Um Meta-editor de Estruturas

Arndt von Staa *†

Resumo

Serão apresentados os requisitos para a construção de um meta-editor de estruturas, visando, primariamente, a edição e geração de programas segundo óticas modernas de programação. Este editor de estruturas está implementado e faz parte do ambiente TALISMAN.

Essencialmente, um editor de estruturas é um hipertexto com "links" de diversos tipos e "frames" contendo diferentes fragmentos de texto. O editor consta de duas ferramentas básicas: o editor de estrutura propriamente dito, capaz de criar e manter grafos, usualmente na forma degenerada de árvores; e o editor de formulários capaz de editar campos contendo fragmentos de texto e relações. O editor de formulários pode ser especificado de modo que linearize o conteúdo do hipertexto, gerando código compilável.

1. Introdução

Para aumentar a produtividade do desenvolvimento de sistemas de programação complexos (unidades de qualidade assegurada entregues por unidades de tempo), estão sendo desenvolvidas diversas ferramentas. Em [ML89 e Ch86] podem ser encontradas discussões sobre tais ferramentas.

Em particular, o desenvolvimento de módulos de programa, segundo a ótica introduzida por Modula2 [Wi83] e por programação orientada a objetos [Me88], pode beneficiar- se do uso de um editor de estruturas de programas. Estes editores compõem módulos a partir de pacotes ("procedures", "functions") e de tipos de dados. Cada um destes por sua vez é composto por uma estrutura em árvore definindo o código executável ou declarativo, conforme o caso. É desejável que um editor de estruturas seja capaz de simular estes conceitos em linguagens mais antigas, tais como PASCAL, COBOL etc. Desta forma poder-se-á aproveitar software já desenvolvido, pessoas cujo conhecimento se limita a tais linguagens, e ambientes de processamento onde não existam compiladores para linguagens de programação modernas.

Dentro do escopo do projeto TALISMAN, um ambiente de engenharia de software apoiado por computador, foi desenvolvido um meta-editor de estruturas. Este editor pode ser utilizado para editar as mais variadas organizações de elementos, que tenham por base um par de relações do gênero composição e decomposição.

Neste artigo será discutida a base conceitual para a construção deste editor.

---

* é PhD em Ciência da Computação, Professor Associado no Departamento de Informática da PUC/RJ. Seus interesses de pesquisa são ambientes de engenharia de software assistidos por computador, metodologias de desenvolvimento e controle de qualidade.

Este trabalho recebeu apoio institucional da FINEP

2. Editor de estrutura

Nesta seção discutiremos o que são editores de estruturas. Para isto serão identificados os requisitos básicos destes editores, sem preocupação com possíveis implementações. O que desejamos é identificar a funcionalidade requerida para editores de estrutura que, em principio, manipulem estruturas de programas e estruturas de dados.

Uma estrutura de programa é tipicamente uma árvore onde cada nó corresponde a um sub-programa aberto ("macro") de corpo constante. O nome do nó é a pseudo-operação correspondente à sub-estrutura que emana deste nó. Ou seja, a árvore representa uma estrutura de refinamento onde, ao descer na estrutura, está-se progredindo do abstrato para "o cada vez mais concreto", culminando com operações concretas nas folhas da estrutura.

Assim, um editor de estrutura simples nada mais é do que um editor de uma organização arborescente. Neste caso, dado estar-se sobre um nó corrente da estrutura, as operações de edição se resumem a:

- criar nós de estrutura nas quatro direções de vizinhança do nó corrente;
- definir e alterar os nomes dos nós;
- eliminar o nó corrente, subindo de nível os nós filho;
- eliminar toda a subestrutura que emana do nó corrente;
- copiar e mover sub-estruturas;
- explorar a estrutura.

Editores com estas características são de pequena serventia. Por exemplo, para editar um programa não basta editar a sua estrutura, é necessário, também, associar texto (fragmentos de código) a esta estrutura. Tornar o nome do nó folha uma instrução concreta na linguagem de programação escolhida, atomiza excessivamente a estrutura, criando um considerável "overhead" de navegação sempre que se queira localizar e/ou alterar determinada porção de código.

Surgem, então, *estruturas anotadas*. Nestas cada nó da estrutura possui diversos fragmentos de texto. Para assegurar uma interpretação uniforme da estrutura anotada, cada um destes textos corresponde a uma determinada classe de informação. Por exemplo, para um determinado nó de uma estrutura de programa, um dado fragmento de texto pode descrever informalmente o objetivo da sub-árvore com raíz no nó; outro pode especificar formalmente as assertivas de entrada; outro pode especificar formalmente as assertivas de saída; finalmente outros podem conter fragmentos de código de programa.

Uma estrutura assim anotada corresponde a um hipertexto muito simples [Co87]. Os quadros ("frames") correspondem aos fragmentos de texto, e os enlaces ("links") são determinados pela estrutura.

Diferentes linguagens de programação tornam necessários diferentes fragmentos de texto. Por exemplo, do ponto de vista de declarações de dados (uma sub-linguagem), a estrutura precisa anotar tipos básicos das variáveis, dimensões de vetores etc. Do ponto de vista de código executável (outra sub-linguagem), os textos precisam anotar porções de código incondicional, porções de código de controle etc. Finalmente, do ponto de vista de sub-rotinas (uma terceira sub-linguagem) precisa-se anotar cabeçalhos, assertivas de entrada e saída, etc.

Para generalizar a composição de anotações de cada nó, pode-se utilizar *formulários*. Um formulário especifica e coordena a edição dos textos de cada nó. Para isto o formulário define campos com títulos e referências a fragmentos de texto. Estas referências correspondem a chaves secundárias que, associadas à chave primária do nó, asseguram acesso único

aos fragmentos de texto. Com esta definição, o editor de estrutura poderá armazenar e recuperar estes fragmentos de texto, bastando conhecer somente a identificação (chave primária) do nó ao qual estarão associados os textos.

Em TALISMAN os formulários são definidos através de *programas de formulário*. Estes, por sua vez, são interpretados pelo editor de formulários, produzindo *formulários editáveis* pelo usuário. Os programas de formulário podem ser criados e/ou alterados pelo usuário, fazendo com que o editor seja, na realidade, um meta-editor.

A seguir ilustramos a definição de um programa de formulário simples:

```
Titulo  Nome do nó
Nome                                    - busca o nome do nó
Titulo  Assertivas de entrada
Texto = 1                               - busca o fragmento 1
Titulo  Código fonte
Texto = 2                               - busca o fragmento 2
Titulo  Assertivas de saida
Texto = 4                               - busca o fragmento 4
```

Note que a semântica é atribuida aos fragmentos através do uso que se faz deles. O uso proposto é definido pelo título. Isto generaliza o uso dos textos que passam a não possuir semântica específica do ponto de vista do editor. Para estabelecer uma interpretação perceptível e respeitada pelo editor, podem-se associar "helps" e validadores especializados aos títulos. Os "helps" explicam ao usuário como preencher determinado texto, e os validadores conferem a correção deste preenchimento.

Embora editores de estrutura integrados com editores de formulários simplifiquem a edição de uma coletânea organizada de textos, não resolvem o problema da exploração e da compreensão local do objetivo da estrutura. Isto decorre do fato que o formulário apresenta somente os textos, sendo que a estrutura apresenta os interrelacionamentos. No entanto, para obter uma compreensão local, é necessário visualizar os textos junto com pelo menos os nomes das pseudo-operações, ou seja, os nomes dos filhos do nó em que se está no momento. Agora, os textos, junto com um conhecimento do que realiza cada pseudo-operação, possibilitam a verificação da correção local do programa, tanto do ponto de vista mecânico como do ponto de vista avaliação humana. Isto reduz significativamente o esforço despendido quando da alteração de programas, uma vez que a verificação pode, agora, ater-se somente às porções efetivamente alteradas.

Para resolver este problema podem-se acrescentar campos de enlace aos formulários. Estes campos chamaremos de *relações*. Genericamente, uma relação discrimina os elementos aos quais o elemento corrente está vinculado (relacionamentos, "links"). Esta discriminação pode ser realizada, por exemplo, listando-se os nomes dos elementos relacionados.

Relações devem permitir a navegação direta para os elementos referenciados, bastando, para isto escolher um dos relacionamentos contidos na relação e solicitar que ele se torne corrente.

As relações podem estar intimamente vinculados à estrutura. Por exemplo, pode-se atribuir um nome único a cada nó da estrutura. A relação de estrutura será, agora, uma lista simples de nomes determinando a existência de uma referência do nó corrente a um outro nó da estrutura. Na realidade são duas as relações, uma vez que se está tratando de árvores (no caso geral, grafos dirigidos). Estas relações são: "pai", que relaciona todos os nós da estrutura que são antecessores, e "filho", que relaciona todos os elementos da estrutura que são sucessores do nó corrente.

Esta organização permite editar diretamente a estrutura, sem olhar para os formulários. Permite, ainda editar os formulários e, simultaneamente, editar a estrutura, através da edição das relações, sem olhar para a estrutura. É evidente que nada impede utilizar indiscriminadamente qualquer uma das duas formas de edição.

A seguir ilustramos um programa de formulário com as características descritas acima:

```
Titulo  Nome do nó
Nome
Titulo  Assertivas de entrada
Texto = 1
Titulo  Código fonte anterior aos filhos
Texto = 2
Titulo  Pseudo instruções filho
Relacao Decomp
Titulo  Código fonte posterior aos filhos
Texto = 3
Titulo  Assertivas de saída
Texto = 4
Titulo  Lista de pais, navegação
Relação Comp
```

No caso geral, um formulário pode referenciar relações adicionais que não estão associados diretamente com a estrutura. Isto estabelece relacionamentos entre elementos, mais complexos do que os que seriam definidos estritamente pela estrutura, permitindo, inclusive, a integração entre diferentes linguagens de representação. Todavia, é essencial que cada relação possua uma semântica bem definida.

Os relacionamentos que constituem uma dada relação podem ser definidos explicitamente pelo usuário, por exemplo através da incorporação do respectivo nome diretamente na relação. Os relacionamentos poderiam ser extraídos automaticamente dos fragmentos de texto. Por exemplo, identificando palavras chave contidas nos fragmentos de texto, onde estas palavras chave correspondem aos nomes dos elementos referenciados. Foge ao escopo deste trabalho aprofundar esta discussão. Assumiremos no restante do trabalho que as relações são criadas explicitamente pelo usário, ou através do editor de estrutura, ou através da edição de relações contidas nos formulários.

Com esta generalização tem-se agora hipertextos convencionais [Co 87], possuindo pelo menos dois mecanismos de navegação: ou diretamente através dos elementos contidos nas relações ("links" explícitos), ou através do editor de estrutura que funciona agora como um explorador ("browser").

As organizações discutidas até agora são suficientes para aplicações onde o objetivo é ter acesso rápido a fragmentos de texto. Por exemplo, pode-se criar desta forma um sistema de auxílio hierarquizado, ou mesmo um manual "on-line".

Em adição à exploração convencional da estrutura, pode ocorrer a necessidade de externar a estrutura de modo seqüencial. Neste caso, percorre-se a estrutura em uma determinada ordem e externam-se os fragmentos de texto ao atingir os elementos. Muitas vezes esta forma de externar utiliza diversas ordens de visitação. A geração de texto seqüencial é realizada por *linearizadores*.

Por exemplo, um linearizador PASCAL pode percorrer a estrutura de um procedimento em ordem prefixada externando todas as declarações de dados associadas aos nós visitados. Após, percorre novamente a estrutura, externando, em ordem prefixada, os fragmentos de código de controle e de código executável, adicionando, em ordem pós-fixada, os marcadores de término de comandos compostos ("END").

A linearização depende do conteúdo dos fragmentos de texto. Por exemplo, ao explorar estruturas de programa, existe uma diferença sensível entre estruturas sem controle e estruturas controladas através de comandos tipo "while" ou "if then else". Ou seja, existem aplicações onde a navegação depende não só da existência de relacionamentos, dependendo, ainda, da semântica associada aos fragmentos de texto. A existência de determinado fragmento de texto altera o comportamento do linearizador.

A seguir ilustramos um linearizador simples para programas "C", admitindo a estrutura de controle "guarded command" [Di75]:

```
InicFrm  Lineariza bloco      - define um formulário
   Se Texto = 5               - TRUE se o fragmento 5 existir
      Titulo  if (
      Texto = 5               - condição do 'guard'
      Titulo    ) {
      Frm  Lineariza filhos   - chamada de formulário
      Titulo  ) /* guard */
   Senao
      Frm  Lineariza filhos
   FimSe
FimFrm
InicFrm  Lineariza filhos
   Texto = 2                  - texto fonte antes
   ParaTodos Decomp           - percorre todos filhos
      Frm  Lineariza bloco    - lineariza o filho
   Fim
   Texto = 3                  - texto fonte após
FimFrm
```

Em resumo, um editor de estrutura, junto com um editor de formulário:

- pemite a edição de duas relações R1 e R2 quaisquer, de cardinalidades m para n, assegurando que, se a relação R1 possui um relacionamento do elemento A para o elemento B, então a relação R2 possui o relacionamento do elemento B para o elemento A.
- permite, opcionalmente, a edição de outros pares de relações.
- permite a edição segundo um programa de formulário identificando diversos fragmentos de texto e relações. Cada formulário acessa os atributos de um determinado nó, ou alcançáveis navegando-se a partir deste nó (linearização parcial).
- permite a navegação entre os elementos da estrutura através do caminhamento na estrutura e/ou através da navegação via relacionamentos exibidos no formulário.
- permite a linearização da estrutura segundo alguma regra de caminhamento definida. Esta regra de caminhamento poderá ser sensível ao conteúdo dos fragmentos de texto.

Cabe salientar que a linearização corresponde, na realidade, à definição de um programa de formulário capaz de fazer uma navegação extensa através da estrutura.

O uso de linearizadores programáveis torna possível a geração de texto fonte em qualquer linguagem, necessitando-se, em adição ao que foi visto, somente operadores e variáveis aritméticas (para criar contadores) e operações de formatação (para assegurar margens). Os linearizadores podem ser tornados mais poderosos (por exemplo: sensíveis ao conteúdo) se forem tornadas disponíveis operações básicas de "pattern matching".

3. Requisitos de um editor de módulos

Nesta seção identificaremos propriedades adicionais de editores de estruturas à luz de conceitos modernos de programação encontrados em linguagens de programação tais como Modula2[Wi83] ou utilizadas em linguagens orientadas a objetos[Me88]. Não discutiremos particularidades destes conceitos, uma vez que isto extrapolaria o escopo deste artigo. Conceitos modernos de programação estabelecem tipicamente as seguintes classes de elementos:

- *estrutura de código de programa,* implementam algum procedimento computacional. Este procedimento requer a disponibilidade de dados de entrada (inclusive os fornecidos pelo usuário interativo) satisfazendo determinadas regras de armazenamento e de interpretação. O procedimento produzirá resultados (inclusive os dirigidos para o usuário interativo) satisfazendo regras de armazenamento e de interpretação.
- *pacotes* (funções, subrotinas) que possuem como corpo uma estrutura de programa, e estabelecem uma interface bem definida para este corpo. Esta interface especifica as regras de entrada e de saída. O pacote pode, ainda, estabelecer requisitos de desempenho a serem satisfeitos pelo corpo. É desejado que o comportamento ideal (especificação essencial) bem como as características de funcionamento (propriedades de implementação) sejam totalmente definidos pelo pacote, de modo que não se necessite recorrer ao corpo para saber o que, e em que condições, o pacote faz (encapsulamento de código). Cabe salientar que a noção pacote aqui utilizada difere da utilizada por Ada. Um "package" Ada corresponde, em linhas gerais, a um módulo.
- *estruturas de dados* ("records", "structs"), definem uma organização de memória, e associam a esta organização uma interpretação estática básica. A interpretação estática básica determina as categorias de valores que são válidos em cada campo da estrutura (inteiro, real, ponteiro para tipo A etc.)
- *tipos de dados* (estáticos) que possuem como corpo estruturas de dados. Ao declarar um nome como sendo de um determinado tipo, pode-se alocar espaço de memória para este nome. A organização deste espaço é dada pelo corpo, ou seja, pela a estrutura de dados. Em adição, o tipo de dados define o que vem a ser uma instância válida de um valor deste tipo. Caso a estrutura possua referências a outros tipos, é a definição do tipo que estabelece a validade semântica destas referências. É desejado que os possíveis estados de armazenamento (conjunto de valores válidos) sejam totalmente definidos pelo tipo, mesmo se o conjunto for infinito. Esta definição deve possibilitar o uso do tipo sem que se precise saber detalhes de sua implementação definidos pela estrutura de dados. Cabe salientar que a noção de tipos aqui utilizada se preocupa somente com o conjunto de dados, sem se preocupar com as operações sobre estes dados. Diverge, portanto, da noção de tipo abstrato de dados.
- *módulo de implementação* é um conjunto de constantes, tipos de dados, áreas de dados organizadas segundo estes tipos, pacotes operando sobre estas áreas de dados e, possívelmente um corpo próprio. Um módulo deve implementar um conceito computacional único e bem definido e dever ser compilável independentemente. Tipicamente um módulo corresponde a um tipo abstrato de dados, uma classe, um objeto, um

"driver" de dispositivo etc. O conjunto de elementos que formam um módulo está sujeito a uma série de regras específicas. Estas regras definem ordens legais de operação (por exemplo somente pode ser desempilhado algo que já foi empilhado) bem como comportamento devido ao estado, ou história do processamento (por exemplo, a próxima coisa a ser desempilhada é a última coisa que foi empilhada). O corpo do módulo destina-se a criar um estado inicial válido, bem como a retirar o módulo de atividade assegurando a integridade do restante sistema, por exemplo fechando arquivos, recuperando espaço de memória alocada etc.

- *módulo interface* define uma coletânea de tipos de dados, constantes, espaços de dados organizados segundo estes tipos (variáveis globais, dados herdados) e pacotes estritamente necessários para que se possa utilizar completa e corretamente o conceito computacional implementado. Da mesma forma como para pacotes e tipos, deseja-se poder fazer uso correto de um módulo exclusivamente a partir do conhecimento do seu módulo interface.

As definições acima evidenciam uma hierarquia de abstrações, onde o módulo estabelece a abstração mais limitante e as estruturas de programa e de dados estabelecem as abstrações menos limitantes. Esta visão de níveis de restrição é particularmete interessante pois permite o reuso de estruturas em diferentes módulos, bem como torna menos difícil a verificação da corretude de um módulo. Adicionalmente, as restrições impostas hierarquicamente permitem desenvolver assertivas executáveis [St89] cuja finalidade é verificar se determinado estado computacional é um estado válido.

É evidente que estão embutidas nas definições acima diversas classes de texto. Por exemplo: texto de código executável; texto de código declarativo; texto descritivo informal de pacotes; texto descritivo formal de pacotes; texto descritivo de características de implementação de módulos etc. É evidente, ainda, que diferentes documentos possuem interseções não vazias se forem vistos como conjuntos formados por estes fragmentos de texto. Por exemplo, o usuário de um módulo desejará ver a descrição de um pacote, vinculado ao conceito implementado pelo módulo; o mantenedor do módulo desejará ver este texto vinculado ao corpo do pacote; já um compilador deseja receber as declarações de dados seguidas do código definido no corpo dos pacotes.

Em outras palavras, existem diversas relações entre estes textos dependendo da semântica da leitura que se quer fazer da documentação resultante. Estas relações, em geral, não podem ser geradas automaticamente. Isto decorre do fato que elas têm uma conotação semântica não dedutível da informação contida exclusivamente em cada fragmento de texto. Por exemplo, qual é a razão para que determinado pacote definido no módulo deva estar na interface deste módulo, enquanto que outro não?

De forma semelhante a módulos de programa, podemos ver documentos compostos por:

- *parágrafos,* que contêm fragmentos de texto descrevendo completamente idéias elementares.
- *estruturas de parágrafos,* que compõem texto descrevendo idéias complexas a partir de idéias menos complexas, procurando estabelecer uma ordem "natural" de leitura.
- *capítulos e seções,* que empacotam idéias complexas correlatas, dando a este conjunto um nome que reflita a intenção do conjunto.
- *documentos,* compostos por diversos capítulos.

Se bem que a semântica é muito mais vaga, é perceptível que a estrutura de um documento é semelhante à estrutura de um módulo. Neste caso um documento corresponde a um módulo, uma seção corresponde a um pacote, e uma estrutura de programa corresponde a uma estrutura de parágrafos. Alguns formatadores, por exemplo Ventura[Ve88], utilizam exatamente esta estrutura para organizar documentos.

4. Elemetos de um meta-editor de estrutura

Nesta seção identificaremos os elementos que compõem um meta editor. A instanciação, ou seja, a associação de um determinado significado a cada um destes elementos, permite definir editores específicos.

Estrutura é uma hierarquia definida por duas relações R1 e R2, tal que, para dois elementos A e B, se A R1 B, então B R2 A. A cardinalidade das relações é qualquer. Os elementos A e B pertencem a uma mesma classe de elementos.

Semântica da estrutura é a forma como uma estrutura deve ser interpretada. A semântica é definida através dos linearizadores. O resultado a ser produzido, determina a composição dos formulários e a composição das relações editáveis pelo editor de estruturas.

Bloco é uma unidade da estrutura e que corresponde a um único elemento. O tipo do elemento bloco depende da linguagem de representação usada corrente e da semântica da estrutura. Por exemplo:

- ao editar uma estrutura tipo programa PASCAL, cada bloco corresponde a uma pseudo-instrução deste programa.
- ao editar uma estrutura tipo dado PASCAL, cada bloco corresponde a um agregado de dados ou a um dado elementar.
- ao editar uma estrutura tipo composição de "procedures" PASCAL, cada bloco corresponde a uma "procedure" e a estrutura define o aninhamento de "procedures".

Pacote é um elemento que define a interface de uma estrutura com outros elementos. A semântica de um pacote depende da semântica da estrutura de seu corpo. Assim se o corpo é uma estrutura de dados, o pacote será um tipo. Se o corpo é uma estrutura de parágrafos, o pacote corresponde a uma seção, etc.

Corpo de pacote é uma estrutura vinculada a um pacote. O corpo define a implementação do pacote. Podem existir diversos corpos associados a um pacote. Cada um deles corresponde a uma versão de implementação do pacote.

Bloco chama é um bloco que referencia um pacote. Por exemplo: em programa é uma chamada de sub-rotina; em dados é uma referência a um valor de determinado tipo de dados. Ou seja, o bloco chama estabelece uma relação de uso de pacotes diferente da relação corpo, que vem a ser a implementação do pacote.

Módulo é uma coletânea de pacotes (programa e dado) e um corpo. O conceito de módulo aqui utilizado é derivado de conceitos de programação orientada a objetos, sendo que um módulo corresponde, em linhas gerais, a uma classe.

Configuração de módulo é um conjunto de definições e linearizadores, identificando as classes de elementos e as relações entre os elementos que compõem um módulo. Do ponto de vista do editor de estruturas, um módulo pode ter diversas finalidades, é a configuração que define para que se serve um módulo.

Especificação do módulo é a coletânea de informações necessárias para o desenvolvedor possa localizar, identificar e fazer correto uso do módulo. Fazem parte da especificação do módulo: as regras de uso dos pacotes, as assertivas de entrada e saída dos pacotes visíveis na interface, os requisitos de desempenho assegurados etc. (módulo de especificação).

Interface do módulo é a coletânea de informações necessárias para que o compositor de módulos (compilador sensível à definição de interface, p.ex. Modula2) possa fazer uso do módulo sem conhecer a sua organização interna. Fazem parte da interface do módulo:

procedimentos ou métodos (pacotes) exportados; tipos de dados (pacotes) exportados etc. (módulo de definição).

Implementação do módulo é a coletânea de pacotes, tipos e corpo necessárias para que o compilador possa gerar código, assegurando a completa e correta implementação dentro das expectativas de desempenho estabelecidas (módulo de implementação).

O ambiente TALISMAN possui um editor de estruturas com as características descritas acima. Este editor é controlado por tabelas que definem:

- lista de classes de elementos, agregadas por linguagem de representação.
- relações direta e correspondente "inversa" editáveis, para cada classe de elemento.
- classes de elementos (dependente da linguagem de representação) que correspondem a blocos, pacotes e módulos.
- relações utilizadas diretamente pelo editor de estruturas (dependente da linguagem de representação) bloco.pai – > bloco.filho, pacote – > corpo, bloco.chama – > pacote, módulo – > pacote, módulo – > pacote.interface, módulo – > corpo. As inversas estão em outra tabela já discutida.
- especificação de formulários para cada classe de elementos.
- linearizadores para cada configuração de módulo.

Com estas tabelas forma definidos diversos editores de estrutura que permitem gerar módulos de especificação, módulos de definição ("include file") e módulos de implementação para módulos a serem compilados em "C", DBaseIII ou PASCAL.

5. Conclusão

Foram identificados os elementos básicos que formam módulos de programa segundo a ótica Modula2. A partir disto foram identificadas as diretivas que instanciam um editor de estruturas e de formulários genéricos, necessárias para editar e gerar código compilável de estruturas que correspondam a módulos em uma linguagem de programação específica.

Em princípio pode-se gerar código para diversas linguagens mesmo que não sigam a estrutura definida por Modula2. Isto é o caso, por exemplo, das linguagens DBase, PASCAL, COBOL e FORTRAN.

Foram desenvolvidos alguns programas utilizando os editores instanciados para gerar código "C". Este uso demonstrou a viabilidade, a praticidade e eficácia desta forma de trabalho. Em virtude dos editores terem sido desenvolvidos recentemente, não se tem, ainda, um conjunto de resultados experimentais expressivo. O desenvolvimento de TALISMAN foi realizado com apoio de MOSAICO, que vem a ser um editor de estruturas com características semelhantes, porém menos poderoso. O uso desta ferramenta é um dos principais fatores para se ter alcançado elevada produtividade (cerca de 70 linhas de código puro, sem comentário, por pessoa.dia). É de se esperar que com o editor de estrutura de TALISMAN venha-se a alcançar resultados no mínimo semelhantes.

Referências bibliográficas

[Ch 86] Charette, R.N.; *Software Engineering Environments;* McGraw Hill; 1986

[Co 87] Conklin, J.; "Hypertext: An Introduction and Survey"; *IEEE Computer* 20(9); setembro 1987; pags 17-41

[Di 75] Dijkstra, E.W.; "Guarded Commands, Nondeterminancy and Formal Derivation of Programs"; *Communications ACM* 18(8); agosto 1975; pags 453-457

[Me 88] Meyer, B.; *Object Oriented Software Construction*; Prentice Hall; 1988

[ML 89] McLure, C.; *CASE is Software Automation*; Prentice Hall; 1989

[St 89] Staa, A.v.; "Teste de programas com o computador"; *9o. Congresso Nacional de Informática*; Uberlândia, MG; Sociedade Brasileira de Computação; julho 1989

[Ve 88] –; *Xerox Ventura Publisher*; Xerox Corp.; 1988

[Wi 83] Wirth, N.; *Programming in Modula 2*; Springer; 1983