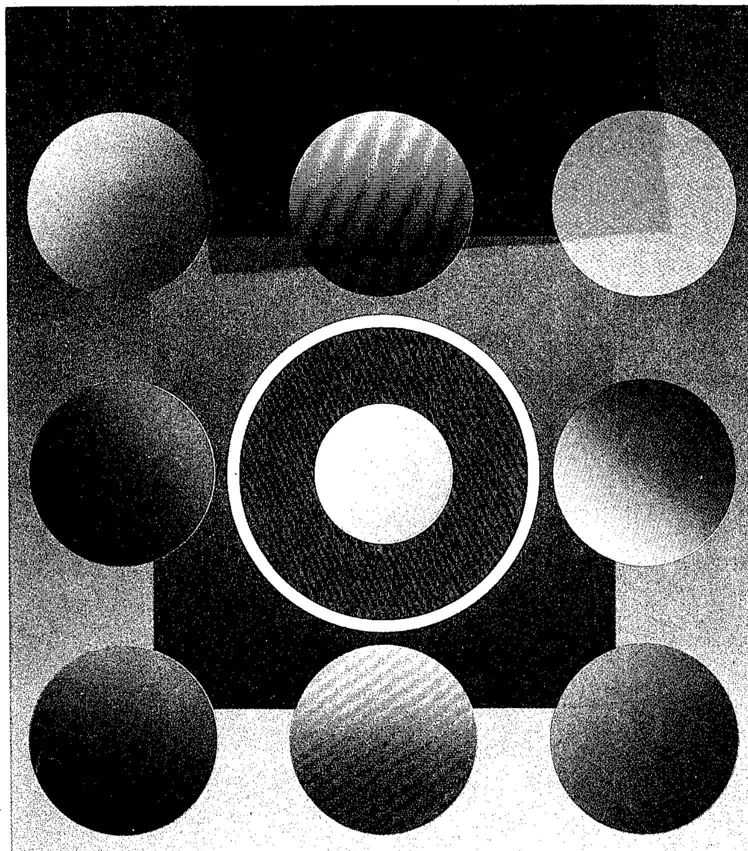
REVISTA
BRASILEIRA
DE COMPUTAÇÃO

RBC

ISSN 0101-0883

Uma publicação da SBC SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

1982, Vol. 2, Nº 2



NESTE NÚMERO

METODOLOGIA DE PROGRAMAÇÃO

LINGUAGEM PARA DESCRIÇÃO DE ARQUITETURAS

EDITORA CAMPUS

RBC, Rio de Janeiro, V. 2, nº 2, 1982, pp. 65 a 100

ESPECIFICAÇÃO DE SISTEMAS AUTOMATIZADOS

ANA REGINA CAVALCANTI DA ROCHA
ARNDT VON STAA
DEPARTAMENTO DE INFORMÁTICA
PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225
CEP 22453 – RIO DE JANEIRO – RJ – BRASIL

SUMÁRIO

São estudadas as características e produtos da fase de especificação baeado em um modelo de qualidade. São discutidas, também, as propriedades das metodologias e linguagens necessárias para o auxílio à construção de boas especificações. Tendo por base o modelo descrito, são examinadas algumas metodologias e linguagens de especificação atualmente disponíveis.
São propostas, ao final, as características de uma família de linguagens de especificação projetada para permitir o controle da qualidade de especificações.

ABSTRACT

Products and properties of the software specification phase are discussed. This discussion is based on a quality model of specifications, methodologies and specification languages. Several existing methodologies are examined from the point of view of this quality model.
We finish describing the properties of a family of specification languages designed to allow to control quality of the resulting specifications.

1. INTRODUÇÃO

O objetivo deste trabalho é definir um modelo de avaliação da qualidade de especificações examinando, após, diversas metodologias e linguagens de especificação à luz deste modelo. O modelo de qualidade empregado possui estrutura semelhante à do modelo utilizado para avaliar qualidade de software [28] [8], [38], [27]. Diferencia-se, no entanto, no que tange aos objetivos e medidas (fatores e métricas) específicos a serem utilizados para avaliar a qualidade de especificações. Para cada uma das metodologias e linguagens de especificação examinadas é dada uma pequena descrição, sendo fornecidas referências à literatura onde o leitor interessado poderá encontrar mais detalhes.

É fato conhecido que grande parte das inadequações observadas em sistemas automatizados está relacionada a erros, omissões e inadequações nas especificações. O que é mais grave, erros nas especificações detectados em estágios avançados do desenvolvimento, ou mesmo durante a produção, exercem uma forte pressão sobre os custos de construção e manutenção do *software*.

O desenvolvimento de *software* de boa qualidade depende, pois, da qualidade de suas especificações. É evidente que especificações de baixa qualidade impedirão a construção de sistemas automatizados de boa qualidade, independentemente do esmero na construção, documentação e gerência desta construção. Neste trabalho procuramos identificar as propriedades que especificações devem possuir para que consigamos desenvolver sistemas automatizados de elevada qualidade. O estudo concentrar-se-á na especificação em si e não no produto que está sendo especificado (o sistema automatizado). De posse destas propriedades e de uma análise crítica das metodologias e linguagens de expecificação existentes, esperamos ser capazes de definir uma família de linguagens de especificação homogênea, evolutível e que proporcione eficazes e econômicos instrumentos para o desenvolvimento de *software* de elevada qualidade, inclusive quando este for de pequeno ou médio porte.

Existem atualmente vários métodos ferramentas e linguagens para especificação. Estes instrumentos muitas vezes carecem de respaldo teórico, e/ou são de difícil emprego em ambientes reais. Isto põe em dúvida sua aplicabilidade. Esta dúvida é corroborada por serem incipientes e pouco conclusivos os resultados do uso experimental de muitos destes instrumentos, principalmente quando os sistemas em apreço são de pequeno e médio porte. Este estudo é, pois, oportuno já necessitamos instrumentos de especificação eficazes que assegurem que o desenvolvimento tenha elevada produtividade e leve a sistemas com qualidade média elevada sem, no entanto, comprometer os custos.

2. QUALIDADE DE SOFTWARE

Qualquer ramo de engenharia preocupa-se em desenvolver de forma econômica

produtos com qualidade preditível e controlável. Assim sendo, também em engenharia de *software* tem sido intensa a busca de melhores conhecimentos com relação à qualidade de *software*, como consegui-la e controlá-la [28], [27], [7].

Para podermos identificar os atributos de qualidade de especificações precisamos saber o que vem a ser qualidade de *software*, já que é nosso desejo construir *software* de qualidade preditível e controlada.

Para sistematizar qualidade de *software* utilizamos um modelo de predição e avaliação da qualidade [38]. Este modelo é hierárquico e baseia-se em quatro conceitos fundamentais:

- Objetivos de Qualidade - que são as propriedades gerais que o produto (*software*) deve possuir,
- Fatores de Qualidade do Produto - que são condições e características determinantes da qualidade do ponto de vista do usuário,
- Fatores de Qualidade da Engenharia - que são condições e características determinantes da qualidade da engenharia e da construção do sistema, e,
- Métricas - que são medidas quantificáveis de propriedades do *software* desenvolvido, ou a ser desenvolvido.

O modelo de qualidade é definido durante a especificação do *software* (especificação de requisitos de qualidade, critérios de sucesso). Ao definir o modelo são determinados os níveis mínimo e ideais para cada métrica e fator, bem como a forma de valorar um fator em função das métricas. Nas fases de especificação e projeto o modelo assim preenchido servirá, portanto, como condicionante do desenvolvimento permitindo ainda predizer-se se o *software* satisfazendo o modelo também satisfaz as necessidades e expectativas do usuário. Já nas fases de teste o modelo servirá de padrão contra o qual as avaliações efetuadas serão comparadas. Desta forma poder-se-á assegurar que o *software* desenvolvido tenha efetivamente o grau de qualidade esperado.

Quando nos referimos à qualidade de *software*, temos em conta os seguintes objetivos:

- Útil - o *software* produz resultados oportunos e úteis. Atingir este objetivo significa ter um produto:
  - que faz o que o usuário deseja e necessita,
  - que o faz na oportunidade certa, e,
  - é confiável.
- Utilizável - o *software* é fácil de usar e é projetado para operar em ambientes reais. Atingir este objetivo significa ter um produto:
  - que seja operável, robusto e seguro,
  - cujos dados sejam fáceis de preparar,

- cujos resultados sejam fáceis de entender e interpretar,
- que possa ser corrigido e reposto em operação após erros ou acidentes, requerendo pouco tempo e esforço para isto,
- que seja ameno ao usuário.

- Monitorável - o *software* e seus resultados podem ser continuamente avaliados em relação ao consumo de recursos e à qualidade dos resultados. Atingir este objetivo significa ter um produto:
  - que permita a detecção de erros de uso, erros decorrentes de falhas, acidentes, fraudes, sabotagens, etc.,
  - que permita a contínua medição do grau de alcance dos objetivos,
  - que permita a execução de auditoria dinâmica (em operação) e estática (entradas/saídas).
- Evolutível - o *software* pode ser mantido, expandido e reutilizado. Atingir este objetivo significa ter um produto:
  - que permita a ampliação da capacidade,
  - que permita a ampliação de funções,
  - que permita a adaptação a novos ambientes e alterações das necessidades.
- Rentável - atingir este objetivo significa ter um produto:
  - parcimonioso no consumo de recursos computacionais e humanos,
  - apresentando baixo risco de dano,
  - auferindo benefícios compatíveis com os gastos exigidos.

Não é suficiente, no entanto, somente desejar-se que o sistema a ser produzido atinja estes objetivos de qualidade. Para que sejam alcançados, é necessário visar seu alcance desde o início do desenvolvimento, uma vez que qualidade satisfatória é assegurada somente se o grupo de desenvolvedores atua consciente e deliberadamente para atingi-la. Para tal deverá ser especificado e validado o modelo de predição e avaliação da qualidade do *software* a ser desenvolvido. Ou seja, deverão ser definidos os fatores e métricas que caracterizam a qualidade do *software*, bem como devem ser definidas as tolerâncias destes fatores e métricas para o *software* em apreço.

A definição de fatores e métricas está profundamente influenciada pela intuição e experiência. Assim sendo, há muito de subjetividade e mesmo imprecisão nos conceitos descritos e, conseqüentemente, nas medições e avaliações efetuadas. Para reduzir esta subjetividade devem ser conduzidos experimentos de calibração e, também, deve ser analisada a concordância de opinião de diversos avaliadores. Na literatura atual são apresentados e validados diversos métodos destinados à eliminação da subjetividade e da imprecisão. Foge ao escopo deste tra

balho aprofundar-se neste estudo. Recomendamos, pois, aos leitores interessados as referências [28], [11], [30], [8].

3. QUALIDADE DE ESPECIFICAÇÕES

O desenvolvimento de um sistema automatizado se dá através de uma seqüência de etapas durante as quais são produzidos especificações e projetos, culminando em um projeto que permita a construção do sistema. Estas especificações e projetos são produzidos através de refinamentos sucessivos, partindo de uma visão abrangente para uma visão detalhada. Os passos de refinamento são usualmente agregados em fases, definindo assim o ciclo de vida do sistema.

O desenvolvimento controlado de sistemas automatizados requer a existência de especificações e projetos de boa qualidade. Tendo em vista a inexistência de um modelo de qualidade de especificações bem fundamentado, aplicamos um método de aprendizado para identificar os fatores de qualidade de especificações. Ao aplicarmos este método tomamos por base as referências [1], [26], [47].

Quando falamos em qualidade de especificações temos em vista os seguintes objetivos de qualidade:

- Evolutibilidade - que se refere à capacidade das especificações evoluírem em detalhe à medida que se avança no ciclo de vida através da criação e do refinamento destas especificações,
- Mensurabilidade - que se refere à capacidade de avaliar as propriedades inerentes às especificações,
- Utilizabilidade - que se refere à capacidade de especificações servirem para predizer o comportamento e, após o desenvolvimento, avaliar o produto especificado,
- Modificabilidade - que se refere à capacidade de modificar especificações sem que isto afete negativamente sua qualidade.

Para que estes objetivos sejam atingidos, devem ser realizados através dos seguintes fatores de qualidade de especificações:

- Detalhabilidade - que se refere à facilidade que especificações devem ter para que sejam agregados detalhes a itens já parcialmente especificados,
- Estrutura - que se refere à facilidade de percorrer-se o conjunto de especificações desde a visão mais geral até a mais detalhada e vice-versa,
- Rastreabilidade - que se refere à facilidade de percorrer a seqüência de agregação de detalhes a um determinado aspecto desde sua visão mais parcial até a mais detalhada e vice-versa,
- Ser não-condicionante - que se refere à inexistência de restrições implícita ou explicitamente estabelecidas em relação à escolha de alternativas em passos posteriores da seqüência de refinamentos,

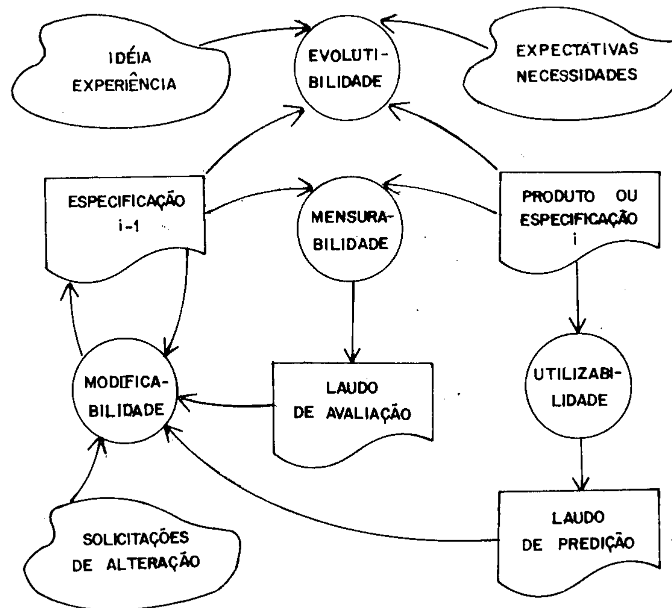


Fig. 1 - Objetivos de qualidade e sua relação no processo de produção de especificações

- Comunicabilidade - que se refere à facilidade de entendimento do conteúdo da especificação,
- Modularidade de comunicação - que se refere à facilidade de compreensão dos elementos da especificação relacionados com determinado aspecto ou assunto. O usuário interessado em um determinado aspecto deverá poder concentrar-se exclusivamente neste aspecto e, ainda assim, deverá obter um perfeito entendimento de tudo relacionado com este aspecto,
- Correção - todos os aspectos especificados estão formalmente corretos e/ou correspondem perfeitamente à percepção que o especificador tem do problema a ser resolvido,
- Completeza - todos os aspectos que deveriam constar da especificação, efetivamente constam dela,
- Necessidade - todos os aspectos tratados são imprescindíveis,
- Uniformidade de detalhe - o grau de detalhe da abordagem dos diversos itens especificados é uniforme,

- Viabilidade - a construção do *software* é possível dos pontos de vista econômico, técnico e administrativo,
- Formalidade - que se refere ao rigor e precisão com que são definidos os aspectos especificados. O grau de formalidade, no entanto, é variável dependendo do ponto em que se encontra no processo de refinamentos sucessivos. Assim, no início o grau de formalidade pode ser baixo, enquanto que mais para o final do processo de refinamento o grau de formalidade deve ser alto,
- Verificabilidade - que se refere à facilidade de examinar-se a especificação com relação a normas e padrões, bem como com relação a especificações anteriores, caso existam,
- Validabilidade - que se refere à facilidade de examinar-se a especificação com relação às necessidades e expectativas do usuário.

A figura 2 mostra a relação entre os objetivos de qualidade de especificações e os fatores de qualidade do produto que os realizam.

Os fatores de qualidade do produto, no entanto, não se atingem com a mera intenção por parte dos desenvolvedores. A qualidade do produto está diretamente relacionada à qualidade da engenharia e construção do sistema. Assim sendo, os fatores de qualidade do produto se realizam através dos fatores de qualidade da engenharia.

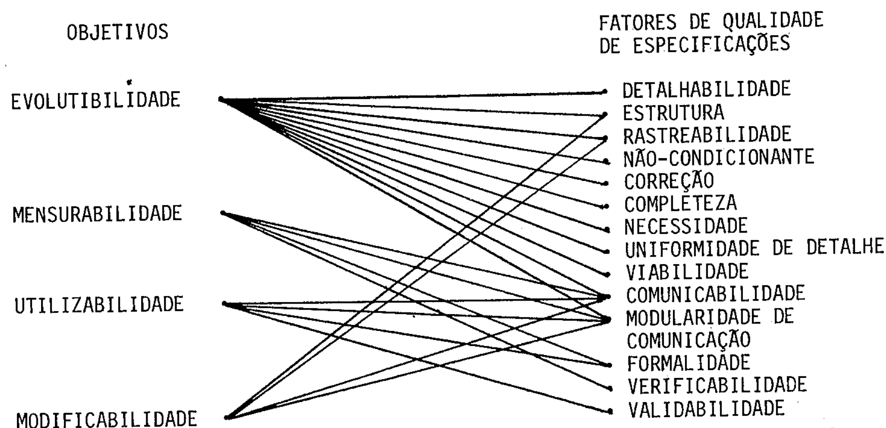


Fig. 2 - Objetivos de qualidade de especificações e os fatores de qualidade do produto que realizam estes objetivos

Para auxiliar na realização dos objetivos e fatores de qualidade do produ-

to (especificações), metodologias e linguagens de especificação devem possuir os seguintes fatores de qualidade:

Quanto ao Processo (Metodologias). Para serem um auxílio eficaz à produção de especificações que assegurem a qualidade do produto especificado, metodologias devem:

- Servir de apoio a todo o ciclo de desenvolvimento facilitando a passagem de uma fase a outra, dado que atualmente se considera de pouca utilidade uma metodologia que apóie apenas a parte do ciclo de vida,
- Possuir um amplo campo de aplicabilidade, isto é, espera-se que a metodologia seja suficientemente geral para que possa ser aplicada a diferentes classes de projetos,
- dar amplo suporte, preferentemente automatizado, para:
  - criação de especificações,
  - verificação,
  - validação,
  - gerência.

Quando automatizado este suporte deve entre outros, auxiliar na predição do comportamento do produto sendo especificado. Isto será alcançado na medida em que especificações sejam executáveis e/ou facilitem a criação de protótipos (*stubs*).

Quanto à Linguagem de Especificação. A partir dos fatores de qualidade do produto, uma linguagem de especificação deve possuir os seguintes fatores de qualidade:

- Construtibilidade, que se refere à facilidade oferecida ao especificador para produzir a especificação desde que conheça a linguagem de especificação e entenda o problema a ser resolvido,
- Inteligibilidade, que se refere à facilidade oferecida ao leitor de entender o que foi especificado, desde que possua nível de formação e treinamento adequados,
- Naturalidade, que se refere à necessidade de que a linguagem de especificação seja natural ao domínio do problema que está sendo especificado,
- Formalidade, que se refere à necessidade de que a linguagem de especificação seja formal com uma notação que permita a verificação formal,
- Auto-verificação, que se refere à facilidade oferecida pela linguagem de através de seu próprio uso auxiliar na detecção de inconsistências e ambigüidades,
- Suporte para validação, que se refere à facilidade oferecida pela lingua-

gem de auxiliar na verificação e validação da especificação,

- Capacidade de produzir índices, que se refere à facilidade oferecida pela linguagem de auxiliar na criação de índices remissivos visando a facilitar o acesso a informações específicas contidas nos diversos documentos.

Quanto ao Uso da Linguagem de Especificação. Além dos fatores de qualidade relativos à metodologia e à linguagem de especificação é necessário, ainda, que o uso da linguagem satisfaça certos fatores de qualidade. Assim sendo, uma linguagem de especificação deve possibilitar seu uso de modo:

- Hierárquico, que se refere à necessidade de que a linguagem ajude o especificador a pensar de modo hierárquico,
- Modular, que se refere à necessidade de que a linguagem ajude o especificador a expressar seu pensamento de forma modular,
- Conciso, que se refere ao volume de informação conduzido por unidade de texto,
- Consistente, que se refere à não-existência de contradições entre quaisquer aspectos desta e/ou de outras especificações na seqüência de refinamentos,
- Não-ambíguo, que se refere à necessidade de não ser deixada margem a interpretações por parte do leitor para qualquer um dos aspectos especificados,
- Uniforme, que se refere ao uso de simbologia, notação e termos técnicos de modo uniforme em todos os documentos produzidos.

A figura 3 mostra a relação entre:

a) objetivos de qualidade de *software* e objetivos de qualidade de especificações,
b) fatores de qualidade de especificações visando o alcance dos objetivos de qualidade das especificações e satisfazendo, assim, indiretamente os objetivos de qualidade do produto,
c) fatores de qualidade da engenharia de especificações visando satisfazer os fatores de qualidade de especificações.

A figura 4 dá uma visão geral dos fatores envolvidos na realização de cada um dos objetivos de qualidade de especificações.

5. METODOLOGIAS E LINGUAGENS PARA ESPECIFICAÇÃO

Descrevemos aqui algumas metodologias, linguagens e ferramentas para especificação propostas na literatura técnica. Se por um lado esta descrição é necessária uma vez que nem todas as metodologias e linguagens são amplamente conhecidas, por outro lado uma descrição detalhada excederia em muito o escopo e a dimensão física deste artigo. Optamos assim por uma descrição extremamente re

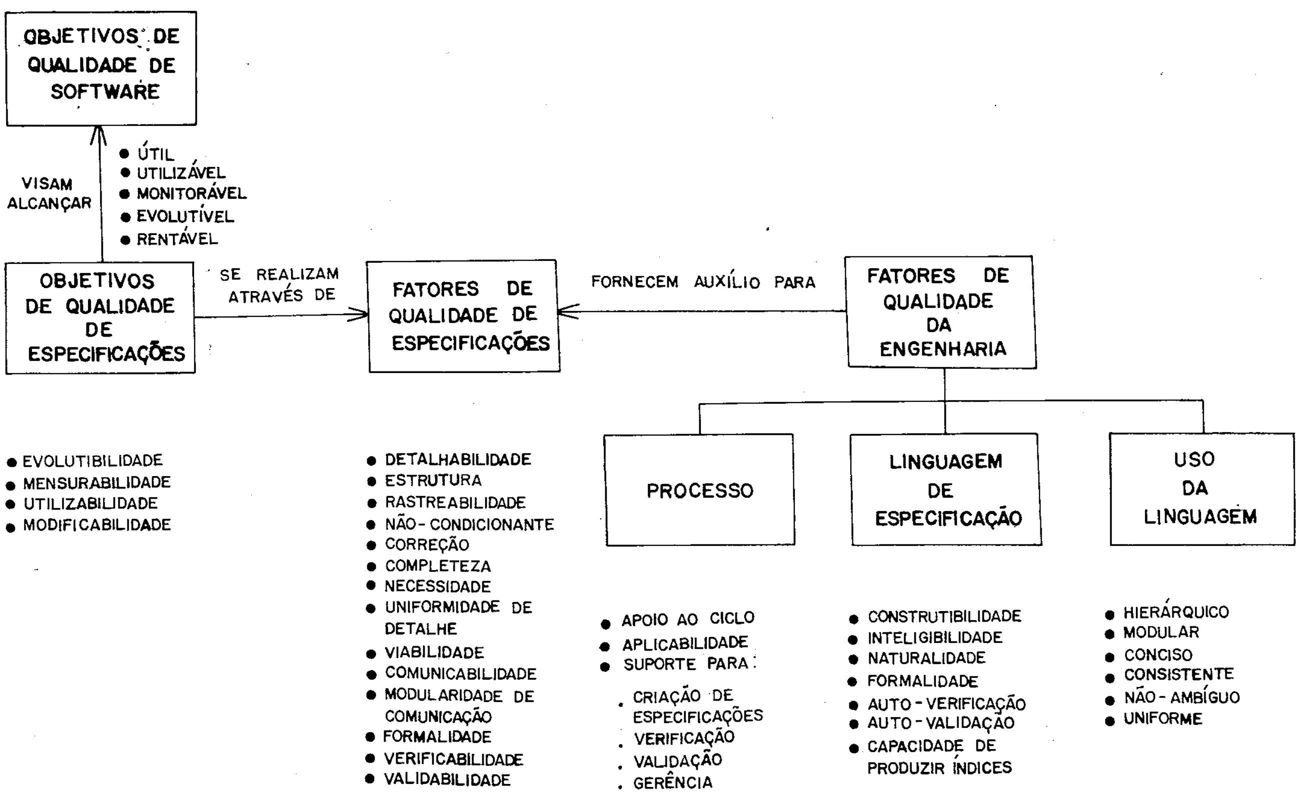


Fig. 3 - Relação entre objetivos e fatores de qualidade

Fig. 3 - Relação entre objetivos e fatores de qualidade

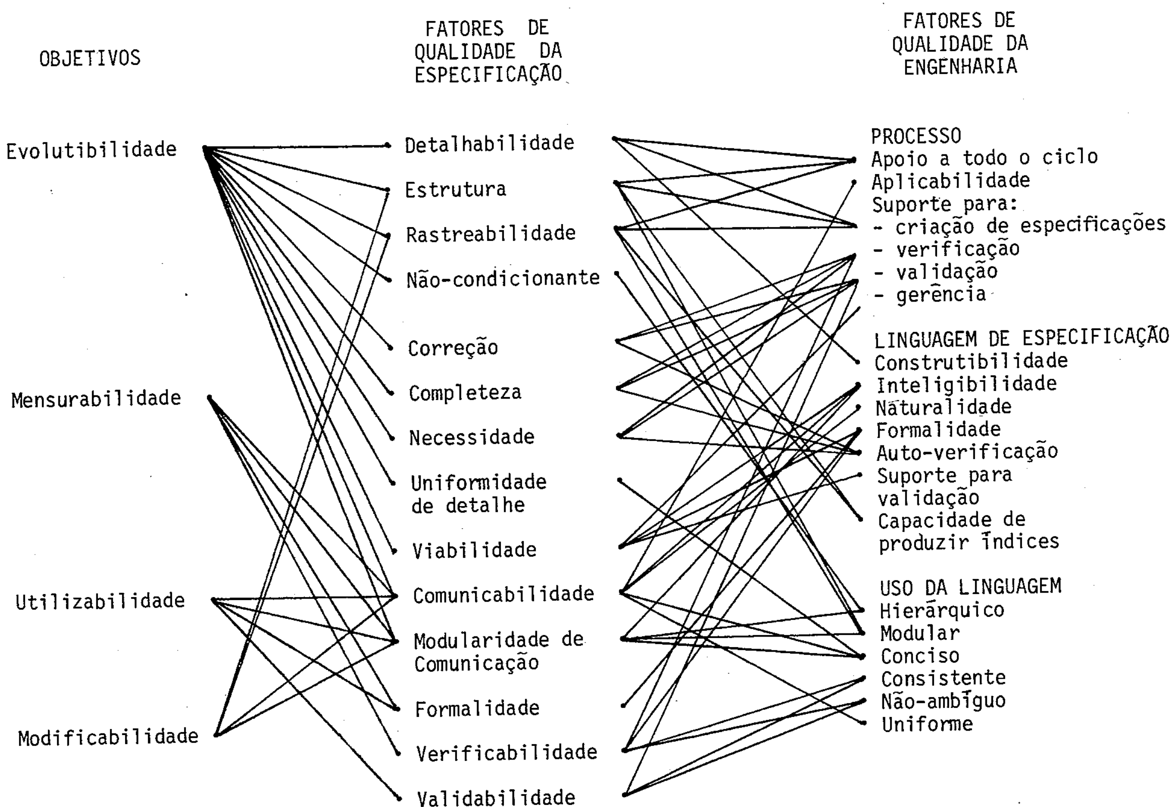


Fig. 4 - Qualidade de especificações

sumida, fornecendo referências bibliográficas onde o leitor interessado poderá encontrar os detalhes desejados.

SADT [34], [35], [5]. SADT (*Structured Analysis and Design Technique*), desenvolvida por Douglas Ross (*SoftTech*) é uma técnica para especificar requisitos de sistemas utilizando diagramas de fluxo de dados (*actigram*) e de estados de dados (*datagram*). SADT é útil também para a fase de projeto. Consiste de:

- um conjunto de métodos que auxiliam a pensar de uma maneira estruturada sobre problemas grandes e complexos, utilizando refinamentos sucessivos,
- uma linguagem gráfica para comunicar especificações de uma maneira clara e precisa, e,
- auxílios para o planejamento e gerência do progresso.

Uma especificação utilizando SADT é uma representação gráfica da estrutura hierárquica de um sistema, sendo esta representação estruturada de modo a aumentar gradualmente o nível de detalhe (figura 5).

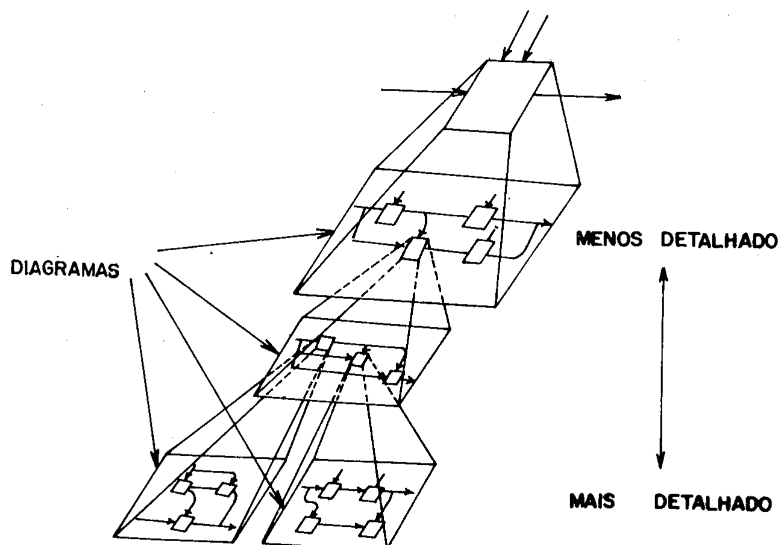


Fig. 5 - Estrutura Hierárquica SADT
Fonte: [35]

A linguagem gráfica SADT fornece um número limitado de primitivos a partir dos quais compõem-se estruturas. A notação consiste de caixas que representam partes de um todo e indicam:

- uma atividade (caso de trate de um *actigram*), ou
- um objeto (no caso de ser um *datagram*).

As flechas representam interfaces entre as partes e,

- no caso de *actigram* descrevem os dados, controles e instrumentos necessários para cada atividade, e possivelmente gerados para serem consumidos em uma atividade posterior,

- no caso de *datagram* descrevem os processos, controles e meios de armazenagem necessários para criar, manter e controlar os diversos objetos (dados).

A preocupação dos diversos diagramas concentra-se no fluxo de transformação de dados, sendo inexistente a preocupação com o fluxo de controle (fluxograma tradicional).

As figuras 6 e 7 mostram exemplos de *actigram* e de *datagram*.

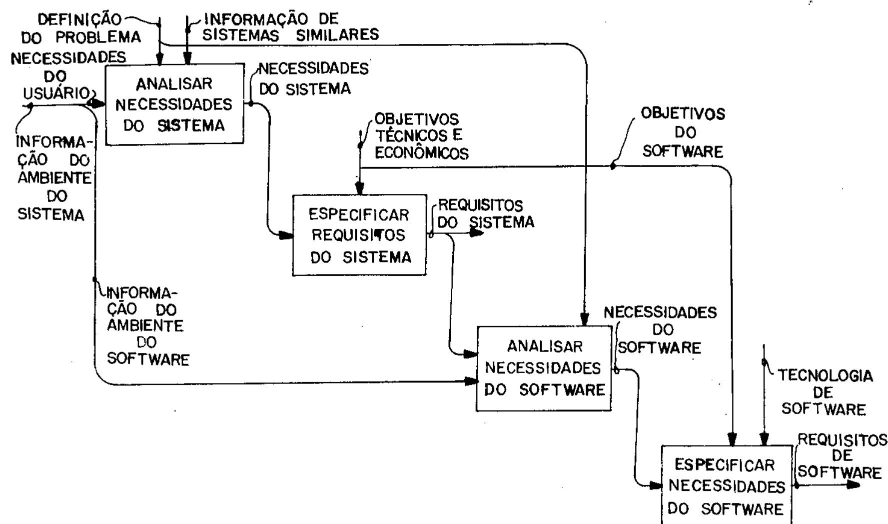


Fig. 6 - Exemplo de *actigram*

SADT vem sendo utilizada desde 1974 por diversas organizações. Seus usuários apontam como aspectos positivos o alto grau de comunicabilidade, o aumento da qualidade do *software* produzido e o fato de assegurar disciplina e facilitar a gerência e o trabalho em equipe.

Atualmente já existe uma proposta de automatização da metodologia SADT. EDDA [36] desenvolvida na Universidade Técnica de Viena (Áustria), combina a técnica gráfica SADT com a teoria de Petri Nets e consegue, assim, uma decomposição rigorosa do problema.

SSA [19], [20], [16]. SSA (*Structured Systems Analysis*) é um conjunto de

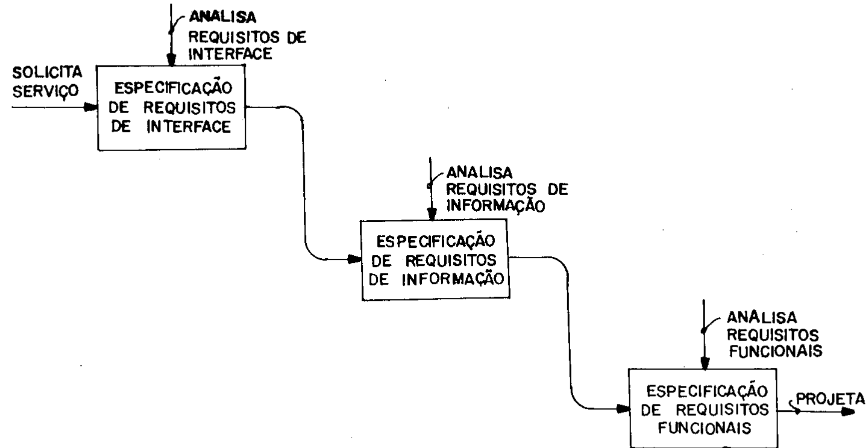


Fig. 7 - Exemplo de *datagram*

técnicas e ferramentas cujo objetivo é auxiliar na análise e definição de sistemas. Como no caso de SADT, o conceito fundamental é a construção de um modelo do sistema utilizando técnicas gráficas. A metodologia envolve a construção *top down* do sistema por refinamentos sucessivos.

SSA consiste em (figura 8):

- Diagramas de fluxos de dados,
- Dicionário de dados, com detalhes sobre todos os dados (estruturas e arquivos) utilizados no fluxo,
- Ferramentas para descrever a organização lógica dos processos (tabelas de decisão, árvores de decisão, pseudo linguagens de programação, linguagem natural estruturada),
- Ferramentas para descrever a organização lógica dos dados (diagrama de acesso indicando como localizar um determinado dado).

Estes documentos fornecem uma descrição do sistema, sua especificação funcional lógica, estabelecendo detalhadamente o que o sistema faz, sendo o mais independente possível de considerações físicas sobre a implementação.

A partir do modelo lógico é posteriormente produzido um projeto (*design*). A metodologia mais apropriada é a conhecida como *Structured Design*, desenvolvida por Larry Constantine e posteriormente refinada por Myers [29] e Yourdon [43]. Os aspectos positivos da metodologia SSA mais comumente apontados são o alto grau de comunicabilidade, a redução do tempo de desenvolvimento do sistema e o aumento de qualidade do sistema produzido.

Recentemente foi desenvolvido pela Tektronix Inc. [14] um conjunto de ferramentas automatizadas de apoio à SSA. A partir da experiência do uso da metodo

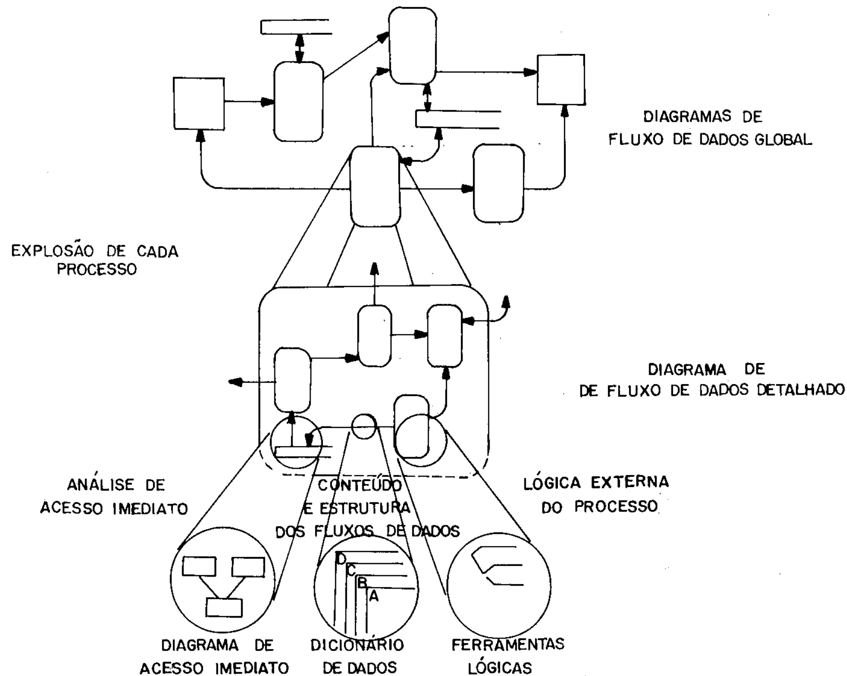


Fig. 8 - *Structured Systems Analysis*
Fonte: [19]

logia foram identificadas áreas a serem automatizadas. Em primeiro lugar gráficos e editores de texto, depois ferramentas para a identificação de erros e inconsistências em documentos e por último a possibilidade de automatizar a tradução de SSA para *Structured Design*. Esta última área foi posteriormente deixada de lado pois só poderia ser construído um tradutor semi-automático, dado que neste processo é necessária a interação humana.

Uma metodologia inspirada no uso de *Structured Design* foi desenvolvida na Hughes Aircraft Company. AIDES (*Automated Interactive Design and Evaluation System*) [48] automatiza vários procedimentos manuais de SD, embora possua também características próprias.

<u>SAMM</u> [40], [31]. SAMM (*Systematic Activity Modeling Method*) é uma técnica de modelagem para descrição de requisitos e projeto que utiliza comunicação gráfico-lingüística. Com esta propriedade pretende satisfazer as necessidades do técnico e do cliente para que este, mesmo sem conhecimentos específicos em

software possa entender os gráficos.

SAMM foi desenvolvida pela Boeing Company e sofreu influência do uso da metodologia SADT. Baseia-se na hipótese de que análise de requisitos e projeto são mais do que duas fases que estão relacionadas e que são, na realidade, apenas artificialmente separadas. Com isso chegou-se a um esquema através do qual os produtos das duas fases podem ser expressos utilizando os mesmos mecanismos.

O projeto da metodologia SAMM teve em conta três conceitos:

- Refinamento semântico, que é um método de abstração hierárquica de significado,
- Racionalidade limitada, que se refere à limitação da capacidade humana e impõe restrições à quantidade de informação que pessoas podem receber, processar e lembrar-se em um determinado intervalo de tempo.
- Teoria dos grafos.

O objetivo da metodologia é modelar um sistema através de uma estrutura ordenada de atividades e fluxo de dados. Sua representação gráfica compreende três elementos (figura 9):

- Uma estrutura de árvore que descreve o contexto do diagrama no sistema,
- Um *Activity Diagram* que descreve o fluxo atividade-dados,
- Um *Condition Chart*, cujos objetivos são indicar quais as entradas necessárias para cada saída do diagrama de atividades e documentar o comportamento funcional do diagrama.

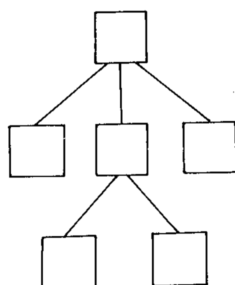
Cada um destes elementos tem uma forma de representação gráfica com uma expressão análoga mais formal. Um *activity diagram*, por exemplo, é análogo a um grafo dirigido.

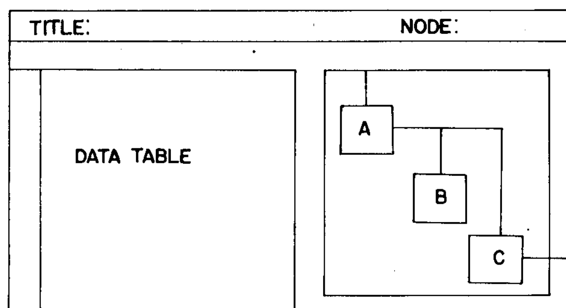
A metodologia fornece meios para a verificação do modelo através de testes de sintaxe e consistência, análise de conectividade do grafo e emissão de relatórios para avaliação. O esquema de representação da metodologia SAMM foi automatizado e sua automatização, um sistema gráfico interativo, chama-se SAMMDF.

PSL/PSA [41], [42]. PSL/PSA (*Problem Statement Language/Problem Statement Analyzer*) é uma ferramenta de auxílio à análise, documentação de requisitos e preparação de especificações funcionais para sistemas de informação. Foi desenvolvida por Daniel Teichroew na Universidade de Michigan (Projeto ISDOS).

São características da metodologia:

- Os resultados de cada uma das atividades no processo de desenvolvimento do sistema são armazenados em forma processável por computador à medida em que são produzidos,
- Um banco de dados é usado para manter todos os dados básicos sobre o sistema,
- O computador é usado na produção de documentação.





TITLE:			NODE
OUT	INP. RQ	CC	CONDITION DESCRIPTION

"CONDITION CHART"

Fig. 9 - Representação gráfica da metodologia SAMM
Fonte: [40]

Os requisitos do sistema devem estar expressos numa linguagem não ambígua e possível de ser processada. Para atingir este objetivo foi desenvolvida PSL, uma linguagem para descrever sistemas. O objetivo de PSL é ser capaz de expressar em forma possível de ser analisada sintaticamente o máximo possível de informações que geralmente aparecem em documentos de descrição de sistemas.

Descrições de sistemas podem ser divididas em oito aspectos principais:

- fluxo de entrada e saída,
- estrutura do sistema,
- estrutura de dados
- derivação de dados,
- tamanho e volume do sistema,
- dinâmica do sistema,
- propriedades do sistema,
- gerência do projeto.

PSL possui tipos de objetos e relacionamentos que permitem a descrição destes aspectos.

À medida que as informações são obtidas, são expressas em PSL e armazenadas no banco de dados usando PSA. A partir deste momento podem ser produzidas saídas e relatórios padronizados sempre que se desejem. Estes relatórios podem ser classificados de acordo com seus objetivos:

- *Data Base Modification Report*, que constitui um registro de alterações que foram feitas para correção e recuperação de erros,
- *Reference Reports*, que apresentam a informação no banco de dados em vários formatos, por exemplo:
  - *Name List Report*, que apresenta todos os objetos no banco de dados com seu tipo e data da última mudança,
  - *Formatted Problem Statement Report*, que mostra todas as propriedades e relacionamentos de um determinado objeto,
  - *Dictionary Report*, que fornece informações do dicionário de dados.
- *Summary Reports*, por exemplo,
  - *Data Base Summary Report*, que fornece informações para a gerência do projeto,
  - *Structure Report*, que mostra hierarquias completas ou parciais,
  - *Extended Picture Report*, que mostram fluxo de dados de forma gráfica.
- *Analysis Reports*, que fornecem vários tipos de análise da informação contida no banco de dados, por exemplo:
  - *Contents Comparison Report*, que analisa as semelhanças entre as entradas e as saídas,
  - *Data Process Interaction Report*, que pode ser usado para detectar falhas no fluxo de informação,
  - *Process Chain Report*, que mostra o comportamento dinâmico do sistema.

Após terminar de estabelecer os requisitos do sistema, pode-se produzir semi-automaticamente a documentação requerida pela organização.

A figura 10 mostra a estrutura do sistema PSL/PSA.

PSL/PSA tem sido utilizada nos últimos anos por um grande número de organi-

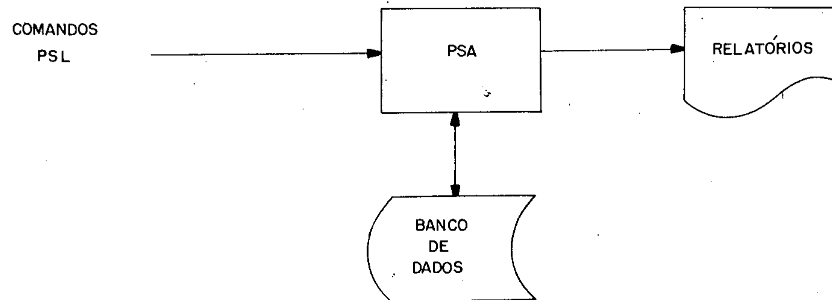


Fig. 10 - O sistema PSL/PSA
Fonte: [41]

zações, inclusive por instituições acadêmicas com finalidade de ensino e pesquisa. Demonstrou ser uma ferramenta eficaz na especificação de sistemas que envolvem grande número de programadores.

SREM [1], [2], [3], [9], [10], [13]. SREM (*Software Requirements Engineering Methodology*), desenvolvida pela TRW como parte do programa US Army Ballistic Defense, tem como objetivo aumentar a qualidade de especificações de requisitos, fornecendo um método para o seu desenvolvimento, junto com um sistema auxiliado por computador, para determinar sua consistência e completeza. É um sistema extenso e poderoso com uma série de inovações que são necessárias para estabelecer requisitos de projetos de desenvolvimento de *software real time*.

SREM consiste em:

- uma metodologia para compor requisitos,
- uma linguagem processável para estabelecer requisitos (RSL- *Requirements Statement Language*),
- um conjunto integrado de ferramentas para auxiliar no desenvolvimento dos requisitos em RSL (REVS - *Requirements Engineering and Validation Systems*).

A linguagem RSL foi concebida para ser um meio de se estabelecerem requisitos de uma maneira natural, embora mantendo um nível de rigor suficiente para ser interpretado por máquina. Para representar requisitos de desempenho *real-time* os requisitos são estabelecidos como seqüências de processos que devem ser executados como respostas a estímulos. Estas seqüências têm uma forma de representação gráfica que é chamada de R-Nets (*Requirements Networks*) (figura 11).

REVS consiste de três segmentos:

- um tradutor para RSL,
- um banco de dados centralizado (ASSM - *Abstract System Semantic Model*),

RBC, Rio de Janeiro, V. 2, nº 2, 1982

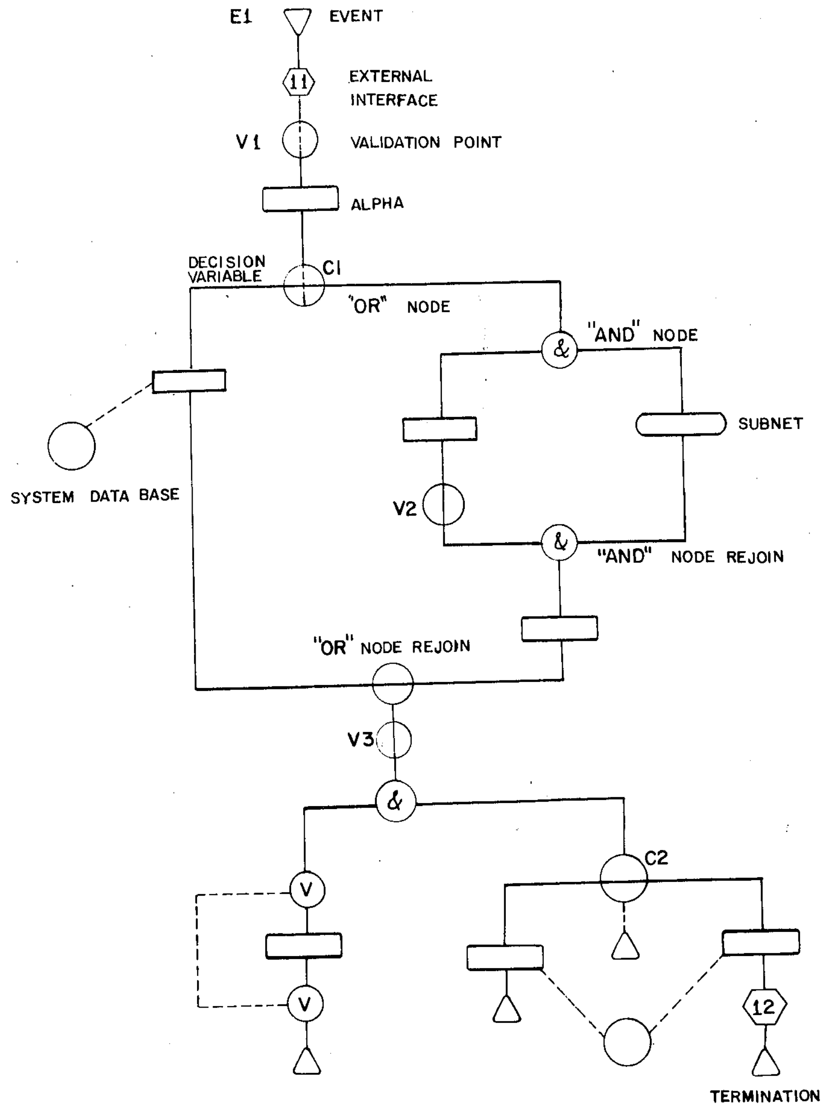


Fig. 11 - Terminologia R-Nets

- um conjunto de ferramentas automatizadas para processar a informação no banco de dados.

As descrições do sistema em RSL podem ser entradas interativamente ou em *batch*, sendo verificada sua consistência com relação a dados armazenados previamente e a completeza da descrição, através de analisadores automáticos. O significado das declarações em RSL é então abstraído e armazenado no banco de dados que é acessado através de um sistema de recuperação flexível. Podem-se construir e executar simulações para validar os requisitos. Após validados, estática e dinamicamente, os requisitos são gerados através de auxílio automatizado.

Assim sendo, o conjunto básico de ferramentas em REVS aceita requisitos expressos em RSL como entrada e contém auxílios para validação e para verificação da completeza, consistência e correção dos requisitos. Contém ainda ferramentas orientadas para a análise de fluxo que fornecem entrada e saída gráfica, realizam análise estática e criam simuladores. A arquitetura flexível de REVS torna ainda possível extensões em dois sentidos; adições de novas ferramentas e criação de relatórios especiais. A figura 12 mostra um diagrama do sistema SREM.

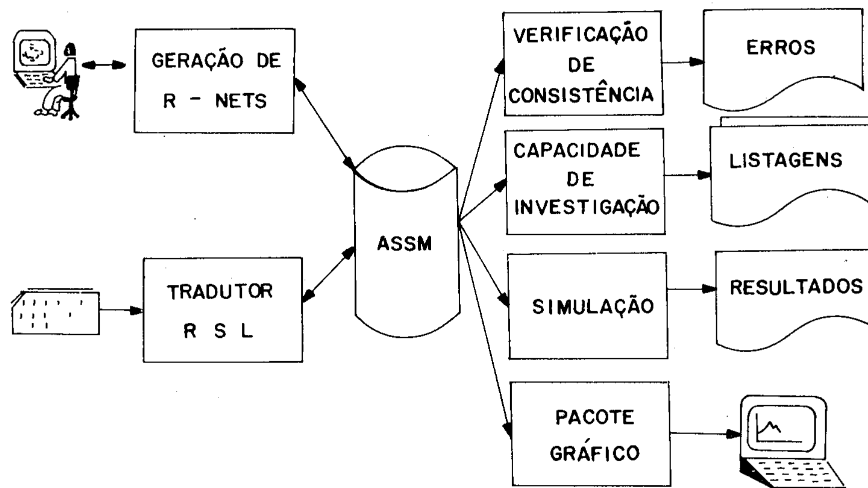


Fig. 12 - Metodologia SREM
Fonte: [13]

SREM apresenta uma série de limitações que refletem decisões deliberadas de seu projeto, voltado para os problemas de controle de processos *real-time* característicos de defesa de misseis.

Portabilidade é outro problema pois partes do sistema operam em máquinas especiais, poderosas e extremamente caras. Além disso, SREM é uma metodologia extensa tornando sua utilização apenas viável no caso de sistemas de grande porte. Observações com relação à utilização da metodologia encontram-se em [5].

HDM [17], [15], [12], [37], [25]. HDM (*Hierarchical Development Methodology*), desenvolvida pelo Stanford Research Institute (SRI), é uma metodologia concebida para ajudar no desenvolvimento de sistemas grandes, complexos e com um alto grau de qualidade.

São características da metodologia:

- O uso de decomposição hierárquica para resolver problemas complexos,
- O uso de métodos de abstração, isolando as propriedades do objeto apropriadas ao seu entendimento e descrição do mesmo, no nível de detalhe sob consideração,
- O uso de especificação formal através de uma linguagem semi-matemática,
- modularidade,
- A possibilidade de a cada passo verificar formalmente a especificação resultante tendo como base as especificações anteriores,
- A existência de ferramentas automatizadas para apoiar o processo de desenvolvimento e verificação.

O desenvolvimento de um sistema em HDM se realiza como uma seqüência de máquinas abstratas, também chamadas níveis. O nível mais alto é a interface com o usuário e o nível mais baixo é chamado de máquina primitiva. Estes dois níveis são chamados de níveis extremos e os demais de níveis intermediários.

Assim sendo, HDM sugere os seguintes estágios no desenvolvimento:

- Conceituação, onde se identifica o problema a ser resolvido e o papel do sistema em sua solução,
- Definição da máquina extrema, onde se define a decomposição modular e as interfaces externas dos níveis extremos,
- Definição da estrutura do sistema, onde se define a decomposição modular e as interfaces externas dos níveis intermediários,
- Especificação dos módulos, onde se escreve uma especificação (em *SPECIAL*) para cada módulo do sistema,
- Representação dos dados, onde se define (em *SPECIAL*) as estruturas internas de cada máquina não-primitiva em termos das estruturas de dados internas da máquina abstrata do nível imediatamente inferior da hierarquia,
- Implementação abstrata, onde se escrevem programas abstratos (utilizando a linguagem de implementação abstrata ILPL) para implementar cada operação de cada uma das máquinas abstratas não primitivas,
- Implementação concreta, onde se traduz o programa abstrato em ILPL em có

digo executável na linguagem que se escolha (por exemplo Modula ou Pascal).

As linguagens de HDM procuram expressar os conceitos descritos acima. A linguagem SPECIAL (*Specification and Assertion Language*) é usada para especificar e projetar módulos. HSL (*Hierarchy Specification Language*) é usada para descrever níveis e hierarquias de níveis. ILPL (*Intermediate Level Programming Language*) é uma linguagem de programação de nível intermediário usada para descrever programas abstratos). Não existe uma linguagem HDM para implementação final. Assim sendo, no final do processo os programas abstratos em ILPL têm que ser traduzidos para uma linguagem de programação executável.

Verificação em HDM tem duas formas: prova do projeto e prova da implementação. Prova do projeto é a verificação de que a especificação do sistema possui certas propriedades. Prova da implementação é a verificação de que a implementação do sistema está de acordo com as especificações. O enfoque que HDM dá à prova de correção da implementação é baseado no método de Floyd-Hoare onde um programa é provado correto em relação a assertivas de entrada e saída.

HDM tem tido mais sucesso como uma ferramenta para a fase de projeto. Os usuários da metodologia, em geral, a identificam com a linguagem SPECIAL, isto é, utilizam HDM para especificação e projeto de módulos. Estes usuários geralmente estão interessados em verificar propriedades do projeto e apreciam o grau de rigor da linguagem SPECIAL.

USE [44], [45], [46]. O projeto USE (*User Software Engineering*), desenvolvido por Anthony Wasserman, foi iniciado na Universidade da California, San Francisco em 1975. Seu objetivo é produzir uma metodologia que sirva de apoio à especificação, projeto e implementação de sistemas de *software* interativos. Assim sendo, USE procura fornecer um conjunto de ferramentas e uma metodologia para o seu uso especificamente dirigidas para a especificação e implementação de sistemas de informação interativos.

A metodologia é baseada no conceito de desenvolvimento sistemático e tem, também, como objetivo favorecer a participação do usuário na especificação do sistema. USE apóia-se em um conjunto de ferramentas automatizadas disponíveis no ambiente UNIX e especificamente dirigidas às necessidades de sistemas de informação interativos. Estas ferramentas são:

- A linguagem de programação PLAIN (*Programming Language for Interaction*), uma linguagem procedural derivada de Pascal que contém facilidades para construção de sistemas de informação interativos com um conjunto de operações para apoiar a definição e a manipulação de bancos de dados relacionais. PLAIN também tem em conta os objetivos de desenvolvimento sistemático de *software* e assim sendo fornece apoio para modularidade e abstração.
- O *Module Control System*, uma ferramenta que apóia a organização modular

do sistema, assiste na manutenção do controle sobre as várias partes da documentação e do código associado com o sistema de informação e controla o processo pelo qual são feitas modificações no sistema,

- O *Transition Diagram Interpreter* (TDI) que pretende apoiar a especificação do sistema via *transition diagrams* (figura 13) e permitir deste modo a rápida construção e modificação de protótipos de interface usuário/programa e de protótipos do sistema. Para isso, o TDI aceita um ou mais *transition diagram* como entrada e produz um programa executável simulando a interface especificada,

- TROLL, uma ferramenta que fornece uma interface de comunicação para um pequeno sistema de banco de dados relacional (*PLAIN Data Base Handler*) e que pode ser conectada ao TDI via UNIX fornecendo assim um maior grau de funcionalidade ao sistema.

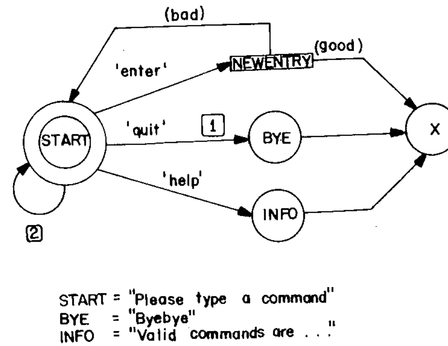


Fig. 13 - Exemplo de *transition diagram*
Fonte: [45]

A figura 14 mostra a organização do protótipo USE.

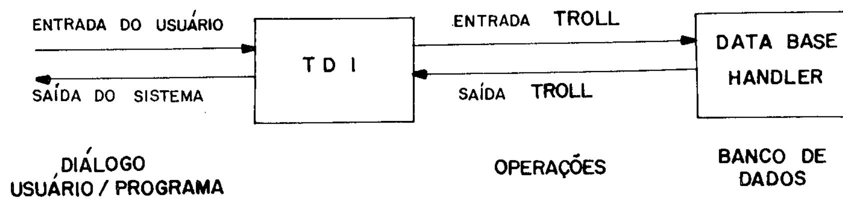


Fig. 14 - Organização do protótipo USE
Fonte: [46]

Estas ferramentas pretendem tornar possível a construção de sistemas que realmente atinjam as necessidades do usuário e ajudar desenvolvedores e usuários a determinarem estas necessidades. Este envolvimento do usuário é favorecido pelo uso de protótipos do diálogo usuário/programa e se possível do próprio sistema.

Especificações Algébricas [21], [23]. Especificações algébricas descrevem propriedades funcionais de operações sobre tipos abstratos de dados e fornecem um conjunto de axiomas que descrevem o comportamento das operações e as interações entre elas (figura 15).

```
type Stack[elem-type: Type, n:Integer]
where ( )

syntax

newstack:         --> Stack
push:     Stack X elem-type --> Stack
pop:      Stack --> Stack
top:      Stack --> elem-type
isnew:    Stack --> Boolean
replace:  Stack X elem-type --> Stack
*depth:   Stack --> Integer

semantics

declare stk:Stack, elm:elem-type

1) pop(push(stk, elm)) = stk
2) top(push(stk,elm)) = elm
3) isnew(newstack) = true
4) isnew(push(stk,elm)) = false
5) replace(stk,elm) = push(pop(stk),elm)
6) depth(newstack) = 0
7) depth(push(stk,elm)) = 1 + depth(stk)

restrictions

pre(pop,stk) = ~isnew(stk)
pre(replace,stk,elm) = ~isnew(stk)
isnew(stk) => failure(top,stk)
failure(push, stk,elm) => depth(stk)_n
```

Fig. 15 - Exemplo de especificação algébrica para pilha
Fonte: [23]

Essas especificações apresentam como aspecto positivo sua precisão e o fato de serem independentes da representação. Como aspecto negativo está sua limitação à classe de objetos e operações que podem representar.

Existe uma linguagem formal, chamada OBJ [22], para escrever e testar espe

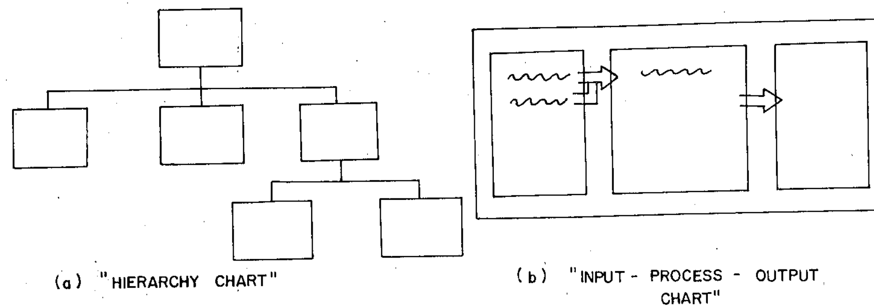
cificações algébricas de programas. OBJ é também uma linguagem de programação uma vez que permite a execução de especificações algébricas.

Especificações algébricas influenciaram outras linguagens de especificação, por exemplo a linguagem *SPECIAL* da medotologia HDM.

HIPO [39]. HIPO (*Hierarchy plus Input-Process-Output*) foi desenvolvida pela IBM, originalmente como uma ferramenta para documentação. É, no entanto, uma técnica útil tanto na documentação do projeto como na fase de análise de requisitos e especificação.

Está baseada em dois conceitos: o modelo entrada-processo-saída e o de hierarquias funcionais. Assim sendo, consiste de uma notação gráfica com dois componentes:

- *Hierarchy Charts* que mostram como uma função se subdivide em várias funções,
- *Input-Process-Output Charts*, que expressam cada função na hierarquia em termos de suas entradas e saídas (figura 16).



(a) "HIERARCHY CHART" (b) "INPUT - PROCESS - OUTPUT CHART"

Fig. 16 - Diagramas HIPO
Fonte: [39]

Como aspectos positivos podem-se assinalar a facilidade com que pode ser aprendida e utilizada e o fato de poder ser aplicada tanto para a especificação como para o projeto.

GYPSY [4]. O projeto de GYPSY, desenvolvido na Universidade do Texas, Austin, teve como objetivo o desenvolvimento de uma metodologia de auxílio à construção de programas verificáveis orientados para o controle de processos.

A metodologia consiste de um sistema integrado de métodos para especificação formal de programas e sua verificação ou por prova formal ou por validação em tempo de execução. Assim sendo, os objetivos específicos presentes no proje-

to de GYPSY foram:

- Verificabilidade, qualquer aspecto da linguagem tem que ser rigorosamente verificável ou por prova formal ou por validação em tempo de execução,
- Oferecer suporte para o desenvolvimento incremental e modular de programas e para verificação,
- Assistir no desenvolvimento de sistemas de software,
- Permitir a execução em ambientes com defeito,
- Capacidade de especificação.

Contém duas ferramentas:

• a linguagem para projeto de programas GYPSY, e,
• um sistema interativo para o projeto e verificação dos programas em GYPSY.

GYPSY fornece meios para expressar programas e sua especificação formal e é o elemento unificador da metodologia. A linguagem fornece meios para expressar um programa através de todos os estágios do desenvolvimento, da especificação inicial à implementação, verificação e evolução. Esta integração de facilidades para especificação e programação em uma única linguagem é a principal característica de GYPSY.

Pelo fato de ter sido especifidamente projetada para verificação, a linguagem evita conceitos que dificultam a verificação por prova formal. Assim sendo, Pascal foi selecionada como modelo, embora GYPSY apresente diferenças significativas em relação a Pascal.

MODEL [32], [33]. MODEL (*MOdule DEscription Language*) foi desenvolvida por N.S. Prywes na Universidade da Pensylvania. Foi projetada para ser usada por especialistas no campo ao qual o programa será aplicado. Estes não são solicitados a ter treinamento em computação, necessitam, porém, de terem amplos conhecimentos de matemática. MODEL procura tornar viável a preparação de programas por estes usuários (interagindo com um gerador automático de programas) sem necessidade de recorrer a um programador de aplicação.

O usuário compõe comandos (através de um terminal e um editor de texto) na linguagem MODEL. Cada um destes comandos é considerado como uma unidade integral e contém informação. Um comando pode descrever um item de dados (descrição de dados) ou uma relação (algébrica ou lógica) entre itens de dados (assertivas).

Podem ser feitas referências a itens que foram previamente armazenados no banco de dados pelo usuário, por outros usuários que especificaram requisitos para aplicações similares ou ainda por outros usuários com quem se quer partilhar dados. O processador MODEL analisa a totalidade dos comandos a ele transmitida e, se necessário, solicita ao usuário adições e alterações, para resolver incompletezas, ambigüidades e inconsistências. Quando estes problemas são resolvidos e o usuário está satisfeito, o processador produz um programa, o compila-

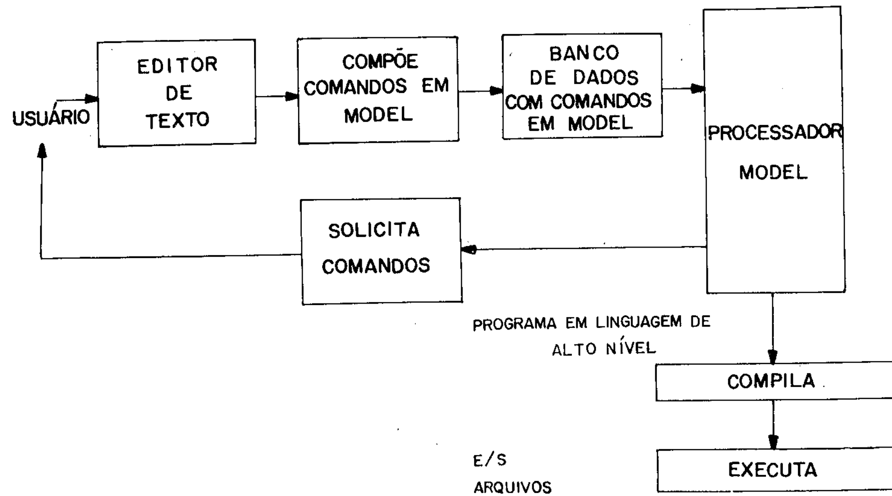


Fig. 17 - Estrutura MODEL
Fonte: [32]

dor produz o código objeto e sistema é carregado para execução (figura 17). MODEL é programado em PL-I e produz código objeto em PL-I.

A linguagem MODEL apresenta as seguintes características:

- Independência dos comandos, isto é, a especificação pode ser vista pelo usuário como um conjunto de unidades de informação independentes e ele pode se concentrar em cada unidade de uma vez,
- É não-procedural, isto é, o usuário não precisa (nem pode) especificar uma ordem de avaliação dos comandos. Isto é importante porque remove pré-requisitos de conhecimento de fluxograma,
- A ordem dos comandos é arbitrária, isto é, podem ser compostas em qualquer ordem,
- Simplicidade de sintaxe, para isto existem dois tipos de comandos, um para descrever dados e outro para descrever relacionamentos entre dados,
- Tolerância e interatividade, isto é, o processador é tolerante às ambigüidades e omissões cometidas pelo usuário e solicita ao mesmo novas informações até resolver estes problemas e completar a especificação.
- Partilhamento e incrementalidade, isto é, a descrição de requisitos é feita incrementalmente através da composição de diferentes partes, algumas das quais compostas por outros usuários. O processador MODEL funciona, assim, como integrador de especificações preparadas independentemente.

SAFE/TI [6], [18], [24]. Os projetos SAFE (*Specification Acquisiton From*

Experts) e TI (*Transformational Implementation*) desenvolvidos por Robert Balzer, Neil Goldman, David Wile e Chuck Willians (University of Southern California) , são dois projetos complementares que veem o processo de desenvolvimento de *software* como consistindo de dois passos:

- obter um conjunto preciso e completo de especificações para o prolema, e,
- converter estas especificações em código executável e eficiente.

O projeto SAFE tem como objetivo automatizar a fase de projeto. Para isso recebe do usuário especificações informais, expressas em uma linguagem natural limitada e procura a partir desta, construir uma especificação formal.

Como um objetivo do sistema é criar especificações formais a partir de informais, isso significa que ele deve completar cada uma das especificações de entrada para produzir a saída, resolvendo as ambigüidades e imprecisões presentes na especificação original. Geralmente cada descrição parcial pode ser completada de diferentes maneiras e torna-se necessária uma decisão separada para selecionar, em cada caso, a maneira correta de completar.

Baseado na descrição parcial e no contexto de decisões anteriores, é criado um conjunto de possíveis conclusões para cada descrição parcial. Uma decisão, no entanto, não pode ser feita isoladamente de outras para que a especificação final tenha sentido como um todo. Para isso, após cada decisão o programa é testado para ver se obedece aos critérios. Caso isso se dê, continua até que todas as decisões tenham sido tomadas e o resultado aceito. Este processo termina ou por se ter encontrado uma solução (especificação formal) ou porque se verificou que não é possível encontrá-la. O objeto resultante (programa) é uma solução aceitável (especificação formal) para o problema (especificação informal).

O projeto TI se dirige ao problema de produzir um código eficiente a partir destas especificações. O enfoque do projeto é fornecer ao programador um conjunto de ferramentas semi-automatizadas que o auxiliem a realizar a transformação.

O sistema TI, no entanto, apresenta certos problemas técnicos. Em primeiro lugar o problema de se é realmente possível encontrar o conjunto de todas as transformações que o programador pode querer empregar na tarefa de converter especificações em código e ao mesmo tempo o problema de que se é possível representar estas transformações de forma compacta. O segundo problema é quanto à real viabilidade de que um sistema deste tipo possa ser usado por programadores.

6. ANÁLISE DAS METODOLOGIAS E LINGUAGENS DE ESPECIFICAÇÃO

Nesta seção fazemos uma análise das metodologias e linguagens de especificação descritas na seção anterior tendo como base o modelo de qualidade descrito na seção 4. Nosso objetivo é mostrar a presença ou não, em cada uma delas, dos fatores de qualidade de engenharia.

FATORES DE QUALIDADE / METODOLOGIAS	SADT	SSA	SAMM	PSL/ PSA	SREM	HDM	USE	ESP. ALG.	HIPO	GYPSY	MODEL	SAFE
QUANTO AO PROCESSO												
•Apoio a todo o ciclo	N	N	N	N	N	N	S	N	N	S	S	S
•Amplo campo aplicabil.	S	S	S	S	N			N	S	N		
•Suporte automatiz. p/												
•criação de especif.	N	N	N	S	S	S	S	N	N	S	S	S
•verificação	N	N	N	S	S	S	S	N	N	S	S	S
•validação	N	N	N	S	S	S	S	N	N	S	S	S
•gerência	N	N	N	S	S		S	N	N	N	N	N
QUANTO Ā LING. DE ESPEC.												
•Construtibilidade	G	G	G	M	M	P	G	P	G	P	G	G
•Inteligibilidade	G	G	G	M	M	P	G	P	G	P	G	G
•Naturalidade	G	G	G	M	G	M	G	M	G	G	G	G
•Formalidade	P	P	P	M	G	G	G	G	P	G		
•Auto-verificação	M	M	G						G			
•Auto-validação							G					
QTO. AO USO DÁ LING.												
•Hierārquico	S	S	S			S	S		S	S		
•Modular	S	S	S			S	S	S	S	S		
•Conciso	S	S	S		S	S		S				
•Consistente	S	S	S	S	S	S	S	S	S	S	S	S
•Não ambīguo	S	S	S	S	S	S	S	S	S	S	S	S
•Uniforme	S	S	S	S	S			S	S	S	S	S

Fig. 18 - Anālise das metodologias e linguagens de especificação segundo os fatores de qualidade da engenharia
Legenda: S - Sim; N - não; G - Grande; M - Mēdio; P - Pequeno

Em alguns casos é difícil indicar a presença ou não de um fator, tendo como base apenas referências bibliográficas. Neste caso preferiu-se deixar em branco no quadro (figura 18).

Esta análise nos faz concluir:

- As metodologias, linguagens e ferramentas descritas, em geral, aplicam-se a uma ou duas fases do ciclo de vida do sistema (figura 19). São poucas as metodologias que pretendem servir de apoio ao ciclo de vida completo, desde a fase de Análise e Especificação de Requisitos até a fase de Construção e Teste. Este, no entanto, é atualmente um dos requisitos indispensáveis a uma metodologia de apoio ao processo de desenvolvimento de *software*.

  Cabe assinalar aqui, a não existência de ferramentas específicas de auxílio à fase de Evolução. A disciplina observada nas fases anteriores e a existência de documentação atualizada são, no entanto, os melhores auxílios para a fase. A realização de uma atividade característica de evolução implica em um retorno às primeiras fases do ciclo de desenvolvimento para o que se deverá utilizar as mesmas metodologias, linguagens e ferramentas utilizadas anteriormente. A efetividade desta etapa é, portanto, dependente das metodologias, linguagens e ferramentas que foram utilizadas durante as fases anteriores.
- As linguagens de especificação descritas apresentam graus bastante variáveis de formalidade. A formalidade da linguagem, no entanto, está diretamente relacionada à possibilidade de realização de verificações formais. Consideramos que um sistema ou programa está correto se satisfaz a suas especificações. Assim sendo, verificar a correção é verificar em que

	ANÁLISE E ESPECIFICAÇÃO DE REQUISITOS	PROJETO	CONSTRUÇÃO E TESTE	EVOLUÇÃO
SADT				
SSA / SD				
SAMM				
PSL / PSA				
SREM				
HDM				
USE				
ESP. ALG.				
HIPO				
GYPSY				
MODEL				
SAFE/TI				

Fig. 19 - Apoio ao ciclo de vida

RBC, Rio de Janeiro, V. 2, nº 2, 1982

grau o sistema ou programa construído atinge as necessidades e requisitos expressos nas especificações.

Para que seja possível realizar esta verificação é necessário que exista um certo grau de formalidade. Sem isso só poderão ser realizadas especificações informais e por isso nem sempre eficazes.

Formalidade, porém, apresenta como efeito secundário um potencial de dificuldades de compreensão. Especificações formais são, muitas vezes, difíceis de serem entendidas sem que se tenha uma formação matemática de alto nível. Este é um fator a ser ponderado no desenvolvimento e utilização de metodologias, pois certamente não se pode esperar, no momento, este grau de sofisticação na formação de programadores e usuários.

- A variedade do campo de aplicabilidade é um objetivo muitas vezes presente nas metodologias mas difícil de ser avaliado dada a inexistência de observações de diferentes tipos de usuários e também, algumas vezes, a seu uso ainda incipiente [5]. Além disso várias destas metodologias encontram-se ainda em fase de desenvolvimento.
- Não é desejável o uso de metodologias onde inexiste suporte automatizado. No caso das metodologias SADT e SSA seus usuários desenvolveram um conjunto de ferramentas automatizadas procurando suprir esta carência [36], [14], [48].

7. CONCLUSÃO

Foi apresentado um modelo de qualidade para especificações e a partir do modelo procurou-se analisar as principais metodologias e linguagens de especificação propostas na literatura atual.

Esta análise nos mostrou uma série de características das metodologias atualmente disponíveis ou em desenvolvimento conforme descrevemos na seção anterior. Mostrou-nos, também, que os fatores de qualidade descritos no modelo são muitas vezes conflitantes. Atingir um deles em maior grau pode significar diminuição no grau de alcance de outro ou outros fatores. O uso de linguagem formal, por exemplo, pode por um lado aumentar o grau de alcance dos fatores formalidade e verificabilidade das especificações e por outro lado dificultar o alcance do fator comunicabilidade.

Outro exemplo de conflito é a naturalidade da linguagem de especificação versus ter um amplo campo de aplicabilidade. Enquanto o fator naturalidade procura que a linguagem seja natural ao domínio do problema que está sendo especificado, possuir um amplo campo de aplicabilidade implica ser suficientemente geral como para poder ser aplicada a diferentes classes de projetos. Estes dois fatores são, claramente, difíceis de compatibilizar.

Na realidade, portanto, o que se deve procurar é um equilíbrio entre os fa-

tores, equilíbrio este, que é ditado pelo conjunto de objetivos de qualidade.

Como vimos, metodologias devem cobrir um espectro mais amplo do que apenas algumas das fases do ciclo de vida. O que se procura, atualmente, são metodologias e ferramentas que sirvam de apoio a todo o ciclo de desenvolvimento ou que possam ser facilmente integradas com outras metodologias e ferramentas, de modo que o conjunto seja um apoio eficaz durante todo o ciclo de desenvolvimento.

Uma solução na tentativa de compaginar estes fatores é a criação de uma família de linguagens de especificação com membros adequados a cada um dos níveis de detalhe, de modo que a passagem de um nível para outro se faça de uma maneira harmônica, sem rupturas desnecessárias. Para isto, esta família de linguagens deve possuir membros progressivamente mais formais, à medida que cresce o nível de detalhe e se avança no ciclo de desenvolvimento do sistema.

No início do desenvolvimento, o conhecimento é ainda bastante precário e é necessário um alto grau de envolvimento do usuário para que se determinem corretamente os requisitos e necessidades do sistema. Neste momento deve-se buscar um menor grau de formalidade, apenas o suficiente para permitir concisão, consistência e evitar ambigüidades. Deste modo torna-se possível o entendimento das especificações por parte do usuário, sem que isto lhe suponha demasiado esforço.

No final o conhecimento deve ser perfeito e, conseqüentemente, as especificações devem ser formais, utilizando uma linguagem formal. Esta linguagem formal, por sua vez, deve ser tal que torne possível a passagem sem rupturas para a fase de construção. Dentro do objetivo de se terem metodologias e ferramentas que apóiem todo o ciclo de desenvolvimento, é importante a existência de auxílio automatizado para esta fase. O ideal será contar com um sistema de programação automática que transforme as especificações produzidas anteriormente em um programa executável.

Uma família de linguagens de especificação com estes objetivos e baseada no modelo de qualidade anteriormente descrito está sendo desenvolvida atualmente na PUC-RJ.

6. REFERÊNCIAS

[1] ALFORD, M.W.; BURNS, I.F. "R-Nets: a graph model for real-time software requirements", *Proceedings of the Symposium on Computer Software Engineering*, New York, 1976.

[2] ALFORD, M.W., "A Requirements engineering methodology for real-time processing requirements", *IEEE Transactions on Software Engineering*, vol SE-3 n1, janeiro 1977.

[3] ALFORD, M.W., "Experience with the software development system", in: *Software Engineering Experience*, Hunke, H. ed. North-Holland Publishing Company, 1981.

[4] AMBLER, A.L. et al, "GYPSY: A Language for specification and implementation of verifiable programs", *Proceedings of ACM Conference on Language Design for Reliable Software*, março 1977.

[5] BAIL, W.G. "User experience with specifications tools". Panel from Specifications of Reliable Software Conference, ACM *SIGSOFT Software Engineering Notes*, vol SE-2, n 3, julho 1979.

[6] BALZER, R; GOLDMAN, N; WILE, D., "Informality in program specifications" *IEEE Transactions on Software Engineering*, vol SE-4, n 2, março 1978.

[7] BARROS, S.C., "Controle de qualidade aplicado ao desenvolvimento de software", Dissertação de Mestrado, Informática, PUC-RJ, 1982.

[8] BARROS, J.C.C., "Medidas de qualidade de software - proposição e aplicação de um modelo", *Anais do XV Congresso Nacional de Informática*, Rio de Janeiro, outubro 1982.

[9] BELL, T.E.; BIXLER, D.C. "A flow-oriented requirements statement language", *Proceedings of the Symposium on Computer Software Engineering*, New York, 1976.

[10] BELL, T.E.; BIXLER,D.C.; CHARLES, R., "The Software development systems", *IEEE Transactions on Software Engineering*, vol. SE-3, n 1, jan. 1977.

[11] BOEHM, B., *Software engineering economics*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

[12] CHANDERSEKARAN, C.S.; LINDER, R.C. "Software specifications using SPECIAL language", *The Journal of Systems and Software*, vol.2, n 1, fev. 1981.

[13] DAVIS, G.; VICK, C., "The Software development system", *IEEE Transactions on Software Engineering*, vol.SE-3, n 1, jan. 1977.

[14] DELISLE, N.M. et al, "Tools for supporting structured analysis", in: *Automated tools for informations systems design*, H.J. Schneider e A.I. Wasserman eds, North-Holland Publishing Company, IFIP 1982.

[15] DISCEPOLO, A.G., Towards a practical specification language", ACM'81 *Conference Proceedings*, Los Angeles, CA, novembro 1981.

[16] DE MARCO, T., *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[17] "Progress Toward System Integrity", *EDP ANALYZER*, vol.17, n 12, dez 1979.

[18] ELSCHLAGER, R.; PHILLIPS, J. "Automatic Programming", Computer Science Department, Stanford University, Report STAN-CS-79-758, agosto 1979.

[19] GANE, C.; SARSON, T., *Structured systems analysis: tools and techniques*, Prentice-Hall Inc., N.J. 1979.

[20] GANE, C., "Data design in structured systems analysis", in: *Tutorial on software design techniques*, P. Freeman and A.I. Wasserman, eds, IEEE Computer

Society, 1980.

[21] GOGUEN, J.A.; THATCHER, J.W.; WAGNER, E.G., "An Initial algebra approach to the specification, correctness and implementation of abstract data types", in: *Current trends in programming methodology*, vol. IV, R.Yeh ed, Prentice Hall, 1978.

[22] GOGUEN, J.A.; TARDO, J.J., "An Introduction to OBJ: a language for writing and testing formal algebraic programs specifications", *Proceedings Specifications for Reliable Software*, 1979.

[23] GUTTAG, J., "Notes on type abstraction", *Proceedings Specifications for Reliable Software*, 1979.

[24] HAMMER, M.; RUTH, G., "Automating the software development process", in: *Research directions in software technology*, Wegner, P. ed, MIT Press, 1979.

[25] LEVITT, K.; ROBINSON, L.; SILVERBERG, B. "Writing simulatable specifications in SPECIAL", in: *The use of formal specification of software*, Berg, H.K.; Giloi, W.K., ed. Springer-Verlag, 1980.

[26] LISKOV, B.; ZILLES, S., "An Introduction to formal specifications of data abstractions", in: *Current trends in programming methodology*, vol.I, Yeh, Raymond ed, Prentice-Hall, New Jersey, 1977.

[27] MAZZONI, C.J., "Um modelo para avaliação da qualidade de software", Tese de Mestrado, Departamento de Informática, PUC/RJ, outubro de 1981.

[28] Mc CALL, J., "An introduction to software quality metrics", in: *Software quality management*, Cooper, J.D.; FISHER, M.J. ed., Petrocelli Books, 1978.

[29] MYERS, G.J., *Reliable software through composite design*, Petrocelli/Charter, N.Y. 1975.

[30] PEERCY, D.E., "Software maintainability evaluation methodology", *IEEE Transactions on Software Engineering*, SE-7(4), julho 1981.

[31] PETERS, L., "Relating software requirements and design", *Proceedings of the Software Quality and Assurance Workshop*, San Diego, CA, novembro 1978.

[32] PRYWES, N.S., "Automatic generation of computer programs", *Proceedings 1977 IFIPS National Conference*.

[33] PRYWES, N.S., "Automatic generation of computer programs", in: *Advances in computers*, vol.16, M. Rubinoff e M.Yovits, eds, Academic Press, 1977.

[34] ROSS, D.T., "Structured analysis for requirements definition", *IEEE Transactions on Software Engineering*, vol. SE-3, n 1, jan 1977.

[35] ROSS, D.T., "Structured analysis (SA): a language for communicating Ideas", *IEEE Transactions on Software Engineering*, vol. SE-3, n 1, jan. 1977.

[36] SCHINDLER, M., "Today's software tools point to tomorrow's tool systems", *Eletronic Design*, julho 1981.

[37] SILVERBERG, B.A., "An Overview of the SRI hierarchical development Methodology", in: *Software engineering environments*, Hunke, H. ed, North-Holland Publishing Company, 1981.

[38] STAA, A.v., *Engenharia de programas*, Livros Técnicos e Científicos, Rio de Janeiro, 1983.

[39] STAY, J.F., "HIPO and integrated program design", *Tutorial on software engineering*, P. Freeman e A.I. Wasserman, eds, Terceira Edição, 1980.

[40] STEPHENS, S.A.; TRIPP, L.L., "Requirements expression and verification Aid", *Proceedings of the 3rd International Conference on Software Engineering*, Atlanta, Georgia, maio 1978.

[41] TEICHROEW, D.; HERSLEY, E.A., "Computer aided structured documentation and analysis of information processing systems requirements", *ISDOS Project*, University of Michigan, agosto 1976.

[42] TEICHROEW, D.; HERSLEY, E.A., "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems"; *IEEE Transactions on software engineering*, vol. SE-3, n 1, jan 1977.

[43] YOURDON, E.; CONSTANTINE, L.L., *Structured design*, Prentice Hall, N. J. 1979.

[44] WASSERMAN, A.I.; STINSON, S.K., "A specification method for interactive information systems", *Proceedings of Specifications for Reliable Software*, 1979.

[45] WASSERMAN, A.I., "User software engineering and the design of interactive systems", *5th International Conference on Software Engineering*, San Diego, CA, março 1981.

[46] WASSERMAN, A.I., *The user software engineering methodology: an overview*, University of California, San Francisco, Technical Report 56.

[47] WASSERMAN, A.I., "Automated tools in the information system development environment", in: *Automated tools for information systems design*, Schneider, H.J.; Wasserman, A.I., ed, Noth-Holland Publishing Company, IFIP 82.

[48] WILLIS, R.R., "AIDES: Computer aided design of software systems-II", in: *Software engineering environments*, Hunke, H. ed, North-Holland Publishing Company, 1981.