31880-7

UM ESTUDO DE TÉCNICAS DE VALIDAÇÃO E DE VERIFICAÇÃO DE PRODUTOS DE SOFTWARE

por

Nina Edelweiss

RP-159 Junho/1991

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av. Osvaldo Aranha, 99
90210 - Porto Alegre - RS - BRASIL
Telefone: (0512) 281633
Telex: (051) 2680 - CCUF BR
FAX: (0512) 244164
E-MAIL: PGCC@sbu.ufrgs.anrs.br

Correspondência: UFRGS-CPGCC
Caixa Postal 1501
90001 - Porto Alegre - RS - BRASIL

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Editor: Ingrid E. S. Jansch Pôrto

Engenharia de Software - SBU
Engenharia: Software
Validação: Software
Verificação: Software

CNPq - 1.03.04.00 - 2

U F R G S
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Nº CHAMADA:		Nº REG.:
FL2068		36215
		DATA: 09/07/91
ORIGEM: D	DATA: 1/7/91	PREÇO: cr$5000,00
FUNDO: CPGCC	FORN.: CPGCC	

UFRGS

Reitor: Prof: TUISKON DICK
Pró-reitor de Pesquisa e Pós-Graduaçao: Prof. ABILIO A. BAETA NEVES
Coordenador do CPGCC:Prof. Ricardo A. da L. Reis
Comissão Coordenadora do CPGCC: Prof. Carlos Alberto Heuser
Prof. Clesio Saraiva dos Santos
Profª Ingrid J. Pôrto
Prof. José Mauro V. de Castilho
Prof. Ricardo A. da L. Reis
Prof. Sergio Bampi
Bibliotecária CPGCC/CPD: Margarida Buchmann

LISTA DE FIGURAS

Fig. 1 - Validação/Verificação no ciclo de desenvolvimento de produtos sequenciais 4

Fig. 2 - Validação/Verificação no ciclo de desnvolvimento de protocolos 5

SÚMULA

1. INTRODUÇÃO 1

2. CONCEITOS GERAIS 2
   2.1 VALIDAÇÃO E VERIFICAÇÃO DE SOFTWARE 2
   2.2 VALIDAÇÃO/VERIFICAÇÃO NO CICLO DE VIDA DE UM SOFTWARE 2
   2.3 CRITÉRIOS UTILIZADOS PARA VALIDAÇÃO E VERIFICAÇÃO ... 6
   2.4 UTILIZAÇÃO DE TÉCNICAS E DE FERRAMENTAS 7

3. TÉCNICAS DE VALIDAÇÃO 8
   3.1 "WALKTHROUGH'S" E INSPEÇÕES 8
   3.2 TESTES 9
      3.2.1 TESTES FUNCIONAIS 12
      3.2.2 TESTES ESTRUTURAIS 12
         3.2.2.1 TESTES DE COBERTURA 12
         3.2.2.2 TESTES BASEADOS EM ERROS 13
         3.2.2.3 TESTES DE MUTAÇÕES 14
      3.2.3 TESTES DE PROTOCOLOS 14
   3.3 PROTOTIPAÇÃO 16

4. TÉCNICAS DE VERIFICAÇÃO 19
   4.1 PROVA POR ASSERÇÕES 20
      4.1.1 ASSERÇÕES DE ENTRADA E DE SAÍDA 21
      4.1.2 PRÉ-CONDIÇÕES MAIS FRACAS 22
      4.1.3 LÓGICA TEMPORAL 23
   4.2 INDUÇÃO ESTRUTURAL 23
   4.3 EXECUÇÃO SIMBÓLICA 23
   4.4 ANÁLISE DE ALCANÇABILIDADE 25
   4.5 DESENVOLVIMENTO DE PRODUTOS FORMALMENTE VERIFICÁVEIS . 26

5. TÉCNICAS DE VALIDAÇÃO E DE VERIFICAÇÃO APROPRIADAS AOS
   DIFERENTES PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE 28
   5.1 IMPERATIVO 28
   5.2 ORIENTAÇÃO A FUNÇÕES 29
   5.3 ORIENTAÇÃO A OBJETOS 31
   5.4 ORIENTAÇÃO A LÓGICA 33

6. CONCLUSÃO 35

REFERÊNCIAS BIBLIOGRÁFICAS 38

RESUMO

No desenvolvimento de um produto de software devem ser efetuadas avaliações, com o objetivo de validá-lo e de verificá-lo, dando assim origem a produtos mais confiáveis. Este trabalho situa a validação e a verificação no ciclo de desenvolvimento de um produto e apresenta algumas técnicas utilizadas em cada um destes processos. Apresenta, também, algumas considerações a respeito de quais as técnicas de validação e de verificação mais apropriadas aos principais paradigmas de desenvolvimento de software (imperativo, orientação a funções, orientação a objetos e orientação a lógica).

ABSTRACT

When verification and validation (V & V) are done during the software development process they give rise to products of better quality. This work presents a study of software validation and verification techniques. The validation and verification activities are related to the software development cycle and some specific techniques are presented. The appropriate V & V techniques to the main software development approaches (imperative, function oriented, object oriented and logic oriented) are identified.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

1. <u>INTRODUÇÃO</u>

O desenvolvimento de sistemas de software é uma tarefa demorada, complexa e dispendiosa. A garantia de que os sistemas desenvolvidos estejam, o mais possível, corretos é muito importante.

No desenvolvimento de sistemas de software podem ser introduzidos erros (1) pelo próprio usuário que solicita o desenvolvimento do sistema, ao não especificar corretamente as suas aspirações; (2) pelo projetista do sistema, ao construir uma especificação do sistema que não atenda corretamente aos requisitos solicitados pelo usuário; e (3) pelo codificador, ao implementar esta especificação. Detectar estes erros é uma tarefa bastante difícil. Para a identificação dos dois primeiros tipos de erros são utilizadas técnicas de <u>validação</u>, através das quais se procura mostrar ao usuário do sistema como foram interpretados os requisitos por ele definidos. Os erros de implementação são detectados através de técnicas de <u>verificação</u>.

Muito tem sido desenvolvido e utilizado efetivamente na área de validação e de verificação de software. Diferentes técnicas, utilizando diferentes enfoques. A importância que este assunto vem apresentando é comprovada pela grande quantidade de pesquisas realizadas na área.

O presente trabalho tem por objetivos:

a) apresentar um estudo das técnicas de validação e de verificação mais utilizadas;

b) analisar os principais paradigmas de desenvolvimento de software, procurando identificar quais as técnicas de validação e de verificação mais apropriadas a cada um deles.

O capítulo 2 do trabalho apresenta conceitos gerais de validação e de verificação. Os capítulos 3 e 4 analisam, respectivamente, técnicas de validação e de verificação. O capítulo 5 é dedicado a uma breve apresentação dos principais paradigmas de desenvolvimento de software, com a análise das técnicas mais apropriadas a cada um deles. As conclusões tiradas da realização do trabalho constituem o capítulo 6.

2. CONCEITOS GERAIS

Este capítulo procura definir validação e verificação de sofware e identificar em que pontos do ciclo de desenvolvimento de um produto cada um destes conceitos se aplica. Apresenta, ainda, alguns critérios básicos que devem ser considerados na validação e na verificação, finalizando com a introdução às diferentes técnicas e ferramentas utilizadas para apoiar estes processos.

2.1 VALIDAÇÃO E VERIFICAÇÃO DE SOFTWARE

Na literatura são encontrados diversos significados para validação e verificação de software. As definições apresentadas a seguir são as mais utilizadas [BER82, BOE84, COH86, DEU81, FAI85].

A validação de um produto tem por objetivo demonstrar que uma especificação ou um sistema implementado satisfaz os requisitos desejados pelos seus usuários. Procura determinar se foi construído o produto realmente desejado.

A verificação, por sua vez, se baseia em provas rigorosas, formalmente justificáveis. Procura saber se o produto está sendo construído corretamente, mostrando a consistência entre duas representações do mesmo comportamento, uma delas substancialmente mais detalhada do que a outra.

2.2 VALIDAÇÃO/VERIFICAÇÃO NO CICLO DE VIDA DE UM PRODUTO DE SOFTWARE

O desenvolvimento de um produto de software é efetuado em diversas etapas. Estas etapas variam de acordo com o problema específico que está sendo tratado. Dois tipos basicamente diferentes de produtos são os sequenciais, nos quais somente uma instrução pode ser executada a cada instante, e aqueles que apresentam processos concorrentes, nos quais mais de um componente pode ser ativado simultaneamente. Vamos analisar cada um deles separadamente.

O ciclo de desenvolvimento convencional de um produto de software sequencial inicia com a coleta dos requisitos funcio-

nais do sistema a ser implementado, efetuada junto aos usuários do sistema. Uma vez definidos os requisitos, o sistema é especificado e, posteriormente, implementado fisicamente. As diferentes etapas percorridas resultam em documentos que apresentam os resultados obtidos. Os principais documentos são os seguintes:

- especificação dos requisitos funcionais;
- especificação do sistema;
- implementação do sistema.

A avaliação do produto obtido não deve ser realizada somente após sua implementação final, mas deve acompanhar todo o ciclo de seu desenvolvimento. A seguir apresentamos alguns aspectos de cada um destes documentos, comentando qual o tipo de avaliação que pode ser realizado ao final de sua elaboração (validação e/ou verificação) e quais os tipos de erros que podem ser detectados.

A especificação dos requisitos funcionais é elaborada pelo projetista do sistema com o auxílio de seu usuário. Nela se baseiam todas as fases subsequentes do desenvolvimento, sendo de fundamental importância a sua validação junto aos usuários logo no início do ciclo de desenvolvimento do sistema, com o objetivo de avaliar se as funções desejadas são realmente as especificadas. A especificação dos requisitos é usualmente representada de forma informal ou, no máximo, semiformal.

Segundo [FAI85], os principais erros na especificação de requisitos são causados por (1) falhas de representação das necessidades do usuário, (2) especificação incompleta dos requisitos e de seu desempenho, (3) inconsistências entre os requisitos, e (4) especificação de requisitos não realizáveis. O processo utilizado na validação de especificações de requisitos deve procurar detectar estes erros.

A especificação do sistema deve ser, sempre que possível, expressa formalmente, possibilitando a sua verificação formal. Esta verificação tem por objetivo detectar erros de projeto do sistema. Mesmo quando não for possível a verificação totalmente formal deverá ser efetuada alguma outra forma de verificação, comparando-a com a especificação funcional dos requisitos e verificando a consistência da própria especificação. Muitas vezes, dependendo do formalismo empregado, é possível a construção de um protótipo do sistema a partir da especificação, dando possibili-

dade a efetuar sua validação junto aos usuários, através da simulação de sua execução.

Uma vez efetuada a implementação do sistema, esta deverá ser validada, sempre com respeito aos requisitos iniciais colocados pelo usuário, à procura de eventuais erros de implementação. Diferentes técnicas de testes e de depuração de programas são empregadas. Sempre que possível, deve ser efetuada a verificação dos programas em comparação com a especificação formal anterior.

Vemos, portanto, conforme esquematizado na figura 2.1, que a validação está presente nas principais fases do desenvolvimento de um produto de software sequencial, devendo ser realizada em cada uma delas para garantir a satisfação final do usuário. Quanto à verificação, esta se apresenta em todas as etapas subsequentes à especificação dos requisitos.

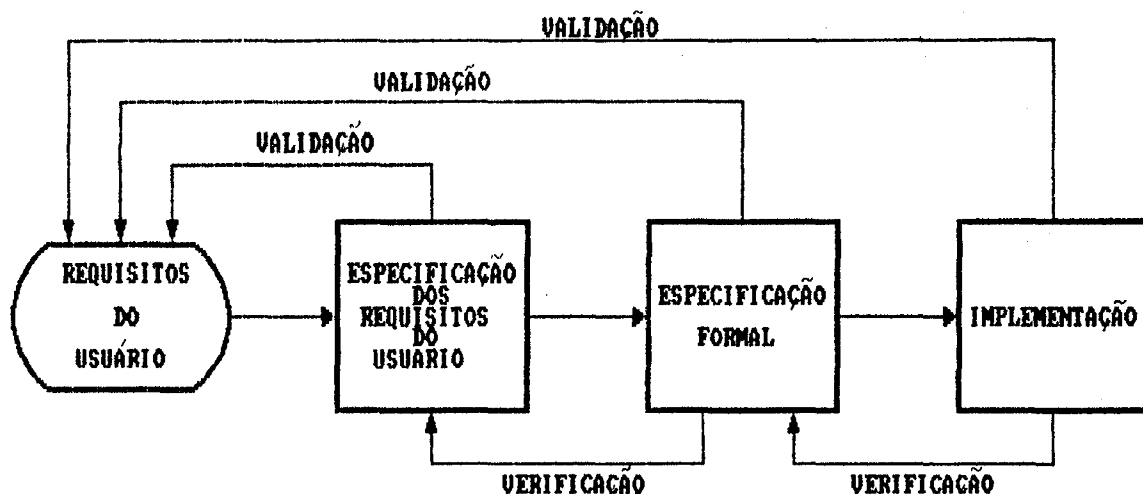


Figura 2.1 - VALIDAÇÃO/VERIFICAÇÃO NO CICLO DE DESENVOLVIMENTO DE PRODUTOS SEQUENCIAIS

As fases de desenvolvimento de sistemas concorrentes são basicamente as mesmas dos sistemas sequenciais. Devido à complexidade destes sistemas, foram desenvolvidas técnicas para sua especificação formal [BOC90], tanto pela ISO (International Stan-

dards Organization), que desenvolveu as linguagens ESTELLE e LOTUS, quanto pela CCITT, com a linguagem SDL. A ISO também desenvolveu o modelo OSI (Open Systems Interconnection) de arquitetura para estes sistemas, composto de 7 camadas que definem serviços específicos. A comunicação entre estas camadas é governada através de regras denominadas protocolos. A especificação e posterior implementação de sistemas concorrentes se concentra em especificar e implementar estes protocolos.

As etapas de especificação de um protocolo são as seguintes [PEH90]: (1) formulação dos requisitos do usuário; (2) especificação dos serviços de comunicação que atendem aos requisitos formulados; (3) especificação de protocolos que forneçam os serviços especificados; e (4) implementação das entidades de protocolo de acordo com a especificação dos protocolos.

A validação e/ou verificação dos protocolos de sistemas de comunicação também deve ser efetuada em todas as fases de seu desenvolvimento, através de processos de validação dos requisitos (serviços desejados) ou de verificação formal. Na figura 2.2 temos um esquema das fases de desenvolvimento de protocolos de comunicação, indicando os processos usualmente utilizados de avaliação dos protocolos gerados.

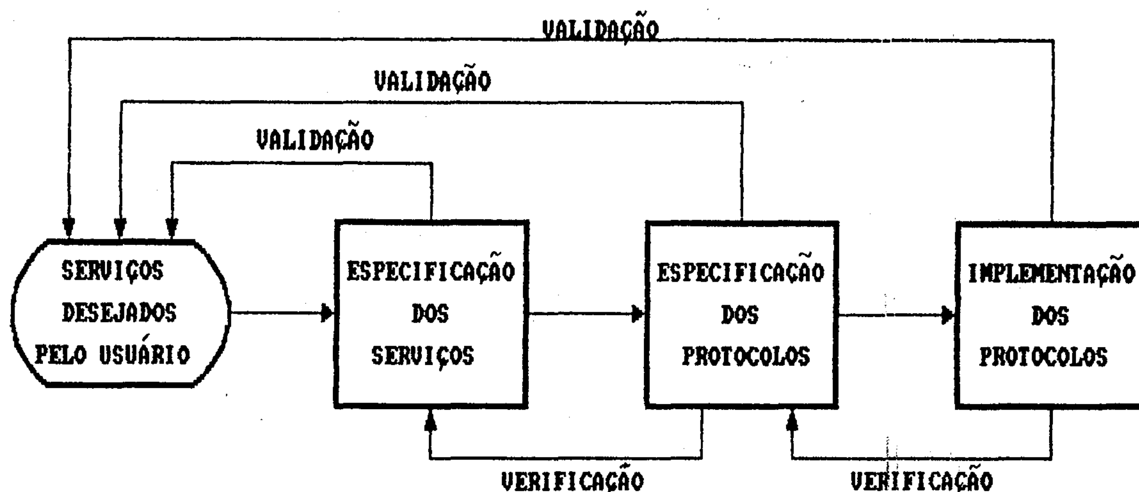


Figura 2.2 - VALIDAÇÃO/VERIFICAÇÃO NO CICLO DE DESENVOLVIMENTO DE PROTOCOLOS

UFRGS.
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Vemos, portanto, que a validação se apresenta nas principais fases de desenvolvimento de um produto, seja sequencial ou concorrente. A verificação se apresenta somente nas fases mais avançadas, uma vez que se baseia sempre em uma forma anterior de representação do produto.

2.3 CRITÉRIOS UTILIZADOS PARA VALIDAÇÃO E VERIFICAÇÃO

Os processos de validação e de verificação avaliam características específicas de produtos de software. Os critérios utilizados nestas avaliações dependem das características que se quer avaliar.

Os critérios a serem utilizados variam de acordo com o tipo de produto que está sendo avaliado - sequencial ou concorrente - e com o estágio de seu desenvolvimento - especificação ou implementação.

Nos produtos sequenciais, uma especificação (de requisitos ou do sistema) deve apresentar as seguintes características [FAI85]: corretude, completeza, consistência, rastreamento, funcionalidade, verificabilidade, não ser ambígua e poder facilmente ser modificada. As características que deve apresentar uma implementação não diferem muito das de uma especificação, incuindo corretude, completeza, consistência, rastreamento, testabilidade, não apresentar ambiguidades e possuir facilidades para modificações e manutenção. A validação de uma especificação usa como principal critério a funcionalidade, enquanto que a sua verificação se preocupa principalmente com a corretude. Os outros aspectos, entretanto, também devem ser avalidados.

As propriedades gerais dos protocolos de processos concorrentes, tanto em sua especificação como na implementação, são [MOU86] ausência de impasse ("deadlock"), atividade ("liveless"), realização de progresso, estabilidade, completeza, terminação, correção parcial e minimidade. Vemos, portanto, que além das características dos produtos sequenciais, ainda devem ser consideradas propriedades de segurança próprias destes processos.

2.4 UTILIZAÇÃO DE TÉCNICAS E DE FERRAMENTAS

Diversas são as técnicas utilizadas tanto para validação como para verificação de produtos de software - especificações ou implementações (programa ou sistema). São geralmente classificadas em técnicas de análise estática e análise dinâmica. Na análise estática é considerado somente o texto do produto de software (especificação ou implementação), enquanto que a análise dinâmica envolve algum tipo de execução. Esta classificação não é muito clara pois algumas técnicas podem ser efetuadas tanto de maneira estática como dinâmica. Existem, ainda, esforços no sentido de unir, em uma só técnica, a análise estática e a dinâmica.

A complexidade crescente dos produtos de software torna necessária a utilização de ferramentas automáticas nos processos de validação e de verificação. O número de ferramentas existentes é bastante extenso, sendo que grande ênfase tem sido dada ao desenvolvimento de novas ferramentas. O ambiente denominado ARCADIA [TAY89], por exemplo, é um ambiente para o desenvolvimento de protótipos de ferramentas para teste de software sequencial e concorrente. Muitos dos ambientes de desenvolvimento de software incluem ferramentas para avaliação dos produtos gerados. Entre eles podemos citar os ambientes TEAM [CLA89], REFINE [KOT89], GYPSY [MEN88] e AFFIRM [MEN88].

As ferramentas também podem ser classificadas como estáticas e dinâmicas [MIL84]. As ferramentas estáticas analisam o texto automaticamente, de maneira sistemática. As ferramentas dinâmicas apoiam a execução do produto, selecionando valores de entrada, processando "stubs", apresentando resultados para a validação, etc.

Nos dois capítulos a seguir apresentaremos um resumo de diferentes técnicas encontradas na bibliografia, tanto para validação e verificação de especificações como de implementações. Usaremos o termo genérico produto para designar especificação ou implementação. Sempre que possível será indicada alguma ferramenta desenvolvida para os métodos específicos.

3. TÉCNICAS DE VALIDAÇÃO

As técnicas de validação tem por objetivo comparar um produto de software com os requisitos solicitados pelo usuário, procurando avaliar se estes estão sendo cumpridos. A seguir apresentaremos algumas das técnicas mais utilizadas.

3.1 "WALKTHROUGH'S" E INSPEÇÕES

Os "walkthrough's" e as inpeções são técnicas de análise estática. Embora bastante semelhantes, apresentam alguns pontos fundamentais diferentes. Ambos utilizam, no processo de validação, somente o texto do produto de software, que é analisado por uma equipe.

Um "walkthrough" [FAI85, LON85] consiste de uma revisão técnica profunda e rigorosa do texto do produto, efetuada por uma equipe de 4 a 6 pessoas. Os membros da equipe são as pessoas envolvidas diretamente no desenvolvimento do produto - o analista do sistema, o projetista e os programadores. O material a ser revisado é apresentado por um dos membros da equipe, o revisor, atuando os demais membros como avaliadores. É feita uma "execução manual" do sistema ou de trechos de programa, utilizando dados especificamente projetados para esta finalidade.

O objetivo desta revisão é somente detectar os erros existentes, ficando a correção para ser efetuada posteriormente. Serve também para integrar a equipe de desenvolvimento.

Uma inspeção [FAG86, BIS89] se diferencia da técnica anterior pela maneira como o texto é revisado e pelos membros da equipe de revisão. Numa inspeção, o próprio texto do produto é analisado, à procura de eventuais erros de lógica e de implementação. Os membros envolvidos em uma equipe de inspeção não participam efetivamente do desenvolvimento do produto. Sua tarefa é simplesmente inspecionar o produto, detectar os erros e solicitar à equipe de desenvolvimento a sua correção. Os participantes típicos de uma equipe de inspeção (3 a 6 pessoas) são o projetista, o codificador, o testador e o moderador. Todos os membros da equipe recebem o material a ser analisado com antecedência, estudando-o em profundidade antes da seção de inspeção. Munidos de listas de itens a serem analisados, cada membro da equipe assume um papel e representa o ponto de vista deste durante a inspeção.

As etapas principais de uma inspeção são: (1) planejamento, quando é preparado o material, escolhidos os membros da equipe, etc; (2) "overview", reunião na qual são definidos o papel de cada elemento da equipe e situado o trabalho no contexto geral; (3) preparação, fase na qual cada membro da inspeção estuda o material a ser analisado; (4) seção de inspeção propriamente dita; (5) alterações do texto, solicitadas pelo moderador da inspeção e realizadas pelo projetista; e (6) acompanhamento das alterações realizadas, efetuado pelo moderador.

"Walkthrough" e inspeções podem ser realizados em todas as fases de desenvolvimento de um produto, para analisar os requisitos especificados, o projeto do sistema e o código implementado.

3.2 TESTES

O teste de um produto de software consiste na sua análise através da avaliação de suas respostas a um conjunto selecionado de valores de entrada. É a técnica de validação mais utilizada, principalmente para validação de produtos sequenciais. Também é utilizada para testes de protocolos de comunicação, embora não seja muito apropriada aos produtos concorrentes devido ao grande número de sequencias possíveis de execução.

Dois aspectos devem ser observados na utilização de testes: (1) a seleção de valores apropriados para entradas dos testes, e (2) a análise das interações observadas no produto ao ser executado com estes valores de entrada, com o objetivo de determinar se o par causa/efeito está de acordo com os requisitos do usuário.

A seleção dos valores de entrada para os testes é fundamental para o sucesso na utilização desta técnica. Os valores de entrada devem ser planejados juntamente com o desenvolvimento do produto. Em cada uma das fases do desenvolvimento devem ser selecionados valores para teste, valores estes apropriados à validação de aspectos relevantes ao nível de abstração considerado na respectiva fase. A partir da especificação dos requisitos são selecionados os primeiros valores, com base nas funções que o sistema pode desempenhar. São selecionados pelo analista ou pelo próprio usuário. A partir da especificação formal podem ser selecionados valores de teste de uma maneira formal [RIC89], embora

esta técnica ainda não esteja muito desenvolvida. Mesmo aquelas técnicas que testam especificações formais (tais como OBJ [GOG82, GOG87], utilizado para testar especificações algébricas, e as técnicas propostas por [KEM85] para testar especificações em Ina Jo), não fazem considerações quanto à seleção do conjunto de valores de entrada a ser utilizado.

Na seleção dos valores devem ser levados em consideração não somente os casos normais, mas também os casos limites e os casos errôneos.

Quanto ao número de valores de entrada utilizados nos testes, estes podem ser classificados em:

(a) Testes exaustivos, nos quais se pressupõe que sejam efetuados testes com todos os possíveis valores do domínio de entrada. Este domínio geralmente possui um número muito grande de elementos, sendo quase sempre infinito. Por maior que seja o número de valores de entrada utilizado, este conjunto dificilmente será formado pela totalidade de valores possíveis. Este tipo de teste, portanto, não é quase utilizado.

(b) Testes randômicos, que utilizam um subconjunto aleatório de valores do conjunto total de valores de entrada possíveis. Embora não seja adotado nenhum critério na seleção destes valores, muitos erros podem ser detectados com a sua execução.

(c) Testes usando partições [JEN89, WEY80] nos quais o domínio de valores de entrada é dividido, segundo algum critério pré-definido, em subconjuntos não necessariamente disjuntos. Os testes são realizados utilizando um ou mais elementos de cada um destes subconjuntos. A divisão deve ser tal que o conjunto de valores de teste retirados dos conjuntos seja representativo em relação ao domínio todo. No caso particular de divisão em subconjuntos disjuntos, em que para cada subdomínio o programa fornece a mesma resposta (correta ou incorreta) para todos os seus elementos, usa-se a denominação de subconjuntos reveladores [WEY80] (subconjuntos de valores equivalentes). Quando da utilização de subconjuntos reveladores, é suficiente a utilização de um só elemento de cada subconjunto para formar o conjunto de valores de testes. Segundo [JEN89], não é usual a divisão em subconjuntos reveladores, sendo mais comum a obtenção de subconjuntos não disjuntos.

Os testes servem somente para detectar a presença de erros no produto, não provando a sua ausência.

Testes são normalmente efetuados somente em implementações. Especificações formais, entretanto, também podem ser testadas, já existindo algumas ferramentas para isto (por exemplo, especificações algébricas descritas em OBJ). Em [RIC89] é apresentada a idéia de estender as técnicas usuais de teste de implementações com a finalidade de testar também especificações formais, exemplificando com especificações nas linguagens LARCH e ANNA.

As técnicas de testes evoluiram muito nos últimos anos, tornando-se mais rigorosas, envolvendo planos e procedimentos formais, através da utilização de ferramentas automáticas. Quatro tipos de ferramentas podem ser utilizadas para apoiar a execução de testes [AND86]: analisadores de cobertura, monitores da execução, processadores dinâmicos de asserções e monitores de desempenho.

Segundo [FAI85], todo produto de software deve satisfazer os seguintes tipos de testes: (1) funcionais, que validam o produto em relação à satisfação requisitos definidos pelo usuário, sem se preocupar com fatores de desempenho; (2) de desempenho, onde são validados os requisitos que envolvem parâmetros de desempenho do produto; (3) de "stress", que analisam o comportamento frente a eventuais sobrecargas do sistema; e (4) estruturais, nos quais é analisada a lógica interna de processamento do produto.

Os três primeiros tipos de testes acima citados são realizados junto às interfaces do sistema, analisando o produto como um todo. Por este motivo são denominados de testes de caixa-preta. São fornecidos dados de teste e medidos os resultados produzido, comparando-os com os resultados previstos na especificação dos requisitos. Já os testes estruturais se propõe a examinar os caminhos lógicos percorridos durante a execução do teste, sendo por isto também chamados de testes de caixa-aberta ou caixa-branca.

Várias técnicas de testes foram desenvolvidas. Elas se diferenciam basicamente pelo modo como são escolhidos os dados de teste. Vamos analisar sucintamente algumas delas.

3.2.1 TESTES FUNCIONAIS

Nos testes funcionais [DEM87, HOW80, HOW86, HOW89] são testadas as funções que o produto deve executar, especificadas nos requisitos do usuário. O conjunto de dados de teste é construído a partir da especificação destes requisitos, com base nas propriedades funcionais que o produto deve apresentar e nos seus domínios de entrada e de saída. Deste modo, este tipo de teste é dependente da qualidade da especificação.

Como o número de elementos do conjunto de valores de entrada pode ser muito elevado, foram desenvolvidas diversas técnicas para diminuí-lo. Em [OST88], por exemplo, é apresentado um método denominado de partição por categorias, que consiste de uma forma sistemática de analisar as especificações funcionais do sistema, buscando construir uma partição bem-definida de cada uma das funções do domínio de entrada, juntamente com a descrição dos resultados esperados de cada uma destas funções. Já em [BAL89] é encontrado um modelo de especificação de teste que permite que especificações escritas em uma linguagem específica, a Test Specification Language (TSL), sejam compiladas gerando roteiros de testes completos e executáveis.

3.2.2 TESTES ESTRUTURAIS

Testes estruturais [DEM87, HUA75, NTA88] examinam a estrutura interna do produto de software e dela derivam os dados de teste. É provavelmente a estratégia de teste mais utilizada, existindo um significativo número de ferramentas para apoiá-la.

Muitas são as estratégias desenvolvidas para testes estruturais. A seguir serão vistas algumas delas, sendo a primeira a mais utilizada.

3.2.2.1 TESTES DE COBERTURA

Nos testes de cobertura os dados são selecionados de modo a causar a execução de partes específicas do texto do produto. Dentre estes testes podemos citar: (1) testes de cobertura de segmentos, nos quais cada segmento do produto (por exemplo, comando) deve ser executado pelo menos uma vez; (2) testes de cobertura de arestas (Linear Code Sequence and Jump, [WOO80]), nos

quais são testadas sequências consecutivas de comandos, sequências estas que iniciam em um ponto de entrada e terminam em um ponto de saída ou de desvio, ocasionando que toda aresta de transferência de controle seja executada pelo menos uma vez; (3) <u>testes de cobertura de decisões</u>, nos quais toda decisão elementar, bem como toda decisão resultante de alguma expressão lógica, é executada pelo menos uma vez; (3) <u>testes de cobertura de caminhos</u>, nos quais todos os possíveis caminhos lógicos através do produto devem ser executados pelo menos uma vez.

Os testes de cobertura de caminhos são dificultados pelo grande número, geralmente infinito, de caminhos possíveis. Isto levou ao desenvolvimento de diversas técnicas para limitar o número de caminhos, entre as quais podemos citar:

(a) <u>Testes de caminhos estruturados</u>, onde o número de caminhos de teste é diminuido pelo agrupamento daqueles que diferem somente pelo número de vezes que iteram ciclos, sendo testados somente alguns caminhos significativos de cada grupo. Casos particulares destes últimos são os "<u>boundary path testing</u>", nos quais são testados os caminhos que entram nos ciclos de repetição mas não executam as iterações, e os "<u>interior path testing</u>", onde são executados todos os caminhos diferentes dentro do ciclo, sendo efetuada somente uma iteração.

(b) <u>Testes baseados em análise de fluxo de dados</u> [LAS82, LAS83, NTA84, RAP85], nos quais se procuram caminhos que envolvam definições de variáveis e subsequentes referências a estas variáveis.

3.2.2.2 <u>TESTES BASEADOS EM ERROS</u>

Nestes testes o conhecimento de erros usualmente cometidos é utilizado para guiar a escolha dos valores para teste [HOW90, NTA84]. A idéia é identificar as classes de erros e efetuar testes específicos para identificar erros de cada uma destas classes. Foram identificadas duas classes básicas de erros: erros de abstração e erros de decomposição. Esta técnica fornece bons resultados para detectar erros cometidos, mas não é apropriada para erros de omissão.

Como exemplos da utilização desta técnica podemos citar:

a) o modelo RELAY [RIC89], o qual apresenta uma técnica para seleção de casos de teste baseada em falhas;

b) em [HOW90] é apresentada uma ferramenta para a realização de testes de detecção de erros de decomposição, denominada QDA (QUICK DEFECT ANALYZER).

3.2.2.3 TESTES DE MUTAÇÕES

Nos testes de mutação [JEN89, NTA84], um conjunto de pequenas modificações, denominadas mutações, é introduzido no produto a ser testado, sendo depois efetuados testes com o objetivo de detectá-las. Antes de serem efetuadas as modificações, é necessário que se determine um conjunto de valores para os quais o produto fornece resposta correta. Cada modificação introduzida deve ser testada para cada um dos elementos deste conjunto de valores. As mutações que não incorrerem em erro podem ser equivalentes ao texto original. Quando este não for o caso, novos valores de teste devem ser adicionados ao conjunto inicial, demonstrando que este não cobria as necessidades. Os novos valores utilizados para detectar falhas provocadas por mutações poderão identificar outros erros do produto, não identificados pelo conjunto inicial de valores de teste. Uma desvantagem desta técnica é a possibilidade de ser necessário um grande número de testes até que a modificação altere o resultado.

Uma ferramenta desenvolvida para apoiar este tipo de teste é a denominada P^{M}othra [CHO89].

3.2.3 TESTES DE PROTOCOLOS

Um protocolo é a especificação abstrata de um sistema envolvendo a coordenação de processos concorrentes. Os testes de protocolos [LIN90, MAT88a, SAR89, ZEN89] são normalmente denominados de testes de conformidade, pois procuram avaliar se o produto desenvolvido está de acordo com a sua especificação.

Este tipo de teste se baseia na análise de PDUs (Protocol Data Unit - unidade de dados do protocolo) trocadas. Esta análise é feita em duas etapas: (1) análise sintática, onde é verificado se a PDU é bem-formada; e (2) análise semântica, quando é analisado o sequenciamento e o conteúdo das PDUs.

Os testes de conformidade podem ser de dois tipos: (1) testes de conformidade formal, realizados em laboratórios de testes para certificação; e (2) testes diagnósticos, realizados pelos implementadores durante o desenvolvimento. O primeiro destes dois tipos de testes caracteriza um método de verificação, enquanto que o segundo constitui uma validação.

O teste de protocolo é realizado considerando uma entidade (uma ou mais camadas do modelo OSI da ISO), estimulando esta entidade através de suas primitivas em pontos específicos de controle e observação, e observando as reações da implementação sob teste. O envio/recebimento destes estímulos é feito através de um sistema de teste.

Quatro são os métodos abstratos de teste de protocolos (identificados pelo ISO): teste local, distribuído, coordenado e remoto. O teste local é o mais poderoso dentre os quatro. Neste, o sistema de teste deve residir no próprio sistema sob teste, o que pode trazer problemas. O método de teste remoto não requer que este sistema de teste esteja na unidade a ser testada, o que traz como consequência uma grande redução de sua capacidade. Os métodos coordenado e remoto apresentam unidades de teste em duas máquinas diferentes, além de uma complexa unidade de teste junto à unidade que está sendo testada, sendo deficientes na garantia de sincronização entre estas unidades.

A ISO propôs dois tipos de critérios para os testes de conformidade de uma implementação: (1) critérios estáticos, que centram sua atenção na formação de estruturas de dados apropriadas, codificação de PDUs, além de critérios condicionais que dependem de escolhas legítimas do implementador; e (2) critérios dinâmicos, que se preocupam com o comportamento da implementação à medida em que interage com outra implementação ou durante um teste do sistema.

O processo de teste de protocolos inicia com a análise estática de conformidade. No caso desta ter sucesso, é feita a seleção de testes, a parametrização dos testes, em seguida a sua execução e a análise dos resultados.

A ISO recomenda que quatro tipos de testes de conformidade sejam aplicados a uma unidade sob teste [SAR89]: (1) testes de interconexão básica, através dos quais são detectados problemas de interconexão da unidade sob teste com a unidade de teste,

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

ou de não atendimento das características do protocolo; (2) testes de capacidade, onde são testados os requisitos de conformidade estática definidos para os protocolos; (3) testes de comportamento, onde é testada a conformidade dinâmica de uma implementação; e (4) testes de resolução de conformidade, que verificam se uma determinada implementação satisfaz requisitos particulares.

Como exemplo de ferramentas implementadas para estes testes podemos citar o ambiente para especificação, projeto, validação e manutenção de testes de conformidade do projeto TOCS (Testing Open Communications Systems) [PRO89]. /

3.3 PROTOTIPAÇÃO

A prototipação [TAY82, WEI82, WAS82] consiste na implementação de um modelo (protótipo) de um sistema com o objetivo de validar algum aspecto específico deste sistema, através da simulação de sua execução. Este protótipo tem por principal característica ser operacional, permitindo ao usuário a visualização do sistema em funcionamento. Desta maneira, o protótipo aumenta o conhecimento do usuário sobre o problema, ajudando-o a esclarecer dúvidas e a fixar idéias.

A criação de um protótipo tem como principal finalidade validar as especificações dos requisitos do sistema antes de sua implementação completa. O usuário geralmente não é profundo conhecedor de computação, não sabendo expressar com clareza o que deseja do sistema. Colocado em frente ao sistema em funcionamento, as divergências de comunicação entre ele e o projetista aparecerão logo de início e não somente após a implementação completa do sistema. Mesmo que o usuário tenha expressado corretamente suas aspirações, ocorre muitas vezes dele mudar as especificações do sistema durante o seu desenvolvimento ou quando completada a implementação, por não saber exatamente o que desejava no início. Uma vez colocado em frente ao sistema em funcionamento, suas dúvidas serão esclarecidas, evitando problemas futuros.

Para que um protótipo seja caracterizado como um modelo do sistema e não como uma versão do produto final, ele deverá representar apenas parte do sistema desejado. Somente desta maneira será possível obtê-lo de modo rápido e econômico. Na pesquisa dos aspectos do sistema que poderão ser negligenciados no desenvolvimento de seu protótipo, aparecem três aspectos diferentes e apro-

ximadamente independentes. Conforme o objetivo a ser verificado pelo protótipo, um dos aspectos deverá ser completamente desenvolvido e os outros serão sacrificados. A prototipação rápida, portanto, deverá verificar somente um dos seguintes aspectos:

(a) desempenho do sistema, devendo neste caso ser implementadas as partes do sistema que consomem mais tempo de execução;

(b) interface com o usuário, verificando os requisitos colocados por ele e suas aspirações a respeito do sistema como um todo;

(c) funcionalidade do sistema, utilizando estruturas completas de dados, embora empregando, possivelmente, algoritmos para a manipulação destes dados, linguagem de programação e máquinas diferentes daquelas que serão utilizadas no sistema final.

A prototipação pode ser efetuada em duas fases distintas do ciclo de desenvolvimento de um produto de software: (1) logo no início, para auxiliar na análise do problema, através de um diálogo entre o usuário e o projetista, com vistas à validação da especificação dos requisitos funcionais do sistema; e (2) para validar a especificação formal do sistema, através de nova iteração entre o projetista e o usuário.

A grande vantagem da prototipação sobre outras técnicas de verificação formais, é a presença de facilidades para comunicação e modificação da especificação, facilidades estas não encontradas na maioria das técnicas formais utilizadas.

Em relação à implementação posterior do sistema, o processo de prototipação pode ser classificado em [LEE86]: (1) "throwaway prototyping", quando o protótipo é utilizado somente como modelo, sendo a implementação do sistema um processo totalmente novo, desenvolvido por métodos tradicionais; e (2) "cornerstone prototyping", quando o sistema final é desenvolvido a partir do protótipo, acrescentando-lhe as características que faltam.

A obtenção de um protótipo de modo rápido e econômico é possível através da utilização de ferramentas que auxiliem o seu desenvolvimento, tais como as linguagens de 4ª geração e os geradores de aplicação (ex: RAPID [WAS82]). Além disso, esta obtenção também pode ser facilitada através de reutilização de software já disponível (tanto de projeto como de código).

Alguns métodos formais de especificação apresentam ferramentas próprias para prototipação. Entre eles podemos citar:

(a) a linguagem de especificação OBJ [GOG82, GOG87], que conjuga os aspectos de prototipação rápida com especificação formal através do método algébrico;

(b) a linguagem de especificação INA JO [KEM85], que possui uma ferramenta para gerar um protótipo que testa a funcionalidade da especificação dos requisitos;

(c) TODOS [HEI88, PER90], um ambiente para desenvolvimento de Sistemas de Informação de Escritórios (SIE), o qual apresenta uma ferramenta de prototipação a partir de especificação formal do escritório, especificação esta feita através da linguagem de especificação C-TODOS [PER89];

(d) a linguagem TRACE [HOF88], a partir da qual podem ser construídos modelos executáveis.

A prototipação é muito útil para a validação de especificações dos serviços desejados pelo usuário em processos concorrentes, através da simulação de sua execução. Dos sistemas existentes podemos citar:

a) SPANNER [AGG88], que implementa um modelo de máquina de estados finitos, através do qual é possível fazer análise de alcançabilidade e simulação de execução;

b) PROTEAN [BIL88], que utiliza redes de Petri numéricas;

c) IC* [CAM88], que implementa protótipos diretamente a partir de especificações na linguagem formal IC*;

d) VÉDA [JAR88], um simulador de especificações na linguagem Estelle;

e) GYPSY [MEN88], que utiliza a linguagem GYPSY para a especificação e a implementação de programas concorrentes e que permite, além da verificação formal, a validação através da sua execução.

4. TÉCNICAS DE VERIFICAÇÃO

As técnicas de verificação se baseiam em uma análise formal do produto de software, seja uma especificação ou uma implementação. O produto é tratado como se fosse um objeto matemático, sendo utilizada a lógica matemática para provar suas propriedades.

Uma vez que não é possível desenvolver automaticamente todas as fases de um produto, deve ser proporcionado ao projetista apoio para a verificação dos passos de transformação que foram efetuados manualmente, o que pode ser feito através de técnicas de verificação. Além disso, sempre que as propriedades específicas do comportamento esperado de um produto puderem ser expressas como teoremas, elas poderão ser verificadas através de técnicas formais, rigorosamente matemáticas.

As propriedades fundamentais avaliadas nos processos de verificação são a terminação e a corretude. A corretude parcial não leva em conta a terminação, sendo que a corretude total engloba a corretude parcial e a terminação.

A verificação de um produto pode ser realizada de duas maneiras distintas [RAM89]: (1) verificar formalmente o produto pronto, comparando-o com a sua especificação (enfoque estático); ou (2) desenvolver o produto já formalmente verificado, com o apoio de alguma ferramenta de desenvolvimento (enfoque construtivo).

A verificação do produto pronto, devido à sua complexidade, é geralmente realizada através de ferramentas. As duas principais classes de ferramentas utilizadas em sistemas de verificação são os geradores de condições de verificação e os provadores de teoremas.

Os geradores de condições de verificação utilizam uma definição axiomática de uma linguagem de programação e uma descrição do comportamento do produto, feita através desta linguagem, para gerar sequências de condições de verificação. Estas condições deverão ser provadas por um provador de teoremas. Um exemplo de gerador de condições de verificação pode ser encontrado no ambiente GYPSY [COH86].

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Os provadores de teoremas provam teoremas sobre o domínio do produto de software. São utilizados para verificar a consistência, a completude e a corretude do produto e para verificar se as transformações efetuadas a partir de uma representação anterior do produto, estão corretas. Existem vários enfoques para provadores de teoremas, variando de acordo com a quantidade e o estilo empregado para as interações com o usuário - de procedimentos totalmente automatizados a provadores baseados em técnicas de Inteligência Artificial, que utilizam heurísticas e grandes bases de conhecimento. Um exemplo de provador de teoremas é o "Interactive Theorem Prover", integrado a um ambiente formal para projetos de sistemas especificados na linguagem de especificação INA JO [BER87].

Os sistemas que proporcionam verificação de produtos geralmente apresentam as duas ferramentas acima citadas. Dentre eles podemos citar: (1) o sistema MUSE [HAL87], que consiste de um sistema de verificação desenvolvido para especificações formais em HDM; e (2) o sistema AFFIRM [HOF88, MUS80], que aceita tipos abstratos de dados e especificações algébricas, além de uma linguagem de programação baseada em Pascal.

A seguir são apresentadas algumas das principais técnicas de verificação. Destas, as primeiras são utilizadas para verificar produtos prontos e a última apresenta o enfoque construtivo, no qual são gerados produtos formalmente verificados.

4.1 PROVA POR ASSERÇÕES

O método de verificação que se baseia em asserções [FAI85, BER82] foi introduzido por Floyd em 1967 e refinado por Hoare em 1973 e por Dijktra em 1976.

Uma asserção é um predicado considerado sempre verdadeiro, sendo representada por alguma linguagem de lógica similar ao cálculo de predicados. Asserções são associadas a determinados pontos do produto, representando condições que devem ser verdadeiras quando a execução atingir cada um destes pontos. As asserções representam o estado do produto no lugar considerado, ou verificam condições específicas de determinadas variáveis.

Também podem ser utilizadas asserções invariantes, ou simplesmente invariantes, as quais devem ser verdadeiras em qual-

quer ponto do produto.

À asserção de entrada de um produto podem ser adicionadas pré-condições que tratem de excessões, tais como variáveis não inicializadas, indexação inválida, "overflow" de variáveis, etc. Neste caso, o tratamento destas excessões é incorporado ao processo de verificação.

Os passos básicos de um processo de verificação através desta técnica são: (1) estabelecer as asserções; (2) construir a prova das asserções; e (3) avaliar os resultados.

A definição das asserções é feita com base na especificação do produto, dependendo diretamente de sua qualidade. É importante lembrar que não é possível definir algoritmicamente se uma especificação está completa.

As verificações realizadas através deste método são muito complexas, levando a execução de linhas de código em número muitas vezes superior às do próprio produto que está sendo verificado. Além disso, erros podem ser introduzidos neste código. A utilização de ferramentas automáticas diminui este problema.

Este método de verificação foi criado para verificar somente programas sequenciais, tendo sido estendido para cobrir também programas que apresentam processos concorrentes, através da utilização de lógica temporal. A seguir apresentamos algumas variantes do método de prova por asserções.

4.1.1 ASSERÇÕES DE ENTRADA E DE SAÍDA

São associadas asserções aos pontos de entrada e de saída, além de, eventualmente, a pontos internos do produto. A asserção de entrada define o domínio de entrada do produto. A asserção de saída define a computação que se deseja que seja realizada pelo produto.

O método que analisa as asserções de entrada e de saída estipula que, se a asserção de entrada for satisfeita por determinadas condições de entrada e se o programa termina a sua execução com estas condições de entrada, satisfazendo a asserção de saída, então o programa está correto. Este princípio pode ser representado pela regra:

$$(P)\ S\ (R)$$

onde (P) e (R) são, respectivamente, as asserções de entrada e de saída, e S representa o produto que está sendo verificado.

No caso de utilização de asserções em pontos intermediários do produto, a regra de composição da lógica permite a conjunção de predicados ao longo de um determinado caminho de execução, levando a:

$$(P)\ S_1\ (I_1)\ ,\ (I_1)\ S_2\ (I_2)\ ,\ \ldots\ ,\ (I_{N-1})\ S_N\ (R)$$

onde P é a asserção de entrada, I_1 a I_{N-1} são as asserções intermediárias, R é a asserção de saída, e S_1 a S_N são as N partes do produto, seguindo um caminho da entrada até o final de execução. Todas estas regras devem ser satisfeitas para que o produto esteja correto.

4.1.2 PRÉ-CONDIÇÕES MAIS FRACAS

Este método [FAI85, LUC87] é similar ao anterior, sendo utilizadas pré-condições mais fracas. Uma pré-condição P é denominada de pré-condição mais fraca se for a condição que caracteriza o conjunto de estados iniciais que, após a ativação do produto S, resulta em um estado final que satisfaz a pós-condição R. Representa-se da seguinte maneira:

$$P = pf(S,R)$$

A pré-condição mais fraca é encontrada analisando-se o produto a partir da pós-condição R e procurando as condições necessárias para que esta última seja satisfeita.

Segundo [BER82], Dijkstra estendeu o conceito de pré-condição mais fraca a uma classe de construções não-determinísticas, denominadas <u>comandos guardados</u>. Para estes, dado um determinado estado que satisfaz as pré-condições, pode existir mais de um caminho de execução através do produto. A execução dos comandos guardados para uma entrada que satisfaz as pré-condições mais fracas leva a um estado que satisfaz a pós-condição usada para gerar estas pré-condições.

4.1.3 LÓGICA TEMPORAL

O método de verificação baseado em lógica temporal [MOU86, PEH90] é uma extensão da prova por asserções. Se aplica à verificação de protocolos de comunicação, verificando propriedades de ativação e de progresso. São utilizadas fórmulas em lógica temporal proposicional para especificar propriedades específicas que os protocolos devem apresentar. Estas fórmulas são associadas a estados ou a sequências de estados em um sistema global de transições que representa o protocolo. A lógica temporal é uma lógica modal, isto é, uma lógica que distingue diferentes estados através da utilização de operadores modais que permitem dizer que uma proposição é verdadeira agora (no estado corrente), ou eventualmente (em algum estado que será eventualmente alcançado), ou daqui para diante (em todos os estados subsequentes), etc.

4.2 INDUÇÃO ESTRUTURAL

A indução estrutural [FAI85] é uma técnica de verificação baseada no princípio geral de indução matemática. É utilizada quando se quer provar uma proposição sobre todos os elementos de um conjunto parcialmente ordenado, bem-formado. Sendo S este conjunto e P a proposição, a prova é feita da seguinte maneira:

a) provar que P é verdadeira para o elemento de menor ordem de S;

b) assumir que P é verdadeira para o elemento de ordem N e provar que é verdadeira para o elemento de ordem N+1 de S.

As provas por indução podem ser utilizadas inclusive em produtos que apresentem recursividade.

4.3 EXECUÇÃO SIMBÓLICA

A execução simbólica [DEM87, FAI85, SAI84] é uma técnica na qual é simulada a execução do produto, utilizando valores de entrada simbólicos. Podem ser empregados valores simbólicos elementares ou expressões. Os valores simbólicos são propagados através dos possíveis caminhos de execução, sendo todas as computações e decisões expressas através destes valores simbólicos. O resultado da execução simbólica de um produto é a função de execução deste produto, que representa a sua semântica independente-

mente dos valores de entrada e de saída.

Através da execução simbólica podem ser detectados erros em caminhos de execução e, se for possível percorrer todos os possíveis caminhos (o que geralmente não acontece), provar a corretude do produto. Podem, ainda, ser detectadas anomalias em fluxos de dados. A técnica é também muito utilizada como ferramenta para gerar dados de teste.

Além disso, através da execução simbólica pode-se efetuar verificação de asserções [HAN76]. Procura-se obter fórmulas algébricas que expressem as variáveis de entrada em função das de saída. Estas fórmulas são então comparadas às asserções de entrada e de saída.

Muitas vezes o objetivo da verificação de um produto através de execução simbólica é a construção da <u>árvore de execução simbólica</u>. Nesta, a raiz modela todos os estados iniciais do produto e as folhas representam os estados finais. Cada nó representa uma classe de estado em um determinado instante. A árvore de execução simbólica representa todas as possíveis execuções do produto. Quando o produto apresenta ciclos, esta árvore se torna infinita. Os valores simbólicos das variáveis de interesse são associados a cada nó. A verificação das propriedades dos estados em um determinado nó se reduz à análise do comportamento destes valores.

A execução simbólica torna-se cada vez mais complexa a medida em que aumenta o tamanho ou a complexidade do produto analisado, fazendo-se necessária a utilização de processadores simbólicos automáticos, tais como EFFIGY [FAI85, KIN76] e SELECT [BOY75, FAI85].

A execução simbólica pode ser utilizada em verificação tanto de especificações como de programas. Como exemplo, existe uma ferramenta para verificar especificações formais escritas em INA JO através de execução simbólica [KEM85].

É apropriada para produtos sequenciais, podendo também ser utilizado para verificação de algumas propriedades de programas concorrentes (por exemplo, programas concorrentes especificados em ADA [DIL90]). Nestes casos, cada processo é executado simbolicamente, sendo verificadas isoladamente como se fossem sequenciais. Em seguida são feitas verificações nos pontos de in-

terferência dos processos.

4.4 ANÁLISE DE ALCANÇABILIDADE

É uma das técnicas mais antigas e mais comuns para a verificação de protocolos. Os protocolos de comunicação representam o comportamento de entidades em sistemas concorrentes. Muitas das diferentes técnicas de especificação utilizadas nestes sistemas representam o comportamento destas entidades através de algum sistema de transição, tal como uma máquina de estados finitos ou uma rede. A análise de alcançabilidade ("reachability analysis") [CHO88, MEN88, MOU86, PEH90] efetua uma exploração exaustiva de todas as possíveis interações entre o conjunto de processos comunicantes modelado através de um sistema de transições. É um método bastante simples, que pode facilmente ser mecanizado.

A análise da alcançabilidade de uma máquina de estados finitos se baseia no conjunto de alcançabilidade - o conjunto de todos os estados que podem ser alcançados pelo estímulo de todas as possíveis entradas da máquina. É obtido através da geração exaustiva de todos os estados alcançáveis a partir de um dado estado inicial. Este conjunto é usualmente representado através de um grafo orientado, o grafo de alcançabilidade. Este grafo contém todas as informações a respeito do comportamento lógico do protocolo, informações estas que podem ser utilizadas para verificar propriedades específicas, tais como ausência de impasses, não ocorrência de recepções não especificadas, etc.

O objetivo da verificação de um protocolo é provar que este protocolo forneça o serviço desejado. Uma maneira de provar isto é comparar a especificação do protocolo com a especificação do serviço, provando que as mesmas relações são satisfeitas nas duas especificações. As relações entre as duas especificações podem ser de equivalência ou de implementação. Relações de equivalência normalmente utilizadas são: (1) relações de equivalência de observação, quando um observador externo não pode diferenciar as duas especificações através de experimentos; (2) relações de equivalência de testes, quando a execução de testes nestas especificações não diferencia uma da outra; e (3) relações de equivalência a "trace", quando as especificações podem executar as mesmas sequências de eventos. As relações de implementação traduzem o fato de que, se um processo implementa outro, então tudo que o primeiro processo pode fazer também pode ser feito pelo segundo,

não sendo a recíproca verdadeira.

A maior dificuldade na análise de alcançabilidade é a explosão de estados causada pela complexidade exponencial dos sistemas. Várias técnicas foram desenvolvidas para contornar este problema. Entre elas podemos citar [CHO88]: (1) "fair reachability analysis", aplicável a uma rede formada por duas máquinas de estados finitos, na qual cada arco do grafo é rotulado simultaneamente por duas transições, uma para cada máquina considerada; (2) projeção de protocolos [CHE86, LAM84], onde são construídos protocolos-imagens, os quais agregam os estados, mensagens e eventos do protocolo original, reduzindo a análise para vários protocolos mais simples; e (3) "multiphase protocol construction", que separa a funcionalidade de um protocolo em funções separadas, construindo uma rede de máquinas de estados finitos (uma "fase") para cada uma das funções identificadas, conectando depois estas máquinas individuais para formar o protocolo desejado.

Como exemplos de ambientes que implementam esta técnica de verificação temos:

a) o ambiente PROSPEC [CHO88], para projeto e verificação de protocolos de forma interativa, que implementa a verificação através de um grafo de alcançabilidade;

b) o ambiente SPANNER [AGG88], que simula a execução destas especificações e permite a sua verificação através de análise de alcançabilidade.

4.5 DESENVOLVIMENTO DE PRODUTOS FORMALMENTE VERIFICAVEIS

O objetivo principal desta técnica [ALV88, GOL86, MAI87] é verificar a consistência e a completeza do produto durante a sua construção. Sua realização é efetuada através de um sistema que controla as informações que estão sendo definidas. Este sistema detém o conhecimento do método utilizado na especificação/implementação e do domínio da aplicação, conhecimento este expresso através de regras que são avaliadas a cada nova informação definida. As regras de verificação da consistência impedem que informações inconsistentes sejam definidas. As de verificação de completude podem interagir com o usuário, indicando o que falta para que a definição esteja completa. O sistema guia a especificação/implementação, verificando a correção do produto

com base nas regras de consistência.

Os ambientes de especificação de Sistemas de Informação baseados em bancos de dados clássicos normalmente não possuem ferramentas para assegurar a qualidade dos sistemas gerados. Com esta finalidade estão sendo utilizados ambientes de resolução de problemas centrados em bases de conhecimento. Se baseiam em técnicas de Inteligência Artificial e de Sistemas Especialistas.

Como exemplos de ambientes que permitem o desenvolvimento de produtos já verificados podemos citar:

a) C-TODOS [PER90], que permite ao projetista definir especificações para Sistemas de Informação de Escritórios, automaticamente verificando sua corretude e consistência com elementos já definidos no banco de dados que armazena a especificação;

b) a técnica de "rewrite rule theorem provers" [COH86] utilizada em AFFIRM, a qual utiliza provadores de teoremas que verificam a consistência dos axiomas de uma especificação algébrica, a medida em que estes vão sendo definidos;

c) PENELOPE [RAM89] - um editor de programas protótipos, que apoia o desenvolvimento de programas e de provas a partir de especificações na linguagem LARCH/ADA-88, para programas sequenciais.

5. TÉCNICAS DE VALIDAÇÃO E DE VERIFICAÇÃO APROPRIADAS AOS DIFERENTES PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE

Os primeiros programas de computação eram escritos em linguagens baseadas em um paradigma de programação que reflete a estrutura da máquina de von Neumann: uma lista de instruções executadas passo a passo, em uma determinada sequência. Estas linguagens, chamadas de imperativas, são ainda hoje as mais utilizadas. Sua principal característica é a existência de um estado implícito, modificado pelas construções (comandos) utilizadas.

A crescente complexidade dos sistemas de computação e a necessidade de produção econômica de sistemas confiáveis levaram ao desenvolvimento de novas técnicas para o desenvolvimento de software, baseadas em paradigmas não convencionais [GHE82, MAC87], como os de orientação a funções, a objetos e a lógica. Nestes são utilizadas linguagens declarativas, que se caracterizam por não possuir um estado implícito. Através delas é construído um modelo do problema. Enquanto que as imperativas expressam como uma determinada computação deve ser executada, as declarativas expressam o que está sendo computado. Cada um destes novos paradigmas possui técnicas específicas de desenvolvimento, utilizando diferentes mecanismos de abstração, com ambientes e ferramentas apropriados. O paradigma mais apropriado a um problema específico depende da área de aplicação do problema e das ferramentas existentes.

O objetivo deste capítulo é analisar os principais paradigmas de desenvolvimento de software, procurando identificar quais as técnicas de validação e de verificação que são mais apropriadas a cada um deles. É feita uma breve apresentação das principais características de cada um dos paradigmas, seguida da análise de quais das técnicas vistas melhor se adaptam a eles.

5.1 IMPERATIVO

É o paradigma tradicional de desenvolvimento de software. Divide um programa em duas partes distintas: (1) uma parte declarativa, onde são descritas as variáveis que serão alocadas em tempo de compilação; e (2) outra imperativa na qual, através de comandos e de uma forma procedimental, é descrita a execução do programa. A unidade principal destes programas é o comando. A solução de um problema é, na realidade, a descrição de um proce-

dimento.

Todas as linguagens tradicionais de programação, tais como FORTRAN, PASCAL e C, baseiam-se neste neste paradigma.

A maior parte das técnicas de validação e de verificação vistas nos capítulos anteriores foi desenvolvida visando à aplicação em programas imperativos, uma vez que estes são os mais utilizados hoje em dia.

Das <u>técnicas de validação</u>, a mais utilizada é a de testes estruturais. A menos utilizada é a prototipação rápida, uma vez que os programas desenvolvidos segundo este paradigma são geralmente especificados em linguagens de especificação também imperativas, que não apresentam ferramentas de prototipação.

Das <u>técnicas de verificação</u>, somente a execução simbólica se apropria integralmente a este paradigma. As provas de correção de programas imperativos são muito complexas, sendo pouco utilizadas. O desenvolvimento de produtos formalmente verificados se baseia geralmente em uma base de conhecimentos, sendo esta inapropriada para programas procedimentais.

5.2 ORIENTAÇÃO A FUNÇÕES

No paradigma de desenvolvimento de software orientado a funções [HUD89, LEA74, LOY90, MEI87, NET89] um programa é visto como uma função encarregada de transformar os valores de entrada nos valores de saída. No desenvolvimento de um produto segundo este paradigma são considerados os aspectos funcionais do produto: que funções o produto deve executar, que subfunções são necessárias para a execução destas funções, quais as funções que cada uma das partes do produto vai executar, como estas funções serão executadas. É uma técnica de desenvolvimento "top-down".

As funções são definidas através da utilização de operações sobre funções, sendo novas funções definidas a partir de outras mais simples ou de funções pré-definidas.

As funções e os porcedimentos para a sua execução são os aspectos principais no paradigma funcional, sendo os dados secundários. Os dados são independentes das funções ou, no máximo, associados aos componentes funcionais.

As técnicas estruturadas [DEM79, DIJ76] se apoiam em uma filosofia de desenvolvimento de sistemas que analisa o sistema sob ponto de vista funcional.

A primeira linguagem funcional foi λ-Calculus, tendo influenciado o desenvolvimento das demais. Entre as linguagens atualmente em uso, a que mais se aproxima do conceito de linguagem funcional é LISP. A linguagem MIRANDA [TUR86] é puramente funcional, não apresentando qualquer efeito colateral nem aspectos imperativos. Também as linguagens de consulta a bancos de dados (SQL, QUEL) se baseiam no paradigma funcional.

Das <u>técnicas de validação</u> vistas, os testes funcionais são os mais apropriados a produtos desenvolvidos segundo este paradigma, podendo cada função ser testada isoladamente antes de serem avaliadas suas inter-relações. Os "walkthrough's" e as inspeções, embora desenvolvidos e utilizados em programas imperativos, podem igualmente ser utilizados em programas funcionais, devendo serem alterados os itens a serem avaliados.

A centralização em funções também propicia facilidades para prototipação. Existem, por exemplo, especificações executáveis usando linguagens funcionais, como no caso de PAISLey [ZAV82], que servem tanto para testar a especificação como para validá-la junto aos usuários.

A execução simbólica é a <u>técnica de verificação</u> que mais se apropria a produtos desenvolvidos segundo este paradigma, uma vez que através dela se obtem expressões que representam as funções consideradas. Como exemplo de sua utilização podemos citar SELECT [BOY75, FAR85], um processador de execuções simbólicas para programas em LISP. Outras técnicas de verificação vistas também se aplicam a produtos funcionais. Em [MEI87], por exemplo, é apresentada uma prova de correção formal completa para um programa funcional, cuja base de prova é um sistema natural de dedução. Ainda segundo [MEI87], já existem vários provadores implementados: "Boyer-Moore Theorem Prover" para funções LISP, LCF e VERITAS. O desenvolvimento de produtos formalmente verificados também pode ser utilizado, devendo as funções serem armazenadas em uma base de conhecimentos, de modo a permitir que o desenvolvimento do sistema seja guiado pelo conteúdo desta base [ALV88].

5.3 ORIENTAÇÃO A OBJETOS

O paradigma de desenvolvimento com orientação a objetos [HOR88, LOY90, ROB81, STR87, TAK88] é aquele que mais interesse tem despertado atualmente nos meios de pesquisa. O desenvolvimento de um sistema segundo este paradigma envolve desde o processo de análise do "espaço do problema", incluindo captura e organização de informações, até a modelagem do problema no "espaço de soluções". Cobre, portanto, todo o ciclo de desenvolvimento de um produto.

Este paradigma se baseia no princípio de que os problemas envolvem objetos do mundo real, com determinados comportamentos evoluindo no tempo. O enfoque principal é dado à identificação e representação destes objetos. Um objeto é uma entidade do mundo real, com capacidade de armazenar e manipular informação. É caracterizado por um conjunto de atributos (ou propriedades) que definem a sua estrutura, e por um conjunto de procedimentos (operações ou métodos) que definem o seu comportamento. Os objetos podem ser agrupados em classes de objetos similares, sendo um objeto encarado como uma instância da classe. Uma classe é também encarada como um objeto, podendo ser formadas hierarquias de classes, através de mecanismos de abstração tais como generalização, especialização, associação e agregação, com herança de atributos e de operações. Relações representam as associações lógicas entre elementos de classes. A solução de um problema é, portanto, obtida pelos possíveis inter-relacionamentos entre classes de objetos reais.

Segundo este paradigma, os dados (objetos) são a principal informação, sendo os procedimentos executados sobre estes dados encarados como secundários - o oposto do que ocorre no paradigma de orientação a funções. O desenvolvimento, segundo este paradigma, é do tipo "bottom-up" - inicialmente são definidas as classes de objetos, depois os inter-relacionamentos entre elas.

Um programa que utiliza uma linguagem de orientação a objetos é composto da definição das classes de objetos e de suas propriedades. Sua execução é iniciada pela criação de instâncias destas classes, seguida da manipulação destas instâncias através de troca de mensagens entre os objetos.

O conceito de Tipos Abstratos de Dados (TAD) é às vezes confundido com orientação a objetos. Segundo [NET89], na utiliza-

ção de TAD as operações que manipulam os objetos pertencentes a um tipo fazem parte do tipo, enquanto que na orientação a objetos estas operações são associadas diretamente aos objetos. Existem diversas técnicas desenvolvidas para especificações com TAD, tais como especificações algébricas [KLA83], OBJ [GOG82, GOG87] e AFFIRM [MEN88].

A linguagem que mais se apropria ao paradigma de desenvolvimento orientado a objetos é SMALLTALK [GOL84], encarada como uma linguagem de orientação a objetos pura. Além disso, linguagens procedimentais foram adaptadas para atender a este paradigma. Entre elas podemos citar C^{++} [STR86] baseada na linguagem C e COMMON LISP OBJECT [BOB88], baseada em LISP. Várias linguagens orientadas a objetos tem sido desenvolvidas, incluindo facilidades para sua aplicação em programas desenvolvidos segundo outros paradigmas - é o caso, por exemplo, da linguagem BETA [MAD88], que também pode ser utilizada em programas procedimentais e funcionais.

Vários ambientes de desenvolvimento de software orientado a objetos estão sendo desenvolvidos. Apresentam modelos orientados a objetos, manipulados por ferramentas próprias para geração, validação e verificação. Entre eles podemos citar o projeto TODOS [HEI88, PER90], cujo modelo conceitual é orientado a objetos, representados através de uma linguagem própria ou através de uma representação gráfica.

Das <u>técnicas de validação</u> vistas, a prototipação é a mais adequada. Os objetos e suas variáveis de instâncias de seus atributos são amarrados em tempo de execução, o que favorece a prototipação. O ambiente TODOS, acima citado, apresenta uma ferramenta de prototipação do modelo conceitual. Além disso, o paradigma de objetos se apropria à reutilização de software - de objetos já definidos - o que proporciona economia no tempo de desenvolvimento de protótipos.

Testes funcionais também podem ser empregados. Inicialmente, cada classe criada deve ser testada individualmente. O teste de uma classe isolada pode ser efetuado da seguinte maneira [MAT88b]: (1) substituir as mensagens enviadas a outras classes por uma mensagem externa (exibição para o depurador) e pela inicialização dos parâmetros que estariam na mensagem de volta; (2) ativar cada um dos procedimentos da classe através do envio de mensagens e verificar os resultados obtidos. Uma vez testadas as

classes isoladas, elas devem ser integradas, sendo testados os inter-relacionamentos entre elas em função das tarefas requeridas do sistema. Deve ser efetuado o seguinte procedimento: (1) integrar e testar as classes de objetos que implementam cada tarefa; (2) integrar e testar conjuntamente as classes de objetos que compõem a totalidade das tarefas do sistema.

Das <u>técnicas de verificação</u> vistas, o desenvolvimento de produtos formalmente verificados poderá ser empregado se fôr utilizado no desenvolvimento um ambiente suportado por uma base de conhecimentos. Esta base de conhecimentos deverá conter informações a respeito da correta definição de novos objetos e dos inter-relacionamentos válidos na aplicação considerada. Existem provas de correção para Tipos Abstratos de Dados, mas nada foi encontrado especificamente para objetos.

5.4 <u>ORIENTAÇÃO A LÓGICA</u>

O desenvolvimento do paradigma de orientação a lógica se deu em consequência ao estudo de correção de teoremas. A utilização da lógica para a solução de problemas [CAS87, KOW79, MEI87] se baseia: (1) no conhecimento do domínio do problema, representado através de fatos em uma base de conhecimentos; e (2) em regras (sentenças em lógica) que permitem deduzir novas informações a partir daquelas armazenadas na base de conhecimentos.

Um programa em lógica é, portanto, um modelo do problema representado através de um conjunto de fatos e de regras. Sua execução constitui uma tentativa de provar a validade de algum predicado. É composto de uma série de asserções que descrevem o problema para o qual se procura uma solução. Estas asserções ou são verdadeiras, ou se tornam verdadeiras através do instanciamento de uma ou mais variáveis. O mecanismo utilizado na computação é o da inferência, através da qual se pode descobrir novos fatos sobre a área de aplicação de sua base de conhecimentos. Apresenta duas partes distintas, fisicamente separadas: o controle e a lógica propriamente dita. Esta separação facilita a prova de correção, pois somente a parte de lógica precisa ser testada.

Para este paradigma, a linguagem mais utilizada é o PROLOG, embora não seja puramente lógica: a lógica e o controle não estão totalmente separados (problemas com o "corte", conhecimento e utilização da ordem de execução das regras, etc.), além

de ser possível fazer atribuições (operador IS).

Das técnicas de validação vistas, a mais apropriada a este paradigma é a prototipação. A implementação de protótipos através, por exemplo, da linguagem PROLOG, é bastante fácil e rápida, permitindo a validação de aspectos relevantes dos requisitos. Existem ferramentas que implementam protótipos rápidos em linguagens de lógica, sem se preocupar com eficiência na execução.

"Walkthrough's" e inspeções também podem ser utilizados para validação, embora não seja comum a estas aplicações.

O paradigma de orientação a lógica é muito apropriado à aplicação das técnicas de verificação, em especial a provas de asserções. A forma habitual de definir asserções e pré-condições é através de fórmulas escritas em alguma linguagem de lógica. Os provadores de teoremas baseados em técnicas de Inteligência Artificial utilizam representações de conhecimento baseadas em lógica. Um produto desenvolvido com base no paradigma de orientação à lógica tem, portanto, todas as condições necessárias à sua verificação através destas técnicas.

O desenvolvimento de produtos formalmente verificáveis também pode ser facilmente utilizado, uma vez que utiliza bases de conhecimento e técnicas de Inteligência Artificial, ferramentas apropriadas a produtos baseados neste paradigma.

Das técnicas de verificação, a menos utilizada em produtos desenvolvidos segundo este paradigma é a execução simbólica. As ferramentas disponíveis para esta técnica são geralmente voltadas a produtos tradicionais, que seguem o paradigma imperativo. Nada impede, entretanto, que seja utilizada em produtos orientados a lógica, tais como programas PROLOG, sendo somente necessária a construção de ferramentas apropriadas.

6. CONCLUSÃO

Durante todo o ciclo de desenvolvimento de um produto de software devem ser efetuadas validações e verificações, de modo a garantir um produto final mais confiável, de melhor qualidade, com menor número de erros e que realmente satisfaça os requisitos solicitados pelo usuário. É importante que a avaliação do produto seja efetuada ao longo de todas as fases de seu desenvolvimento e não somente ao final, quando este estiver completo. Deste modo, problemas de requisitos mal formulados ou mal expressos, erros de especificação, etc. podem ser detectados logo, evitando o desenvolvimento completo com a propagação destes erros.

Diversas técnicas de validação e de verificação podem ser empregadas no desenvolvimento de produtos de software. A escolha da melhor técnica vai depender das características do produto que está sendo desenvolvido e, principalmente, das ferramentas disponíveis para sua aplicação. O paradigma de desenvolvimento de software empregado também influi na escolha das técnicas que serão utilizadas.

Este trabalho apresenta um estudo a respeito das técnicas mais utilizadas nos dias de hoje - tanto de validação como de verificação. Procurou-se identificar em que fases do desenvolvimento de produto cada uma das técnicas apresentadas pode ser aplicada e para que tipo de produto deve ser empregada - sequencial e/ou concorrente.

Dentre as possíveis técnicas de validação, foram identificadas como as mais comuns "walkthrough's", inspeções, testes e a prototipação. As duas primeiras se caracterizam por serem técnicas estáticas, nas quais é analisado o texto do produto, não sendo efetuada qualquer execução. Os testes requerem a execução do produto com determinados valores de entrada, dependendo os resultados obtidos do conjunto de valores selecionados. É a técnica mais utilizada nos produtos desenvolvidos atualmente. A prototipação se constitui em uma forte ferramenta de validação dos requisitos, colocando o usuário em contato direto com o sistema e permitindo que avalie se especificou corretamente o que desejava.

As técnicas de verificação não são tão utilizadas quanto as de validação devido à sua complexidade implícita. Neste trabalho foram apresentadas as características de prova por asserção, indução estrutural, execução simbólica, análise de alcan-

çabilidade e do desenvolvimento de produtos formalmente verificáveis. Dentre elas, uma das mais utilizadas é a execução simbólica, através da qual é possível simular a execução do produto com valores de entrada simbólicos, permitindo a verificação dos possíveis caminhos de execução do produto. A análise de alcançabilidade é outra técnica muito utilizada, sendo específica para produtos concorrentes. Através desta técnica são analisadas todas as possíveis interações entre um conjunto de processos comunicantes, modelado através de um sistema de transições. As técnicas de prova por asserções e de indução estrutural, que empregam métodos matemáticos para provar a correção dos produtos, são bastante complexas e difíceis de serem utilizadas. Uma nova tendência é o desenvolvimento de produtos formalmente verificáveis, nos quais a verificação é feita simultaneamente ao desenvolvimento. Sendo uma técnica nova, apresenta poucas ferramentas implementadas.

O trabalho apresenta, também, uma análise de quais as técnicas mais apropriadas ao desenvolvimento de produtos segundo os principais paradigmas de desenvolvimento de software: imperativo, orientado a funções, orientado a objetos e orientado a lógica.

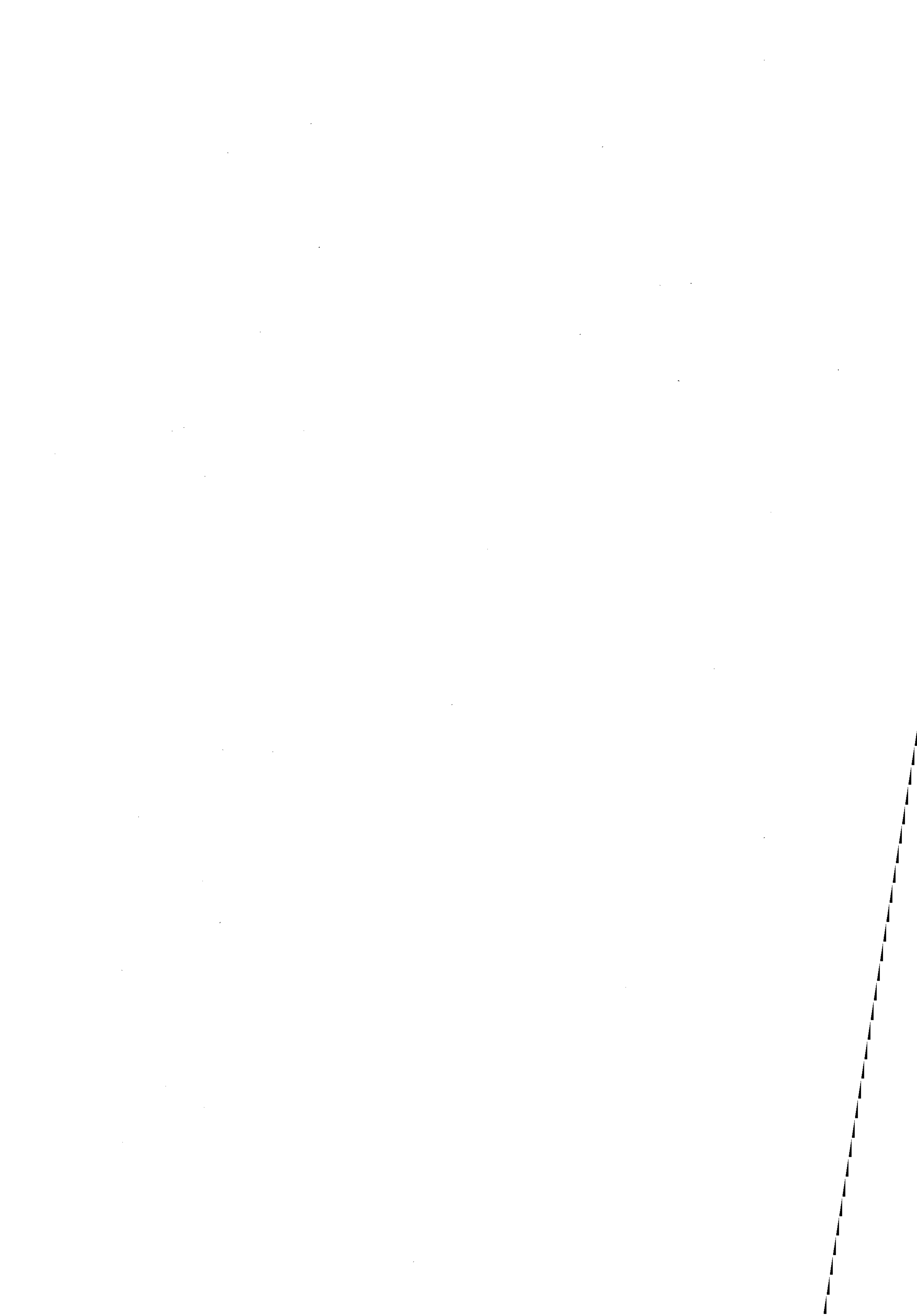
O paradigma imperativo, sendo o tradicional, é aquele que apresenta o maior número de técnicas disponíveis para validação e verificação. Na validação são empregados principalmente testes estruturais, existindo grande número de técnicas e de ferramentas para determinar os valores de entrada apropriados. "Walkthrough's" e inspeções foram criadas especialmente para a validação deste tipo de produto. A execução simbólica é a técnica mais utilizada para a verificação destes produtos.

Os testes estruturais são a técnica mais apropriada para a validação de produtos desenvolvidos segundo o paradigma de orientação a funções, sendo também muito utilizada a prototipação. Para a verificação é indicada a execução simbólica.

A validação de produtos desenvolvidos segundo o paradigma de orientação a objetos geralmente é efetuada através de prototipação, podendo também ser empregados testes funcionais. Para a verificação é indicado o desenvolvimento de produtos formalmente verificáveis.

Para os produtos desenvolvidos segundo o paradigma de orientação a lógica, a técnica mais apropriada para validação é a

prototipação. As técnicas de verificação através de provas por asseções e por indução estrutural podem ser empregadas nestes produtos, embora não sejam muito utilizadas. O desenvolvimento de produtos formalmente verificáveis também é uma técnica que se apropria a este paradigma de desenvolvimento.



REFERÊNCIAS BIBLIOGRÁFICAS

[AGG88] AGGARWAL,S.; BARBARÁ,D.; METH,K.Z. A Software environment for specification and analysis of problems of coordination and concurrency. IEEE Transactions on Software Engineering, New York, 14(3):280-90, Mar. 1988.

[ALV88] ALVARES,L.O.C. Contribution à l'étude du pilotage de la modelisation des systèmes d'information. Grenoble, Université Joseph Fourier, 1988. 225p. (Tese de doutorado).

[AND86] ANDRIOLE,S.J. (ed.) Software validation, verification,testing and documentation. Princeton, Petrocelli Books, 1986. 389p.

[BAL89] BALCER,M.; HASLING,W.; OSTRAND,T. Automatic generation of test subscript from formal test specifications. Software Engineering Notes, New York, 14(8):210-8, Dec. 1989.

[BER82] BERG,H.K. et al. Formal methods of program verification and specification. New Jersey, Prentice-Hall, 1982. 207p.

[BER87] BERRY,D.M. Towards a formal basis for the Formal Development Method and the Ina Jo specification language. IEEE Transactions on Software Engineering, New York, SE-13(2):184-201, Feb. 1987.

[BIL88] BILLINGTON,J; WHEELER,G.R.; WILBUR-HAM,M.C. PROTEAN: A high-level Petri Net tool for the specification and verification of communication protocols. IEEE Transactions on Software Engineering, New York, 14(3):301-16, Mar. 1988.

[BIS89] BISANT,D.B & LYLE,J.R. A Two-person inspection method to improve programming productivity. IEEE Transactions on Software Engineering, New York, 15(10):1294-1304, Oct. 1989.

[BOB88] BOBROW,D.G. Common Lisp Object. Sigplan Notices, New York, 23:1.1-1.48, Sept. 1988.

[BOC90] BOCHMANN,G. Protocol specifications for OSI. Computer Networks and ISDN Systems, Netherlands, 18(3):167-84, Apr. 1990.

[BOE84] BOEHM,B.W. Verifying and validating software requirements and design specifications. IEEE Software, Los Alamitos, 1(1):75-88, Jan. 1984.

[BOY75] BOYER, R.S.; ELPAS,B.; LEVITT,K.N. SELECT - A Formal system for testing and debugging programs by symbolic execution. SIGPLAN Notices, New York, 10(6):234-45, June 1975.

[CAM88] CAMERON,E.J. et al. The IC* Model of parallel computation and programming environment. IEEE Transactions on Software Engineering, New York, 14(3):317-26, Mar. 1988.

[CAS87] CASANOVA,M.A.; GIORNO,F.A.C.; FURTADO,A.L. Programação em lógica e a linguagem Prolog. São Paulo, Edgard Blucher, 1987. 461p.

[CLA89] CLARKE,L.A.; RICHARDSON,D.J.; ZEIL,S.J. TEAM: A Support environment for testing, evaluation, and analysis. SIGPLAN Notices, New York, 24(2):153-62, Feb. 1989.

[CHE86] CHEUNG,T.-Y. On the Projection method for protocol verification. IEEE Transactions on Software Engineering, New York, SE-12(11):1088-89, Nov. 1986.

[CHO88] CHOW,C.-H. & LAM,S.S. PROSPEC: an Interacting programming environment for designing and verifying communication protocols. IEEE Transactions on Software Engineering, New York, 14(3):327-38, Mar. 1988.

[CHO89] CHOI,B.; MATHUR,A.P.; PATTISON,B. P^{M}othra: Scheduling mutants for execution of a hypercube. Software Engineering Notes, New York, 14(8):58-65, Dec. 1989.

[COH86] COHEN,B.; HARWOOD,W.T.; JACKSON,M.I. The Specification of complex systems. Great Britain, Addison-Wesley, 1986. 143p.

[DEM79] DEMARCO,T. Structured analysis and system specification. New York, Yourdon, 1979. 353p.

[DEM87] DEMILLO,R.A. et al. Software testing and evaluation. Menlo Park, Benjamin/Cummings, 1987. 537p.

[DEU81] DEUTSCH,M.S. Software project verification and validation. Computer, Los Alamitos, 14(4):54-70, Apr. 1981.

[DIL90] DILLON,L.K. Verifying general safety properties of Ada tasking programs. IEEE Transactions on Software Engineering, New York, 16(1):51-63, Jan. 1990.

[DIJ76] DIJKTRA,E.W. A Discipline of programming. Englewood Cliffs, Prentice-Hall, 1976. 217p.

[FAG86] FAGAN,M.E. Advances in software inspections. IEEE Transactions on Software Engineering, New York, SE-12(7):744-51, July 1986.

[FAI85] FAIRLEY,R. Software engineering concepts. Singapore, McGraw-Hill, 1985. 364p.

[GHE82] GHEZZI,C. & JAZAYERI,M. Programming language concepts. New York, John Wiley & Sons, 1982. 428p.

[GOG82] GOGHEN,J.A. & MESEGUER,J. Rapid prototyping in OBJ executable language. Software Engineering Notes, New York, 7(5):75-84, Dec. 1982.

[GOG87] GOGHEN,J.A. Principles of parameterized programming. Stanford University/Center of Study of Language and Information, 1987. 67p.

[GOL84] GOLDBERG,A. Smalltalk-80: The Interactive programming environment. Reading, Addison-Wesley, 1984. 516p.

[GOL86] GOLDBERG,A.T. Knowledge-based programming: a survey of program design and construction techniques. IEEE Transactions on Software Engineering, New York, SE-12(7):752-68, July 1986.

[HAL87] HALPERN,J.D. et al. Muse - A Computer assisted verification system. IEEE Transactions on Software Engineering , New York, SE-13(2):151-6, Feb. 1987.

[HAN76] HANTLER,S.L. & KING,J.C. An Introduction to proving the correctness of programs. ACM Computing Surveys, Baltimore, 8(3):331-53, Sept. 1976.

[HEI88] HEIJMINK,F. et al. Development of tools for designing OIS. In: BULLIGER,U.-J. et al (eds.), Information Technology for Organizational Systems, Brussels, Elsevier Science Publishers (North-Holland), 1988. p.66-73.

[HOF88] HOFFMAN,D. & SNODGRASS,R. Trace specifications: methodology and models. IEEE Transactions on Software Engineering, New York, 14(9):1243-52, Sept. 1988.

[HOR88] HORN,B.L. An Introduction to Object-Oriented programming, inheritance and method combination. Pittsburgh, CMU, 1988. 30p. (Relatório de pesquisa).

[HOW80] HOWDEN,W.E. Functional program testing. IEEE Transactions on Software Engineering, New York, SE-6(2):160-9, Mar. 1980.

[HOW86] HOWDEN,W.E. A Functional approach to program testing and analysis. IEEE Transactions on Software Engineering, New York, SE-12(10):997-1005, Oct. 1986.

[HOW89] HOWDEN,W.E. Validating programs without specifications. Software Engineering Notes, New York, 14(8):2-9, Dec.1989.

[HOW90] HOWDEN,W.E. Comments analysis and programming errors. IEEE Transactions on Software Engineering, New York, 16(1):72-81, Jan. 1990.

[HUA75] HUANG,J.C. An Approach to program testing. ACM Computing Surveys, New York, 7(3):113-28, Sept. 1975.

[HUD89] HUDAK,P. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, New York, 21(3):359-411, Sept. 1989.

[JAR88] JARD,C.; MONIN,J-F.; GROZ,R. Development of Véda, a prototyping tool for distributed algorithms. IEEE Transactions on Software Engineering, New York, 14(3):339-52, Mar. 1988.

[JEN89] JENG,B. & WEYUKER,E.J. Some Observations on partition testing. Softsare Engineering Notes, New York, 14(8):38-47, Dec. 1989.

[KEM85] KEMMERER,R.A. Testing formal specifications to detect design errors. IEEE Transactions on Software Enghineering, New York, SE-11(1):32-42, Jan. 1985.

[KIN76] KING,J.C. Symbolic execution and program testing. Communications of the ACM, New York, 19(7):385-94, July 1976.

[KLA83] KLAEREN,H.A. Algebraiche Spezification - eine Einfuhrung. Berlin, Springer-Verlag, 1983. 235p.

[KOT89] KOTIK,G.B. & MARKOSIAN,L.Z. Automating software analysis and testing using a program transformation system. Software Engineering Notes, New York, 14(8):75-84, Dec. 1989.

[KOW79] KOWALSKI,R. Logic for problem solving. New York, North-Holland, 1979. 287p.

[LAM84] LAM,S.S. & SHANKAR,A.U. Protocol verification via projections. IEEE Transactions on Software Engineering, New York, SE-10(4):325-42, July 1984.

[LAS82] LASKY,J. On data flow guided program testing. SIGPLAN Notices, New York, 17:62-71, Sept. 1982.

[LAS83] LASKY,J. & KOREL,B. A Data flow oriented program testing strategy. IEEE Transactions on Software Engineering, New York, SE-9(3):347-54, May 1983.

[LEA74] LEAVENWORTH,B.M. Nonprocedural programming. In: Programming methdology. Berlin, Springer-Verlag, 1974. p.362-385.

[LEE86] LEE,A.F. A Plan for prototyping. In: Computer Programming Management, Auerbach, 1986. p.1-10.

[LIN90] LINN JR,R.J. Conformance testing for OSI protocols. Computer Networks and ISDN Systems, Netherlands, 18(3):203-20, Apr. 1990.

[LON85] LONGWORTH,G. Padrões em Programação. Rio de Janeiro, Campus, 1985. 234p.

[LOY90] LOY,P.H. A Comparison of object-oriented structured development methods. Software Engineering Notes, New York, 15(1):44-49, Jan. 1990.

[LUC87] LUCENA,C.J.P. Inteligência artificial e engenharia de software. Rio de Janeiro, Jorge Zahar, 1987. 305p.

[MAC87] MACLENNAN,B.J. Prociples of programming languages: design, evaluation, and implementation. New York, CBS College Publishing, 1987. 568p.

[MAD88] MADSEN,O.L. & MOLLER-PEDERSEN,B. What object-oriented programming may be - and what it does not have to be. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING. Oslo, Aug. 15-17, 1988. Proceedings. Berlin, Springer-Verlag, 1988. p.1-20.

[MAI87] MAIOCCHI,R. & PERNICI,B. Verification and refinement of office procedures. In: IEEE COMPUTER SOCIETY OFFICE AUTOMATION SYMPOSIUM, Gaithersburg, Apr. 27-29, 1987. Proceedings. Washington, IEEE, 1987. p.206-16.

[MAT88a] MATTHEWS,R.S.; MURALIDHAR,K.H.;SPARKS,S. MAP2.1 Conformance testing tools. IEEE Transactions on Software Engineering, New York, 14(3):363-74, Mar. 1988.

[MAT88b] MATTOSO,A.L.Q & BLUM,H. Proposta de desenvolvimento de software com orientação a objetos. In: SIMPóSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 2., Canela, 27-28 out. 1988. Anais. p.7-17.

[MEI87] MEIRA,S.L. Programação funcional. Sociedade Brasileira de Computação, 1987. 34p.

[MEN88] MENDES,S.B.T. & AGUIAR,T.C. Métodos para especificação de sistemas. Curitiba, Ebai, 1988. 288p. (III EBAI).

[MIL84] MILLER,E.F. Software testing technology: an overview. In: VICK,C.R. & RAMAMOORTHY,C.V. (eds.) Handbook of software engineering. New York, Van Nostrand, 1984. p.359-79.

[MOU86] MOURA,J.A.B. et al. Redes locais de computadores. São Paulo, McGraw Hill, 1986. 446p.

[MUS80] MUSSER,D.R. Abstract data type specification in the AFFIRM System. IEEE Transactions on Software Engineering, New York, SE-6(1):24-32, Jan. 1980.

[NET89] NETTO,J.L.M.R. Projeto de linguagens de programação. Rio de Janeiro, Sociedade Brasileira de Computação, 1989. 73p.

[NTA84] NTAFOS,S.C. On Required element testing. IEEE Transactions on Software Engineering, New York, SE-10(6):795-803, Nov. 1984.

[NTA88] NTAFOS,S.C. A Comparison of some structural testing strategies. IEEE Transactions on Software Engineering, New York, 14(6):868-74, Jun. 1988.

[OST88] OSTRAND,T.J. & BALCER,M.J. The Category-Partition Method for specifying and generating functional tests. Communications of the ACM, New York, 31(6):676-86, June 1988.

[PEH90] PEHRSON,B. Protocol verification for OSI. Computer Networks and ISDN Systems, Netherlands, 18(3):185-201, Apr. 1990.

[PER89] PERNICI,B. et al. C-TODOS: An Automatic tool for office system conceptual design. ACM Transactions on Information Systems, New York, 7(4):378-419, Oct. 1989.

[PER90] PERNICI,B. & ROLLAND,C. Automatic tools for designing Office Information Systems - the TODOS approach. Politecnico di Milano/Université de Paris I, Feb. 1990.

[PRO89] PROBERT,R.L.; URAL,H.; HORNBEEK,M.W.A. A Comprehensive software environment for developing standardized conformance test suites. Computer Networks and ISDN Systems, Netherlands, 18(1):19-30, Nov. 1989.

[RAM89] RAMSEY,N. Developping formally verified Ada programs. Software Engineering Notes, New York, 14(3):257-65, May 1989.

[RAP85] RAPPS,S. & WEYUKER,E.J. Selecting software test data using data flow information. IEEE Transactions on Software Engineering, New York, SE-11,(4):367-75, Apr. 1985.

[RIC89] RICHARDSON,D.J.; O'MALLEY,O.; TITTLE,C. Aproaches do specification-based testing. Software Engineering Notes, New York, 14(8):86-96, Dec. 1989.

[ROB81] ROBSON,D. Object-oriented systems. Byte, Peterborough, 6(8):74-86, Aug. 1981.

[SAI84] SAIB,S.H. Formal verification. In: VICK,C.R. & RAMAMOORTHY,C.V. (eds.) Handbook of software engineering. New York, Van Nostrand Reinhold, 1984. p.380-91.

[SAR89] SARIKAYA,B. Conformance testing: architectures and test sequences. Computer Networks and ISDN Systems, Netherlands, 17(2):111-26, July 1989.

[STR86] STROUSTRUP,B. The C^{++} Programming language. Menlo Park, Addison-Wesley, 1986.

[STR87] STROUSTRUP,B. What is "Object-Oriented programming"? In: BÉZIVIN,J. et al. (Eds.) ECOOP'88 - EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING. Paris, June 15-17, 1987. Proceedings. Berlin, Springer-Verlag, 1987. p.51-70.

[TAK88] TAKAHASHI,T. Introdução à programação orientada a objetos. Curitiba, EBAI, 1988. 148p.

[TAY82] TAYLOR,T. & STANDISH,T.A. Initial thoughts on rapid prototyping techniques. ACM Sigsoft Software Engineering Note, New York, 7(5):160-6, Dec. 1982.

[TAY89] TAYLOR,R.N. et al. Foundations for the Arcadia environment architecture. SIGPLAN Notices, New York, 24(2):1-13, Feb. 1989.

[TUR86] TURNER,D.A. A Non-strict functional language with polimorphic types. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE DA SBC, 13., Olinda, 19-25 jul., 1986. Anais. p. 351-66.

[WAS82] WASSERMAN,A.I. & SHEWMAKE,D.T. Rapid prototyping of interactive information systems. ACM Sigsoft Software Engineering Notes, New York, 7(5):171-80, Dec. 1982.

[WEI82] WEISER,M. Scale models and rapid prototyping. ACM Sigsoft Software Engineering Notes, New York, 7(5):181-5, Dec. 1982.

[WEY80] WEYUKER,E.J. & OSTRAND,T.J. Theories of program testing and the application of revealing subdomains. IEEE Transactions on Software Engineering, New York, SE-6(3):236-46, May 1980.

[WOO80] WOODWARD,M.R.; HEDLEY,D.; HENNELL,M.A. Experience with path analysis and testing of programs. IEEE Transactions on Software Engineering,New York, SE-6(3):278-86, May 1980.

[ZAV82] ZAVE,P. An Operational approach to requirements specification for embedded systems. IEEE Transactions on Software Engineering, New York, SE-8(3):250-69, Mar. 1982.

[ZEN89] ZENG,H.X.; CHANSON,S.T.; SMITH,B.R. On Ferry clip approaches in protocol testing. Computer Networks and ISDN Systems, Netherlands, 17(2):77-88, July 1989.