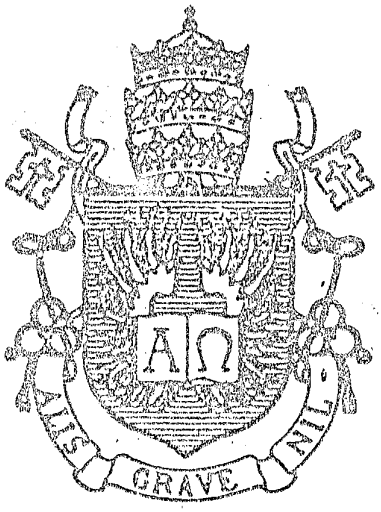

PUC

Série: Monografias em Ciência da Computação, 12/89

UMA PROPOSTA DE ARQUITETURA PARA AMBIENTES PRÁTICOS DE ESPECIFICAÇÃO DE LINGUAGENS

EDWARD HERMANN HAEUSLER
CLOVIS TORRES FERNANDES
JOSÉ LUCAS MOURÃO RANGEL NETTO

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453

RIO DE JANEIRO – BRASIL

PUC/RIO - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, 12/89

Editor: Paulo Augusto Silva Veloso julho, 1989

UMA PROPOSTA DE ARQUITETURA PARA AMBIENTES PRÁTICOS DE ESPECIFICAÇÃO DE LINGUAGENS

EDWARD HERMANN HAEUSLER[1]
CLOVIS TORRES FERNANDES[2]
JOSÉ LUCAS MOURÃO RANGEL NETTO[3]

[1]Professor Assistente do Departamento de Computação da UFF; parcialmente financiado pela CAPES.
[2]De licença da Divisão de Ciências de Computação do ITA – Instituto Tecnológico de Aeronáutica; parcialmente financiado pela CAPES/PICD.
[3]Parcialmente financiado pelo MCT.

RESUMO

Este trabalho apresenta uma proposta de arquitetura para um tipo de ambiente de software, ao qual nos referimos como ambiente de especificação de linguagens. Esta proposta acompanha a direção de evolução dos ambientes gerais de desenvolvimento de software, rumo à estruturas interativas e incrementais. Apoiando um ciclo de desenvolvimento reduzido, ambientes deste tipo oferecem facilidades para validação parcial de especificações de linguagens através de prototipação rápida. Tais ambientes, por serem mais práticos, deverão incentivar o uso de métodos formais.

Palavras-chaves:

ambiente de especificação de linguagens, ambiente sequencial, ambiente interativo e incremental, prototipação rápida.

ABSTRACT

We introduce a proposed structure for a kind of software environment, which we refer to as a language specification environment (LSE). This proposal follows the trend of general software development environments, towards interactive, incremental structures. Due to the reduced development cycle they support, environments of this kind offer facilities for partially validating language specification, through rapid prototyping. We foresee that LSE's, due to the ease of practical use, will forward the use of formal methods.

Keywords:

language specification environment, sequential environment, interactive, incremental environment, rapid prototyping.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

SUMÁRIO

1. Introdução

Procuramos neste trabalho discutir as características que devem ter ferramentas e ambientes para apoiar o uso de métodos formais no desenvolvimento de software. Nosso principal objetivo é a determinação de quais dessas características são adequadas para permitir a usuários não treinados no uso de métodos formais (ou não especialmente inclinados ao seu uso) o emprego de tais métodos, de forma que as vantagens superem de maneira sensível as dificuldades iniciais encontradas. Em teoria, sabe-se que o uso de tais técnicas traz mais vantagens do que problemas, mas isso só é facilmente observável para pessoas que não oferecem resistência ao uso de formalismos.

Os ambientes estudados neste trabalho são chamados *ambientes de especificação de linguagens*. Mostramos aqui que os ambientes de especificação atuais podem não ser muito práticos, o que evidencia a necessidade de se projetar ambientes mais práticos e amigáveis. Tais ambientes - *ambientes de especificação de linguagens interativos e incrementais* - tem suas principais características derivadas dos ambientes interativos atuais.

Esses ambientes tem, em geral, como característica cobrir o ciclo completo de desenvolvimento escolhido, oferecendo facilidades desde a fase de especificação até a fase de manutenção. No nosso caso, entretanto, usaremos um ciclo de desenvolvimento mais reduzido, voltado para a validação de especificações através de prototipação rápida. Por essa razão, fatores como eficiência, estrutura, manutenibilidade, manipulação completa de erros e documentação não são dominantes.

Tais ambientes encontram uma vasta gama de aplicações, entre as quais podemos destacar:

- no ensino de técnicas formais de especificação de software e de construção de compiladores.

- no projeto de linguagens de programação e na construção de compiladores usando técnicas formais e automatizadas.

- no projeto e construção de editores, formatadores de documentos, e software similar.

No projeto de linguagens de programação, esse sentido de prototipação pode ser extremamente útil. Podemos experimentar formas distintas de tratar tipos em uma linguagem, diferentes mecanismos de abstração, de tratamento de exceções ou de comunicação entre unidades de programa, aprender com seus efeitos e, só após essa avaliação, fazer as escolhas finais de projeto da linguagem. A situação no caso das demais aplicações é semelhante.

Nas seções a seguir fazemos uma revisão das características principais das ferramentas e ambientes destinados à implementação e/ou especificação de linguagens de programação. A seção 6 apresenta nossa proposta de arquitetura para um ambiente de

especificação interativo e incremental, como acreditamos que devam ser tais ambientes.

2. Implementação de linguagens de programação

O problema da compilação (ou tradução) de linguagens de programação leva, como se sabe, a programas grandes e complexos. Por essa razão, um esforço considerável tem sido dispendido no estudo das técnicas de compilação, e, para a maioria das fases em que se subdivide o processo de compilação, já é usual, hoje, o emprego de algum tipo de ferramenta automática, durante a geração de um compilador.

Entretanto, ainda se sente falta de um *compilador de compiladores* de qualidade industrial ("production-quality"), completo. Assim, mesmo quando as linguagens possuem especificações formais, essas especificações são tipicamente usadas para documentação não ambígua, e não para implementação ou prototipação rápida. Como formalização significa possibilidade de mecanização, seria de se esperar que como resultado do esforço de pesquisa dispendido nesses problemas, pelo menos já estivessem disponíveis:

- ambientes que permitissem a especificação completa de uma linguagem de programação, e sua validação, com apenas um razoável esforço, de forma simples, sem ambiguidade, através de um conjunto adequado de ferramentas integradas.

- a partir dessa especificação, ferramentas capazes de produzir compiladores com qualidade de produto, com um mínimo de esforço dispendido adicional.

Ambos os objetivos, entretanto, ainda não foram atingidos; ainda não é possível, por exemplo, construir, confortavelmente,uma especificação formal (por exemplo uma gramática de atributos) e uma implementação (por exemplo um compilador) para uma linguagem, que seja, digamos, do porte da linguagem Ada[1] [Ada 83]. Neste artigo procuramos apresentar soluções para o primeiro desses problemas, para o caso de linguagens de dimensões reais.

Falando do caso particular da especificação algébrica, Sueli Mendes [MeAg88] mostra as dificuldades de produzir especificações manualmente, sem ajuda automatizada. Acrescenta ainda que esse tipo de dificuldade é comum a todos os métodos de especificação. A produção de boas especificações sem erros sintáticos, e, principalmente sem erros semânticos é uma tarefa árdua.

Uma maneira de resolver este problema consiste em desenvolver ferramentas automatizadas, que ajudam na construção das especificações, e verificam sua correção, dos pontos de vista sintático, e semântico. Ambientes que incluem ferramentas para, a

---

[1] Ada é uma marca registrada do DoD, o Departamento da Defesa americano.

partir dessas especificações, passar à construção automatizada de implementações formam a continuação natural desse processo.

3. Desenvolvimento de programas e ambientes

Para provar que um programa está correto com relação à sua especificação, esta deve ser formalizada, para que possa ser usada com métodos de prova (formais), e portanto, podemos pressupor o uso de alguma linguagem de especificação formal.

Na área particular de implementação de linguagens de programação, os sistemas formais, nos quais os métodos formais estão baseados, são matematicamente bem definidos. Falta ainda, no entanto, uma linguagem de especificação formal padronizada, como fica bem claro no caso da semântica denotacional [Tenn76, Tenn77, Stoy77, Moss89], para a qual se podem encontrar diversas notações na literatura. Um dos primeiros passos é a seleção de uma linguagem de especificação formal para o método formal escolhido.

Isto posto, podemos aplicar metodologias de desenvolvimento de programas ao projeto de linguagens de programação, compiladores e intepretadores. Nesta situação particular onde se vislumbra um uso intensivo de ferramentas poderosas para o desenvolvimento de software, do ponto de vista geral do projetista de software temos um cenário que pode ser representado pelo ciclo da Figura 1:

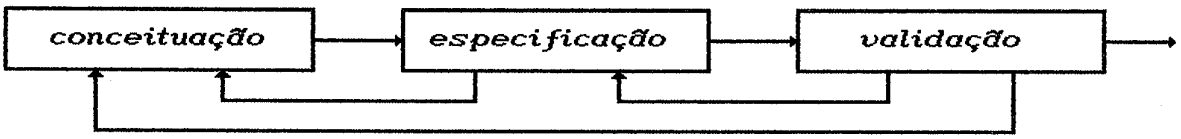

Fig.1: O processo de desenvolvimento de software escolhido

O diagrama acima explicita os laços de realimentação entre as diversas fases, adaptando ao nosso caso os conceitos de [LeST84] e [Blum86] e de desenvolvimento com uso intensivo de ferramentas e prototipação [Grah89].

Nesse ciclo, o primeiro passo, o passo de *conceituação* se relaciona com o processo de definição da linguagem fonte, e ocorre, em geral, de maneira informal, ou apenas parcialmente formal. Um exemplo desta última situação é aquele em que se pretende uma segunda implementação (ou a implementação de um dialeto) de uma linguagem de programação já existente. Neste caso, a implementação anterior e sua documentação formam, em conjunto, uma especificação da linguagem, à qual se devem acrescentar os detalhes que diferenciam a nova implementação da anterior.

O passo a seguir, o passo de *especificação*, é aquele em que é feita, de acordo com o método formal escolhido, a formalização do produto do passo anterior, que pode então ser visto como uma

especificação informal (ou parcialmente formal, como vimos acima) e incompleta da mesma linguagem de programação.

O último passo é o passo de *validação*. Neste passo devemos validar o que foi especificado em relação à conceituação pretendida. Neste caso, devemos preliminarmente obter uma implementação correta (em relação à especificação). Devemos então verificar, através do exame dos resultados da implementação, a concordância da especificação formal em relação à conceituação, que, como vimos, pode ser informal e incompleta. Isto corresponde, na realidade, a determinar (informalmente) se o implementado é o que se pretendia. Assim, os resultados da implementação são confrontados com a conceituação informal, para que se possa ter algum grau de certeza de que a especificação obtida tem, realmente, todas aquelas características desejadas pelo usuário. Esta estratégia segue Turski [Turs86], que assevera que o único modo de validar uma especificação é por meio de testes.

Completado o ciclo acima, com a validação de uma especificação (em relação à conceituação), estarão também validadas as implementações corretas em relação à especificação validada. Essa é a principal vantagem de um ambiente em que se possa validar especificações, com conforto e segurança.

A validação de uma especificação de linguagem é feita através do exame de um protótipo corretamente gerado pelo ambiente, a partir da especificação. A validação pode ser dita parcial, uma vez que se baseia na validação do protótipo pelo uso das facilidades do ambiente que permitem o teste sistemático do protótipo. No que se segue, faremos referência a estes ambientes como *ambientes de especificação de linguagens*. Em princípio, um único ambiente pode simultaneamente incorporar facilidades para especificação e implementação, mas as únicas facilidades para implementação que devem obrigatoriamente fazer parte de um ambiente de especificação de linguagens são aquelas que permitam a construção de protótipos rápidos corretos em relação a uma dada especificação considerada.

4. Caracterização dos ambientes de especificação

Podemos caracterizar os tipos de ambientes de acordo com a taxonomia apresentada em Fernandes e Lucena [FeLu89]. Segundo ela, os ambientes de desenvolvimento de software são caracterizados segundo três critérios: quanto aos paradigmas do processo de software (por exemplo tem-se o paradigma "análise e projeto estruturado"), quanto à escala (número de usuários e tamanho do sistema computacional pretendido) e quanto ao uso de técnicas de IA (inteligência artificial, (por exemplo, bases de conhecimento, provadores de teoremas, interfaces inteligentes).

Quanto aos paradigmas do processo de software, podemos ter fragmentos de paradigmas, como por exemplo "codificação e teste", paradigmas completos, como o já citado "análise e projeto estruturado", e mais de um paradigma, como por exemplo ambos os

paradigmas "análise e projeto estruturado de Yourdon e Constantine" e "metodologia de Jackson".

Quanto à escala, modelam-se os ambientes de acordo com o número de usuários do ambiente e com a complexidade do sistema alvo. Essencialmente, temos quatro níveis: nível individual, nível família (aproximadamente dez usuários), nível cidade (vários projetos, ambiente possivelmente distribuído) e nível estado (onde se fazem necessárias políticas de padronização de linguagens, de protocolos, e semelhantes).

Quanto ao uso de técnicas de IA, temos ambientes que não fazem uso de tais técnicas (os ambientes de programação comuns, como o Cornell Program Synthesizer - CPS [TeRe80], os ambientes gerados pelo Gerador de Sintetizador (Synthesizer Generator) [ReTe88], o Interlisp [TeMa81]), o Cépage [Meye88], e os ambientes gerados pelo Gandalf [HaNo86]; temos os que apresentam alguma ferramenta inteligente (base de conhecimentos e máquinas de inferência) e temos, finalmente, os que apresentam um projeto arquitetural fortemente influenciado por técnicas de IA, como os projetos ETHOS [Luce87] e SAFO [ViCa85]. No nosso caso, como não pretendemos utilizar, pelo menos por enquanto, técnicas de IA, vamos caracterizar os ambientes sem esse critério.

A área A da Figura 2 abaixo caracteriza os ambientes de programação comuns que suportam, por exemplo, a codificação, testes e depuração em uma linguagem. A área B caracteriza os ambientes de programação que suportam todo um ciclo de desenvolvimento de software segundo um paradigma. Atualmente, a maioria dos ambientes de especificação baseados em especificação formal se encontram na área A, fornecendo apenas algumas ferramentas que auxiliam o processo de construção de compiladores. Como exemplo, temos o YACC em ambiente Unix, para especificação da parte sintática [John75].

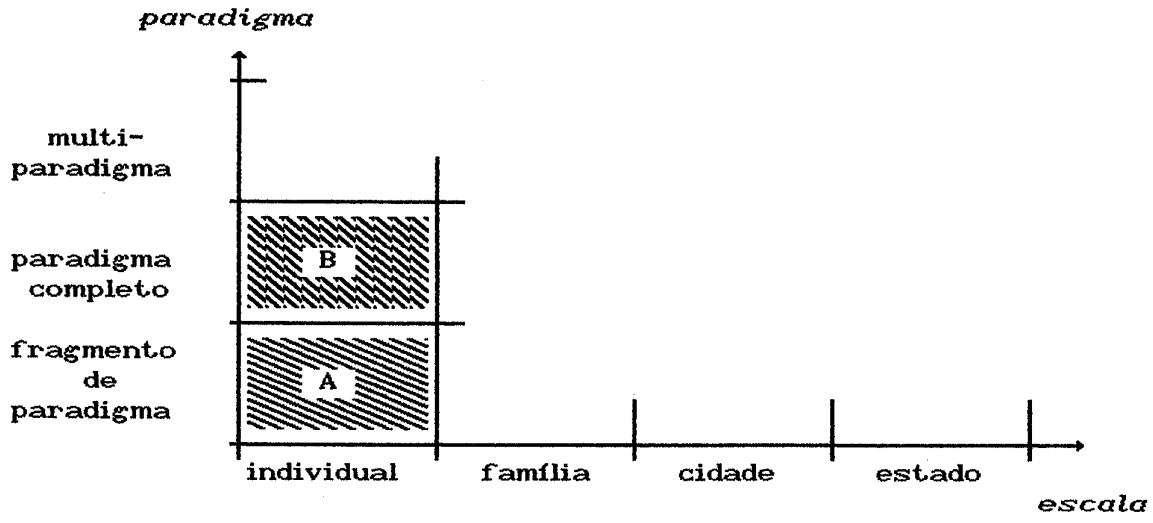

Fig.2 - Caracterização dos ambientes de programação

Alguns sistemas se aproximam dos ambientes da área B, mas falham em aspectos essenciais, no que se refere ao apoio à especificação. Um desses sistema é o sistema GAG [KaHZ82], um meta-compilador, onde a especificação de um compilador é feita manualmente, na linguagem de especificação ALADIN. Além disso, todo o processamento nesses sistemas é sequencial ("em batch"). Assim, quando um erro é descoberto em alguma fase do ciclo de desenvolvimento, a especificação fonte é corrigida e toda ela tem de ser re-submetida ao sistema.

Sistemas semelhantes ao GAG não oferecem facilidades para a construção ou alteração de uma especificação: uma especificação é tratada monoliticamente. No entanto, de acordo com o ciclo apresentado na Figura 1, a alteração de especificações é um passo normal durante a fase de validação, e deve ser apoiada pelo ambiente. Assim, três pontos devem ser considerados quando se contempla a construção de um ambiente de especificação:

- devem ser oferecidas facilidades para a construção de especificações de forma interativa e incremental. Essas facilidades devem apoiar também o processo de alteração de especificações.

- devem ser oferecidas facilidades para a verificação da sintaxe, consistência e de outras propriedades das especificações sendo todo o processamento interativo e incremental.

- devem ser oferecidas facilidades que possibilitem o desenvolvimento incremental de protótipos, para facilitar a validação de especificações incompletas (isto é, que só podem gerar protótipos incompletos).

5. Ambientes de especificação sequenciais

Um ambiente de especificação sequencial é um ambiente preparado para receber uma especificação construída manualmente, através de um analisador ("front-end"), cuja saída é uma representação interna da especificação, usada nas fases posteriores, de maneira sequencial. Em tais ambientes, quando um erro é encontrado, a especificação fonte deve ser corrigida manualmente, e re-submetida por inteiro.

Um exemplo deste tipo de ambiente é o que apoia a linguagem OBJ [GoTa86], [DuFi87] no contexto de especificação algébrica. OBJ é uma linguagem formal para escrever e testar especificações algébricas, mas, além disso, é também uma linguagem de programação, e as especificações geradas podem ser executadas.

Outros exemplos, agora no contexto de gramáticas de atributos, são os sistemas LinDA [RaWi89], HLP84 [KNRS86] e o já citado GAG. Também nesses ambientes uma especificação completa da linguagem fonte (por exemplo sintaxe e semântica) deve ser construída inteiramente de forma manual para submissão ao sistema.

Esta especificação é relativamente fácil de se obter no caso de linguagens de programação pequenas; no caso de uma linguagem com as dimensões da linguagem Ada, essa tarefa é bem difícil, embora não impossível [Uhl 81]. Essa tarefa é dificultada pela necessidade de, a cada vez que erros são encontrados, corrigir e re-submeter a especificação por inteiro.

No caso do ambiente LinDA, um compilador para uma linguagem L pode ser gerado a partir de uma especificação que consiste em uma gramática de atributos, com atributos que podem ser avaliados em um passo da esquerda para a direita [Boch76], e que se baseia, portanto, em um analisador descendente (sintaxe abstrata). A análise sintática do compilador de L, por seu lado, se baseia em outra gramática (sintaxe concreta), que cobre a gramática da sintaxe abstrata, e será feita usando um analisador ascendente R*S simples [Rang88]. A especificação inclui adicionalmente a correspondência semântica entre as regras das duas gramáticas. A Figura 3 reflete esse processo, e inclui um analisador sintático simulado para L, que recebe como entrada uma árvore sintática concreta.

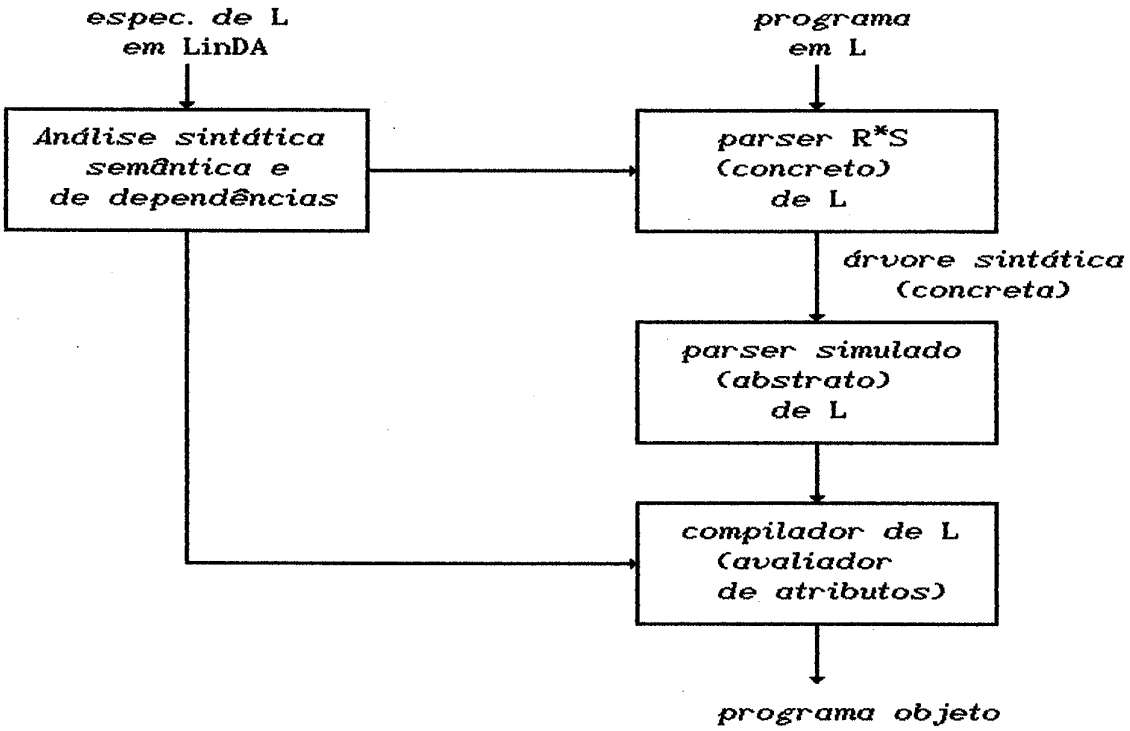

Fig.3: A estrutura do ambiente LinDA

A especificação completa em LinDA é submetida ao ambiente, e deve ser re-submetida cada vez que um erro sintático, semântico, ou nas dependências entre os atributos é encontrado. O sistema não dispõe de facilidades que evitem esse tedioso processo. No entanto, esta é uma atitude consciente, uma vez que o objetivo

principal desta pesquisa está na definição de uma linguagem de especificação adequada para gramáticas de atributos baseadas em duas gramáticas e dois tipos de analisadores acoplados.

No que se segue, não vamos discutir ambientes específicos, mas vamos apenas supor que o ambiente em questão visa atender um ciclo de desenvolvimento reduzido como já apresentado. Uma arquitetura abstrata adequada para ambientes que suportam este tipo de ciclo está descrita na Figura 4.

A primeira idéia para melhorar a interação de tais ambientes com o usuário é construir um editor dirigido pela sintaxe para a linguagem de especificação - o editor de especificação. Assim, o projetista pode usar o sistema para ajudar a elaborar a especificação do compilador, tendo garantia de que, ao terminar a tarefa, possui uma especificação fonte sem erros sintáticos ou de semântica estática. Esse editor substitui o bloco *Analisador de especificações* do ambiente abstrato da Figura 4, de forma que passamos a ter a arquitetura abstrata da Figura 5. Um exemplo é o meta-compilador MetaUncle [Tarh89], que possui um editor de especificação da gramática de atributos aceita pelo ambiente.

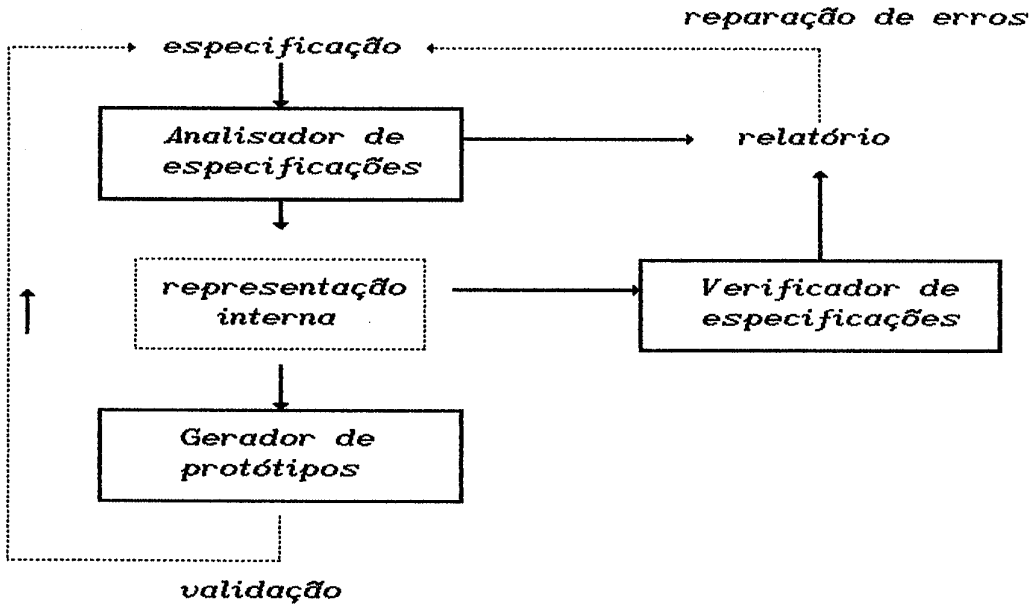

Fig.4: Ambiente de especificação sequencial

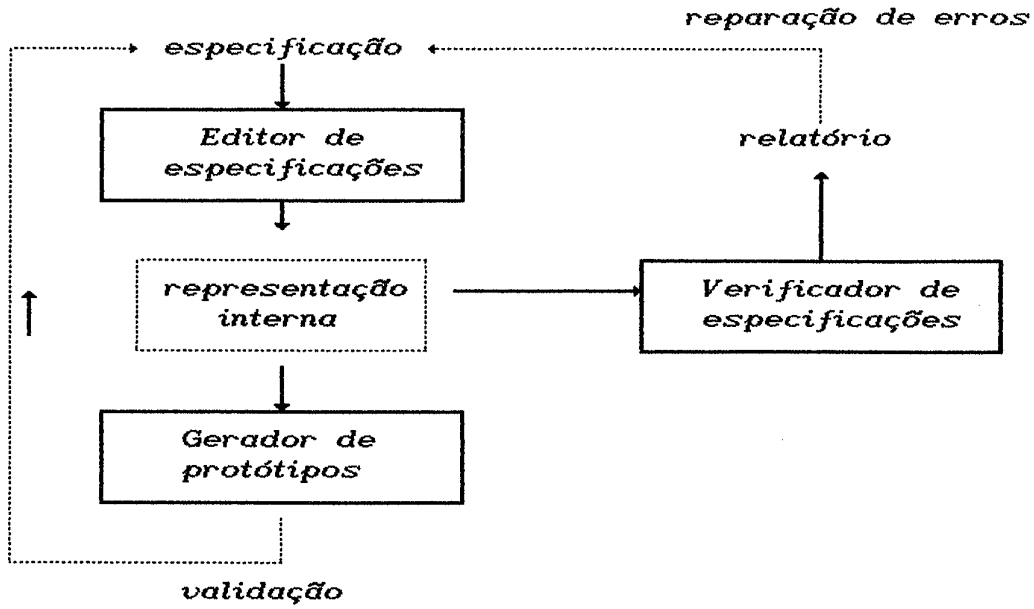

Fig.5:
Ambiente de especificação sequencial com editor de especificações

Devemos observar que o editor de especificações facilita o trabalho do projetista quanto à introdução das especificações no ambiente, mas não elimina os problemas dos ambientes sequenciais. Veremos, na seção seguinte, como, através de um paralelo com os ambientes de programação interativos usuais, poderemos determinar algumas outras características desejáveis para os ambientes de especificação.

6. Ambientes de especificação interativos e incrementais

Para determinar de forma sistemática as características que devem ser oferecidas em um ambiente de especificação, examinamos algumas das características dos ambientes de programação interativos usuais, que geralmente se destinam apenas a apoiar as fases de codificação e teste e depuração de programas. Dentre os ambientes examinados, citamos o CPS, o Interlisp, os ambientes para Ada [BuDr81] e o Gerador de Sintetizador. Após este estudo, consideramos que as seguintes propriedades devem ser previstas no projeto de nossa arquitetura:

Quanto à linguagem de especificação:

> *Modularidade:* Definir elementos de uma linguagem (construções tais como comandos e declarações) sem que seja necessário definir toda a linguagem, e sem afetar outras definições já feitas. Após ter sido escolhido o nível de modularidade, pode-se realizar o teste de elementos escolhidos de uma linguagem não completamente especificada,

através de consulta ao usuário sobre o efeito de elementos ainda não definidos, em situações particulares (por exemplo, para estabelecer o efeito de algum comando ainda não definido, durante o teste de um outro comando).

Consistência: Verificar o interrelacionamento entre os elementos já definidos da linguagem, sintática e/ou semanticamente.

Extensibilidade - Permitir ao usuário a introdução de mecanismos auxiliares novos na linguagem de especificação.

Quanto ao ambiente de especificação:

Monitoração: Permitir o acompanhamento contínuo dos testes realizados, oferecendo recursos para que esse acompanhamento seja feito com a velocidade e o detalhe desejados pelo usuário, em cada situação específica.

Reversibilidade: Permitir que qualquer teste seja repetido a partir de um ponto posteriormente especificado, permitindo-se a alteração do contexto em que o teste se realiza.

Integração: Todo o controle do ambiente deve ser feito através da mesma interface, que permite acesso a todas as ferramentas do ambiente. A interface deve ser simples, amigável, e oferecer facilidades de auxílio adequadas; as ferramentas, em seu conjunto, devem funcionar de forma harmônica e integrada.

Interatividade: permitir fazer todas as alterações em tempo real, usando técnicas incrementais.

Com base nas propriedades acima, propomos aqui uma arquitetura geral para ambientes de especificação interativos e incrementais, descrita na Figura 6. Esta arquitetura apresenta todas as propriedades listadas acima.

O usuário interage com o ambiente através da interface comum, que é utilizada para a criação ou alteração de uma especificação, permitindo amplo acesso às outras ferramentas e facilidades disponíveis. As especificações geradas por este editor são sintaticamente corretas por construção, e satisfazem adicionalmente algumas propriedades semânticas. O teste semântico completo, entretanto, deve ser, posteriormente, acionado através da interface comum, e nessa ocasião será feita a verificação semântica restante. Naturalmente, o ambiente só permite a passagem às fases subsequentes após a verificação da especificação.

Dispondo de uma especificação verificada, o usuário pode passar à fase de obtenção da implementação (protótipo). Isto é feito através do acionamento dos módulos *Gerador Incremental de Editor*, e *Gerador Incremental de Interpretador*. O primeiro destes módulos gera um editor de estruturas específico para a linguagem considerada, cuja saída é uma representação de um programa que,

por construção, já é correto sintaticamente, e que já teve algumas propriedades semânticas verificadas. Essa representação interna (possivelmente uma árvore abstrata decorada) é guardada na *Base de Objetos*, e pode ser submetida ao Interpretador gerado pelo segundo módulo acima mencionado. Este se encarrega da verificação semântica adicional, e da execução do programa, para que seus efeitos possam ser confrontados com os efeitos pretendidos.

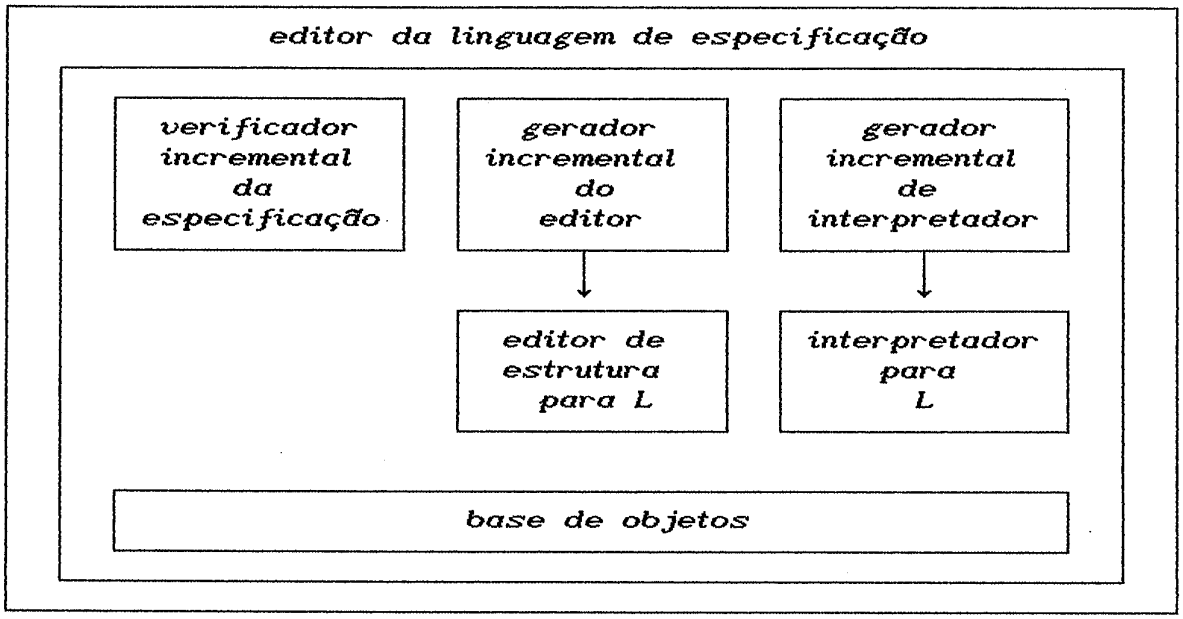

Fig.6: Proposta do ambiente de especificação

Uma observação que cabe fazer aqui é que o nome *interpretador* é usado apenas por ser sugestivo; na realidade a forma de implementação da máquina virtual da linguagem especificada é irrelevante.

O *Base de Objetos* deve ser capaz de armazenar todas as classes de objetos manipulados pelo ambiente: especificações, funções auxiliares, programas em diversas linguagens, e módulos auxiliares do interpretador. Especificada (total ou parcialmente) uma linguagem L, o usuário pode criar ou alterar, através do editor específico de L, um programa, cuja representação será armazenada na Biblioteca; qualquer programa assim gerado pode ser executado pelo interpretador específico de L. Naturalmente, no caso de uso de construções ainda não introduzidas na especificação de L, a execução do interpretador será interrompida, e o usuário será consultado para fornecer instruções quanto à continuação ou interrupção da execução. A execução pode também ser re-inicializada a partir de um ponto indicado pelo usuário, como mencionado acima.

Instâncias da arquitetura aqui proposta podem ser ambientes para apoiar abordagens de especificação semântica, como semântica

denotacional e semântica operacional, em particular usando técnicas como VDL (Vienna Definition Language) [Paga81], VDM (Vienna Development Method) [JoBj82] e gramáticas de atributos. As classes de ambientes assim obtidas deverão diferir fundamentalmente no módulo *Verificador de Especificações*, que deverá refletir a técnica específica usada; algumas outras alterações menores poderão também ser necessárias.

Conclusão

Discutimos, neste trabalho, uma proposta de arquitetura de ambientes a que chamamos de ambientes de especificação. Essa proposta segue a direção de evolução dos ambientes em geral rumo a estruturas interativas e incrementais. Apoiando um ciclo de desenvolvimento reduzido, esses ambientes oferecem facilidades para a validação de especificações por meio de prototipação rápida. Esses ambientes mais práticos deverão incentivar o uso de métodos formais, mesmo no caso de usuários não treinados em tais métodos.

Nossa análise se refere especificamente a ambientes isolados destinados à criação de especificações, que possam ser usadas para produzir implementações de alta qualidade, confiáveis e fáceis de manter. Essas implementações podem ser efetivamente construídas usando ambientes e ferramentas de desenvolvimento usuais, externos ao ambiente em que as especificações foram desenvolvidas. Uma alternativa interessante é a de adicionar tais ferramentas a um ambiente de especificação, que, assim ampliado, constituir-se-ia então em um ambiente maior, destinado a suportar o ciclo de desenvolvimento completo.

A estrutura aqui proposta ainda não se encontra implementada, havendo planos nesse sentido [FeRa89]. Somente a experiência com um ou mais ambientes construídos de acordo com as idéias aqui apresentadas poderá realmente servir para sua validação. Por outro lado, diversas extensões possíveis poderão também ser consideradas, entre as quais a introdução de técnicas de Inteligência Artificial, principalmente no projeto da interface do Ambiente, mas preferimos deixar tais extensões para uma fase posterior.

Reconhecimento: Agradecemos a Júlio César Leite pela leitura de versões anteriores deste trabalho, e pela proveitosa troca de idéias sobre o assunto. Este trabalho reflete, entretanto, a opinião de seus autores.

Bibliografia:

[Ada 83] Ada Joint Program Office. **Ada programming language**. ANSI/MIL-STD-1815A, , US DoD.

[Blum86] BLUM, B.I. Thoughts on the software process. **ACM SIGSOFT Software Engineering Notes, 11**(4):25-27, Aug. 1986.

[Boch76] BOCHMANN, G.V. Semantic evaluation from left to right. **CACM, 19**(2):52-62, Feb 76.

[BuDr81] BUXTON, J. N. ; DRUFFEL, L. E. Requirements for an Ada Programming Support Environment: rationale for Stoneman. **Eng. Environments**, H. Hunke (ed.) North Holland, 1981.

[ClUz77] CLEAVELAND, J.C.; UZGALIS, R.C. **Grammars for programming languages**. Elsevier, New York, NY, 1977.

[DuFi87] DUCE,D.A.; FIELDING, E.V. Formal specification - a comparison of two techniques. **Computer Journal, 30**:316-327, 1987.

[FeLu89] FERNANDES, C.T.; LUCENA, C.J.P. Uma taxonomia para ambientes. **Monografia em Ciências da computação - PUC/RJ**, 1989 (em preparação).

[FeRa89] FERNANDES, C. T.; RANGEL, J. L. **Towards language specification environments.** Em preparação.

[GoTa86] GOGUEN, J.A.; TARDO, J.J. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In: **Software specification techniques**, Gehani e McGettrick (eds.). Addison Wesley, Reading, MA, 1986.

[Grah89] GRAHAM, D.R. Incremental development: review of nonmonolithic life-cycle development models. **Information and Software Technology, 31**(1):7-20, Jan/Feb 1989.

[HaNo86] HABERMANN, A. N.; NOTKIN, D. Gandalf: software development environments. **IEEE Trans. Software Engineering, SE-12**(12):1117-1127, Dec 1986.

[John75] JOHNSON, S.C. Yacc - yet another compiler compiler. **Computing Science Tech. Report 32**, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[KaHZ82] KASTENS, U.; HUTT, B.; ZIMMERMANN, E. GAG A pratical compiler generator. **Lecture Notes on Computer Science, v. 141**. Springer-Verlag, Berlin, 1982.

[KNPS88] KOSKIMIES, K.; NURMI, O.; PAAKI, J.; SIPPU, S. The design of a language processor generator. **Software Practice and Experience, 18**(2): 107-135, 1988

[LeST84] LEHMAN, M.M.; STENNING, V.; TURSKI, W.M. Another look at Software Design Methodology. **ACM SIGSOFT Software Engineering Notes, 9**(2):38-53, 1984.

[Luce87] LUCENA, C.J.P. Subsídios à estação de trabalho ETHOS. **Reunião de trabalho do projeto ETHOS,** Petrópolis, Julho, 1987.

[MeAg88] MENDES, S.B.T.; AGUIAR, T.C. **Métodos para especificação de sistemas.** III EBAI, Curitiba, 1988.

[Meye88] MEYER, B. Cépage: toward computer-aided design of software. **The Journal of Systems and Software, 8,** 1988, 419-429.

[Moss89] MOSSES, P. A practical introduction to denotaional semantics. **Lectures Notes for IFIP State of the Art Seminar on formal Description of Programming Concepts,** Petrópolis, April 1989.

[Paga81] PAGAN, F.G. **Formal specification of programming languages.** Prentice-hall, Englewood Cliffs, NJ, 1981.

[Rang88] RANGEL NETTO, J.L.M. Manual de aplicação do sistema de geração de analisadores sintáticos R*S Simples. **Monografias em Ciências da Computação, nr. 11,** PUC/RJ, 1988.

[RaWi89] RANGEL, J.L.; WIENKOWSKI, E. O sistema gerador de avaliadores de atributos LinDA. Em preparação.

[ReTe88] REPS,T. W.; TEITELBAUM, T. **The Synthesizer Generator: a system for constructing language-based editors.** Springer-Verlag, New York, N.Y. 1988.

[Stoy77] STOY, J.E. **Denotational Semantics: the Scott-Strachey approach to programming language theory.** MIT Press, Boston, MA, 1977.

[Tarh89] TARHIO, J. ATTE Editor for attribute grammars. **First Finnish-Hungarian Workshop on Programming Languages and Software Tools,** Szeged, Hungary, August, 8-11, 1989.

[TeMa81] TEILTELMAN, W.; MASINTER, L. The Interlisp programming environment. In: **Software Development environments,** Computer Society Press, Los Alamitos, CA, 1981. pp. 73-81.

[Tenn76] TENNENT, R.D. The denotational semantics of programming languages. **CACM, 19**(8):437-453, Aug. 1976.

[Tenn77] TENNENT, R.D. Language design methods based on semantics principles. **Acta Informatica, 8**:97-112, 1977.

[TeRe80] TEILTELBAUM, T.; REPS, T.W. The Cornell Program Synthesizer: a syntax-directed programming environment. **Tech. Report TR 80-421**, Department of Computer Science, Cornell University, May 1980.

[Turs86] TURSKI, W. M. And no philosopher's stone, either. **Information Processing, 86**(10): 1077-1080, 1986.

[Uhl 81] UHL, J. An attributed grammar for Ada. **Report #25/81**, Universitat Karlsruhe, Inst. Informatik II, 1981.

[ViCa85] VIEIRA, N.J.; CARVALHO, R.L. SAFO - um ambiente para desenvolvimento de protótipos de sistemas especialistas baseado em lógica. In: **Anais do II Simpósio Brasileiro de Inteligência Artificial**, São José dos Campos, 1985.