os tipos determinados pelo usuário.

A partir deste subconjunto, tornam-se mais amenos os próximos passos de análise e modificação. A quantidade de componentes a ser analisada é sensivelmente menor e todos compartilham características comuns definidas pelo usuário. Outro ponto importante é a possibilidade da seleção de componentes relativamente semelhantes ao desejado. Caso o usuário não encontre na biblioteca exatamente a função que procure, o que é o mais provável de acontecer, a ferramenta fornece a possibilidade de expandir a consulta, definindo-se distâncias de similaridade (pesos) para cada termo, no primeiro filtro. De acordo com estas distâncias, são recuperados componentes semelhantes, que podem ser modificados e, assim, passam a desempenhar a função realmente esperada.

Outra característica da ferramenta é a mesclagem de técnicas formais e informais durante o processo de seleção. O primeiro filtro é utilizado de maneira independente de qualquer linguagem de programação ou formalismo, enquanto o segundo trabalha com interfaces de especificações formais definidas em LOTOS. Assim sendo, o usuário tem a liberdade de poder escolher o nível de abstração em que deseja realizar a seleção. Em outras palavras, os dois filtros trabalham de maneira independente, de forma que o usuário possa obter conjuntos distintos de componentes ao longo do processo de seleção.

A maioria das pesquisas na área [MMM97] é desenvolvida com ênfase na manipulação de componentes de software sequenciais. Sob este foco, acabam de certa forma ignorando as potencialidades do emprego de componentes concorrentes na reutilização, os quais são fundamentais em sistemas de tempo-real ou distribuídos, por exemplo. Neste trabalho, procuramos contornar esta lacuna, estendendo as características iniciais da classificação por atributos [Pri91], de modo a conseguir classificar os dois tipos de componentes. Neste sentido, criamos novos atributos, com o intuito de identificar o tipo de processamento do componente em questão. Além disso, implementamos um segundo filtro, o qual trabalha com especificações formais LOTOS, que é uma linguagem elaborada basicamente para se especificar componentes concorrentes.

Assim como todo trabalho de pesquisa tem uma série de limitações, este também não é uma exceção à regra. O principal problema do primeiro filtro é o grande esforço inicial necessário para se definir os termos que poderão compor as descrições dos componentes, juntamente com seus termos sinônimos e semânticos. Isto exige um trabalho árduo de pesquisa, no sentido de selecionar aqueles termos que poderão ser necessários nas futuras classificações. Além disso, a definição dos termos semânticos e suas respectivas distâncias (pesos) a um outro termo é uma tarefa subjetiva, que depende da experiência e interpretação da pessoa que estiver montando a rede semântica. Entretanto, conforme os componentes vão sendo selecionados, esta estrutura tende a se tornar estável, devendo ser pouco alterada. Ainda assim, caso surja a necessidade

de tal modificação, o usuário pode remover ou adicionar novos termos. Ao remover um determinado termo, o sistema automaticamente apresenta as opções de também eliminar todos os componentes já classificados com este termo, ou senão, apenas substituir este termo por outro. Por outro lado, ao adicionar um novo termo, o usuário tem apenas o trabalho de determinar seus termos sinônimos e semânticos.

Com relação ao segundo filtro, sua principal limitação é o fato de ser específico à seleção de interfaces definidas apenas com os quatro tipos básicos de LOTOS: Int-Sort, Bool-Sort, Nat-Sort e Nat0-Sort. Como esta ferramenta é um protótipo inicial, decidimos não trabalhar com os tipos abstratos de dados, mais complexos e que exigiriam algoritmos mais elaborados de unificação de tipos [Zar96]. Neste caso, apenas os tipos básicos já são suficientes para demonstrar a viabilidade do segundo filtro, baseado no casamento de interfaces LOTOS.

6.4 Trabalhos Futuros

Nesta seção, apresentamos alguns pontos do trabalho que necessitariam de mais pesquisas. De certa forma, são questões que recaem sobre as limitações especificadas anteriormente.

Rede Semântica A rede semântica, utilizada no primeiro filtro para fazer a seleção de componentes similares, é montada sobre 'pesos' fixos, escolhidos de maneira subjetiva pelo usuário no momento de inserir o termo na base. O grande problema é que esses 'pesos' ou distâncias entre os termos, são representados por valores exatos. O ideal seria que, ao invés de utilizar tais valores exatos, fossem empregados valores *fuzzy*, tais como 'muito próximo', 'próximo', 'distante', 'muito distante', etc. Dessa forma, a representação das distâncias se tornaria mais simples e natural, reduzindo o esforço necessário para se criar a lista de termos semânticos para cada termo.

Tipos Abstratos Como já foi anteriormente comentado, o segundo filtro baseia-se no 'casamento' de tipos básicos de LOTOS. Para aumentar sua abrangência, seria fundamental expandi-lo, de maneira a conseguir 'casar' também os tipos abstratos de dados. Dessa forma, componentes com especificações mais complexas, não criariam dificuldades para serem selecionados.

Terceiro Filtro Esta ferramenta é composta por dois filtros básicos: o primeiro seleciona os componentes que executam a função desejada, ou alguma função similar, enquanto o segundo filtro seleciona apenas os componentes que manipulam os tipos especificados pelo usuário. O

ideal seria criar um terceiro filtro, baseado na análise do comportamento de cada componente. Assim sendo, componentes que executam funções semelhantes, mas de maneiras variadas, seriam tratados de forma diferenciada. Por exemplo, existem vários algoritmos de ordenação de dados: *MergeSort*, *ShellSort*, *BubbleSort*, *QuickSort*, etc. Todos executam a mesma função (ordenação) e manipulam as mesmas informações. Entretanto, cada um executa um algoritmo próprio, comportando-se de maneira totalmente diferente um do outro. Apenas com os dois primeiros filtros da ferramenta, todos estes algoritmos seriam selecionados indistintamente; entretanto, caso fosse utilizado um terceiro filtro, seria selecionado somente aquele algoritmo que se comportasse da maneira desejada, reduzindo consideravelmente o conjunto final de componentes recuperados.

Ambiente Real Os testes executados na ferramenta foram muito limitados em escopo, empregando um pequeno conjunto de termos, e uma biblioteca de componentes reduzida. Para conduzir experimentos mais significativos, precisaríamos testá-la em um verdadeiro ambiente de desenvolvimento de software, utilizando uma grande biblioteca de componentes e usuários reais.

Apêndice A

Exemplos de classificação

Neste apêndice serão apresentados vários exemplos de classificação de componentes empregando os seis atributos criados (Capítulo 3): <Função, Objeto, Meio, Processamento, Comunicação, Composição>. Estes exemplos foram obtidos integralmente de trabalhos variados, de maneira a demonstrar a flexibilidade do método.

A.1 Exemplos obtidos de [Hoa78]

1. COPY

Problema: escreva um processo X para copiar caracteres do processo west para o processo east.

Solução:

$$X :: *[c : character; west?c \rightarrow east!c]$$

Classificação:

<copy,characters,file,concurrent,asynchronous,interleaving>

2. SQUASH

Problema: adapte o programa anterior para substituir todos os pares de asteriscos consecutivos "**" por uma seta"↑". Considere que o último caracter não é um asterisco.

Solução:

$$\begin{aligned} X \quad :: \quad & *[c : character; west?c \rightarrow \\ & [c \neq asterisk \rightarrow east!c \end{aligned}$$

$$
\begin{array}{l}
[]c = asterisk \rightarrow west?c; \\
\quad [c \neq asterisk \rightarrow east!asterisk; east!c \\
\quad []c = asterisk \rightarrow east!upwardarrow \\
]] \quad]
\end{array}
$$

Classificação:

<substitute,character,file,concurrent,synchronous,interleaving>

3. DISASSEMBLE

Problema: ler cartões de um arquivo e escrever em X a string de caracteres que eles contém. Um espaço adicional deve ser colocado ao final de cada cartão.

Solução:

$$
\begin{array}{l}
*[cardimage : (1..80)character; cardfile?cardimage \rightarrow \\
\quad i : integer; i := 1; \\
\quad *[i \leq 80 \rightarrow X!cardimage(i); i := i + 1] \\
\quad X!space \\
]
\end{array}
$$

Classificação:

<copy,string,cardfile,concurrent,synchronous,interleaving >

4. ASSEMBLE

Problema: ler uma string de caracteres de um processo X e imprimí-las em linhas de 125 caracteres. A última linha deve ser completada com espaços, se necessário.

Solução:

$$
\begin{array}{l}
lineimage : (1..125)character; \\
i : integer; i := 1; \\
*[c : character; X?c \rightarrow \\
\quad lineimage(i) := c; \\
\quad [i \leq 124 \rightarrow i := i + 1 \\
\quad []i = 125 \rightarrow lineprinter!lineimage; i := 1
\end{array}
$$

$$
\begin{array}{l}
] \quad]; \\
[i = 1 \rightarrow skip \\
[]i > 1 \rightarrow *[i \leq 125 \rightarrow lineimage(i) := space; i := i + 1]; \\
\quad lineprinter!lineimage \\
]
\end{array}
$$

Classificação:

<print,string,lineprinter,concurrent,asynchronous,interleaving>

5. REFORMAT

Problema: ler uma sequência de cartões de 80 caracteres cada e imprimir 125 caracteres por linha. Cada cartão deve ser seguido por um espaço adicional, e a última linha deve ser completada com espaços em branco, se necessário.

Solução:

$$[west :: DISASSEMBLE||X :: COPY||east :: ASSEMBLE]$$

Classificação:

<print,string,lineprinter,concurrent,asynchronous,interleaving>

6. REFORMAT II

Problema: adapte o programa anterior de modo a substituir todo par de asteriscos consecutivos por uma seta "↑".

Solução:

$$[west :: DISASSEMBLE||X :: SQUASH||east :: ASSEMBLE]$$

Classificação:

<print,string,lineprinter,concurrent,asynchronous,interleaving>

7. Função: divisão com resto

Problema: construa um processo para representar uma subrotina que recebe um dividendo e um divisor (positivos), e retorna seu quociente (inteiro) e o resto.

Solução:

$$
\begin{array}{l}
[DIV :: *[x, y : integer; X?(x, y) \rightarrow \\
\quad quot, rem : integer; quot := 0; rem := x; \\
\quad *[rem \geq y \rightarrow rem := rem - y; quot := quot + 1]; \\
\quad X!(quot, rem) \\
\quad] \\
||X :: USER \\
]
\end{array}
$$

Classificação:
<divide,integer,keyboard,concurrent,asynchronous,interleaving>

8. Recursão: fatorial

Problema: calcule o fatorial por um método recursivo, até um dado limite.

Solução:

$$
\begin{array}{l}
[fac(i : 1..limit) :: \\
*[n : integer; fac(i-1)?n \rightarrow \\
\quad [n = 0 \rightarrow fac(i-1)!1 \\
\quad []n > 0 \rightarrow fac(i+1)!n - 1; \\
\quad\quad r : integer; fac(i+1)?r; fac(i-1)!(n * r) \\
\quad]] \\
||fac(0) :: USER \\
]
\end{array}
$$

Classificação:
<factorial,integer,keyboard,asynchronous,interleaving>

9. Representação de dados: pequeno conjunto de inteiros

Problema: representar um conjunto de no máximo 100 números inteiros como um processo, S, o qual aceita dois tipos de instruções do processo X: (1) S!insert(n), insere o inteiro n no conjunto, e (2) S!has(n);...;S?b, onde b recebe Verdadeiro se n está no conjunto ou Falso, caso contrário. O valor inicial do conjunto é vazio.

Solução:

$$
\begin{aligned}
&S :: \\
&content(0..99)integer; size : integer; size := 0; \\
&*[n : integer; X?has(n) \rightarrow SEARCH; X!(i < size) \\
&[]n : integer; X?insert(n) \rightarrow SEARCH; \\
&\quad [i < size \rightarrow skip \\
&\quad []i = size; size < 100 \rightarrow \\
&\qquad content(size) := n; size := size + 1 \\
&] \quad]
\end{aligned}
$$

Classificação:
<insert,integer,set,concurrent,asynchronous,interleaving>

Onde SEARCH é uma abreviação de:

$$
\begin{aligned}
&i : integer; i := 0; \\
&*[i < size; content(i) \neq n \rightarrow i := i + 1]
\end{aligned}
$$

Classificação:
<search,integer,string,concurrent,synchronous,interleaving>

A.2 Exemplos obtidos de [Dub94]

1. Create

Interface:

```
int creat(const char *path, mode_t mode);
```

Parâmetros:
path: [in] o caminho do novo arquivo a ser criado.
mode: [in] a nova permissão do arquivo.

Descrição:
Cria um novo arquivo.

Classificação:

<create,file,file system,concurrent,synchronous,real>

2. Read

Interface:

```
int read(int fd, char *buf, size_t count);
```

Parâmetros:

fd: [in] o identificador do arquivo a ser lido.

buf: [out] o buffer que irá conter a informação lida.

count: [in] o tamanho máximo de *buf*.

Descrição:

Lê até *count* bytes de *fd* para dentro de *buf*.

Classificação:

<read,bytes,file,concurrent,asynchronous,real>

3. Write

Interface:

```
size_t write(int fd, const char *buf,size_t count);
```

Parâmetros:

fd: [in] o identificador do arquivo onde os dados serão escritos.

buf: [out] o buffer onde os dados a serem escritos estão armazenados.

count: [in] o número máximo de bytes a ser escrito.

Descrição:

Lê até no máximo *count* bytes de *buf* e escreve-os em *fd*.

Classificação:

<write,bytes,file,concurrent,synchronous,real>

4. Mkdir

Interface:

```
int mkdir(const char *path, mode_t mode);
```

Parâmetros:
path: [in] o caminho do diretório a ser removido.
mode: [in] o direito de acesso ao novo diretório.

Descrição:
Cria um novo diretório.

Classificação:
<create,directory,file system,sequential,null,null>

5. Rmdir

Interface:

```
int rmdir(const char *path);
```

Parâmetros:
path: [in] o caminho do diretório a ser removido.

Descrição:
Remove um diretório. O diretório a ser removido deve estar vazio.

Classificação:
<remove,directory,file system,sequential,null,null>

6. Fork

Interface:

```
pid_t fork(void);
```

Descrição:
Cria um processo-filho a partir do processo atualmente em execução. O novo processo é uma cópia exata do processo-pai, com exceção do seu identificador.

Classificação:

<create,process,operational system,concurrent,asynchronous,real>

7. Stime

Interface:

```
int stime(time_t *t);
```

Parâmetros:

t: [in] o novo horário.

Descrição:

Estabelece o novo horário do sistema.

Classificação:

<set,time,operational system,sequential,null,null>

A.3 Exemplo obtido de [Ben90]

1. Merge

```
procedure Merge_Sort is
   A:array(1..N) of Integer;
   procedure Sort(Low,High:Integer);
   procedure Merge;
begin
   Sort(1,N/2);
   Sort(N/2 + 1,N);
   Merge;
end;
```

Classificação:

<merge,integer,array,concurrent,asynchronous,interleaving>

<sort,integer,array,concurrent,asynchronous,interleaving>

Bibliografia

[Bac92] Jean Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database and Distributed Systems.* Addison Wesley, 1992.

[Ben90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall International, 1990.

[BH94] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. *Lecture Notes in Computer Science*, 873:105–117, 1994.

[BP89a] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Volume I. Concepts and Models.* Addison-Wesley, 1989.

[BP89b] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Volume II. Applications and Experience.* Addison-Wesley, 1989.

[BR89] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability - Vol. 1. Concepts and Models*, pages 1–17. Addison-Wesley, 1989.

[BR98] Luiz E. Buzato and Cecília M. F. Rubira. Construção de sistemas orientados a objetos confiáveis. In *11a. Escola de Computação*, Rio de Janeiro, Jul. 1998. Universidade Federal do Rio de Janeiro.

[CB91] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, Feb. 1991.

[CDG94] Joachim Cramer, Ernst-Erich Doberkat, and Michael Goedicke. Formal methods. In Wilhelm Schäfer, Rubén Prieto-Díaz, and Masao Matsumoto, editors, *Software Reusability*, pages 79–111. Ellis Horwood, 1994.

[Dub94] Louis-Dominique Dubeau. *Linux 1.0 Syscalls*, 1994.

[EVD89] P. H. J. Van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS.* Elsevier Science Publishers, 1989.

[FG90] W. B. Frakes and P. B. Gandel. Representing reusable software. *Information and Software Technology*, 32(10):653–664, Dec. 1990.

[FKS94] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: a VDM-based software component retrieval tool. Technical Report 94-08, Technical University of Braunschweig, Germany, Nov. 1994.

[FLGD87] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, Nov. 1987.

[FP94] William B. Frakes and Thomas P. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, Aug. 1994.

[Gog86] Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, Feb. 1986.

[HM89] Ellis Horowitz and John B. Munson. An expansive view of reusable software. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability - Vol. 1. Concepts and Models*, pages 19–41. Addison-Wesley, 1989.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[ISO87] ISO - International Standardization Organization. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807 edition, Jul. 1987.

[JC93] Jun-Jang Jeng and Betty H. C. Cheng. Using formal methods to construct a software component library. *Lecture Notes in Computer Science*, 717:397–417, Sep. 1993.

[JC95] Jun-Jang Jeng and Betty H. C. Cheng. Specification matching for software reuse: A foundation. In *Proc. of ACM Symposium on Software Reuse*, pages 97–105. ACM Press, Apr. 1995.

[KRT89] Shmuel Katz, Charles A. Richter, and Khe-Sing The. Paris: A system for reusing partially interpreted schemas. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability - Vol. 1. Concepts and Models*, pages 257–273. Addison-Wesley, 1989.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, Jun. 1992.

[LFH92] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[MBK91] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, Aug. 1991.

[McC97] Carma McClure. *Software Reuse Techniques.* Prentice Hall, 1997.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[MMM95] Hadefh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, Jun. 1995.

[MMM97] A. Mili, R. Mili, and R. T. Mittermeir. A survey of software storage and retrieval. http://www.isr.wvu.edu/services/publications/publications.html, Oct. 1997.

[Nei89] James M. Neighbors. Draco: A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability - Vol. 1. Concepts and Models*, pages 295–319. Addison-Wesley, 1989.

[OHPB92] Eduardo Ostertag, James Hendler, Rubén Prieto-Díaz, and Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, Jul. 1992.

[PA95] John Penix and Perry Alexander. Design representation for automating software component reuse. In *Proc. of the First International Workshop on Knowledge-based Systems for the (re)use of Program Libraries*, France, Jun. 1995.

[PA97] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, Jun. 1997.

[PF87] Rubén Prieto-Díaz and Peter Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, Jan. 1987.

[PP92] Andy Podgurski and Lynn Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In ACM Press, editor, *14th International Conference on Software Engineering*, pages 300–304, New York, 1992.

[Pri85] Rubén Prieto-Díaz. A software classification scheme. Technical Report 85-19, Information and Computer Science - University of California, Irvine, 1985.

[Pri91] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[RS94] Hans Dieter Rombach and Wilhelm Schäfer. Tools and environments. In Wilhelm Schäfer, Rubén Prieto-Díaz, and Masao Matsumoto, editors, *Software Reusability*, pages 113–152. Ellis Horwood, 1994.

[RW91] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In Koichi Furukawa, editor, *Proc. of the Eighth International Conference on Logic Programming*, pages 173–187, Paris-France, Jun. 1991. MIT Press.

[Sal86] Gerard Salton. Another look at automatic text-retrieval systems. *Communications of the ACM*, 29(7):648–656, Jul. 1986.

[SPM94] Wilhelm Schafer, Rubén Prieto-Díaz, and Masao Matsumoto, editors. *Software Reusability*. Ellis Horwood, 1994.

[Tur96] Kenneth J. Turner. The formal specification language LOTOS: A course for users. Technical report, Department of Computing Science - University of Stirling, Stirling - Scotland, Apr. 1996.

[Wir89] Niklaus Wirth. *Algoritmos e Estruturas de Dados.* Prentice Hall do Brasil, 1989.

[Zar96] Amy Moormann Zaremski. *Signature and Specification Matching.* PhD thesis, Carnegie Mellon University - School of Computer Science, Pittsburgh, PA 15213, Jan. 1996.

[ZW95a] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):147–170, Apr. 1995.

[ZW95b] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. In Gail E. Kaiser, editor, *Proc. of the 3rd ACM Sigsoft Symposium on the Foundations of Software Engineering*, pages 6–17, Washington DC-USA, Oct. 1995. ACM Press.