UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELETRICA
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇAO
E AUTOMAÇAO INDUSTRIAL

COMPOSIÇAO DE COMPONENTES DE SOFTWARE:

ESPECIFICAÇAO FORMAL DE REQUISITOS E

AUTOMATIZAÇAO DO PROCESSO DE VALIDAÇAO

por: Fani Barbosa Camargo

Orientador : Prof. Dr. Mário Jino

Co-Orientador: Dr. Fuad Gattaz Sobrinho

Dissertaçao de Mestrado apresentada
à Faculdade de Engenharia Elétrica
da Universidade Estadual de Campinas

Dezembro de 1991

Este exemplar corresponde à redação final da tese defendida por Fani Barbosa Camargo e aprovada pela Comissão Julgadora em 20 12 91.

Mario Jino Orientador

A Cilas, Vania e Erika

AGRADECIMENTOS

A EMPRESA BRASILEIRA DE PESQUISA AGROPECUARIA, EMBRAPA, pelo apoio e incentivo, fundamentais à realizaçao deste trabalho.

Ao Dr. Raymond T. Yeh pela preciosa paciência, incentivo e disposiçao a contribuir com este trabalho.

Ao Prof. Dr. Herbert Weber pela vital colaboraçao técnica, sempre marcada pela boa vontade e fundamental estímulo.

Ao Prof. Dr. Mário Jino pela constante dedicaçao e paciência, fundamentais à elaboraçao deste documento e conclusao deste trabalho.

Ao Dr. Fuad Gattaz Sobrinho que, desde o "início dos tempos", tem inspirado meus primeiros passos na Ciência da Computaçao, com incansáveis doses de paciência, amizade, carinho e dedicaçao (e rigor!).

Aos colegas do NTIA pelo constante apoio, estímulo e amizade.

A equipe de trabalho cuja participaçao enriqueceu o conteúdo deste projeto. Em especial à amiga Carla Geovana do Nascimento Macário, peça fundamental na "existência" do GAT.

A Antônio Luiz Peloso, cujo brilhante trabalho tem sido uma grande estrela guia, sem falar do apoio, incentivo...

A minha família pelo incentivo e afeto decisivos na realizaçao deste trabalho.

A todos aqueles que, direta ou indiretamente, colaboraram para a execuçao deste trabalho.

RESUMO

A atual tecnologia de produçao de software encontra-se em estágio artesanal e caracterizada por baixa qualidade e produtividade. Formas promissoras de ir de encontro a uma produçao industrial de sistemas de software encontram-se em paradigmas de reuso e formalismo.

O objetivo deste trabalho é propor uma alternativa para assegurar a qualidade de sistemas de software, em seus diversos níveis de abstraçao, através de um processo de validaçao com uso de formalismo. Para tal, procurou-se aliar os conceitos de composiçao de Componentes de Software Reusáveis da Linguagem-Pi à técnica de especificaçao formal de Traços, tornando possível a automatizaçao do processo de validaçao. Uma ferramenta de software foi produzida dentro deste contexto - o Gerador Automático de Teste, GAT, que efetua a validaçao do código de um componente de software contra seus requisitos, expressos por uma especificaçao formal.

ABSTRACT

The current software production technology is found to be in a craftsman stage, with low quality and productivity standards. Promising avenues towards an industrial production of software systems lie in paradigms of reuse and formalism.

The objective of this work is to propose an alternative for the assurance of high quality levels of software systems, within its different levels of abstraction, through a validation process, with the use of formal techniques. In this context, the concepts of Reusable Software Components from the Pi-Language model are alied to the formal Trace specification technique, making possible an automatic validation process. A software tool is here produced - the Automatic Test Generator, ATG, which performs the validation of a software component's code against its requirements, expressed by a formal specification.

INDICE

1. Introduçao 01

2. O Problema de Software 03

2.1. Crescimento da Demanda de Software 03

2.2. Carência de Recursos Humanos 06

2.3. Mecanismos para Capturar as Necessidades Tecnológicas 06

2.4. Carência de Tecnologia de Software 07

3. Uma Estratégia de Soluçao 11

3.1. Orientaçao por Domínio e Produçao Industrial de Software 11

3.2. Fundamentos para a Produçao Industrial de Software 12

3.2.1. Reuso e Formalismo: Produtividade e Qualidade 13

3.2.2. Unidade de Programa: A Abstraçao de Dados 15

3.2.3. Componentes de Software Reusáveis: A Unidade de Produçao Industrial de Software ...19

4. Composiçao de Componentes: Uma Estratégia para Produçao Industrial de Software 24

4.1. Produçao de Sistemas de Software: do Abstrato ao Concreto 24

4.2. Interconexao de Componentes: A Linguagem-Pi ..27

4.2.1. Separaçao de "Concerns" - "Views" 28

4.3. A Linguagem de Especificaçao de Traços 32

4.4. Um Modelo de Reuso de Componentes 36

4.4.1. O Mundo Abstrato: "View" de Tipo 36

4.4.2. O Mundo Concreto: "View" de Objeto 37

4.5. Concreto x Abstrato: Uma Proposta para Validaçao de Componentes de Software 38

4.5.1. Princípios de Validaçao 38

4.5.2. Representaçao do Abstrato: A Interface de Exportaçao da "View" de Tipo 39

4.5.3. Representaçao do Concreto: O Corpo da "View" de Objeto 41

5. Uma Ferramenta para a Validaçao da Composiçao de Componentes 44

5.1. Como o Teste é Efetuado pelo GAT 45

5.2. Arquitetura de Implementaçao 46

5.3. Formato das Entradas do GAT 48

5.3.1. Formato da Especificaçao 48

5.3.2. Formato da Amostra de Traços 50

5.3.3. Formato do Código 51

5.3.4. Formato do Validador 52

5.4. Passos de Utilizaçao do GAT 54

6. Conclusões 57

7. Referências Bibliográficas 59

ANEXO A. Arquivos Gerados pelo GAT para o Componente Fila

ANEXO B. BNF do Analisador Sintático da Especificaçao

ANEXO C. Um Exemplo do Modelo de Composiçao de Componentes - Sistema de Controle de Aeroporto

ANEXO D. Um Experimento de Validaçao de Software

LISTA DE FIGURAS

2.1. O Cenário de Software 08

2.2. Intervalo de Expectativa 07

3.1. Intervalo de Expectativa - RIPP 12

3.2. CSR's Incorporáveis 21

4.1. Mapeamento de Tipos 26

5.1. GAT - Estrutura Funcional 44

5.2. GAT - Arquitetura de Implementaçao 47

LISTA DE TABELAS

2.1. Contratos de Software do Governo Americano 04

2.2. Hardware x Software 09

2.3. Utilizaçao de Métodos ou Ferramentas 10

CAPITULO 1 - Introduçao

Avanços recentes da tecnologia de microeletrônica possibilitaram a produçao em massa de dispositivos computacionais miniaturizados, tendo causado uma revoluçao sem precedentes na história, na tecnologia da informaçao. No coraçao de tais dispositivos há uma entidade denominada software, a qual é responsável pela alma da funcionalidade produzida pelos dispositivos computacionais.

A tecnologia de produçao do hardware de tais dispositivos é efetivamente dominada pelo homem, através de técnicas clássicas de engenharia e gerência, com alta produtividade e qualidade [TANI,91]. Por outro lado, a tecnologia utilizada na alma de tais sistemas computacionais, i.e., software, tem-se mostrado inadequada, estando em estágio artesanal e apresentando baixa qualidade e produtividade. A produçao de software atualmente consome 80% do custo total de sistemas computacionais, enquanto a produçao de hardware consome apenas 20% [CHEN,90], o que ilustra o quao crítica a produçao de software vem se tornando.

Nas próximas décadas o domínio da tecnologia de produçao de software pelas naçoes possuirá papel estratégico na sua adaptaçao à revoluçao da tecnologia da informaçao, de tal forma que aquelas naçoes que nao derem atençao a este elemento estratégico de seu desenvolvimento, ficarao para trás competitivamente em todas as áreas que envolvam atividades tecnológicas [TANI,91].

Tendo em vista este cenário mundial, este trabalho concentra-se em problemas de software, buscando alternativas em paradigmas de produçao industrial de software. Uma forma promissora de aumentar efetivamente a qualidade e produtividade de software é o reuso. Sistemas de software necessitam ser construídos a partir de elementos reusáveis, assim como os "blocos de encaixe" da tecnologia de produçao de sistemas de hardware.

Um paradigma para a construçao de produtos de software através da composiçao de Componentes Reusáveis de Software - CSR's - é apresentado neste trabalho, utilizando-se o modelo da Linguagem-Pi [WEBE,88] desenvolvido no projeto tecnológico europeu ESPRIT.

O objetivo deste trabalho é propor uma alternativa para assegurar a qualidade de sistemas de software, em seus diversos niveis de abstraçao, através de um processo de validaçao com uso de formalismo. Para tal, procurou-se aliar os conceitos da Linguagem-Pi à técnica de especificaçao formal de Traços [PARN,86], tornando possivel a automatizaçao do processo de validaçao.

Uma ferramenta de software foi produzida dentro deste contexto - o Gerador Automático de Teste, GAT. Este efetua a validaçao do código de um CSR contra seus requisitos, expressos por uma especificaçao formal. Para tal é necessária uma amostra de dados de entrada, dita amostra de traços canônica.

O paradigma de composiçao de componentes reusáveis apresentado, aliado ao modelo de validaçao aqui proposto, constitui uma alternativa para minorar problemas criticos associados à qualidade e produtividade de sistemas de software.

Esta dissertaçao é composta por seis capitulos. O primeiro deles constitui esta Introduçao, que descreve sucintamente os temas abordados nos demais capitulos. No Capitulo 2 sao identificados os principais problemas relacionados a software existentes. A partir desta informaçao, uma estratégia de soluçao é indicada no Capitulo 3, consistindo de orientaçao por domínios de aplicaçao e produçao industrial de software.

O processo de Composiçao de Componentes de Software é apresentado no Capitulo 4 como alternativa para a produçao industrial de software. Uma estratégia de validaçao para tais componentes de software é aqui proposta.

O quinto capitulo contém a descriçao do Gerador Automático de Teste - GAT, que consiste de uma ferramenta de software para a validaçao de componentes anteriormente proposta.

As conclusoes do trabalho encontram-se no Capitulo 6, juntamente com propostas de pesquisas futuras.

Nos Anexos estao apresentados os arquivos gerados pela ferramenta GAT, em um exemplo para o componente fila, e a BNF de seu analisador sintático da especificaçao; é ainda descrito um exemplo de composiçao de componentes e relatado um experimento de validaçao de software utilizando-se de formalismo.

CAPITULO 2 - O Problema de Software

"Crise de software" é a expressao comumente utilizada para expressar o estado da arte de produçao de software nos países desenvolvidos. A má qualidade e a baixa produtividade de software sao os principais problemas atualmente enfrentados em tal crise. A Tabela 2.1 mostra o conjunto de produtos de software entregues ao governo americano, em contratos de um valor total de US$ 6.8 milhoes [GATT,84]. Tais números sao de 1981, mas provavelmente nao mudaram significativamente devido ao fato de que os métodos, técnicas e ferramentas utilizados ainda permanecem praticamente os mesmos.

De acordo com Yeh [YEH,91], um problema de produçao de software é que os custos de manutençao estao crescendo, já sendo responsáveis por 80% do orçamento de aplicaçoes de software. Da mesma forma existe uma grande demanda de novos sistemas computacionais, em uma realidade de recursos escassos para desenvolvimento de produtos novos.

Neste capítulo discutem-se as principais causas de tais problemas de software existentes atualmente, que podem ser qualificados através de:

- o crescimento da demanda de software;

- carência de recursos humanos;

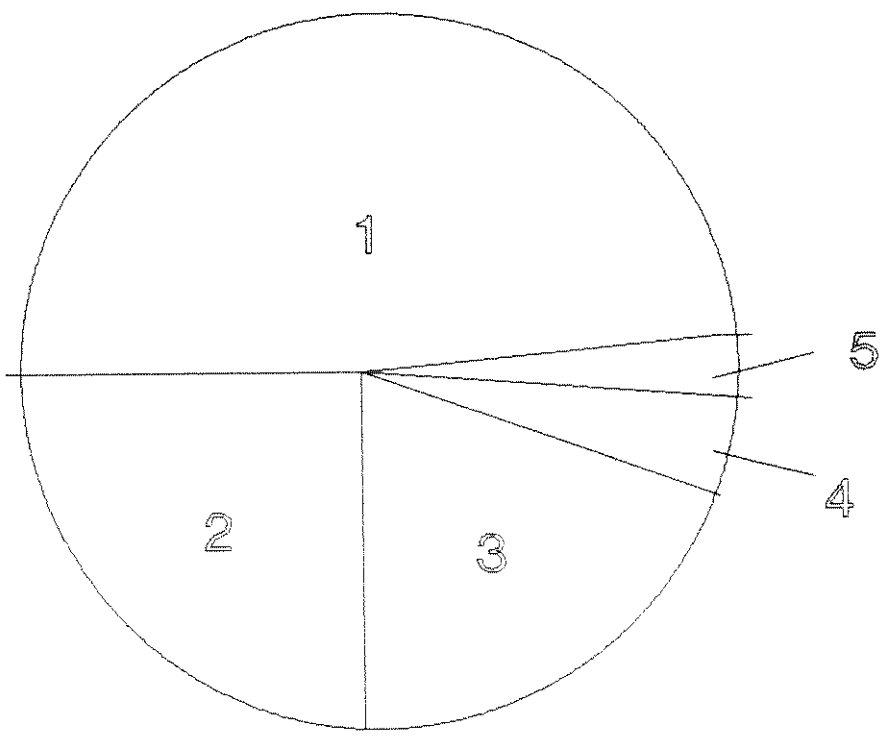
- mecanismos inadequados para capturar as necessidades tecnológicas;

- carência de tecnologia de software.

2.1. Crescimento da Demanda de Software

Na medida em que o hardware de computadores vai se tornando mais popular, a automaçao das atividades humanas vem crescendo rapidamente. Desta forma, o nível das necessidades humanas tem sido elevado, bem como os requerimentos e expectativas em relaçao a sistemas computacionais têm avançado enormemente [FUJI,89].

TABELA 2.1. Contratos de Software do Governo Americano



1. SOFTWARE ENTREGUE MAS NUNCA
UTILIZADO COM SUCESSO (US$ 3.2 MILHOES)

2. SOFTWARE PAGO MAS NUNCA
ENTREGUE (US$ 1.95 MILHOES)

3. SOFTWARE UTILIZADO COM GRANDES MUDANCAS
OU ABANDONADO MAIS TARDE (US$ 1.3 MILHOES)

4. SOFTWARE UTILIZADO COM MUDANCAS (US$ 198.000)

5. SOFTWARE UTILIZADO COMO ENTREGUE (US$ 119.000)

E bem sabido que há um enorme arsenal de sistemas de software que necessitam de constante manutençao e reprogramaçao. A indústria de software americana possui mais de US$ 300 bilhoes de software mal-estruturado e de difícil manutençao [YEH,90]. Tais sistemas representam o "coraçao" das funçoes de informaçao das organizaçoes, sendo responsáveis pelas operaçoes diárias das mesmas; as organizaçoes nao estao dispostas a redesenvolver os sistemas de software existentes, devido aos altos custos envolvidos e pela falta de recursos humanos. Evoluir é a soluçao mais plausível para tornar as aplicaçoes existentes flexíveis, portáveis e amigáveis [RUOT,90]. Desta forma há uma enorme demanda de **evoluçao** de produtos de software existentes.

Um aspecto importante é que os sistemas de software existentes nao sao, na maioria das vezes, produzidos para serem portáveis [ZELK,84]. Normalmente o esforço de portabilidade é muito alto, com ferramental inadequado. Em consequência, surgem produtos de software dependentes de máquina, sistemas operacionais, etc., sendo que uma mudança na tecnologia de hardware geralmente acarretará uma grande demanda de software, mesmo nao havendo mudanças funcionais.

As estruturas organizacionais têm mudado da tradicional hierarquia funcional para uma estrutura orientada para redes ("network-oriented organizations"). Os sistemas de informaçao devem suportar tais transformaçoes. Do lado da tecnologia de hardware, um grande potencial para o processamento cooperativo é oferecido por computadores poderosos, memória barata e uma tecnologia madura de redes de computadores. Software, por outro lado, é um fator limitante. Um grande desafio da atualidade é o da realizaçao do "**downsizing**" do código existente para código distribuído. Também para suportar tais mudanças organizacionais, sistemas de software necessitam ser feitos de forma mais barata, melhor e mais rápida [YEH,91].

2.2. Carência de Recursos Humanos

Recursos humanos possuem um crescimento vagaroso. O treinamento e qualificaçao de pessoas demanda um longo período, além de nao ser um processo fácil [KAMI,90] [GATT,90a]. Segundo Fujino [FUJI,89] a atual tecnologia de software torna a produçao deste uma atividade altamente dependente do esforço humano. A alta demanda de software tem causado uma grande alta na demanda de engenheiros de software. A demanda de pessoal na área de software tem crescido a uma taxa de 20% ao ano, enquanto os recursos humanos têm crescido apenas 4%. Desta forma, há uma grande demanda de recursos humanos qualificados. Princípios de engenharia de software, aliados à automaçao da produçao de software, seriam uma alternativa para tal questao [GATT,84].

2.3. Mecanismos para Capturar as Necessidades Tecnológicas

Os mecanismos disponíveis sao inadequados para capturar as necessidades tecnológicas da sociedade [GATT,90b], o que contribui com a crise de qualidade existente na indústria de software.

Para lidar com tal crise a indústria tem desenvolvido os chamados produtos de software de "propósito geral", os quais:

- possuem tecnologia de produçao inadequada (resultando em software de baixa manutenibilidade, por exemplo);

- nao sao orientados para o domínio, ou seja, nao capturam as reais necessidades do usuário.

A "guerra de mercado" ("marketing war") [GATT,90b] é entao utilizada para criar uma demanda no mercado para os ditos produtos de "propósito geral". Para ir de encontro às reais necessidades do usuário é necessário que uma enorme quantidade de novo software seja produzida, utilizando os de "propósito geral". Tais novos sistemas de software :

- sao mais uma vez produzidos com tecnologia inadequada;

- requerem um esforço de produçao muitas vezes maior do que aquele requerido inicialmente para o desenvolvimento do software de "propósito geral";

- geram uma grande demanda de recursos humanos para tamanho esforço de produçao.

A Figura 2.1 ilustra o cenário anteriormente descrito. Vale ressaltar que, apesar do enorme esforço **E** colocado nos novos sistemas de software, existe ainda um intervalo entre o software e o sistema (sistema tal como definido por Bertalanfy [BERT,77]) do usuário o que é também representado na Figura 2.2 [YEH,91].

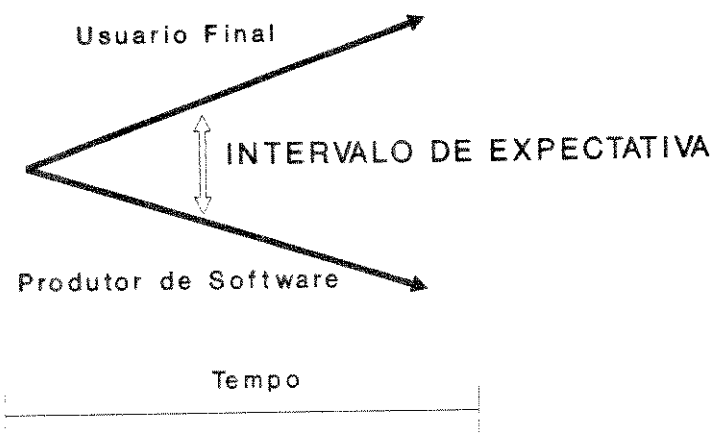


FIG. 2.2. Intervalo de Expectativa

2.4. Carência de Tecnologia de Software

A tecnologia de hardware encontra-se em um estágio industrial, com uma alta produtividade e produtos de qualidade. O processo de produçao de hardware é estruturado no desenvolvimento de "blocos de encaixe", i.e., componentes de hardware, que sao compostos até resultar no produto final [MANO,82]. Por outro lado, a tecnologia de software encontra-se em um estágio artesanal [MITT,90] [GATT,90a]. De forma diferente do hardware, os produtos de software nao sao constituidos por componentes de qualidade; nao podem ser prontamente produzidos nem tampouco reusados. Tal baixa produtividade aumenta significativamente seus custos e agrava ainda mais a alta demanda existente, como mostra a Tabela 2.2. Sistemas de software estao se tornando dominantes e criticos na construçao de sistemas computacionais, ultrapassando

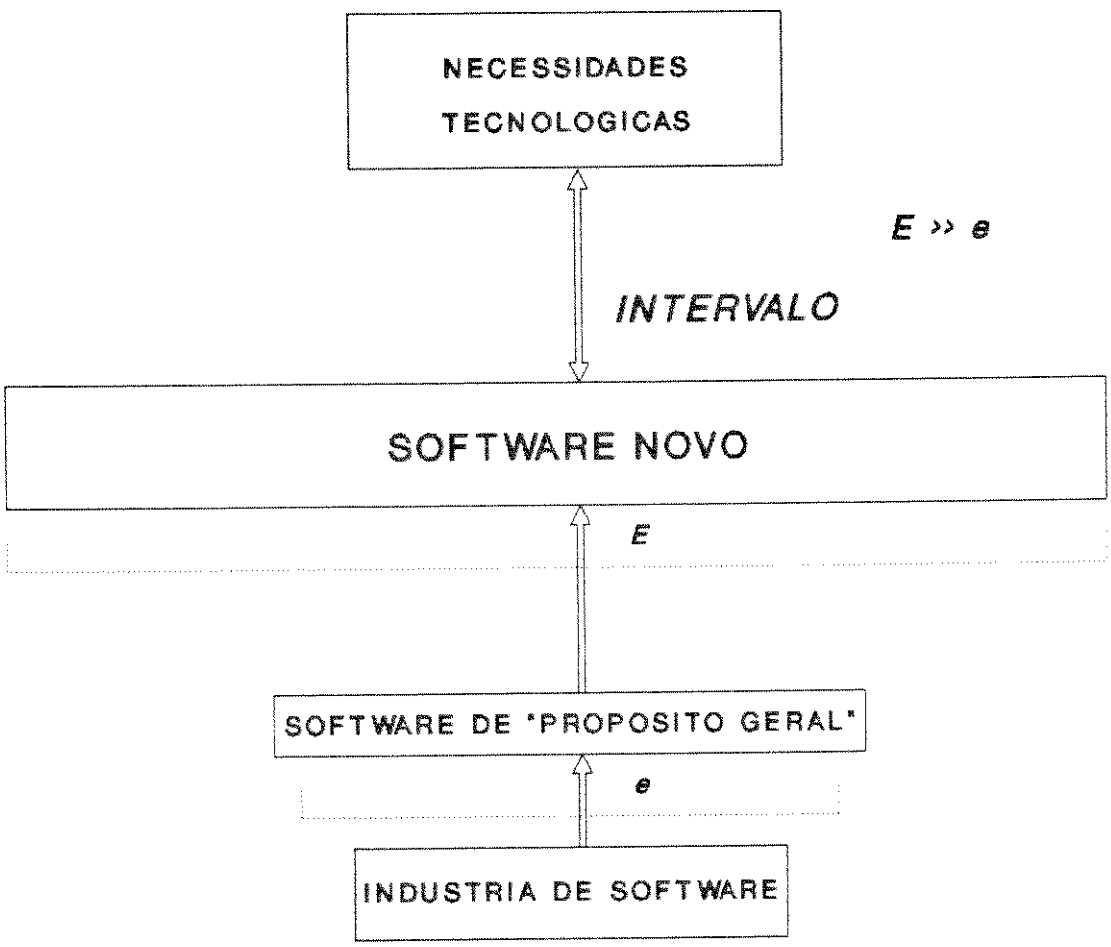


FIG. 2.1. O Cenario de Software

TABELA 2.2. Hardware x Software

	HARDWARE	SOFTWARE
PRODUTIVIDADE	*ALTA*	*BAIXA*
QUALIDADE	*ALTA*	*BAIXA*
CUSTO	*BAIXO*	*ALTO*
TECNOLOGIA	*COMPONENTES de HARDWARE*	*?*

intensamente o hardware no que diz respeito a custo e complexidade [GEHA,86] . Para ilustrar tal fato, o desenvolvimento do software na indústria consome 80% do custo total de um novo sistema computacional, enquanto o desenvolvimento de hardware consome 20% [CHEN,90]. Tal cenário demonstra a **tecnologia de produçao inadequada** atualmente utilizada para os sistemas de software.

Uma pesquisa feita pela Universidade de Maryland, EUA, com o objetivo de investigar os métodos e as ferramentas utilizadas na indústria nos Estados Unidos e no Japao, determinou que há pouco uso de **práticas de Engenharia de Software** nas organizaçoes [ZELK,84]. Existe um intervalo significativo entre as práticas recomendadas na literatura de Engenharia de Software e aquelas realmente seguidas na indústria. O suporte através de ferramentas durante todo o ciclo de produçao de software é exemplo de uma recomendaçao nao amplamente seguida na indústria. A intensidade do uso de ferramentas varia inversamente com a distância de que estas se encontram da fase de codificaçao. A Tabela 2.3 fornece uma idéia das ferramentas e técnicas utilizadas. Um número maior de princípios de Engenharia de Software precisam ser efetivamente utilizados na indústria para que se possa lidar com o problema de software.

TABELA 2.3. Utilizacao de Metodos ou Ferramentas

METODO OU FERRAMENTA	% DE COMPANHIAS
Linguagens de alto nivel	100
Acesso "on-line"	93
Revisoes	73
Linguagens de Desenho de Programas	63
Alguma metodologia formal	41
Algumas ferramentas de teste	27
Auditores de codigo	18
Equipe de programador chefe	7
Qualquer verificacao formal	0
Requisitos ou especificacoes formais	0

CAPITULO 3. Uma Estratégia de Soluçao

Uma estratégia de soluçao de problemas de software é apresentada neste capitulo, utilizando conceitos de qualidade e produtividade discutidos na literatura de Engenharia de Software, que sao apresentados seguir.

3.1. Orientaçao por Dominio e Produçao Industrial de Software

Ao invés de utilizar a "guerra de mercado" como estratégia, a indústria de software deve produzir software: (1) orientado para o sistema (sistema tal como definido por Bertalanfy [BERT,77]) do usuário, para um ganho de qualidade e (2) de maneira industrial através de componentes de software, para um ganho de produtividade. Este trabalho concentra-se no segundo item.

Software Orientado para o Dominio

Pesquisas acerca da importância de produzir-se software orientado para o domínio de aplicaçao (orientado para o sistema) foram feitas por Neighbors através da técnica utilizada no sistema de software Draco [NEIG,84]. Yeh esclarece que "a experiência mostra que o reuso bem sucedido de software precisa ser orientado para o domínio" [YEH,91]. Uma característica importante da produçao orientada para o domínio sao os mecanismos adequados necessários para a captura das necessidades do usuário [GATT,90b]. Prototipaçao incremental, praticada na DuPont IEA através do modelo RIPP ("Rapid Iteractive Production Prototyping") [SCHU,90], é um exemplo de tais mecanismos.

Ao contrário do que ocorre com os produtos de software de "propósito geral", nesta abordagem há um esforço mínimo a ser colocado no sistema de software para que este vá de encontro às necessidades do usuário. Desta forma, produtos de software ganham qualidade no sentido de satisfazer às expectativas tecnológicas do cliente em questao, como mostra a Figura 3.1 [YEH,91].

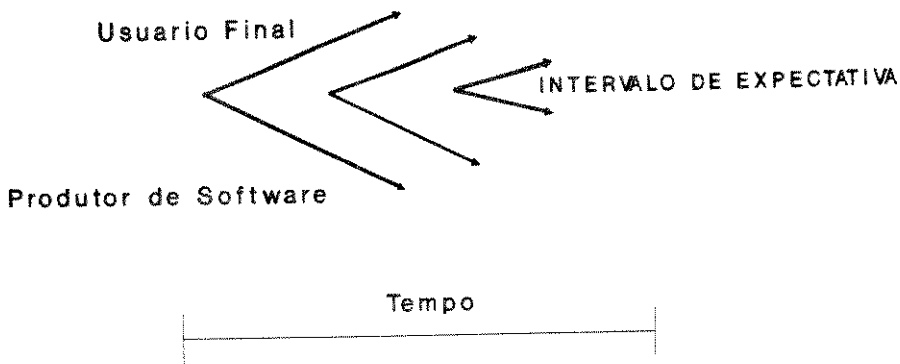


FIG. 3.1. Intervalo de Expectativa - RIPP

Produçao Industrial de Software

Como descrito em 2.4, a tecnologia de produçao de sistemas de software atualmente utilizada verifica-se inadequada para atender satisfatoriamente à crescente demanda existente. De maneira semelhante à produçao de sistemas de hardware (com seus "blocos de encaixe"), faz-se necessária uma produçao industrial de sistemas de software, com ganhos significativos de qualidade e produtividade.

Práticas de Engenharia de Software necessitam ser efetivamente utilizadas na indústria, como descrito em 2.4, para que seja factível a implantaçao de alguma forma de produçao industrial de software.

Este trabalho tem por objetivo a apresentaçao de um paradigma de composiçao de componentes como uma alternativa para a produçao industrial de software. Dentro deste contexto um processo de validaçao é aqui proposto para assegurar um ganho qualitativo de produtos de software.

3.2. Fundamentos para a Produçao Industrial de Software

Nesta seçao sao apresentados conceitos fundamentais de reuso, formalismo, abstraçao de dados e finalmente componentes de software, discutidos na literatura de Engenharia de Software como alternativas promissoras para a produçao de software. Tais conceitos sao mais adiante

utilizados na apresentaçao de um paradigma de composiçao de componentes a ser utilizado em um processo de produçao industrial de sistemas de software.

3.2.1. Reuso e Formalismo: Produtividade e Qualidade

O reuso tem sido amplamente discutido na literatura como meio de produçao de software de forma mais barata, mais rápida e melhor [YEH,91] [HORO,84] [RAMA,77] [STAN,84]. Roussopoulos [ROUS], por exemplo, afirma ser fato que a reusabilidade pode:

- melhorar a **produtividade**, removendo todo o processo de desenvolvimento do que é reusado;

- melhorar a **qualidade**, porque um componente reusável já terá alcançado um certo nivel de qualidade, devido ao seu uso repetido.

Gattaz recomenda expressivamente que um investimento maciço seja feito no reuso de software para diminuir significativamente seus custos [GATT,84].

Para que um pedaço de software seja reusável este deve ser [YEH,90]:

a. **interpretável:** usuários em potencial têm acesso a qualquer informaçao necessária para o entendimento da funcionalidade, ambiente operacional e outros atributos relevantes do software;

b. **incorporável:** o software pode ser usado para a construçao de software de maior tamanho;

c. **portável:** o software pode ser utilizado em diferentes ambientes, com diferentes máquinas.

Para assegurar a **qualidade** de um pedaço de software, a correteza deste deve ser reconhecida, i.e., deve ser assegurado que o programa faz o que ele deveria fazer. Correteza é a propriedade mais fundamental de software confiável. Se este nao estiver correto, outras propriedades como eficiência e tolerância a falhas nao fazem sentido. Para que a propriedade de correteza possa ser assegurada, é necessária uma descriçao precisa e independente do comportamento desejado de determinado programa. A esta descriçao denominamos **especificaçao formal**.

Especificaçao de programas pode ter vários graus de formalidade. No extremo do espectro informal, as especificaçoes podem ser expressas em alguma combinaçao conveniente de Inglês (Português), diagramas e uma variedade de notaçoes padrao da matemática. Algumas vezes as especificaçoes sao requisitadas num formato pré-determinado, onde a ordem das seçoes e a informaçao a ser encontrada em cada seçao sao dados, mas o conteúdo de cada seçao pode estar em qualquer linguagem. Estes tipos de especificaçoes estao em uso comum hoje.

Uma especificaçao é formal se ela é escrita totalmente em uma linguagem com uma sintaxe e semântica explícitas e precisamente definidas. Exemplos de linguagens formais sao cálculo de predicados e uma linguagem de programaçao para a qual a semântica foi definida por uma das técnicas já conhecidas (por exemplo, a axiomatizaçao de parte do PASCAL). Entretanto, um programa nao deveria ser sua própria especificaçao, pois isso elimina a redundância necessária para fazer com que a **validaçao** seja significativa. Uma descriçao que seja independente do comportamento desejável é sempre um requisito. As especificaçoes formais surgem, desta forma, entre o conceito existente na mente humana - ou seja, o que o programa deve fazer - e o próprio programa. A correteza do programa é estabelecida provando-se que o programa é equivalente à sua especificaçao [LISK,75].

Várias sao as vantagens associadas a **especificaçoes formais** [LISK,75] [LISK,86] [PARN,86] [JONE,86]. Algumas delas sao listadas a seguir:

a. podem ser estudadas matematicamente;

b. podem ser processadas por computador, detectando-se algumas formas de incompleteza ou inconsistência antes da etapa de implementaçao [GUTT,75];

c. provas de correteza podem ser auxiliadas por computador;

d. sao valiosas para tornar "público" o código fonte , já que as propriedades dos programas sao especificadas de maneira formal e nao ambigua;

e. auxiliam no entendimento do conceito envolvido, devido ao fato de que uma descriçao completa, concisa e inteligível deve ser produzida;

f. sao um bom meio de comunicaçao, por causa de sua linguagem nao ambigua e da interpretaçao formal de seu significado (comunicaçao entre "designers" e programadores, programadores e programadores, etc.);

g. útil na documentaçao de programas, que poderá servir de auxilio à manutençao e evoluçao dos mesmos.

h. sao provavelmente tao dificeis de serem construidas quanto as especificaçoes informais - estas podem parecer mais fáceis porque sao geralmente incompletas.

Especificaçoes formais e informais podem perfeitamente complementar umas às outras. Especificaçoes informais têm a virtude de que os pontos principais do comportamento podem ser transmitidos numa maneira efetiva e inteligível; infelizmente, detalhes do comportamento geralmente nao estao especificados. Especificaçoes formais contêm todos os detalhes, mas pode nao haver suficiente ênfase nos pontos principais. Desta forma, é recomendado que especificaçoes formais sejam acompanhadas por especificaçoes informais como comentários. Desta maneira, o leitor pode captar a idéia da especificaçao rápida e facilmente, mas também tem informaçao suficiente para entender profundamente o seu significado [LISK,75].

A próxima seçao apresenta consideraçoes acerca da unidade de programa a ser representada por uma especificaçao, buscando-se um tipo de abstraçao que facilite o reuso.

3.2.2. Unidade de Programa : A Abstraçao de Dados

A qualidade de uma especificaçao é em grande parte dependente da unidade de programa sendo especificada. Caso a especificaçao esteja associada a uma unidade muito pequena, esta pode tornar-se desinteressante, além de serem geradas mais descriçoes do que poderiam ser manipuladas de forma conveniente. O exemplo seguinte mostra a especificaçao associada a uma linha de código, que seria nada mais do que um comentário normal de programa [LISK,79]:

```
x := x + 1      "incremente x de 1"
```

Uma especificaçao de uma unidade muito pequena nao traduz nehum conceito interessante. Busca-se uma unidade de especificaçao que corresponda naturalmente a um conceito ou abstraçao.

O tipo de abstraçao mais comumente utilizada é a **abstraçao procedural ou funcional**, onde faz-se um mapeamento de um conjunto de valores de entrada para um conjunto de valores de saida [LISK,75].

Outra forma de abstraçao é a **abstraçao de dados**, que fornece um conjunto de valores de dados e um conjunto de operaçoes para manipulaçao destes valores, sendo que o comportamento do tipo abstrato só pode ser observado pela aplicaçao das operaçoes. Um exemplo clássico de abstraçao de dados é o da pilha, onde a classe de objetos consiste de todas as pilhas possíveis e o grupo de operaçoes incluem push e pop, além de um operador para consulta ao topo da pilha [CAMA,90]. Outros exemplos comumente usados sao processos, filas, listas, tabelas de símbolos e máquinas abstratas. Em cada caso, a implementaçao da abstraçao é dada na forma de um módulo "multiprocedure". Cada procedimento do módulo implementa uma das operaçoes. Os procedimentos sao agrupados porque interagem de alguma das seguintes formas [LISK,75]:

- compartilhamento de recursos (base de dados);

- compartilhamento de informaçao (formato e significado dos dados da base compartilhada, e significado dos estados do recurso compartilhado).

Considerar-se todo o grupo de procedimentos como um módulo permite que a informaçao acerca das interaçoes seja ocultada de outros módulos, isto é, outros módulos obtêm informaçao a respeito das interaçoes invocando os procedimentos do grupo. Tal **ocultamento de informaçao** simplifica a interface entre módulos e leva diretamente a especificaçoes mais simples, pois é exatamente a interface que estas devem descrever.

Uma descriçao simples e precisa da interface de um módulo facilita o acesso à informaçao necessária para o entendimento de sua funcionalidade, tornando-o potencialmente mais interpretável, o que é requisito fundamental para seu reuso.

O exemplo que se segue [LISK,75] ilustra problemas que podem surgir quando a abstraçao de dados é ignorada e utilizam-se especificaçoes procedurais para descriçao das operaçoes do grupo independentemente uma das outras. A operaçao push (pilha) é considerada uma funçao da seguinte forma:

```
push: pilha x inteiro -> pilha
```

A especificaçao procedural deve definir o conteúdo do valor de saída de push (objeto pilha retornado por push), em termos dos valores de entrada de push (objeto pilha e um inteiro). Isto pode ser feito definindo-se uma estrutura para objetos pilha e descrevendo-se o efeito de push em termos desta estrutura. Poderíamos ter, em PASCAL, a seguinte estrutura:

```
type pilha = record top: integer,
                    data: array [1...100] of integer
                    end
```

Teríamos entao que o significado de

```
t := push(s, i)
```

poderia ser descrito da seguite maneira:

```
true {t := push(s,i)} ∀j [1 <= j <= s.top
                    t.data[j] = s.data[j]
                    & t.data[t.top] = i
                    & t.top = s.top + 1]
```

Alguns problemas associados à especificaçao anterior:

- **excesso de detalhes** - nao descreve o comportamento da pilha, sendo que conceitos de tal comportamento podem somente ser extraídos dos detalhes. Tal ponto é indesejável pois:

  - O inventor do conceito deve se atolar em detalhes ao invés de descrever o conceito diretamente;

  - A interpretabilidade do comportamento da pilha é prejudicada com a dificuldade de abstraçao do conceito, advinda do excesso de detalhes. Tal fato enfraquece o seu potencial de reuso.

  - a inclusao de detalhes fere o critério de especificaçao mínima, além da prova de correteza do push para uma pilha de estrutura diferente ser difícil.

- **independência ilusória** - na verdade as especificaçoes dos operadores push e pop, por exemplo nao sao independentes, já que uma mudança na especificaçao de um deles implicaria na mudança do outro.

Caso uma abstraçao de dados como a pilha for especificada como uma única entidade, detalhes desnecesssários desaparecerao (interaçoes entre as operaçoes) [LISK,75].

Algumas técnicas de especificaçao para abstraçoes de dados utilizam especificaçoes procedurais para descrever os efeitos das operaçoes. Tais especificaçoes sao expressadas em termos de objetos abstratos com propriedades abstratas, ao invés de propriedades especificas. Outras técnicas nao exigem uma descriçao individual e separada das operaçoes, sendo que os efeitos destas podem ser descritos em termos umas das outras. Como exemplo deste tipo de técnica, o operador pop pode ser descrito em termos de push da seguinte forma:

pop(push(s,v)) = v , pop retorna o último valor empilhado.

O aspecto de ocultamento de informaçao de abstraçoes de dados contribui fortemente para que estas sejam consideradas unidades de programa interessantes a serem especificadas [LISK,75], por permitirem que estas unidades sejam mais facilmente interpretáveis, reforçando seu reuso.

Uma unidade de alto grau de reusabilidade é apresentada na próxima seçao, tendo em vista a produçao industrial de software.

3.2.3. Componentes de Software Reusáveis - CSR: A Unidade de Produçao Industrial de Software

Segundo Yeh, uma forma promissora de ir de encontro a desafios existentes na área de software está na "Engenharia de Componentes", na qual sistemas de software sao construídos através de "blocos de encaixe" ("building blocks"), que podem ser prontamente reusados [YEH,91], assim como na tecnologia de produçao de hardware. Em outras palavras, sistemas de software devem constituir-se de composiçoes sucessivas de blocos de encaixe, que sao as unidades básicas de produçao de tais sistemas.

Uma forma reconhecida na literatura de aumentar a produtividade e qualidade de software é o reuso. Uma abstraçao de dados foi anteriormente descrita como uma unidade de representaçao de software que favorece o reuso, devido ao princípio de ocultamento de informaçao; este princípio permite um alto grau de interpretabilidade, fundamental para o reuso.

A linguagem precisa e nao ambígua das especificaçoes formais, aliada à interpretaçao formal (e única) de seu significado, fazem com que as mesmas proporcionem um alto grau de interpretabilidade a uma unidade de software qualquer. O fato das especificaçoes formais poderem ser processadas por computador faz com que a qualidade do conceito por elas representado apresente alto grau de qualidade através de provas de correteza, consistência e/ou completeza automáticas. Para que a idéia central da unidade sendo especificada seja captada rapidamente, descriçoes informais devem acompanhar a especificaçao formal, como recomendado em 3.2.1.

Este trabalho apresenta **Componentes de Software Reusáveis** - CSR's - como a unidade para a produçao industrial de software. O CSR é constituído por paradigmas de abstraçao de dados e formalismo, visando o reuso, como estratégia para melhorar a qualidade e produtividade de sistemas de software.

Para proporcionar o reuso CSR's sao:

- **incorporáveis:** Um mecanismo sucessivo de composiçao de componentes permite que sistemas de software sejam construídos a partir da incorporaçao de CSR's ditos subordinados. A Figura 3.2 ilustra tal mecanismo através de um grafo, onde os nós representam os CSR's e o conjunto de arcos a relaçao de composiçao C. A arquitetura de componentes [WEBE,88] apresentada na sequência, materializa tal mecanismo, assegurando a incorporabilidade de componentes de software.

- **interpretáveis:** A informaçao necessária para o entendimento dos atributos relevantes de um CSR é representada em descriçoes parciais, denominadas **"views" ou projeçoes**. A vantagem desta forma de descriçao em relaçao a uma descriçao monolítica, está no fato de descriçoes parciais serem mais inteligíveis e mais facilmente manipuláveis. A superposiçao de todas as projeçoes de um CSR forma uma descriçao completa. A Linguagem-Pi [WEBE,88] é utilizada neste trabalho para a especificaçao de componentes por suportar este paradigma de descriçoes parciais. Além disto, esta linguagem caracteriza-se pelo uso de especificaçoes formais, aliadas a descriçoes informais, proporcionando um alto grau de interpretabilidade à unidade sendo especificada.

- **portáveis:** De acordo com Kernighan a portabilidade do sistema operacional UNIX é um fator em grande parte responsável pelo seu sucesso [KERN,84]. Componentes construídos em ambiente UNIX, por exemplo, poderiam ser **portáveis** para outros ambientes computacionais. Procura-se, desta forma, indicar um caminho para a produçao industrial de software, de maneira análoga à produçao de hardware, através de paradigmas de reuso e formalismo.

O paradigma de componentes de software descrito a seguir baseia-se no trabalho descrito por Weber [WEBE,88].

Componentes de Software Reusáveis sao unidades autônomas de processamento a serem utilizadas para a construçao de sistemas de software complexos. Cada CSR representa um tipo abstrato, em equivalência ao paradigma de abstraçao de dados descrito em 3.2.2. Associados a cada tipo estao os objetos de dados, que sao instâncias do tipo encapsuladas pelo componente.

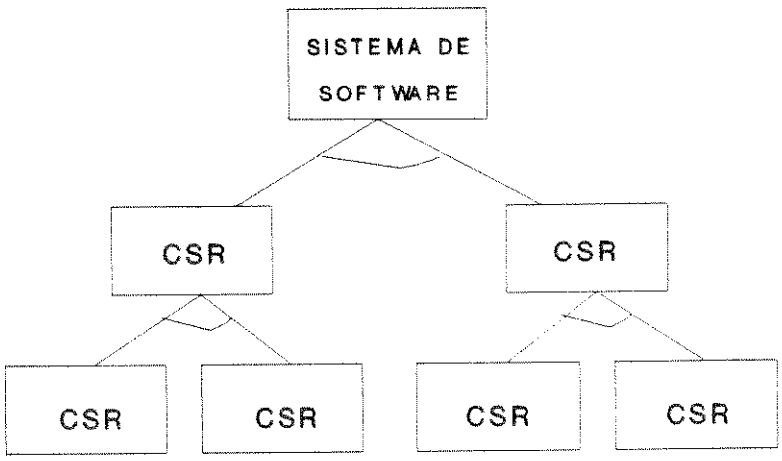


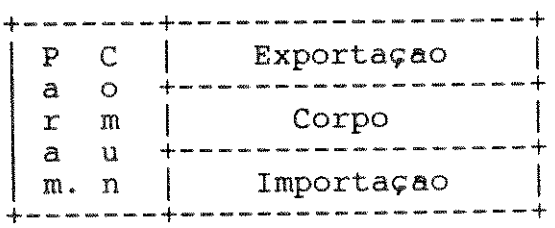
FIG. 3.2. CSR's incorporaveis

CSR's concretizam conceitos de ocultamento de informaçao e abstraçao de dados na medida em que nenhuma operaçao definida fora de um CSR (dentro de outro CSR) pode acessar diretamente e modificar o objeto de dados encapsulado. Operaçoes de outro CSR somente podem utilizar operaçoes associadas com um CSR para modificar o objeto de dados encapsulado. Além disto a presença de um objeto de dados é conhecida somente dentro do CSR.

CSR's sao **unidades executáveis de software**, cuja execuçao é iniciada por usuários ou outros CSR's. Os mesmos podem ser executados de várias formas diferentes, através da invocaçao de diferentes operaçoes que estes exportam para uso. Um CSR exporta nomes de operaçoes e um conjunto de parâmetros para a eventual chamada a estas operaçoes.

Arquitetura de CSR's

O CSR foi criado para prover um mecanismo de construçao de objetos encapsulados a partir de objetos de dados mais primitivos. Para a construçao de objetos de dados e operaçoes a partir de CSR's subordinados, o CSR construido importa operaçoes do subordinado. A estrutura geral de um CSR é mostrada a seguir:



A interface de **Exportaçao (Export)** mostra aquelas partes do CSR que necessitam ser visíveis para a invocaçao de suas operaçoes, ou seja, para que estes possam tornar-se **incorporáveis**, permitindo seu reuso.

O **Corpo (Body)** provê os mecanismos para construçao do objeto de dados encapsulado e das operaçoes associadas a partir dos objetos e operaçoes subordinados importados, i.e., o Corpo descreve a construçao das operaçoes exportadas pelo CSR a partir das operaçoes dos componentes importados.

A **Interface de Importaçao (Import Interface)** contém os CSR's subordinados que devem ser importados para possibilitar o funcionamento correto do CSR.

A seçao de **Parâmetros Comuns** consiste de tipos atômicos importados pelo CSR e novamente exportados por este. Um componente é aqui uma unidade pela qual estes parâmetros passam, sem serem modificados, para o próximo nível mais alto de abstraçao na hierarquia do sistema de software.

O mecanismo de importaçao de CSR's permite que componentes hierarquicamente superiores reusem outros componentes, denominados CSR's subordinados. Tal estrutura de importaçao-exportaçao permite que sistemas de software sejam constituídos por uma hierarquia de componentes reusáveis, onde componentes nao primitivos sao construídos a partir de outros componentes, tornando esta unidade de representaçao de software **altamente incorporável.**

Aplicaçoes do Paradigma de CSR's

Como descrito em 3.2.1, as especificaçoes formais encontram-se entre o conceito existente na mente humana (o que o programa deve fazer) e o próprio programa. Ela consiste de uma descriçao independente e precisa do comportamento do programa desejado, provendo a redundância necessária para uma validaçao significativa. Tal validaçao consiste em assegurar que o programa é equivalente à sua especificaçao. Este trabalho apresenta, no Capitulo 4 uma proposta para a **validaçao de componentes de software**, e no Capitulo 5 uma ferramenta de software construida para prover automaticamente tal validaçao. Para que esta validaçao formal e automática fosse possivel, a técnica formal de Traços [PARN,86] foi utilizada, em conjunçao com a Linguagem-Pi. Desta forma procura-se assegurar a qualidade de sistemas de software construidos dentro do paradigma aqui apresentado.

Além da construçao de sistemas de software novos, a técnica de "componentizaçao" ("componentry") pode ser também utilizada para a **evoluçao** de produtos de software existentes [YEH,91] [RUOT,90]. A presença de uma grande demanda de evoluçao de produtos existentes foi identificada na seçao 2.1 deste trabalho. Moura apresenta uma forma de identificaçao de CSR's a partir de código fonte existente [MOUR,90]. Software mal-estruturado e de dificil manutençao é entao reestruturado em uma hierarquia de CSR's, sendo esta mais flexivel e de mais fácil manutençao .

O próximo capitulo apresenta uma forma de construçao de software a partir de CSR's, tendo por base paradigmas utilizados na Linguagem-Pi. Neste contexto, um processo de validaçao de componentes é nele ainda proposto.

CAPITULO 4. Composiçao de Componentes - Uma Estratégia para Produçao Industrial de Software

Este capítulo apresenta um processo de construçao industrial de sistemas de software a partir da unidade de produçao de software descrita no capítulo anterior: Componentes de Software Reusáveis.

E aqui apresentada uma metodologia baseada no mapeamento de tipos como metodologia de produçao de sistemas de software, sendo esta materializada na composiçao de componentes pela Linguagem-Pi. A qualidade dos componentes compostos é assegurada pelo processo de validaçao proposto ainda neste capítulo.

4.1. Produçao de Sistemas de Software: do Abstrato ao Concreto

Esta seçao descreve princípios desejáveis em uma composiçao de componentes de software, tendo por base os paradigmas de reuso e formalismo descritos anteriormente. Uma metodologia para construçao de sistemas de software é aqui apresentada sendo, em seçoes posteriores, mapeada na metodologia de composiçao adotada.

O desenvolvimento de um sistema de grande porte se dá pela quebra da tarefa de desenvolvimento em um número de subtarefas, manipuláveis pela mente de cada um de seus executores. Faz-se entao necessária uma integraçao das partes produzidas [WEBE,91].

Seguindo o conceito de divisao de tarefas , surge o princípio de "**dividir para conquistar**" [WEBE,88]. Este princípio é aplicável na divisao de um sistema grande de software em partes manipuláveis. A quebra de um sistema grande resulta em uma hierarquia de componentes, que descreve a construçao hierárquica dos componentes a partir de componentes primitivos (indivisíveis) **incorporáveis**, o que é fundamental para seu efetivo reuso.

O princípio de "dividir para conquistar" provê base para o princípio de **ocultamento da informaçao**. A informaçao é ocultada em componentes subordinados somente para uso interno. O componente construído nao possui nenhum conhecimento acerca da informaçao oculta nos

subordinados. O componente construido representa uma linguagem de descriçao abstrata do sistema, escondendo informaçao nos componentes que o constituem.

Um sistema de software complexo possui inevitavelmente uma descriçao complexa, o que é indesejável para qualquer atividade de reuso. Para que tal complexidade se torne manipulável, é necessária uma sub divisao da descriçao em descriçoes parciais mais inteligiveis que, superpostas, constituem a descriçao completa. Tal mecanismo representa o principio de "separaçao de concerns", desejável em um processo de composiçao de componentes [WEBE,88]. Este principio é uma forma de tornar um componente **interpretável** o que, como descrito em 3.2.1, é fundamental para seu reuso.

O principio de separaçao de "concerns" provê base conceitual para permitir a **negligência de informaçao** em uma descriçao parcial de um sistema de software. Informaçao adicional faz-se necessária para uma descriçao completa. A descriçao parcial é uma imagem abstrata do sistema, ignorando informaçao a ser representada em outra descriçao parcial.

Os dois principios de abstraçao anteriores sao ortogonais:

a) ocultamento de informaçao - sistemas de software sao descritos como consistindo de componentes em diferentes niveis de detalhe. Esta forma de descriçao permite o desenvolvimento de descriçoes do sistema com refinamentos graduais e sucessivos.

b) negligência de informaçao - sistemas de software podem ser descritos em fatias, onde cada uma representa a descriçao de uma ou mais de suas caracteristicas.

Uma técnica para produçao de sistemas de software, baseado no mapeamento de tipos abstratos de dados em tipos do mundo real (concreto), é apresentada por Booch [BOOC,83], esquematizada na Figura 4.1.

A produçao de um sistema de software tem inicio com a identificaçao e análise do domínio de aplicaçao do usuário. Uma especificaçao abstrata do sistema é entao produzida, onde sao descritos os **tipos abstratos** que representam o domínio. Tal tipo abstrato consiste de uma estrutura de dados e um conjunto de operaçoes que, juntos, produzem resultados (estados) do mundo abstrato. Tais resultados sao entao mapeados para o mundo computacional através de uma linguagem de programaçao, por exemplo,

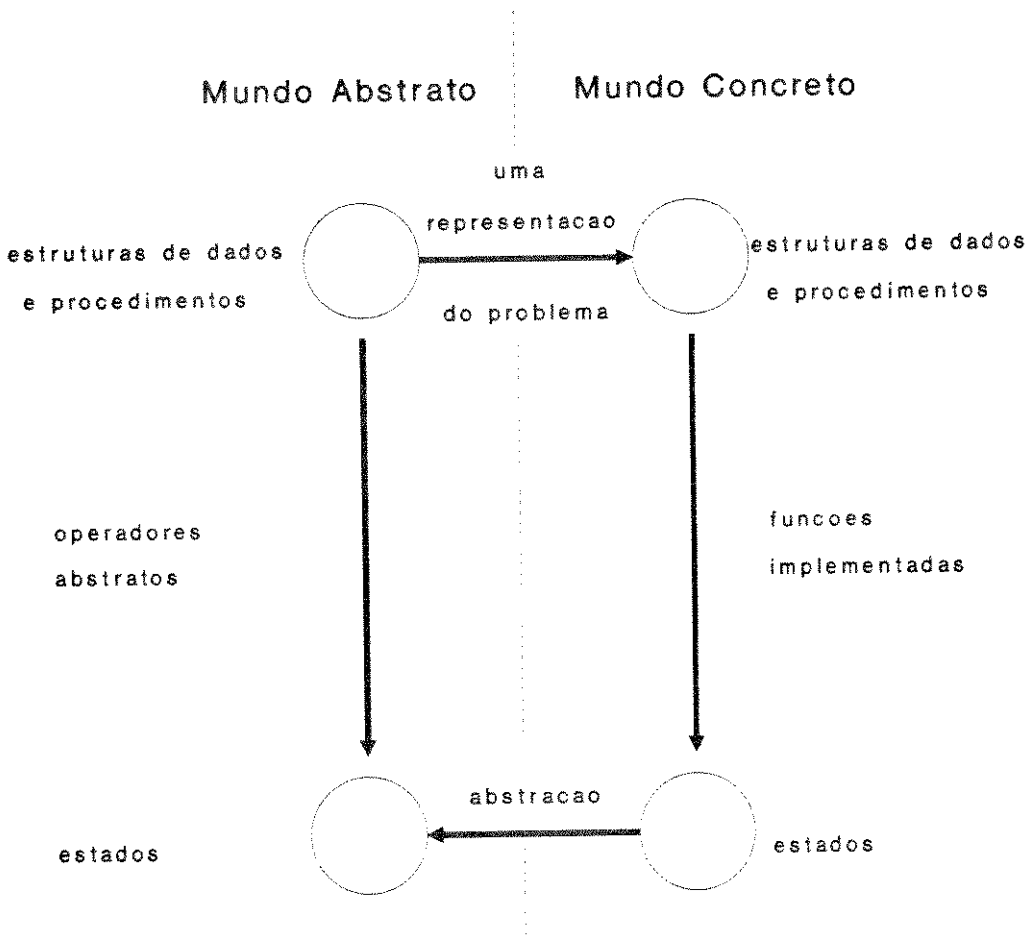


FIG. 4.1. Mapeamento de Tipos

criando **tipos concretos**. Da mesma forma, estes sao constituidos por uma estutura de dados e operaçoes, que produzem resultados do mundo real. Os resultados do mundo abstrato e do mundo concreto sao entao validados (abstraçao) e quanto mais próximos forem, maior qualidade possuirá o sistema de software.

Desta forma um sistema de software é dividido em partes manipuláveis, i.e., tipos abstratos de dados que juntos representam o espaço do domínio de aplicaçao, o que denota o princípio de "dividir para conquistar" anteriormente citado. O princípio de "separaçao de concerns" é aqui aplicado quando da divisao da descriçao do componente em mundo abstrato e mundo concreto. No momento da especificaçao abstrata do domínio, informaçoes de seu mundo concreto sao "negligenciadas", para que se tenha uma descriçao manipulável e inteligível do sistema. Aspectos concretos do domínio sao considerados posteriormente. Sobrepostas, ambas as especificaçoes formam uma descriçao completa do sistema.

A seçao seguinte apresenta a linguagem-pi, que tem por base os conceitos de composiçao anteriormente colocados, como uma alternativa para a construçao de sistemas de software.

4.2. Interconexao de Componentes: A Linguagem-Pi

Esta seçao apresenta uma técnica de construçao de sistemas de software através da composiçao de CSR's, como uma estratégia para a produçao industrial de software. Esta técnica baseia-se na **Linguagem-Pi** [CRAM,90] [WEBE,88], utilizada no projeto tecnológico europeu ESPRIT no âmbito da ESF (EUREKA Software Factory).

A Linguagem-Pi é uma linguagem de especificaçao para descriçao de sistemas de software nos diferentes estágios do processo de desenvolvimento de software. Tem por objetivo possibilitar a produçao industrial e sistemática de sistemas de software grandes, servindo de suporte para o "programming in the large". A linguagem nao procura suportar um processo particular de produçao de software. Busca-se a determinaçao de propriedades ou caracteristicas de sistemas de software grandes.

Na construçao da linguagem-pi procurou-se gerar uma linguagem expressiva porém simples, utilizando-se de descriçoes formais e informais; a linguagem-pi possui um número pequeno de tipos de elementos, com um número reduzido de conceitos e primitivas, procurando mantê-la inteligível e aplicável. Tais caracteristicas foram possíveis devido ao fato de todo sistema de software ser encarado como algo estruturado uniformemente e composto de CEM's ("Concurrently Executable Modules"). Estes, definidos no escopo da ESF, possuem um mapeamento de um para um com os CSR's anteriormente conceituados.

O CEM é, desta forma, a "única" coisa a ser descrita pela linguagem-pi. Esta pode ser vista, portanto, como uma linguagem de descriçao e interconexao de CEM's, de acordo com o princípio de **"dividir para conquistar"** descrito em 4.1, que se refere'a estruturaçao do sistema de software.

O princípio de dividir para conquistar da Linguagem-Pi materializa a propriedade de **incorporabilidade** de Componentes de Software Reusáveis descrita anteriormente neste trabalho.

O princípio de **"separaçao de concerns"**, que trata da estruturaçao do processo de desenvolvimento de software, é aplicado na divisao da descriçao complexa em (espera-se) descriçoes parciais mais inteligíveis chamadas de **views ou projeçoes**. Estas sao posteriormente superpostas formando uma descriçao completa.

A estruturaçao de descriçoes parciais de CEM's, proporcionada pela Linguagem-Pi, materializa a caracteristica de **interpretabilidade**, fundamental em componentes reusáveis. Para tal, descriçoes formais sao utilizadas, aliadas a especificaçoes informais - o que permite um entendimento ao mesmo tempo fácil, com ênfase nos principais pontos, e completo e preciso, respectivamente.

A seguir sao apresentados os paradigmas de CEM's e "views" que materializam os requisitos de incorporabilidade e interpretabilidade de componentes de software, fundamentais para um efetivo reuso.

4.2.1. Separaçao de "Concerns" - "Views"

A especificaçao de CEM's se dá através de "views" ou projeçoes, que sao definidas pela aplicaçao do princípio de separaçao de "concerns", provendo uma alta interpretabilidade ao sistema de software. Uma "view" fornece um especificaçao parcial de um sistema, sendo que uma especificaçao completa consiste da combinaçao de várias "views". Tal combinaçao é possível pelo fato de uma "view" ser definida como o resultado da projeçao de um CEM em uma propriedade particular. As projeçoes diferentes serao posteriormente juntadas, formando a especificaçao completa do CEM.

A linguagem-pi suporta várias projeçoes de um CEM: projeçao de tipo, de objeto, de concorrência, de inteconexao e de distribuiçao. Sao comentadas a seguir as projeçoes de tipo e de objeto, as quais sao as projeçoes necessárias para uma validaçao entre o mundo abstrato e o concreto, repectivamente, representado, uma **especificaçao abstrata** e uma **especificaçao concreta** do sistema de software, segundo descrito em 4.1.

a. "view" de tipo: fornece a descriçao dos tipos abstratos representados por um CEM. Nao sao descritas as instanciaçoes individuais de um tipo, mas sim a coleçao de todos os objetos do mesmo tipo encapsulados por um CEM. A "view" de tipo nao considera conceitos como objetos, variáveis ou valores, concentrando-se em conceitos mais abstratos do tipo, ou seja, em propriedades comuns de todos os objetos daquele tipo.

b. "view" de objeto: trata do mundo concreto de um CEM, especificando os objetos individuais de um dado tipo. Formaliza assuntos tais como a criaçao e posterior destruiçao de objetos de dados, ou o número (finito ou nao) de objetos de um dado tipo que podem ser criados e mantidos.

As "views" anteriores sao abordadas em mais detalhes a seguir.

"View" de Tipo

Consiste de uma especificaçao da interface de exportaçao, uma especificaçao do corpo, uma da interface de importaçao e uma seçao de parâmetros comuns.

A interface de **exportaçao** denota um tipo de dados com suas operaçoes associadas e as regras que definem as propriedades das operaçoes, a qual denomina-se **especificaçao do tipo exportado**.

O **corpo** denota o tipo construído com suas operaçoes associadas, o mecanismo de construçao e as regras que regem tal mecanismo a partir de tipos subordinados importados.

A interface de **importaçao** denota um ou mais tipos de dados com suas operaçoes associadas e, para cada tipo de dados importado, as regras a serem obedecidas quando da aplicaçao de suas operaçoes a qualquer objeto do respectivo tipo.

A seçao de **parâmetros comuns** consiste de tipos atômicos importados pelo CEM e novamente exportados por este. Um componente é aqui uma unidade pela qual estes parâmetros passam, sem serem modificados, para o próximo nível mais alto de abstraçao na hierarquia do sistema de software.

Os tipos importados e o tipo construído encontram-se na relaçao de construçao previamente mencionada, a qual se materializa através do mecanismo e das regras de construçao fornecidas pelo corpo do CEM.

O tipo construído e o tipo exportado possuem uma relaçao de representaçao: o tipo exportado pode ser representado pelo tipo construído.

A "view" de tipo define aquelas propriedades de um CEM que independem de uma execuçao particular deste, sendo assim denominadas **propriedades estáticas**. Estas tratam da estrutura do CEM, (ou seja, de sua construçao a partir de CEM's subordinados importados) com o efeito de uma operaçao sobre um objeto do tipo e com as relaçoes entre os efeitos das operaçoes diferentes.

O esqueleto de um sistema de software é descrito pela "view" de tipo, com os tipos que constituem todos os CEM's que farao parte do software. Como os próprios CEM's sao hierarquicamente estruturados, tais tipos também o devem ser. A descriçao "view" de tipo nao se refere a objetos e execuçoes, representando uma visao abstrata do sistema. Uma hierarquia de tipos que representa uma "view" de tipos de um sistema é chamada de **arquitetura abstrata do sistema**.

"View" de Objeto

A "view" de objeto do sistema também consiste de uma especificaçao de interface de exportaçao, uma especificaçao de corpo, uma especificaçao da interface de importaçao e uma seçao de parâmetros comuns. Como esta "view" lida com objetos de dados individuais de um dado tipo, a especificaçao correspondente também necessita fazê-lo.

Assim a interface de **exportaçao** denota um objeto de dado de exportaçao e suas operaçoes associadas. O **corpo** descreve a construçao de um objeto de dado a partir de seus objetos de dados subordinados importados e a construçao das operaçoes associadas a partir das operaçoes subordinadas importadas. A interface de **importaçao** denota os objetos de dados de importaçao com suas operaçoes associadas. A seçao de **parâmetros comuns** descreve os objetos de dados atômicos importados pelo CEM e novamente exportados por este.

A "view" de objeto tem o objetivo de ser a caracterizaçao de objetos de dados individuais passíveis de manipulaçoes na execuçao das operaçoes. As propriedades estáticas de um objeto estabelecidas na "view" de tipo de um CEM devem ser preservadas na "view" de objeto. As mesmas operaçoes associadas ao tipo de objeto na "view" de tipo sao associadas a objetos individuais na "view" de objeto. Desta forma, a descriçao de CEM's na "view" de objeto pode ser encarada como a descriçao de instanciaçoes de um CEM como descrito na "view" de tipo.

E ainda descrito na "view" de objeto a possível criaçao e destruiçao de objetos individuais durante a execuçao das operaçoes de "create" e "destroy". Estas podem acontecer em momentos diferentes. Esta "view" deve fixar as regras de criaçao e abandono de objetos. E descrito também aqui precisamente quantos objetos de um determinado tipo de objeto podem ser criados e quando estes podem ser criados e destruídos.A "view" de objeto é dita ser a descriçao da **arquitetura concreta do sistema**.

4.3. A Linguagem de Especificaçao de Traços

O método de traços [PARN,86] para a especificaçao de software consiste em uma linguagem de especificaçao formal e abstrata, que permite a descriçao do comportamento de operaçoes associadas a um tipo abstrato em termos de outras operaçoes com os quais ele interage.

Camargo [CAMA,90] apresenta uma sucinta análise comparativa de especificaçoes formais, onde a técnica de traços satisfaz os critérios de avaliaçao de técnicas de especificaçoes formais de formalismo, facilidade de construçao, facilidade de compreensao, minimalidade, escopo de aplicaçao e extensibilidade [LISK,75] Neste trabalho é concluído que a técnica de traços pode representar abstraçoes de dados de forma satisfatória .

Uma especificaçao segundo esta técnica consiste de duas partes: a sintaxe e a semântica. A primeira define a funcionalidade das operaçoes do componente, i.e., o domínio dos valores de suas entradas e saídas [LISK,75]. Tais operaçoes (ou funçoes) sao divididas da seguinte forma:

1. Funçoes-O: causam mudança no estado do componente.

2. Funçoes-V: nao ocasionam mudança de estado, mas permitem que algum aspacto do estado do componente seja observado.

A parte de semântica consiste de três tipos de assertivas:

1. Legalidade: denota a aplicabilidade das operaçoes;

2. Equivalência: define uma relaçao de equivalência de traços, especificando uma álgebra para as operaçoes;

3. Valores: descreve os valores retornados pelas funçoes-V.

Um traço é uma aplicaçao das operaçoes de um componente, sendo constituído por chamadas sucessivas a tais operaçoes. Uma amostra de traços canônica é um conjunto de traços que expressam todos os possíveis estados do tipo pelo uso de funçoes-O.

A sintaxe da parte sintática é dada a seguir:

<nome da funçao>: <tipo de parâmetro> x...x <tipo de parâmetro> -> <tipo de resultado>

onde sao fornecidos os nomes de todas as funçoes e o tipo de cada parâmetro.

A segunda parte da especificaçao de traços é a **semântica**, que fornece o comportamento que os procedimentos do módulo devem exibir. Isto é feito utilizando-se os três tipos de assertivas anteriormente citados. Estas assertivas sao baseadas em lógica de primeira ordem complementadas pelo predicado L, que é verdadeiro quando aplicado a um traço legal, e pelos Operadores_V. As mesmas sao descritas a seguir:

1. Legalidade: Estas afirmativas identificam um subconjunto do conjunto de traços legais, i.e., um conjunto tal que ao chamar as funçoes como descrito no traço (módulo no estado inicial), nao resultará em uma "trap", ou seja, situaçao de erro. Qualquer traço que nao puder ser considerado legal pelo uso dessas afirmaçoes será considerado traço ilegal.

2. Equivalência: Especifica uma relaçao de equivalência de traços, tal que:

- Traços equivalentes possuem a mesma legalidade, i.e., ambos sao legais ou nao legais;

-Possuem o mesmo efeito visível externo no módulo.

3. Valores: Afirmaçoes que descrevem os valores retornados pelos Operadores_V para um subconjunto de traços legais.

A notaçao abaixo foi utilizada para descrever os traços:

<subtraço> ::= Ø |<chamada a funçao>|
<subtraço>.<chamada a funçao>

<traço> ::= Ø |<subtraço>[.<subtraço>]

[<T>] denota um número qualquer de ocorrências de <T> (traços)

"Ø" denota um traço vazio. (Nunca ocorre em um traço)

O traço pressupoe a execuçao das funçoes da esquerda para a direita.

Para a descriçao da Legalidade, introduzimos o predicado L(T), onde T é um traço. L(T) é verdadeiro se T for um traço legal.

O predicado L(Ø) é sempre assumido como legal.

Se T é um traço e S um subtraço, entao

$$L(T.S) \Rightarrow L(T) ,$$

i.e., o prefixo de um traço legal é sempre um traço legal.

A representaçao dos valores dos Operadores_V é feita da seguinte forma:

V(T.X) descreve o valor retornado por X quando chamado após a execuçao de T

onde: T é um traço legal;

X é uma chamada sintaticamente correta a um Operador_V;

L(T.X) é verdadeiro.

Na notaçao da equivalência, se T1 e T2 sao traços entao T1 == T2 (T1 equivalente a T2) implica:

- para qualquer subtraço S, L(T1.S) = L(T2.S);

- para qualquer subtraço S e Operador_V X,
  L(T1.S.X) => V(T1.S.X) = V(T2.S.X)

Traços e a Validaçao de CSR's

A Linguagem-Pi, desenvolvida pelo projeto ESPRIT europeu, utiliza especificaçoes algébricas [CRAM,90] para a descriçao formal dos tipos abstratos encapsulados por seus componentes de software.

As especificaçoes algébricas assemelham-se às especificaçoes de Traços. As primeiras também possuem uma parte de sintaxe e semântica e também descrevem os operadores do tipo abstrato em termos uns dos outros. Elas nao possuem, no entanto, nenhum artifício que permita a expressao de um traço, i.e., de uma aplicaçao de operaçoes do tipo, dentro de suas assertivas. Isto faz com que seja difícil a inserçao das regras associadas ao tipo em um contexto do mundo concreto, o que é altamente desejável em um processo de validaçao de concreto x abstrato. O exemplo seguinte ilustra tal fato.

E dada a seguinte assertiva de uma especificaçao algébrica para o tipo pilha [CAMA,90]:

```
pop( push(P,elem) ) = P;          (1)
```

onde P representa o tipo pilha sendo especificado, e elem o elemento sendo empilhado.

Da mesma forma, a seguinte assertiva de equivalência é dada para o mesmo tipo pilha, utilizando-se a notaçao de traços [PARN,86]:

```
T.Push(P, elem).Pop() == T;       (2)
```

Ambas as assertivas traduzem o mesmo conceito: a aplicaçao da operaçao Push, seguida da aplicaçao de Pop, nao provoca uma mudança de estado, sendo estas operaçoes complementares na álgebra da pilha.

A assertiva (2), porém, afirma que, para qualquer traço T (que nao resulte em "trap"), ou seja, para qualquer estado válido da pilha, a aplicaçao de Push e Pop consecutivamente nao altera o estado resultante da aplicaçao do traço T. Tal fato abre espaço para que as assertivas sejam inseridas dentro de um contexto do mundo real, onde qualquer execuçao de um T legal poderá exprimir a propriedade descrita pela assertiva.

A possibilidade de inserçao de contexto do mundo real na especificaçao é de extrema importância para um processo de validaçao que consista em assegurar que o produto construído no mundo concreto satisfaz os seus requisitos iniciais, descritos pela especificaçao formal, no mundo abstrato.

No caso da assertiva (1), um processo de validaçao poderia se dar, por exemplo, verificando se no mundo concreto da pilha a aplicaçao de uma operaçao Push seguida de Pop nao altera o estado resultante da pilha. Isto seria feito, no entanto, sem levar em conta os possíveis estados iniciais do tipo: como assegurar que esta assertiva é verdadeira para todo o espaço de estados do tipo? Ou mesmo para um sub-conjunto "significativo" de estados?

A assertiva (2) abre espaço para que esta questao seja resolvida através da amostra de traços. Esta constitui-se de um conjunto de aplicaçoes das operaçoes do tipo, exprimindo possíveis estados resultantes. Uma amostra de traços é dita canônica caso ela exprima todos os possíveis estados do tipo, pela invocaçao de suas funçoes-O. Pesquisas futuras necessitam ser realizadas no sentido de identificar amostras canônicas, sendo por ora assegurada a validaçao para um sub-conjunto de estados do tipo.

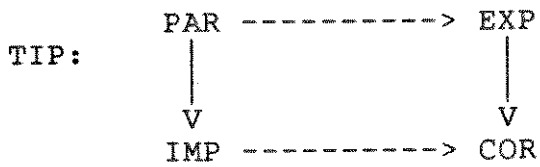
A Linguagem-Pi possui a virtude de nao atrelar sua utilizaçao às especificaçoes algébricas, sendo possível o uso de seus princípios e mesmo de sua estrutura aliada a outras técnicas de especificaçoes formais. Como o objetivo central deste trabalho é a proposiçao de um paradigma de validaçao de CSR's, a técnica formal de traços é aqui aliada à Linguagem-Pi, proporcionando um mecanismo automático de validaçao de componentes de software.

4.4. Um Modelo de Reuso de Componentes

Esta seçao trata do modelo de reuso de componentes de software proposto por este trabalho, utilizando os princípios da Linguagem-Pi comentados em 4.2. Tal modelo explora as "views" de tipo e de objeto. No ANEXO C é fornecido um exemplo de um "Airport Scheduling" utilizando este paradigma.

4.4.1. O Mundo Abstrato: "View" de Tipo

A especificaçao da "view" de tipo TIP=(EXP,IMP,PAR,COR) consiste, assim como na Linguagem-Pi, de interface de exportaçao (EXP), interface de importaçao (IMP) , seçao de parâmetros comuns (PAR), e corpo (COR). O diagrama seguinte ilustra a relaçao entre tais partes.



EXP, IMP e PAR constituem-se de especificaçoes formais utilizando a técnica de Traços [PARN,86], que exprimem propriedades dos domínios de dados e operaçoes. O COR consiste de um conjunto de Traços T para cada operaçao exportada pelo componente. O formalismo associado à especificaçao permite o seu processamento por computador, viabilizando a construçao, execuçao, validaçao, verificaçao, evoluçao e reuso de CSR's de forma eficaz.

A interface de exportaçao da "view" de tipo é responsável pela definiçao da parte visível do CSR, que possibilitará o seu reuso. Ela contém os requerimentos que o componente deve satisfazer, expressos em termos de seu comportamento.

A interface de importaçao contém a especificaçao dos componentes a serem importados para a construçao do CSR em questao. Sao aqui incluídas as assertivas necessárias para o entendimento do comportamento do tipo, no que diz respeito àquelas operaçoes a serem efetivamente importadas. O mesmo vale para a seçao de **parâmetros**, sendo que esta denota partes comuns de importaçao e exportaçao do componente.

O **corpo** define a construçao de cada operaçao exportada pelo CSR em termos das operaçoes da interface de importaçao e das operaçoes da seçao de parâmetros.

4.4.2. O Mundo Concreto: "View" de Objeto

A "view" de objeto OBJ=(EXP,IMP,PAR,COR) assemelha-se em estrutura à "view" de tipo. A "view" de objeto, no entanto, trata da arquitetura concreta do sistema de software, fazendo com que a notaçao de suas partes esteja mais próxima do mundo computacional (mundo concreto).

EXP, IMP e PAR fornecem uma descriçao funcional de suas operaçoes. COR apresenta a construçao das operaçoes de EXP de forma procedural, assemelhando-se à linguagem de programaçao C, como pode ser observado no ANEXO C.

4.5. Concreto x Abstrato: Uma Proposta para Validaçao de Componentes de Software

Nesta seçao é proposto um modelo de validaçao de componentes de software, tendo por base os paradigmas de composiçao de componentes anteriormente considerados neste trabalho.

4.5.1. Principios de Validaçao

"Nao se preocupe com este trabalho com esta papelada de especificaçao. E melhor nós começarmos rapidamente a codificaçao, pois temos pela frente um grande trabalho de depuraçao a fazer."

Quantos projetos começam desta forma e terminam como um fracasso total? Ainda há projetos demais nesta situaçao, mas cada vez mais surgem mais esforços no sentido de revertê-la. Através de um investimento maciço em verificaçao e validaçao de software, tais esforços estao apostando em menores custos de integraçao e teste e em software mais confiável e de mais fácil manutençao [BOEH, 84].

Os termos **verificaçao** e **validaçao** anteriormente mencionados sao definidos no IEEE Standard Glossary of Software Engineering Terminology [IEEE,83] da seguinte forma:

- Verificaçao: O processo de determinar se os produtos de uma dada fase do ciclo de desenvolvimento de software atendem os requisitos estabelecidos durante a fase anterior.

- Validaçao: O processo de avaliaçao de software ao final do processo de desenvolvimento de software, para assegurar coerência com os requerimentos do produto de software.

Os dois termos anteriores podem ainda ser definidos através das seguintes perguntas [BOEH,84]:

- Verificaçao: "O produto está sendo construído corretamente?"

- Validaçao: "O produto correto (desejado) está sendo construído?"

Diferentes processos de verificaçao de software sao amplamente discutidos na literatura [LOND,77] [ROBI,77] [MORR,77] [YEH,77]. No ANEXO D é apresentada um experimento de validaçao formal de software feito com o Software de Recuperaçao de Informaçao - SRI [CAMA,88].

Este trabalho concentra-se na validaçao de Componentes de Software Reusáveis, propondo para tal o uso de conceitos de formalismo e abstraçao de dados, já anteriormente apresentados. Tal proposta de validaçao baseia-se em grande parte no experimento realizado com o SRI (ANEXO D).

A validaçao de um componente de software consiste em mostrar que as propriedades desejáveis descritas pela sua **especificaçao abstrata** sao "respeitadas" em seu **mundo concreto** (implementaçao), de forma semelhante ao descrito em 4.1.

Considerando-se a estrutura da linguagem-pi proposta em 4.5, a especificaçao a ser aqui validada constitui-se da **interface de exportaçao da "view" de tipo** do componente, dado que esta exprime os requerimentos do CSR, i.e., o que este deve fazer. O "programa" a ser testado encontra-se no **corpo da "view" de objeto**, já que é aqui especificada de forma procedural a construçao das operaçoes do componente.

As seçoes seguintes apresentam a forma proposta para as partes das "views" a serem validadas, com exemplos ilustrativos.

4.5.2. Representaçao do Abstrato: A Interface de Exportaçao da "View" de Tipo

O mundo abstrato de um CSR é representado através da interface de Exportaçao da projeçao de Tipo. Esta interface constitui-se da linguagem-pi [WEBE,88] aliada à técnica de Traços [PARN,86], descrita em 4.3. Um exemplo ilustrativo de uma interface de exportaçao da "view" de tipo é apresentado para o componente fila.

O exemplo seguinte apresenta a interface de exportaçao da "view" de tipo de um CSR fila.

```
CSR FILA

descricao { Exemplo utilizando o componente computacional
fila }

view_de_tipo

exportacao

tipo: Fila;

descricao {
   Enqueue insere um elemento no final da fila
   Dequeue deleta um elemento do comeco da fila
   Front exibe o primeiro elemento da fila
   Empty retorna o valor booleano 1 caso a fila esteja
vazio e 0 caso contrario
}

Func_O:
{
      Enqueue          : Fila, str      -> Fila;
      Dequeue          : Fila           -> Fila;
}

Func_V:
{
      Front            : Fila           -> str;
      Empty    : Fila          -> int;

}

legalidade:
{
      L(T.Enqueue(a));
      L(T.Dequeue) == L(T.Front);
}

equivalencia:
{
      T.Enqueue(a).Dequeue==T.Dequeue.Enqueue(a);
}

valor:
{
      V(Empty) == TRUE;
      V(T.Enqueue(a).Empty) == FALSE;
      V(Enqueue(a).Front) == a;
}
```

No ANEXO C é apresentado um exemplo desta interface de um CSR, para um sistema de controle de aeroporto.

4.5.3. Representaçao do Concreto: O Corpo da "View" de Objeto

O Corpo da projeçao de objeto de um componente representa o seu mundo concreto, possuindo uma linguagem próxima do mundo computacional. Desta forma, na seçao de Corpo sao utilizadas construçoes da linguagem C, com uma pequena extensao. A linguagem C permite a implementaçao de componentes de software utilizando-se paradigmas de abtraçao de dados e ocultamento de informaçao. Ela permite ainda que os componentes produzidos sejam **portáveis**, o que é de grande importância para seu reuso, como descrito em 3.2.1. As construçoes de seleçao, iteraçao e atribuiçao desta linguagem de programaçao sao mantidas. Para a chamada às operaçoes dos componentes, a seguinte gramática é utilizada:

```
composiçao: /* nada */
          | funçao '(' lista_parm ')'
          | funçao '(' STRING ',' composiçao
          | funçao '(' composiçao ',' STRING ')'
          | funçao '(' composiçao ')'
          ;

lista_parm: /* nada */
          | STRING
          | STRING ',' lista_parm
          ;

funçao: STRING '.' STRING
      ;
```

Na produçao funçao o primeiro token STRING denota o nome do componente ao qual pertence a operaçao e o segundo token STRING representa a operaçao sendo invocada. Tal extensao permite a identificaçao, de forma clara, do componente de importaçao sendo utilizado na construçao da operaçao a ser exportada pelo componente em questao. A seguir é mostrado um exemplo de tal composiçao, para o

componente fila. No ANEXO C é apresentado outro exemplo - de controle de aeroporto.

view_de_objeto

corpo

descricao { descricao procedimental de fila }

construcao de fila **igual** lista

```
#include      <stdio.h>
#include      <string.h>

#include      "memoria.h"
#include      "fila.h"


Fila  Fil_Create()
{
      return ( (Fila) Lista.Lis_Create());
}

Fila  Fil_Destroy(F)
Fila  F;
{
      return( (Fila) Lista.Lis_Destroy(F) );
}

Fila  Enqueue(F, Item)
Fila  F;
Dado  Item;
{
      Node    pos;

      if (Lista.L_First(F) == (Dado) NULL)
            pos = 1;
      else
            pos = ( Lista.L_End(F) + 1);
      return( (Fila) Lista.L_Insert(F ,Item, pos) );
}
```

```
Fila  Dequeue(F)
Fila  F;
{
      Dado    Daux;
      Node    pos;

      Daux = Lista.L_First(F);
      if (Daux == (Dado) NULL)
        printf("Erro: Fila Vazia!!!\n");
      else {
        return((Fila)Lista.L_Delete(F,Lista.L_Locate(F,Daux)));
      }
}

Dado  Front(F)
Fila  F;
{
      return( Lista.L_First(F) );
}

int Empty(F)
Fila  F;
{
      return( (Lista.L_First(F) == (Dado) NULL) ? (int) -1
: (int) 0 );
}
```

CAPITULO 5. Uma Ferramenta para a Validaçao da Composiçao de Componentes

O **Gerador Automático de Teste (GAT)** é uma ferramenta desenvolvida com o objetivo de introduzir a automaçao no processo de validaçao de componentes de software, tornando viável a utilizaçao deste processo na produçao industrial de sistemas de software.

Dentro do modelo exposto na seçao anterior, o GAT utiliza a linguagem-pi [WEBE,88] juntamente com a técnica de traços [PARN,86] para a validaçao formal de um CSR. Sua estrutura funcional é mostrada na Figura 5.1.

O GAT tem como entradas a especificaçao da composiçao a ser validada, uma amostra de traços canônicos para o componente em questao e uma biblioteca com o código objeto dos CSR's importados ligados. A implementaçao destes deverá ter sido feita na linguagem de programaçao C. A partir daí o teste se dá como descrito por Camargo [CAMA,90].

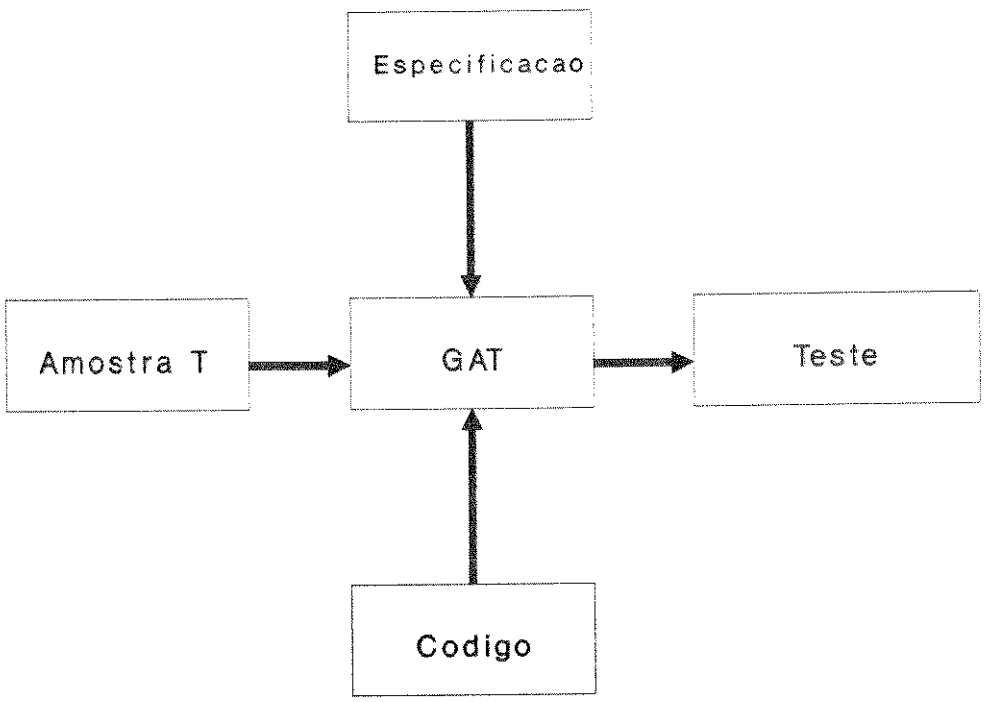


FIG. 5.1. GAT - Estrutura Funcional

5.1. Como o Teste é Efetuado Utilizando-se o GAT

Uma implementaçao de um componente de software será considerada satisfatória caso qualquer propriedade que possa ser deduzida a partir das assertivas de especificaçao também seja uma propriedade da implementaçao do componente.

O teste é feito pela substituiçao das ocorrências de T da especificaçao pela amostra de Traços, de forma análoga à experiência realizada com o SRI (ANEXO D). Como exemplo, podemos ter a assertiva de Equivalência, para o componente Fila (4.5.2):

```
T.Enqueue(a).Dequeue==T.Dequeue.Enqueue(a);
(1)
```

Além disto, a amostra T={T1, T2, T3} é dada:

```
Enqueue(a).Dequeue().Front();     (2)
Front().Enqueue(a).Enqueue(a).Dequeue();
Enqueue(a).Dequeue().Dequeue().First();
```

As ocorrências de T (1) sao, entao, substituídas por T1 (2), resultando:

```
Enqueue(a).Dequeue().Front().Enqueue(a).Dequeue()==
Enqueue(a).Dequeue().Front().Dequeue().Enqueue(a);
(3)
```

A operaçao descrita anteriormente é feita pelo GAT para todas as assertivas de Legalidade, Equivalência e Valor, isto é, a operaçao é repetida para todas as ocorrências de T da especificaçao, utilizando toda a amostra de Traços. As seçoes de Legalidade, Equivalência e Valor sao validadas da seguinte forma:

1. Legalidade: Estas assertivas sao examinadas através de uma simulaçao do estado do componente, para determinada amostra de Traços. Tomando novamente o exemplo da Fila (4.5.2), a primeira assertiva de Legalidade descreve que o operador Enqueue é sempre legal, desde que T seja um traço legal:

```
L(T) -> L(T.Enqueue(a));                    (4)
```

Para a validaçao desta propriedade contra o respectivo código, uma funçao_L é associada a cada operador do componente, que verifica se uma operaçao pode ser legalmente aplicada em determinado instante. Substituindo-se T1 (2) na assertiva (4), temos:

```
L(Enqueue(a).Dequeue().Front()) ->
 L( Enqueue(a).Dequeue().Front().Enqueue(a));
```

O GAT executa, assim, as funçoes L_Enqueue(a), L_Dequeue(), L_Front() e L_Enqueue(a) respectivamente, examinando a legalidade destas por simulaçao. Mensagens de erro sao emitidas caso alguma assertiva seja violada e, neste caso, as assertivas de Equivalência e Valor nao sao testadas.

2. Equivalência: Neste caso o teste é feito pela execuçao direta do código do componente, caso nao tenha sido detectada nenhuma violaçao de Legalidade. Utilizando a assertiva de Equivalência de (3), o GAT executaria o primeiro lado da equivalência, armazenando o estado resultante do componente e, em seguida, executaria o segundo lado da equivalência armazenando novamente o estado resultante. Ambos os resultados seriam entao comparados pelo GAT e, caso nao fossem iguais, uma violaçao a esta assertiva teria ocorrido. Mensagens de erro sao emitidas no caso de violaçao.

3. Valor: Da mesma forma que na Equivalência, esta seçao é validada caso nao haja violaçao de Legalidade. Uma execuçao do código do componente também é aqui feita. Tomando a assertiva de Valor (4.5.2, componente Fila): L(T) -> V(T.Enqueue(a).Front()) == a;, o GAT executaria as operaçoes de T (que é legal), dadas pela amostra de Traço, seguido pela execuçao dos operadores Enqueue e Front. Caso o valor retornado por Front seja a, nao há violaçao a essa assertiva.

5.2. Arquitetura de Implementaçao

O ambiente de implementaçao do GAT é ilustrado na Figura 5.2.

A **Especificaçao** da composiçao define os requerimentos do componente e apresenta a implementaçao a ser testada. No primeiro passo do teste, o **Analisador Sintático da Especificaçao** extrai as informaçoes necessárias à validaçao, e chama o **Gerador de Código** para gerar código necessário para a validaçao das assertivas de equivalência e valores do componente. O **Validador** mostrado na Figura 5.2 contém funçoes para o teste da seçao de Legalidade da especificaçao que devem ser implementadas segundo [CAMA,90].

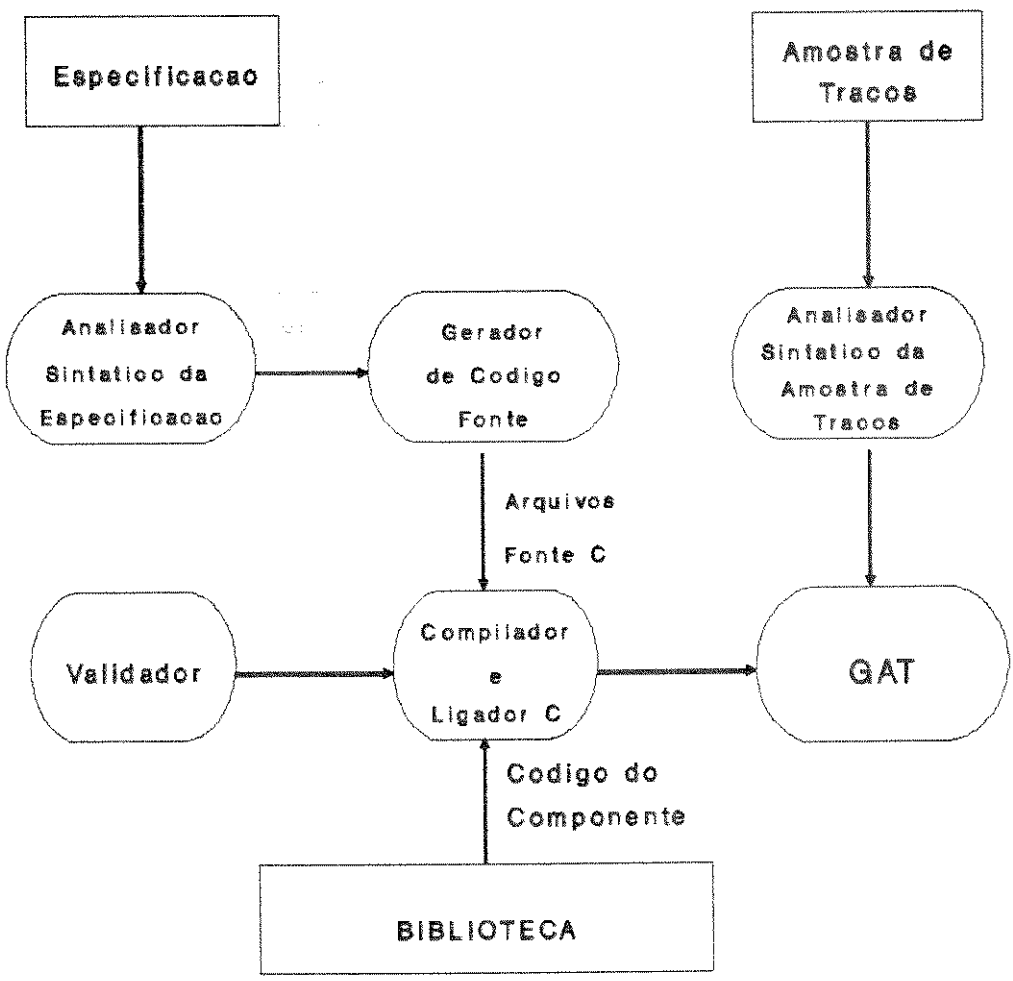


FIG. 5.2. GAT - Arquitetura de Implementacao

Em um segundo passo, código fonte anteriormente gerado **é compilado e linkeditado** juntamente com o Validador e com a **biblioteca**, resultando no **GAT executável**. A biblioteca deve conter o código objeto de todos os componentes importados e das operaçoes exportadas pelo CSR em questao. Uma **Amostra de Traços** serve entao de entrada para o **Analisador Sintático da Amostra de Traços** que alimenta o GAT com dados de entrada. A ferramenta entao realiza o processo de validaçao anteriormente descrito, emitindo mensagens de erro caso quaisquer propriedades especificadas sejam violadas pela implementaçao.

5.3. Formato das Entradas do GAT

5.3.1. Formato da Especificaçao

Para que fosse possivel extrair-se as informaçoes de requisitos necessárias para a validaçao, foi desenvolvido um **analisador sintático de especificaçao**, que reconhece uma linguagem baseada na linguagem-pi [CRAM,90] e na técnica de traços [PARN,86]. A BNF de tal linguagem encontra-se no ANEXO B. A seguir é mostrada uma descriçao sucinta dos elementos desta que formam a interface de exportaçao da "view" de tipo de um CSR.

```
CSR comp_a

descricao geral { ... }

view de tipo

exportacao

Tipo: Tipo_a;

descricao geral { descricao informal dos operadores de
exportacao do componente }

O_Func:
{
   Func_1: parm1, parm2      -> parm3;
   Func_2: parm1             -> parm2;
}
V_Func:
{
   Func_4: parm1             -> parm2;
}
```

```
legalidade:
{
        L(T);
        L(T1) == L(T2);
        L(T1) -> L(T2);
        cond -> traco_l;
}

equivalencia:
{
          T1 == T2;
          T == boolean;
          T != boolean;
          cond -> traço_e;
          L(T) -> T1 == T2;
      }

     valores:
     {
          V(T) op boolean;
          V(T) op número;
          V(T1) == V(T2);
          V(T1) == V(T2) opa atomo;
          V(T1) == atomo opa V(T2);
          cond -> traço_v;
          L(T) -> V(T) op boolean;
          L(T) -> V(T) op numero;
     }

     onde:
          T, T1, T2 : Traços que podem assumir uma ou mais
                      das seguintes formas, unidas por
                      ponto:
                       a
                       b(c,d)
                       e(f)
                       for  a = b, c { T }
          ==   : é igual a, ou
                 é  equivalente  a  (  no  caso  da
                 equivalência);
          ->   : implica;
          =    : atribuição.
          !=   : é diferente de;
          traço_l : traço definido na seção de legalidade;
          traço_e : qualquer um dos traços legais para  a
             equivalência;
          traço_v : qualquer um dos traços legais para  os
             valores;
          a         : uma string qualquer;
          atomo     : string ou numero;
          boolean   : TRUE ou FALSE;
```

```
cond      : V(T) op boolean;
            V(T) op número;
            V(T) op string;
            T == T;
            a op número;
            a op string;

op        : >=;
            <=;
            >;
            <;
            ==;
            !=.
opa       : +;
            -;
            /;
            *;
```

Em 4.5.2 tem-se um exemplo de especificaçao, para o componente Fila, reconhecível pelo GAT. Tal especificaçao deve estar contida em um arquivo com nome na forma:

<nome _do_componente>.t , exemplo: fila.t.

5.3.2. Formato da Amostra de Traços

O GAT possui um **analisador sintático** para reconhecer uma **amostra de Traços**. De forma análoga ao reconhecedor da especificaçao, este foi construído com a ferramenta YACC, cuja BNF é mostrada no ANEXO B. O arquivo de amostras poderia ter a seguinte forma:

```
Enqueue(a).Dequeue().Front();     (1)
Front().Enqueue(a);
Dequeue().Enqueue(a).Enqueue(a);
```

A chamada a um operador, de mesmo nome e número de parâmetros que aquele especificado, é separada por ponto (.) e dois Tracos sao separados por ponto e vírgula (;). Quando da execucao de um traço, o componente é colocado pelo GAT em seu estado inicial.

Dado determinado Traço T1 o GAT irá primeiramente, com base no Validador, avaliar se este é um Traço legal. A partir daí será verificado se sao satisfeitas as assertivas de Equivalência e de Valores do componente.

A questao de técnicas de amostragem tem sido discutida na literatura, especialmente na área de Estatística. Idealmente a escolha de amostras de Traços deveria basear-se em alguma destas técnicas, excluindo a subjetividade desta etapa. Pesquisas futuras certamente serao realizadas nesta área. Neste momento será adotado um método de trabalho informal para a escolha de amostras de teste, comentado a seguir.

Este método tem por base a idéia de tentar incluir na amostra os possíveis casos de erro, por exemplo, da Legalidade. A Seçao 4.5.2 mostra a especificaçao da Fila, que define que a operaçao Dequeue é legal somente caso já tenha sido realizada a operaçao Enqueue. E importante que sejam incluídas nos casos de teste situaçoes onde tenta-se fugir a esta regra (Ex: T3 de (1)) e outras onde esta nao é infringida (Ex: T1 de (1)). Estaremos testando, assim, casos críticos das operaçoes do componente.

5.3.3. Formato do Código

O código do componente a ser validado pelo GAT, assim como os componentes subordinados importados, devem estar escritos na linguagem de programaçao C. Em todos os casos as funçoes implementadas devem possuir o mesmo nome dos operadores e o mesmo número e tipo de parâmetros da especificaçao. Em 4.5.2 o tipo Fila é definido com os operadores Enqueue, Dequeue, Front e Empty. Estes deverao ser os nomes das funçoes eventualmente implementadas. Estas devem estar contidas em um arquivo de nome fila.c, ou seja, o nome do componente (definido em Tipo:) em letras minúsculas com a terminaçao .c.

Associado ao código do componente está o seu desenho, que define a estrutura de dados escolhida para representá-lo. Este deve estar contido em um arquivo .h (ex: fila.h) na linguagem C. A seguir é mostrado um possível desenho para o tipo Fila. A declaraçao das funçoes implementadas também estao incluídas.

```
typedef        struct _Lista    *Fila;

Fila  Fil_Create();
Fila  Fil_Destroy();
Fila  Enqueue();
Fila  Dequeue();
```

```
  str   Fi_Front();
  int   Fi_Empty();
  int   Fi_Print();
/*
//      Final de fila.h */
```

As funçoes que implementam os operadores do componente, assim como dos importados por este, devem utilizar-se de princípios de abstraçao de dados e "information hiding" [PARN,86], acessando somente as informaçoes de seu desenho e escondendo detalhes de implementaçao do resto do sistema. Em 4.5.3 é mostrada uma implementaçao para o tipo Fila.

Associadas ao código dos componentes em questao, devem existir as funçoes xxx_Create e xxx_Destroy, onde xxx sao as três primeiras letras do nome do componente definido na especificaçao (em Tipo:). Estas sao utilizadas pelo GAT para colocar o componente em seu estado inicial a cada execuçao de um Ti (i = 1, 2, ..., n) da amostra de Traços T={T1, T2, ..., Tn}. A funçao xxx_Create deve ser implementada sem parâmetros, e a xxx_Destroy deve ter o componente como parâmetro, como exemplificado em 4.5.2.

5.3.4. Formato do Validador

O Validador é utilizado pelo GAT para validar as assertivas de Legalidade contra o código do componente. Faz-se, para tal, uma simulaçao do comportamento deste, diante da amostra.

Partindo-se da especificaçao por Traços em 4.5.2 para o componente Fila tem-se que:

- é sempre legal a execuçao da funçao Enqueue;

- a operaçao Dequeue só é legal caso uma operaçao Enqueue já estiver sido realizada, ou seja, se houver algum elemento na Fila;

- a legalidade do operador First é análoga à do operador Dequeue.

A avaliaçao de legalidade deste componente em particular pode, portanto, ter por base o número de elementos nele contidos em determinado instante.

O GAT utiliza o tipo 'Simula' no processo de simulaçao, cuja estrutura é definida de acordo com o componente. No exemplo da Fila poderia ter-se um Simula do tipo inteiro, que simplesmente contaria o número de elementos enfileirados em determinado instante. A cada operador do componente será associada uma Funçao_L, que utilizará o tipo Simula para a avaliaçao da Legalidade do componente.

A seguir é mostrado um exemplo de uma possível implementaçao de Funçoes_L para o componente Fila. Tais funçoes retornam o valor 0 caso a operaçao seja legal, ou o número da regra de Legalidade violada, caso contrário. A funçao L_First, por exemplo, retorna o valor 2 caso nao haja nenhum elemento na Fila (traço ilegal) significando que a segunda regra da Legalidade foi infrigida. O valor do tipo Simula é alterado quando conveniente pelas Funçoes_O.

```
  /* Declaracao do tipo Simula e Funcoes_L (lfila.h) */
int     Simula = 0;

int     L_Empty();
int     L_Enqueue();
int     L_Dequeue();
int     L_Front();
int     Fil_Reset();

  /* Final de lpilha.h */

  /* Codigo de Funcoes_L para Fila (lfila.h) */

int     L_Enqueue()
{
        ++Simula_F;
        return(0);
}

int     L_Dequeue()
{
        if (Simula_F) {
                --Simula_F;
                return(0);
        }
        else
                return(2);
}

int     L_Front()
{
        return( (Simula_F) ? 0 : 2 );
}
```

```
int     L_Empty()
{
        return(0);
}

int     Fil_Reset()
{
        Simula_F = 0;
        return(0);
}
```

Chamadas sucessivas às Funçoes_L do componente, a partir de uma amostra de Traços T, possibilitarao a avaliaçao da sua Legalidade. Tais Funçoes devem estar contidas em um arquivo da forma:

l<nome_componente>.c , exemplo: lfila.c

(Obs: nome em letras minúsculas).

A declaraçao do tipo Simula deve estar contida em um arquivo l<nome_componente>.h (exemplo:lfila.h).

Quando da execuçao de um novo T da amostra, o tipo Simula deve ser zerado, voltando a simulaçao do componente ao seu estado inicial. Como este tipo depende das caracteristicas do componente, o GAT necessita da implementaçao de uma funçao xxx_Reset, onde xxx sao as três primeiras letras do nome do componente definido na especificaçao (em Tipo:). No caso do componente Fila, teriamos a funçao Fil_Reset, que deve estar contida juntamente com as Funçoes_L do componente, como ilustrado no exemplo anterior.

5.4. Passos de utilizaçao do GAT

A utilizaçao da ferramenta GAT é resumida nos passos seguintes:

PASSO 1. O GAT gera, a partir da interface de exportaçao da "view" de tipo do componente, arquivos fontes na linguagem de programaçao C ('leg.c', 'leg.h','equ.c', 'equ.h', 'val.c', 'val.h', 'guarda.c', 'setgato.h')., mostrados no ANEXO A para o componente Fila.

PASSO 2. Tais arquivos devem ser compilados e linkeditados juntamente com o arquivo que contém as Funçoes_L do componente sendo exportado, com a gramática de reconhecimento da amostra de Traços, com o código C do componente sendo exportado e com os arquivos que implementam as operaçoes dos componentes subordinados importados.

PASSO 3. O código executável resultante aceita como entrada sequências de Traços T e efetua o processo de validaçao.

O ambiente de teste acima descrito foi projetado da mesma forma que ferramentas do ambiente UNIX, como por exemplo o YACC (Yet Another Compiler-Compiler). Neste, a partir de uma gramática BNF, sao gerados arquivos C que, quando compilados e linkeditados juntamente com as açoes da gramática, geram código executável para o reconhecimento de determinada linguagem.

De forma análoga ao YACC, as etapas de execuçao do GAT podem ser controladas pelo uso do Makefile, outra ferramenta disponível no ambiente UNIX. A seguir ilustramos o uso do Makefile para o GAT.

```
#
#       Makefile para o GAT
#
#       Onde:
#         traco    : Analisador Sintático de Especificaçao
#
#          fila.t: Interface de exportaçao da "view" de tipo
#               do componente Fila
#
#             leg.c, equ.c, val.c exe.c, guarda.c: arquivos
#               gerados pelo Gerador de Código
#
#          lfila.c: Contém as Funcoes_L do componente
#
#             fila.c: Contém os operadores implementados do
#               componente
#
#          gat.o    : Gramática de reconhecimento de amostra
#               de Tracos (Código objeto)
#
#           libcomp.a: Biblioteca que contém o código objeto
#               dos componentes subordinados importados
#
```

```
SRCS = guarda.c exe.c leg.c equ.c val.c lfila.c fila.c
OBJS = gat.o guarda.o exe.o leg.o equ.o val.o lfila.o
fila.o
LIB = libcomp.a
INCL = ../include
CFLAGS = -g -DDEBUG -DULTRIX -I$(INCL)

gat:            $(OBJS)
                cc $(CFLAGS) $(OBJS) $(LIB) -o gat
                #execuçao do PASSO2
exe.c:
                traco < fila.t
                #execuçao do PASSO1

exe.o:                  exe.c

lfila.o:                lfila.c

leg.o:                  leg.c

equ.o:                  equ.c

val.o:                  val.c

guarda.o:       guarda.c

rm:             rm  exe.*  leg.*  equ.*  val.*  guarda.*
setgato.h      #remoçao dos arquivos gerados no PASSO1
```

Os seguintes comandos seriam emitidos para a utilizaçao do GAT com o Makefile acima:

a. 'make' (execuçao dos passos 1 e 2);

b. 'gat < traco.fil' (passo 3), onde traco.fil e' o arquivo que conte'm a amostra de traços para o componente Fila. Pode-se, ainda, simplesmente executar-se 'gat', entrando com a amostra de Traços pelo stdio.

Quando de uma nova geraçao dos arquivos do GAT (comando 'traco < fila.t' do Makefile, PASSO1), é preciso que os arquivos que foram anteriormente gerados sejam removidos. Isto pode ser feito através do Makefile, como mostrado no exemplo anterior. Bastaria, no exemplo, que o comando 'make rm' fosse executado antes da execuçao do PASSO1 do GAT.

6. Conclusoes

O objetivo central deste trabalho foi propor uma metodologia (método, técnica e ferramenta de software) para validaçao de componentes de software, dentro do paradigma de composiçao apresentado.

A primeira atividade neste sentido foi realizada através de um experimento com o Software de Recuperaçao de Informaçao - SRI - [CAMA,88], onde o código do software foi manualmente validado contra a especificaçao formal de Traços [PARN,86] de sua Linguagem de Recuperaçao. O ANEXO D do presente trabalho descreveu este experimento.

A conclusao atingida com o experimento acima foi que o processo de validaçao adotado foi de grande auxílio para encontrar-se erros críticos nao determinados anteriormente por testes convencionais. Além disto concluiu-se que o grande volume de trabalho do processo de validaçao feito manualmente poderia ser automatizado por uma ferramenta de software.

Posteriormente no escopo do Projeto Fábrica de Software [GATT,90a], foi produzida uma metodologia de produçao de componentes onde, a partir dos resultados obtidos com o SRI, foi desenvolvida uma metodologia de validaçao de componentes e implementada uma ferramenta de software para validaçao de componentes primitivos - o Gerador Automático de Teste, GAT. Um trabalho acerca destes dois resultados foi publicado em [CAMA,90].

Tendo em vista estes resultados com componentes de software primitivos, deu-se início ao estudo de uma forma de composiçao de componentes para a produçao de sistemas de software, onde tais componentes funcionariam como "blocos de encaixe". O modelo da Linguagem-Pi [WEBE,88] do projeto ESPRIT foi entao adotado, aliando sua estrutura à técnica de especificaçao formal de Traços, dando continuidade aos trabalhos de validaçao de software. O capítulo 4 deste trabalho descreveu o paradigma de composiçao adotado e a metodologia de validaçao proposta.

Neste contexto este trabalho contribuiu com a utilizaçao de paradigmas de formalismo, abstraçao de dados e reuso em um processo de validaçao de software, por toda sua hierarquia de componentes.

O GAT foi entao modificado para suportar caracteristicas de validaçao automática de componentes nao primitivos. O capitulo 5 apresentou este protótipo desenvolvido em ambiente UNIX, na linguagem de programaçao C, em equipamento Elebra MX-850. Parte do protótipo encontra-se portado para micros do tipo IBM PC/XT e compatíveis em ambiente MS-DOS e para estaçoes de trabalho Sun com sistema operacional SunOs. O restante das atividades associadas'a portabilidade do software devem ser concluídas em breve.

A implementaçao do protótipo de software GAT deverá ter sequência, incluindo a "linkediçao" de tipos genéricos complexos quaisquer à hierarquia de componentes. Além disto, pelo menos duas linhas de pesquisa deverao se iniciar.

A primeira delas diz respeito à amostra de Traços (dados de entrada) necessária para a realizaçao automática da validaçao. Para que esta tenha maior eficácia na certificaçao da qualidade de componentes, é desejável uma amostra que expresse todos os possíveis estados de um Componente Reusável de Software, através de seus operadores-O [PARN,86]. Uma amostra com esta caracteristica é dita uma amostra canônica de traços. Pesquisas futuras deveriam ser realizadas no sentido de determinar tais amostras, para componentes de software quaisquer.

Outra linha de trabalho envolve a descriçao de todos os atributos relevantes de componentes de software, o que é desejável em peças de software reusáveis, segundo o atributo de interpretabilidade [YEH,91]. Neste trabalho foram apresentadas as "views" de tipo e de objeto como representaçao da informaçao associada ao mundo abstrato e concreto do componente, respectivamente. Aspectos de distribuiçao, interconexao e configuraçao do processo de produçao do componente devem ser abordados em pesquisas futuras, possivelmente utilizando-se o conceito de "views" apresentado neste trabalho, para que todos os aspectos relevantes de software possam ser efetivamente descritos, levando estes a adquirir um maior grau de reusabilidade.

7. Referências Bibliográficas

[BERT,77] BERTALANFY L. **Teoria Geral dos Sistemas.** Petropolis, Vozes, 1977.

[BOEH,84] BOEHM, B. W. "Verifying and Validating Software Requirements and Design Specifications." **IEEE Software.** 1(1):75-88, 1984.

[BOOC,83] BOOCH, G. "Object Oriented Design." **Software Engineering with Ada**, 1983.

[CAMA,88] CAMARGO, Fani B. & MACARIO, C. G. N. "**Software de Recuperaçao de Informaçao - SRI. Módulo Linguagem de Recuperaçao** Dissertaçao de Projeto Final de Graduaçao do curso de Processamento de Dados da Universidade de Brasilia. Brasilia, DF, julho, 1988.

[CAMA,90] CAMARGO, Fani B. & MACARIO, C. G. N. "**The Components Production Line.**" II Workshop Internacional - Projeto Fábrica de Software. Campinas, SP, março, 1990.

[CHEN,90] CHENG, F. & NG, P.A. "Diagramming Techniques in CASE". In:Ng, P.A. & YEH, R.T. **Modern Software Engineering; foundations and current perspectives.** New York, Van Nostrand Reinold, 1990. p. 23-52.

[CRAM,90] CRAMER, J. & SCHUMANN, H. **Sintax Description of the Pi-Language with Examples (Version 2.0).** SWT Memo, Universidade de Dortmund, Dortmund, Alemanha, 1990.

[FUJI,89] FUJINO, K. **Concept of Software Factory Engineering.** NEC Research and Development, No.94 p. 103-119, 1989.

[GATT,84] GATTAZ SOBRINHO, F.; FERRARETTO, M.D.; TECHIMA, K.; BRESSAN, G. & ROCHA, L.E. **The Software Plant Project Master Plan,** Rio de Janeiro/RJ, EMBRAPA/SEI-CTI/SERPRO/EMBRATEL, 1985.

[GATT,90a] GATTAZ SOBRINHO, F. "The Model for the Brazilian Software Plant." In:Ng, P.A. & YEH, R.T. **Modern Software Engineering; foundations and current perspectives.** New York, Van Nostrand Reinold, 1990. p. 638-650.

[GATT,90b] GATTAZ SOBRINHO, F. "The Software Plant Project." In: HUNNEKENS, H. & SCHAFER, W. **Summaries of talks of ESF Distinguished Lecturers Series in Software Engineering**. Dortmund, University of Dortmund, 1990. p. 104-108.

[GEHA,86] GEHANI, N. & MCGETTRICK, A.D. **Software Specification Techniques**. Wokingham, Addison-Wesley, 1986. p. vii.

[GUTT,75] GUTTAG, J.V. **The Specification and Application to Programming of Abstract Data Types**. PhD Dissertation, Report No. CSRG-59, Computational Sciences Group, Univ. of Toronto.

[HORO,84] HOROWITZ, E. & MUNSON, J.B. "An Expansive View of Reusable Software." **IEEE Transactions on Software Engineering**, SE10(5):447-487, 1984.

[IEEE,83] **IEEE Standard Glossary of Software Engineering Terminology**. IEEE Std. 729-1983, IEEE-CS order no. 729, Los Alamitos, California, 1983.

[JONE,86] JONES, C.B. "Systematic Program Development." In: GEHANI, N. & MCGETTRICK, A.D. **Software Specification Techniques**. Wokingham, Addison-Wesley, 1986. p. 89-108.

[KAMI,90] KAMIJO, F. "Japan's Sigma Project." In:Ng, P.A. & YEH, R.T. **Modern Software Engineering; foundations and current perspectives**. New York, Van Nostrand Reinold, 1990. p. 603-612.

[KERN,84] KERNIGHAN, B.W. "The UNIX System and Software Reusability." **IEEE Transactions on Software Engineering**, SE10(5):513-518, 1984.

[LISK,75] LISKOV, B.H. & ZILLES, S.N. Specification Techniques for Data Abstractions. **IEEE Transactions on Software Engineering**, 1:7-19, 1975.

[LISK,86] LISKOV, B.H. & BERZINS, V. "An Appraisal of Program Specifications." In: GEHANI, N. & MCGETTRICK, A.D. **Software Specification Techniques**. Wokingham, Addison-Wesley, 1986. p. 3-23.

[LIVT,84] LITVINTCHOUK, S.D. & MATSUMOTO, A.S. "Design of Ada Systems Yelding Reusable Components." **IEEE Transactions on Software Engineering**, SE10(5):447-487, 1984.

[LOND,77] LONDON, R. L. "Perspectives on Program Verification" In: YEH, R. T. **Current Trends in Programming Methodology** Volume II - Program Validation. New Jersey, Prentice-Hall Inc., 1977. p. 151-172.

[MANO,82] MANO, H. **"Computer Systems Architecture."** Englewood Cliffs, Prentice-Hall, 1982.

[MITT,90] MITTTERMEIR, R.T. & ROSSAK, W. "Reusability." In: Ng, P.A. & YEH, R.T. **Modern Software Engineering; foundations and current perspectives**. New York, Van Nostrand Reinold, 1990. p. 205-235.

[MORR,77] MORRIS, J. H. "Program Verification by Subgoal Induction" In: YEH, R. T. **Current Trends in Programming Methodology** Volume II - Program Validation. New Jersey, Prentice-Hall, Inc., 1977. p. 197-227.

[MOUR,90] MOURA, M.F.; CARNASSALE, M.; EVANGELISTA, S.R.M. & TERNES, S. "The Product Evolution Line: A Model for Preparing Software for Changes." **Proceedings of the Second International Workshop on the Brazilian Software Plant Project**, Campinas, 1990. p. 31-38.

[NEIG,84] NEIGHBOURS, James, M. - **"The Draco Approah to Constructing Software from Reusable Components"** - IEEE Transactions on Software Engineering - SE10(5):564-574, 1984.

[PARN,86] PARNAS, D.L. "A Technique for Software Specification with Modules." In: GEHANI, N. & MCGETTRICK, A.D. **Software Specification Techniques**. Wokingham, Addison-Wesley, 1986. p. 75-88.

[RAMA,77] RAMAMOORTHY, C. V. & SO, H. H. "Survey of Principles and Techniques of Software Requirements and Specification." In: YEH, R. T. **Software Engineering Techniques** Infotech International Limited, 1977. p. 265-318.

[ROBI,77] ROBINSON, L. "Proff Techniques for Hierarchially Structured Programs" In: YEH, R. T. **Current Trends in Programming Methodology** Volume II - Program Validation. New Jersey, Prentice-Hall, Inc., 1977. p. 173-197.

[ROUS] ROUSSOPOULOS, N. & CHU, B. **Software Base: An integrated environment for managing reusable software. s.n.t.**

[RUOT,90] RUOTOLO, T.F. "**Surround Re-Engineering; Principles and Fundamentals.**" Computrol, 1990.

[SCHU,90] SCHULTZ, S. "**Rapid Iteractive Production Prototyping Guide**", Dupont Information Engineering Associates, 1990.

[STAN,84] STANDISH, T.A. "An Essay on Software Reuse." **IEEE Transactions on Software Engineering**, SE10(5):494-497, 1984.

[TANI,91] TANIK, M. M. "**Software Plant Project - Preliminary Findings and Evaluation**" Relatório Interno - Projeto Fábrica de Software - CTI/EMBRAPA/Banco do Brasil, Campinas, SP, 1991.

[WEBE,88] WEBER, H. & EHRIG, H. "Specification of Concurrently Executable Modules and Distributed Modular Systems" In: **Proceedings of the Workshop on the Future Trends of Distributed Computing Systems in the 1990s**, Hong Kong, 1988.

[WEBE,91] WEBER, H. "The Integration of Reusable Software Components" **Journal of Systems Integration**, 1(1):55-80, July, 1991.

[YEH,77] YEH, R. T. "Verification of Programs by Predicate Transformation" In: YEH, R. T. **Current Trends in Programming Methodology** Volume II - Program Validation. New Jersey, Prentice-Hall, Inc., 1977. p. 228-247.

[YEH,90] YEH, R.T. "An alternative paradigm for software evolution." In: Ng, P.A. & YEH, R.T. **Modern Software Engineering; foundations and current perspectives**. New York, Van Nostrand Reinold, 1990. p. 7-22.

[YEH,91] YEH, R.T. "System development as a Wicked Problem." To appear in the **International Journal of Software Engineering and Knowledge Engineering**, 1991.

[ZELK,84] ZELKOWITZ, M.V.; YEH,R.T.; HAMLET, J.D. & BASILI, V.B. "Software Engineering Practices in the US and Japan." **IEEE Computer**, 18(9):57-66, 1984

ANEXO A. Arquivos gerados pelo GAT para o componente Fila

```
/* Arquivo batch criado para execuçao do gat */

make rm
make
gat < traco.fila
```

```
/* Arquivo contendo as mensagens advindas do passos de
execuçao do GAT */

rm -f exe.* leg.* equ.* val.* setgato.h guarda.* setgato.h res
analyser < fila.t
/bin/cc -g -DDEBUG -DULTRIX -I../../../include -c exe.c
/bin/cc -g -DDEBUG -DULTRIX -I../../../include -c leg.c
/bin/cc -g -DDEBUG -DULTRIX -I../../../include -c equ.c
/bin/cc -g -DDEBUG -DULTRIX -I../../../include -c val.c
/bin/cc -g -DDEBUG -DULTRIX -I../../../include -c guarda.c
cc -g -DDEBUG -DULTRIX -I../../../include exe.o leg.o equ.o val.o
guarda.o gat.o fila.o lfila.o /usr/users/LPPC/traco/grammar/lib/liblpc.a
-o gat
[0] Traco 1
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[1] Traco 2
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[2] Traco 3
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[3] Traco 4
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[4] Traco 5
Testando a Legalidade ...
Fi_Dequeue (Operador numero 1) nao legal.
[5] Traco 6
Testando a Legalidade ...
Fi_Front (Operador numero 1) nao legal.
[6] Traco 7
Testando a Legalidade ...
Fi_Dequeue (Operador numero 3) nao legal.
```

```
[7] Traco 8
Testando a Legalidade ...
Fi_Front (Operador numero 3) nao legal.
[8] Traco 9
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[9] Traco 10
Testando a Legalidade ...
Testando a Equivalencia ...
Testando os Valores ...
[10]
```

```
  /* Arquivo contendo uma amostra de traços para o
  componente fila */

Fi_Enqueue(qw).Fi_Enqueue(abd).Fi_Front();
Fi_Enqueue(a).Fi_Dequeue().Fi_Enqueue(s).Fi_Enqueue(s).Fi_Front().Fi_Emp
Fi_Enqueue(das).Fi_Dequeue().Fi_Enqueue(aa).Fi_Empty().Fi_Front();
Fi_Enqueue(jk).Fi_Dequeue().Fi_Enqueue(s).Fi_Front().Fi_Enqueue(abd).Fi_
Fi_Dequeue();
Fi_Front();
Fi_Enqueue(sa).Fi_Dequeue().Fi_Dequeue();
Fi_Enqueue(sa).Fi_Dequeue().Fi_Front();
Fi_Enqueue(uy);
Fi_Empty().Fi_Enqueue();
```

```
/*
//
//      Arquivo para execucao dos operadores do componente
//      (exe.c)
*/
#include        <stdio.h>
#include        <ctype.h>
#include        <string.h>

#include        "memoria.h"

#ifndef __FILA
#include        "fila.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

#ifndef __PILHA
#include        "pilha.h"
#endif

#ifndef __LISTA
#include        "lista.h"
#endif

#define CONST   32

extern  _Gato   Gato[];
int     i;
char    *buf_gd;

extern  Pilha   T_Stack;
extern  Lista   L_Traco;
int     main()
{
        extern  Fila    tt_Tipo;
        int     trno = 0;

        T_Stack = Pil_Create();
        L_Traco = Lis_Create();

        buf_gd = (char *) M_Aloca(128, sizeof(char), __EXE);

        do {
                printf("[%d] ",trno++);
        } while (yyparse());
}
```

```
/* Chama a funcao a ser executada */
int     tt_Exec(strg)
char    *strg;
{
        for (i = 0; Gato[i].Nome; ++i) {
                if (!strcmp(Gato[i].Nome, strg)) {
                        return((*Gato[i].Inter) ());
                }
        }
        return (0);
}

int     tt_Fi_Enqueue()
{
        str     D0;

        D0 = (str) P_Top(T_Stack);
        T_Stack = P_Pop(T_Stack);

        tt_Tipo = Fi_Enqueue(tt_Tipo, D0);
        return (1);
}
int     tt_Fi_Dequeue()
{

        tt_Tipo = Fi_Dequeue(tt_Tipo);
        return (1);
}
int     tt_Fi_Front()
{
        str     Ret;

        Ret = (str) Fi_Front(tt_Tipo);
        return ( (int) Ret);
}

int     tt_Fi_Empty()
{
        int     Ret;

        Ret = (int) Fi_Empty(tt_Tipo);
        return ( (int) Ret);
}

void    tt_Create()
{
        tt_Tipo = Fil_Create();
}
```

```
void    tt_Destroy()
{
        tt_Tipo = Fil_Destroy(tt_Tipo);
}

/*
//
//      Arquivo para teste da legalidade
//      (leg.c)
*/
#include        <stdio.h>
#include        <ctype.h>
#include        <string.h>

#ifndef __GUARDA
#include        "guarda.h"
#endif


#ifndef __LEG
#include        "leg.h"
#endif

#ifndef __PILHA
#include        "pilha.h"
#endif

#ifndef __LISTA
#include        "lista.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

#include        "lfila.h"

extern  _Gato   Gato[];
extern  Pilha   T_Stack;
extern  Lista   L_Traco;

extern  int     numfunc;
extern  int     i;
```

```
/* Chama as funcoes L que testam a legalidade */
int     tt_Leg(strg)
char    *strg;
{
        int     Ret;

        if (strg) {
                for (i = 0; Gato[i].Nome; ++i) {
                        if (!strcmp(Gato[i].Nome, strg)) {
                                return((*Gato[i].Leg) ());
                        }
                }
        }
        return (-1);
}

/* Chama a funcao tt_Leg para cada operador do traco a ser testado */
int     tt_Legal(L_Traco)
Lista   L_Traco;
{
        Traco   T;
        int     Ret;
        int     pos, j, k;
        int     fl_eq;
        int     oper;

        oper = 0;

        fl_eq = 0;

        printf("Testando a Legalidade ...\n");

        for (pos = 1; pos <= numfunc; ++pos) {
                T = (Traco) L_Retrieve(L_Traco, pos);
                ++oper;
                if (T_Nome(T)) {
                        for (i = 0; Gato[i].Nome; ++i) {
                                if (!strcmp(Gato[i].Nome, T_Nome(T)))
                                        break;
                        }
                        tt_EmpPar(T,i);
                }
                Ret = tt_Leg(T_Nome(T));
                if (Ret > 0 ) {
                        printf("%s (Operador numero %d) nao legal. \n",
                         Gato[i].Nome, oper);
                        fl_eq = 1;
                        break;
                }
```

```
                else    if (Ret == -1) {
                        printf("Operador (%d) Invalido \n", oper);
                        fl_eq = 1;
                        break;
                }
        }
        return((fl_eq) ? 0 : 1);
}

int     tt_L_Fi_Enqueue()
{
int     Ret;

        str     D0;

        D0 = (str) P_Top(T_Stack);
        T_Stack = P_Pop(T_Stack);
        Ret = L_Fi_Enqueue( D0);
        return (Ret);
}
int     tt_L_Fi_Dequeue()
{
int     Ret;

        Ret = L_Fi_Dequeue();
        return (Ret);
}
int     tt_L_Fi_Front()
{
int     Ret;

        Ret = L_Fi_Front();
        return (Ret);
}
int     tt_L_Fi_Empty()
{
int     Ret;

        Ret = L_Fi_Empty();
        return (Ret);
}
/* Reseta um componente para sua proxima execucao */
void    Ini_Sim()
{
        Fil_Reset();
}
```

```
/*
//
//      Arquivo para teste da equivalencia
//      (equ.c)
*/
#include        <stdio.h>
#include        <ctype.h>
#include        <string.h>

#include        <math.h>

#ifdef  MSDOS
#include        <process.h>
#endif

#include        "guarda.h"
#ifndef __PILHA
#include        "pilha.h"
#endif

#ifndef __LISTA
#include        "lista.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

#ifndef __LEG
#include        "leg.h"
#endif

#ifndef __VAL
#include        "val.h"
#endif

#ifndef __EQU
#include        "equ.h"
#endif

extern  int     numfunc;
extern  _Gato   Gato[];
extern  Pilha   T_Stack;
extern  Lista   L_Traco;
extern  GVar    Table_var[];
extern  Fila    tt_Tipo;
```

```
int     tt_Eq_1(L_Traco, nval)
Lista   L_Traco;
int     nval;
{

        int     RetP1, RetP2;
        int     Ret, Cp1, Cp2;
        char    *temp;

        RetP1 = 0;
        RetP2 = 0;
        Ret = Cp1 = Cp2 = 0;

        tt_TrataT(L_Traco, nval);
        {
                if ((temp = V_Valor(nval, "a")) == (char * ) NULL) {
                        temp = M_Aloca(128, sizeof(char), __EQU);
                        strcpy(temp,"a");
                }
                T_Stack = P_Push(T_Stack, (char *) temp);
        }
        tt_Exec("Fi_Enqueue");
        tt_Exec("Fi_Dequeue");
        RetP1 = Fil_Print(tt_Tipo, 1);
        tt_ResetT();

        tt_TrataT(L_Traco, nval);
        tt_Exec("Fi_Dequeue");
        {
                if ((temp = V_Valor(nval, "a")) == (char * ) NULL) {
                        temp = M_Aloca(128, sizeof(char), __EQU);
                        strcpy(temp,"a");
                }
                T_Stack = P_Push(T_Stack, (char *) temp);
        }
        tt_Exec("Fi_Enqueue");
        RetP2 = Fil_Print(tt_Tipo, 2);
        tt_ResetT();

        if (RetP1 == RetP2)
                Ret = RetP1? tt_CompT() : 1;
        else
                Ret = 0;

        return((Ret) ? 1 : 0);
}
```

```
/* Chama as funcoes para teste da equivalencia */
void    tt_Equiv(L_Traco, nval)
Lista   L_Traco;
int     nval;
{
        int     Ret = 0;

        printf("Testando a Equivalencia ...\n");

        if (L_Traco != (Lista) NULL) {
                if ( !(Ret = tt_Eq_1(L_Traco, nval)) )
                        printf("Regra de Equivalencia numero 1 violada\n
                else if ( Ret == 2 )
                        printf("Regra de Equivalencia numero 1 nao testa
                         - Condicao nao satisfeita\n");

        }
}

/* Executa os operadores de um traco */
void    tt_TrataT(L_Traco)
Lista   L_Traco;
{
        Traco   T;
        int     pos;
        int     j, i, k;

        for (pos=1; pos <= numfunc; ++pos) {
                T = (Traco) L_Retrieve(L_Traco, pos);
                for(i=0; Gato[i].Nome; ++i) {
                        if (!strcmp(Gato[i].Nome, T_Nome(T)))
                                break;
                }
                tt_EmpPar(T,i);
                tt_Exec(T_Nome(T));
        }
}

/* Executa um traco para o teste da legalidade */
int     tt_TrataL(L_Traco)
Lista   L_Traco;
{
        Traco   T;
        int     pos, Ret;
        int     j, i, k;

        Ini_Sim();
```

```
        for (pos=1; pos <= numfunc; ++pos) {
                T = (Traco) L_Retrieve(L_Traco, pos);
                for(i=0; Gato[i].Nome; ++i) {
                        if (!strcmp(Gato[i].Nome, T_Nome(T)))
                                break;
                }
                tt_EmpPar(T,i);
                Ret = tt_Leg(T_Nome(T));
        }
        return(Ret);
}
/* Declara os parametros da funcao, empilhando-os */
void    tt_EmpPar(T,i)
Traco   T;
int     i;
{
        int     j, k;

        for(i=0; Gato[i].Nome; ++i) {
                if (!strcmp(Gato[i].Nome, T_Nome(T)))
                        break;
        }
        for(j=0, k=Gato[i].Par - 2; j < Gato[i].Par - 1; ++j, --k) {
                if ( !(strcmp(Gato[i].TPar[j], "char")) ) {
                        char    *Parm;
                        Parm = (char *) M_Aloca(1, sizeof(char), __EQU);
                        *Parm = *(T_Parm(T,k));
                        T_Stack = P_Push(T_Stack, (char *) Parm);
                }
                else if ( !(strcmp(Gato[i].TPar[j], "int")) ) {
                        int     *Parm;
                        Parm = (int *) M_Aloca(1, sizeof(int), __EQU);
                        *Parm = atoi(T_Parm(T,k));
                        T_Stack = P_Push(T_Stack, (char *) Parm);
                }
                else if ( !(strcmp(Gato[i].TPar[j], "float")) ) {
                        float   *Parm;
                        Parm = (float *) M_Aloca(1, sizeof(float), __EQU
                        *Parm = atof(T_Parm(T,k));
                        T_Stack = P_Push(T_Stack, (char *) Parm);
                }
                else if ( !(strcmp(Gato[i].TPar[j], "double")) ) {
                        double  *Parm;
                        Parm = (double *) M_Aloca(1, sizeof(double), __E
                        *Parm = atol(T_Parm(T,k));
                        T_Stack = P_Push(T_Stack, (char *) Parm);
                }
                else
                        T_Stack = P_Push(T_Stack, (char *) T_Parm(T,k));
        }
}
```

```
/* Compara os resultados de dois tracos armazenados em arquivo */
int     tt_CompT()
{
        FILE    *fd;
        int     Ret;
        int     i;

        system("diff comp1 comp2 > res");
        fd = fopen("res", "r+");

        Ret = fgetc(fd);
        if (Ret == -1) {
                i = feof(fd);
                if (i) {
#ifdef  MSDOS
                        system("rm comp*");
#else
                        system("rm comp*");
#endif
                        return(1);
                }
                else {
                        printf("Erro na abertura de arquivo (equ.c)\n");
#ifdef  MSDOS
                        system("rm comp*");
#else
                        system("rm comp*");
#endif
                        return(0);
                }
        }
        else {
#ifdef  MSDOS
                        system("rm comp*");
#else
                        system("rm comp*");
#endif
                return(0);
        }
}

/* Reseta um componente para sua execucao */
void    tt_ResetT()
{
        if ( tt_Tipo != NULL )
                tt_Destroy();
        tt_Create();
}
```

```
/*
//
//      Arquivo para teste do valor
//      (val.c)
*/
#include        <stdio.h>
#include        <ctype.h>
#include        <string.h>

#include        "guarda.h"
#ifndef __PILHA
#include        "pilha.h"
#endif

#ifndef __LISTA
#include        "lista.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

#ifndef __LEG
#include        "leg.h"
#endif

#ifndef __EQU
#include        "equ.h"
#endif

#ifndef __VAL
#include        "val.h"
#endif

extern  int     numfunc;
extern  _Gato   Gato[];
extern  Pilha   T_Stack;
extern  Lista   L_Traco;
extern  GVar    Table_var[];
extern  Fila    tt_Tipo;

int     tt_Va_1(L_Traco, nval)
Lista   L_Traco;
int     nval;
{
        int     Ret, Cp1, Cp2;
        char    *temp;
        Ret = Cp1 = Cp2 = 0;
```

```
        {
        int     Ret1, Ret2;
        Ret1 = (int) tt_Exec("Fi_Empty");
        tt_ResetT();

        Ret2 = 255;
        if( Ret1 == Ret2 )
                return(1);
        }
}

int     tt_Va_2(L_Traco, nval)
Lista   L_Traco;
int     nval;
{
        int     Ret, Cp1, Cp2;
        char    *temp;
        Ret = Cp1 = Cp2 = 0;

        tt_TrataT(L_Traco, nval);
        {
                if ((temp = V_Valor(nval, "a")) == (char * ) NULL) {
                        temp = M_Aloca(128, sizeof(char), __EQU);
                        strcpy(temp,"a");
                }
                T_Stack = P_Push(T_Stack, (char *) temp);
        }
        tt_Exec("Fi_Enqueue");
        {
        int     Ret1, Ret2;
        Ret1 = (int) tt_Exec("Fi_Empty");
        tt_ResetT();

        Ret2 = 0;
        if( Ret1 == Ret2 )
                return(1);
        }
}

int     tt_Va_3(L_Traco, nval)
Lista   L_Traco;
int     nval;
{
        int     Ret, Cp1, Cp2;
        char    *temp;
        Ret = Cp1 = Cp2 = 0;
```

```
        {
                if ((temp = V_Valor(nval, "a")) == (char * ) NULL) {
                        temp = M_Aloca(128, sizeof(char), __EQU);
                        strcpy(temp,"a");
                }
                T_Stack = P_Push(T_Stack, (char *) temp);
        }
        tt_Exec("Fi_Enqueue");
        {
        str     Ret1, Ret2;
        Ret1 = M_Aloca(128, sizeof(char), __VAL);
        Ret2 = M_Aloca(128, sizeof(char), __VAL);

        Ret1 = (str) tt_Exec("Fi_Front");
        tt_ResetT();

        if ((temp = V_Valor(nval, "a")) != (char * ) NULL)
                strcpy (Ret2, temp);
        else
                strcpy (Ret2, "a");
        if ( !(strcmp( Ret1, Ret2)) )
                return(1);
        }
}

/* Chama as funcoes para teste dos valores */
void    tt_Valor(L_Traco, nval)
Lista   L_Traco;
int     nval;
{
        int     Ret = 0;

        printf("Testando os Valores ...\n");
        if (L_Traco != (Lista) NULL) {
                if ( !(Ret = tt_Va_1(L_Traco, nval)) )
                        printf("Regra de Valor numero 1 violada\n");
                else if ( Ret == 2 )
                        printf("Regra de Valor numero 1 nao testada
                         - Condicao nao satisfeita\n");
                if ( !(Ret = tt_Va_2(L_Traco, nval)) )
                        printf("Regra de Valor numero 2 violada\n");
                else if ( Ret == 2 )
                        printf("Regra de Valor numero 2 nao testada
                         - Condicao nao satisfeita\n");
                if ( !(Ret = tt_Va_3(L_Traco, nval)) )
                        printf("Regra de Valor numero 3 violada\n");
                else if ( Ret == 2 )
                        printf("Regra de Valor numero 3 nao testada
                         - Condicao nao satisfeita\n");
        }
}
```

```
/*
//
//      Arquivo que armazena o traco a ser testado
//      (guarda.c)
*/
#include        <stdio.h>
#include        <string.h>

#ifndef __GUARDA
#include        "guarda.h"

#endif

#ifndef __LISTA
#include        "lista.h"

#endif

#ifndef __PILHA
#include        "pilha.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

#ifndef __LEG
#include        "leg.h"
#endif

#include        "setgato.h"

#include        "erro.h"

extern  int     numfunc;
extern  _Gato   Gato[];
extern  Pilha   T_Stack;
extern  Lista   L_Traco;
extern  GVar    Table_var[];

/* Armazena o traco a ser testado */
void    T_Guarda(strg)
char    *strg;
{
        Traco   T;
        char    *parm;
        Dado    elemento;
        int     numpar;
        int     i, j;
```

```
        T = (Traco) M_Aloca(1, sizeof(struct _Traco), __GUARDA);

        for ( i = 0; Gato[i].Nome; ++i) {
                if ( !strcmp(Gato[i].Nome, strg) ) {
                        if ( !T->Nome )
                                T->Nome = (char *) M_Aloca(128, sizeof(c
                                        __GUARDA);
                        T->Nome = strcpy(T->Nome, strg);
                        numpar = Gato[i].Par - 1;
                        T->Parm = (char **) M_Aloca(numpar, sizeof(char
                                        __GUARDA);

                        for ( j = 0; j < numpar; ++j) {
                                if ( !T->Parm[j] )
                                        T->Parm[j] = (char *) M_Aloca(12
                                        sizeof(char), __GUARDA);
                                parm = P_Top(T_Stack);
                                strcpy(T->Parm[j], parm);
                                T_Stack = P_Pop(T_Stack);
                        }
                        break;
                }
        }

        elemento = (Dado) T;
        L_Traco = L_Insert(L_Traco, elemento, numfunc);
}

/* Retorna o nome da funcao armazenda */
char    *T_Nome(T)
Traco   T;
{
        return(T->Nome);
}

/* Retorna um parametro da funcao armazenda */
char    *T_Parm(T, i)
Traco   T;
int     i;
{
        return(T->Parm[i]);
}

/* Armazena o nome da variavel na tabela */
void    V_SetNom(nvar, strg)
int     nvar;
char    *strg;
{
        if ( !Table_var[nvar] )
                Table_var[nvar] = ( GVar ) M_Aloca(1, sizeof(struct _GVa
                __GUARDA);
```

```
        Table_var[nvar]->Nome = (char *) M_Aloca(128, sizeof(char),
            __GUARDA);
        strcpy(Table_var[nvar]->Nome, strg);
}

/* Armazena o valor da variavel na tabela */
void    V_SetVal(nvar, nval, strg)
int     nval, nvar;
char    *strg;
{
        Table_var[nvar]->Valor[nval] = M_Aloca(128, sizeof(char), __GUAR
        strcpy(Table_var[nvar]->Valor[nval], strg);
}

/* Retorna o valor da variavel */
char    *V_Valor(nval, var)
int     nval;
char    *var;
{
        int     i;

        for ( i = 0; Table_var[i]; ++i)
                if( !strcmp(Table_var[i]->Nome, var) )
                        return(Table_var[i]->Valor[nval]);

        return((char *) NULL);
}

/*
//
//      Arquivo que armazena o traco a ser testado - definicao das funco
//      (guarda.h)
*/
#include        <stdio.h>

#ifndef __ATOMIC
#include        "atomic.h"
#endif

#ifndef __MEMORIA
#include        "memoria.h"
#endif

#ifndef __LISTA
#include        "lista.h"
#endif

#ifndef __GUARDA
#define __GUARDA        ("GUARDA")
#endif
#include        "erro.h"
```

```
struct  _Traco {
        char    *Nome;
        char    **Parm;
};

struct  _GVar {
        char    *Nome;
        char    *Valor[10];
};

typedef struct  _Traco  *Traco;
typedef struct  _GVar   *GVar;

#ifdef  MSDOS
void    T_Guarda(char *);
char    *T_Nome(Traco);
char    *T_Parm(Traco, int);
void    V_SetNom(int, char *);
void    V_SetVal(int, int, char *);
char    *V_Valor(int, char *);
#else
void    T_Guarda();
char    *T_Nome();
char    *T_Parm();
void    V_SetNom();
void    V_SetVal();
char    *V_Valor();
#endif

/*
//
//      Arquivo que contem informacoes sobre cada operador do componente
//      (setgato.h)
*/
#include        <stdio.h>
#ifndef __ATOMIC
#include        "atomic.h"
#endif

#ifndef __EXE
#include        "exe.h"
#endif

_Gato   Gato[] = {
"Fi_Enqueue",   2,      "str",  "Fila", 0,      (Funcao) Fi_Enqueue,
tt_Fi_Enqueue,  tt_L_Fi_Enqueue,
"Fi_Dequeue",   1,      "Fila"," ",     0,      (Funcao) Fi_Dequeue,
tt_Fi_Dequeue,  tt_L_Fi_Dequeue,
"Fi_Front",     1,      "str"," ",      1,      (Funcao) Fi_Front,
tt_Fi_Front,    tt_L_Fi_Front,
"Fi_Empty",     1,      "int"," ",      1,      (Funcao) Fi_Empty,
```

```
tt_Fi_Empty,    tt_L_Fi_Empty,
0,              0,      0,      0,      0,      0,
};
```

ANEXO B. BNF do analisador sintático da interface de exportaçao da "view" de tipo de um CSR

```
traco    : sintaxe semantica
         | traco error
         ;

sintaxe  : tipo  operadores
         ;

semantica: legalidade equivalencia valores
         ;

tipo     : TIPO ':' atomo ';'
         ;

operadores: oper
         | operadores oper
         ;

oper     : FUNC_O ':' parametros
         | FUNC_V ':' parametros
         ;

parametros: funcao
         | lst_de_par
         ;

lst_de_par: '{' lst_de_fncts '}'
         ;

lst_de_fncts:  funcao
         | lst_de_fncts ';' funcao
         ;

legalidade: LEGAL ':' decl_l
         ;

equivalencia: EQUIV ':' decl_e
         ;

valores: VALOR ':' decl_v
         ;

decl_l   : traco_l ';'
         | '{' lst_traco_l '}'
         ;
```

```
decl_e  : traco_e ';'
        | '{' lst_traco_e '}'
        ;

decl_v  : traco_v ';'
        | '{' lst_traco_v '}'
        ;

lst_traco_l: traco_l
        | lst_traco_l ';' traco_l
        ;

lst_traco_e: traco_e
        | lst_traco_e ';' traco_e
        ;

lst_traco_v: traco_v
        | lst_traco_v ';' traco_v
        ;

traco_l : /* nada */
        | expr_l
        | expr_l IMPL expr_l
        | expr_l IGUAL traco_l
        | lst_cond IMPL traco_l
        | atomo ASSGN traco_l
        ;

traco_e : /* nada */
        | expr_ig
        | lst_expr IGUAL bool
        | lst_expr DIF bool
        | lst_cond IMPL traco_e
        | atomo ASSGN traco_e
        ;

traco_v : /* nada */
        | expr_v
        | expr_val IGUAL expr_val
        | expr_val IGUAL expr_val OPAINT atomo
        | expr_val IGUAL atomo OPAINT expr_val
        | lst_cond IMPL traco_v
        | atomo ASSGN traco_v
        ;

lst_cond: cond
        | lst_cond OPB cond
        ;
```

```
cond     : expr_v
         | expr_ig
         | atomo OPINT expr_bool
         ;

expr_v:    expr_val IGUAL expr_bool
         | expr_val DIF expr_bool
         | expr_val OPINT atomo
         ;

expr_ig  : lst_expr IGUAL lst_expr
         ;

expr_val: VAL lst_expr ')'
         ;

expr_l   : LEG lst_expr ')'
         ;

lst_expr: expr
         | lst_expr '.' expr
         ;

funcao   : /* nada */
         | atomo ':' lst_atomo  IMPL atomo
         ;

lst_atomo: atomo
         | lst_atomo ',' atomo
         ;

expr_bool: atomo
         | bool
         ;

expr     : atomo
         | atomo '(' lst_atomo ')'
         | FOR atomo ASSGN atomo ',' atomo ']' '{' lst_expr '}'
         ;

atomo    : STRING
         | NUMBER
         | MENOS atomo %prec MENOS
         ;

bool     : VERD
         | FALSO
         | NULO
         ;
```

ANEXO C. Um Exemplo do Modelo de Composiçao de Componentes - Sistema de Controle de Aeroporto

Este exemplo é apresentado em [CRAM,90] utilizando-se especificaçoes algébricas com a linguagem-pi. Neste trabalho o mesmo é apresentado tendo em vista o paradigma aqui proposto.

O Sistema de Controle de Aeroporto é responsável pela gerência dos vôos e avioes de um aeroporto. Por uma questao de simplicidade, esta tarefa é reduzida à gerência da programaçao de vôos, que consiste do número do vôo, horário de origem e destino para todos os vôos, e da programaçao de avioes, que contém número do vôo número e tipo dos assentos para todos os avioes e a relaçao entre vôos e avioes.

A arquitetura resultante consiste basicamente de três componentes: programaçao de vôo, programaçao de aviao e controle aéreo. Os dois primeiros consistem de listas de tuplas, onde o primeiro item consiste de uma chave e o segundo item da parte de informaçao. Para o componente programaçao de vôo, a parte chave consiste do número do vôo e a parte de informaçao consiste do horário de origem e de destino do vôo. No caso do componente programaçao de aviao, a parte chave é constituída pelo número do aviao e o tipo e número dos assentos formam a parte de informaçao.

CSR CONTROLE_AEREO

descriçao { Controle das atividades de vôo de um aeroporto. Provê operaçoes para a gerência de vôos e suas relaçoes com avioes }

"view"_de_tipo

exportaçao

tipo: cta

descriçao geral { CONTROLE_AEREO - operaçoes de exportaçao:
Insere: pega uma descriçao de vôo e de um controle_aereo e insere aquela neste;
Muda_hori: modifica o horário de origem de um dado número de vôo se este já existir no controle aéreo;
Procura: procura determinado número de vôo no controle_aéreo e retorna o valor booleano 1 caso o encontre, e 0 caso contrário;
Pega_hdest: informa o horário de destino para um dado número de vôo. Caso este nao exista no controle_aéreo, é retornado FALSO.
}

```
Func_O:
{
   Insere: nv, hori, hdes, na, ti, ase, cta -> cta;
   Muda_hori: nv, hori, cta                 -> cta;
}

Func_V:
{
   Procura: nv, cta     -> bool;
   Pega_hori: nv, cta   -> hori;
}

Legalidade:
{
L(T);
}

Equivalencia:
{
T.Insere(nv, horil, hdes, na, ti, ase).Muda_hori(nv,
hori2) == T.Insere(nv, hori2, hdes, na, ti, ase);

Insere(nv, hori, hdes, na, ti, ase). Muda_hori(nv2, hori2)
== Insere(nv, hori, hdes, na, ti, ase);
```

```
nv1 == nv2 | na1 == na2 -> T.Insere(nv1, hori1, hdes1,
na1, ti1, ase1).Insere(nv2, hori2, hdes2, na2, ti2, ase2)
== T;
}

Valores:
{
V(T.Insere(nv, hori, hdes, na, ti, ase).Procura(nv)) ==
VERD;
V(T.Insere(nv, hori, hdes, na, ti, ase).Pega_hori(nv)) ==
hori;
V(Procura(nv)) == FALSO;
V(Pega_hori(nv)) == -1;
}
```

importaçao

tipo: prv

descriçao geral { a programacao de voo (prv) possui suas entradas identificadas por um numero de voo (nv). A informacao aqui contida e o horario de origem (hori) e o de destino (hdest) de um voo }

```
Func_O:
{
Insere_prv: nv, hdes, hori, prv  -> prv;
Muda_hori:  nv, hori, prv        -> prv;
}

Func_V:
{
Procura_nv: nv, prv    -> bool;
Pega_hori:  nv, prv    -> hori;
}

Legalidade:
{
L(T);
}

Equivalencia:
{
T.Insere_prv(nv,hdes,hori).Insere_prv(nv,hdes,hori)==T;
T.Insere_prv(nv,hdes,hori).Muda_hori(nv,hori1) ==
== T.Insere_prv(nv,hdes,hori1);
Insere_prv(nv,hdes,hori).Muda_hori(nv1,hori1) ==
== Insere_prv(nv,hdes,hori);
}
```

```
Valores:
{
V(T.Insere(nv, hdes, hori).Procura_nv(nv)) == VERD;
V(T.Insere(nv, hdes, hori).Pega_hori(nv)) == hori;
V(Procura_nv(nv)) == FALSO;
V(Pega_hori(nv)) == -1;
}
```

tipo: prav

descriçao geral { a programacao de aviao (prav) possui suas entradas identificadas por um numero de voo (nv). A informacao aqui contida e o tipo e o numero de assentos (ti e ase, respectivamente) }

```
Func_O:
{
Reserva: nv, ti, ase, prav  -> prav
}

Func_V:
{
Procura_nva: nv, prav  -> bool;
}

Legalidade:
{
L(T);
}

Equivalencia:
{
T.Reserva(nv,ti,ase).Reserva(nv,ti,ase)==T;
}

Valores:
{
V(T.Reserva(nv, ti, ase).Procura_nva(nv)) == VERD;
V(Procura_nv(nva)) == FALSO;
}
```

```
tipo: tupla_va

descricao geral: { tupla_va e uma tupla que consiste dos
tipos prv e prav }

Func_O:
{
constroi    : prv, prav       -> tupla_va;
atualiza_pv: prv, tupla_va   -> tupla_va;
atualiza_pa: prav, tupla_va -> tupla_va;
}

Func_V:
{
Pega_pa: tupla_va -> prv;
Pega_pv: tupla_va -> prav;
}

Legalidade:
{
L(T);
}

Equivalencia:
{
T.constroi(prv,prav).atualiza_pv(prv).atualiza_pv(prv) ==
== T.constroi(prv,prav).atualiza_pv(prv);

T.constroi(prv,prav).atualiza_pa(prav).atualiza_pa(prav)
== T.constroi(prv,prav).atualiza_pa(prav);

T.constroi(prv,prav).atualiza_pv(prv).atualiza_pa(prav) ==
==T.constroi(prv,prav).atualiza_pa(prav).atualiza_pv(prv);

T.constroi(prv,prav).atualiza_pv(prv1) ==
== T.constroi(prv1,prav);

T.constroi(prv,prav).atualiza_pa(prav1) ==
== T.constroi(prv,prav1);
}

Valores:
{
V(T.constroi(prv,prav).Pega_pa()) == prv;
V(T.constroi(prv,prav).Pega_pv()) == prav;
V(Pega_pa()) == -1;
V(Pega_pv()) == -1;
}
```

parametros comuns

tipo: nv;

descricao geral: { nv consiste de um numero de voo. Todo tipo de dados no qual uma operacao de comparacao e definida pode ser utilizado como numero de voo. Os outros parâmetros comuns devem ser feitos de forma análoga }

```
Func_V:
{
==: nv, nv -> bool;
}

Valores:
{
V(==(nv,nv)) == VERD;
}
```

Corpo

descricao geral: { cta e realizado por uma tupla_va }

construcao de cta **e** tupla_va

operacao Insere

equacoes:

```
{
A := prv.Procura_nv(nv, tupla_va.Pega_pv(cta));
B := prav.Procura_nv(na, tupla_va.Pega_pa(cta));

!A & !B -> tupla_va.constroi(prv.Insere_prv(nv, hdes,
hori, tupla_va.Pega_pv(cta)), prav.Reserva(nv, ti, ase,
tupla_va.Pega_pa(cta)));
}
```

operacao Muda_hori

equacoes:

```
{
prv.Muda_hori(nv,hori,tupla_va.Pega_pa(cta));
}
```

operacao Procura

equacoes:

```
{
prv.Procura_nv(nv,tupla_va.Pega_pa(cta));
}
```

```
operacao Pega_hori

equacoes:
{
prv.Pega_hori(nv,tupla_va.Pega_pa(cta));
}
```

"View" de Objeto

exportacao

tipo: cta

descricao geral { mundo concreto do tipo controle_aereo }

```
procedimento Insere(in: nv, hori, hdes, na, ti, ase
                    inout: cta)

procedimento Muda_hori(in: nv, hori
                       inout: cta)

procedimento Procura(in: nv, cta
                     returns: bool)

procedimento Pega_hori(in: nv, cta
                       returns: hori)
```

importacao

tipo: prv

descricao geral { mundo concreto do tipo programacao de voo }

```
procedimento Insere_prv(in: nv, hdes, hori
                        inout: prv)

procedimento Muda_hori(in: nv, hori
                       inout: prv)

procedimento Procura_nv(in: nv, prv
                     returns: bool)

procedimento Pega_hori(in: nv, prv
                       returns: hori)
```

```
tipo: prav

descricao geral { mundo concreto do tipo programacao de
aviao }

procedimento Reserva(in: nv, ti, ase
                     inout: prav)

procedimento Procura_nva(in: nv, prav
                         returns: bool)

tipo: tupla_va

descricao geral { mundo concreto do tipo tupla de voo e
aviao }

procedimento constroi(in: prv, prav
                      inout: tupla_va)

procedimento atualiza_pv(in: prv
                         inout: tupla_va)

procedimento atualiza_pa(in: prav
                         inout: tupla_va)

procedimento Pega_pa(in: tupla_va
                     returns: prv)

procedimento Pega_pv(in: tupla_va
                     returns: prav)


corpo

descricao { descricao procedimental de cta }

construcao de cta e tupla_va

cta     Insere(Nv, Hori, Hdes, Na, Ty, Ase, Cta)
nv      Nv;
hori    Hori;
hdes    Hdes;
na      Na;
ty      Ty;
ase     Ase;
cta     Cta;
{
bool    A, B;
prav    Aviao;
prv     Voo;
```

```
A = prv.Procura(Nv, tupla_va.Pega_pv(cta));
B = prav.Procura(Na, tupla_va.Pega_pa(cta));

if(A     {
   return(cta);
}
else {
   Voo = prv.Insere_prv(Fn, Hori, Hdes, tupla_va.
Procura_prv(Cta));
   Aviao = prav.Reserva(na, ti, ase, tupla_va.
Pega_pa(cta));

   return(tupla_va.constroi(Voo, Aviao);
}
}

cta      Muda_hori(Nv, Hori, Cta)
nv       Nv;
hori     Hori;
cta      Cta;
{
prv      prvoo;

prvoo = prv.Muda_hori(nv,hori,tupla_va.Pega_pa(cta));

return( atualiza_pv(prvoo, cta) );

}

bool     Procura(Nv, Cta)
nv       Nv;
cta      Cta;
{
return( prv.Procura_nv(Nv,tupla_va.Pega_pa(Cta)) );
}

hori     Pega_hori(Nv, Cta)
nv       Nv;
cta      Cta;
{
return( prv.Pega_hori(Nv,tupla_va.Pega_pa(Cta)) );
}
```

ANEXO D. Um Experimento de Validaçao de Software

Como descrito em III.2.1, as especificaçoes formais surgem entre o conceito existente na mente humana - ou seja, o que o programa deve fazer - e o próprio programa. A correteza do programa é estabelecida provando-se que o programa é equivalente à sua especificaçao [LISK,75]. A tal prova denominamos aqui **validaçao**.

Uma experiência de validaçao, dentro deste paradigma, foi realizada quando da produçao do Software de Recuperaçao de Informaçao - SRI [CAMA,88]. A linguagem de recuperaçao deste sistema de software foi formalmente especificada através da técnica de Traços [PARN,86].

A especificaçao da linguagem de recuperaçao do SRI é mostrada a seguir.

1. Sintaxe

Operadores_O :

INDEXE	:	nome da base	-> base

Operadores_V :

ABRA	:	nome da base	-> base
APAGUE	:	nome do arquivo	-> booleano
BASE	:		-> base
BUSQUE	:	string	-> base
CRIE	:		-> booleano
DEFINA	:	nome do arquivo	-> booleano
FIM	:		-> booleano
IMPRIMA	:		-> booleano
INFO	:		-> base
MOSTRE	:		-> base
RECUPERE	:	nome do arquivo	-> base
SALVE	:	nome do arquivo	-> base

```
SUB      : nome da base     -> base

?        : expr             -> base

!        : comando do SO


expr : string |
       string op string

op   : '&'  |
       '|'  |
       '~'  |
       'adj' num
```

2. Semântica

2.1. Legalidade

(1) L(T) -> L(T.!(a))

(2) L(T) -> L(T.INDEXE(a).ABRA(a))

(3) L(T.INDEXE(a)) = L(T.INFO) = L(T.FIM)

(4) L(T.ABRA(a)) -> L(T.ABRA(a).BUSQUE(b))

(5) L(T.ABRA(a)) -> L(T.ABRA(a).?(a) (b))

(6) L(T.BUSQUE(a)) = L(T.BASE) = L(T.?(a))

(7) L(T.?) -> L(T.?(a).MOSTRE)

(8) L(T.MOSTRE) = L(T.SALVE(a)) = L(T.SUB(a))

(9) L(T.SALVE(a)) -> L(T.SALVE(a).RECUPERE(a))

(10) L(T.RECUPERE(a)) = L(T.APAGUE(a))

2.2. Equivalência

(11) T.INDEXE(a).INDEXE(a) == T.INDEXE(a)

(12) T.INDEXE(a).T1 == T.INDEXE(a)

(13) T.INDEXE(a).INDEXE(b) == T.INDEXE(b).INDEXE(a)

2.3. Valores

(14) L(T.ABRA(a)) -> V(T.ABRA(a).BASE) = a

(15) L(T.?(a)) -> V(T.?(a).SALVE(a).RECUPERE(a).MOSTRE) =

= V(T.?(a).MOSTRE)

(16) L(T.?(a)(b)) -> V(T.ABRA(a).?(a)(b)) =

V(T.ABRA(a).?(b))

(17) L(T.?(a)(b)) -> V(T.ABRA(a).ABRA(b).?(a)(c).?(a)(d)) =

= V(T.ABRA(a).ABRA(b).?(a)(c).?(d))

O método de validaçao do código do SRI contra a especificaçao anterior utilizado é descrito em seguida.

A validaçao se deu pela substituiçao das ocorrências de T da especificaçao pela amostra de Traços. Como exemplo, podemos ter a assertiva de Equivalência, para o componente Fila (4.5.2):

```
T.Enqueue(a).Dequeue==T.Dequeue.Enqueue(a);
(1)
```

Além disto, a amostra T={T1, T2, T3} é dada:

```
Enqueue(a).Dequeue().Front();     (2)
Front().Enqueue(a).Enqueue(a).Dequeue();
Enqueue(a).Dequeue().Dequeue().First();
```

As ocorrências de T (1) sao, entao, substituídas por T1 (2), resultando:

```
Enqueue(a).Dequeue().Front().Enqueue(a).Dequeue()==
Enqueue(a).Dequeue().Front().Dequeue().Enqueue(a);
(3)
```

A operaçao descrita anteriormente é feita para todas as assertivas de Legalidade, Equivalência e Valor, isto é, a operaçao é repetida para todas as ocorrências de T da especificaçao, utilizando toda a amostra de Traços. Um arquivo "batch" foi montado para cada assertiva (simples) da especificaçao com substituiçoes para todo o conjunto de amostras de traços dado. Uma assertiva simples é aquela que nao contém dois lados a serem validados. Em (1) é mostrada uma assertiva com 2 lados de equivalência (nao-simples). Neste caso seriam necessários um arquivo para cada lado. Se a especificaçao contivesse 3 assertivas simples e uma amostra de 4 traços fosse utilizada, seriam necessários 12 arquivos de teste. Cada arquivo foi entao executado (execuçao do código do SRI) e o resultado visualmente conferido, à procura de violaçoes por parte do código de propriedades definidas na especificaçao.

As seçoes de Legalidade, Equivalência e Valor foram validadas da seguinte forma:

1. Legalidade: Arquivos "batch" foram montados para essas assetivas como descrito anteriormente. Caso a execuçao de cada um destes nao resultasse em situaçao de erro, nao havia violaçoes às assertivas.

2. Equivalência: Um arquivo foi montado para cada lado de cada assertiva, com toda a amostra. Cada arquivo era executado e gerado um arquivo de saída com o estado resultante. Ambos eram entao comparados e, caso nao fossem iguais, uma violaçao àquela assertiva havia ocorrido.

3. Valores: Feito de forma semelhante à equivalência, sendo que aqui o arquivo de saída consistia de valores resultantes de aplicaçoes de operaçoes-V. Caso estes nao fossem os valores esperados, dizia-se que uma violaçao à assertiva havia ocorrido.

Considerando que a especificaçao da linguagem do SRI anteriormente mostrada, que nao se apresenta como uma linguagem com um grande número de comandos, possui cerca de 30 assetivas simples e que foi reunida uma amostra de dez traços, seriam necessários, para um teste completo, a montagem e execuçao de cerca de trezentos arquivos de entrada e a análise de cerca de trezentos arquivos de saída! Tal tarefa exige um esforço manual extrememente exaustivo. Desta forma, o processo de validaçao do SRI nao foi feito com dez amostras de traços como previsto inicialmente. Mesmo assim erros considerados graves foram encontrados com este processo, sem terem sido detectados anteriormente por teste convencional de programas.

A partir deste esforço concluiu-se que uma validaçao completa, que assegurasse satisfatoriamente a qualidade do software, só seria viável com a automatizaçao de pelo menos parte do processo [CAMA,90].