Concepção, Especificação e Implementação de Processadores de Linguagens Visuais

Maria João Varanda Jorge Gustavo Rocha
Pedro Rangel Henriques

{mjp,jgr,prh}@di.uminho.pt
Grupo de Especificação e Processamento de Linguagens
Departamento de Informática
Universidade do Minho
Braga – Portugal

Resumo

Com o estudo aqui relatado —realizado no âmbito do projecto SARA— pretendeu-se avaliar a adequabilidade dos métodos e técnicas usadas no desenvolvimento de compiladores tradicionais à construção de processadores para linguagens visuais.

Defendemos a ideia pragmática de que o processador final pode ser criado pelo acréscimo de um editor gráfico, especializado para uma linguagem visual, no topo de um compilador textual, desenvolvido com base numa gramática de atributos.

Para especificar o processador visual faz, ainda, parte da nossa abordagem recorrer a um formalismo modular —MASOViLa— desenvolvido pelo nosso grupo e que aqui iremos, também, apresentar.

Mostraremos a viabilidade das ideias, aplicando-as a um caso prático —surgido no contexto do projecto LEPAForM— que tem como objectivo a criação e a especificação de uma linguagem visual para descrição de diagramas de estados temporizados, *Timed State Charts* (TSC), e a construção de um tradutor dessa mesma linguagem para axiomas em *Real Time Logic* (RTL). Esse ambiente permite concretizar o princípio da *aplicação pragmática de métodos formais na descrição do comportamento de sistemas reactivos.* Do ponto de vista do utilizador, a linguagem visual TSCvl substitui a linguagem textual TSC, permitindo manipular directamente os diagramas de transição de estados temporizados.

A linguagem visual foi formalmente definida, em termos sintáticos e semânticos, no formalismo MASOViLa; o compilador, baseado no princípio da *tradução orientada pela semântica*, foi produzido automaticamente pelo gerador Eli.

1 Introdução

As linguagens (de programação, de especificação, de controlo, de interrogação a bases de dados, ...) visuais estão actualmente em franca expansão. O indiscutível aumento de amigabilidade na interacção Homem-Computador é suficiente para justificar essa aceitação e consolidação crescentes.

É, então, lícito que as pessoas ligadas à produção de *software* se interroguem sobre como desenvolver (especificar e construir os programas) processadores para linguagens visuais. Na secção 2, são aprofundados os argumentos que levam a esta expansão.

O nosso grupo vem acumulando, há vários anos, experiência no desenvolvimento formal de processadores para as convencionais linguagens textuais, usando gramáticas de atributos para descrever a semântica das linguagens, e mesmo o comportamento dos tradutores, e recorrendo a ferramentas sofisticadas para gerar os programas finais a partir das especificações gramaticais.

Por isso, a questão acima enunciada —*o que fazer, e como fazer, para tratar linguagens visuais*— pareceu-nos bastante pertinente e carente duma resposta urgente.

Formamos, assim, a proposta que discutimos neste artigo.

Quanto à especificação, propomos a adopção do formalismo MASOViLa[Roc95], que é uma adaptação do MASLP[Hen92] para tratar as questões espaciais que caracterizam as linguagens visuais. O princípio subjacente ao MASLP, e por consequência ao MASOViLa, é a fusão da ideia básica da gramática de atributos transformacional com a filosofia da modularidade orientada ao símbolo da gramática. O formalismo MASOViLa será tratado com maior detalhe na secção 3.

No que respeita à arquitectura do processador para linguagens visuais, dado que ainda não o geramos automaticamente, apostamos num editor gráfico específico que, além de permitir a escrita da frase visual, produz uma descrição textual dessa frase. A partir daí, usa-se um processador para linguagens textuais especificado e gerado segundo o método tradicional (acima referido). Este assunto será detalhado na secção 4, onde descreveremos a técnica seguida para a implementação do processador; abordaremos sumariamente, quer a questão do editor, quer a construção do compilador.

Era-nos, porém, fundamental arranjar um caso concreto ao qual pudessemos aplicar as ideias propostas, para validarmos pragmaticamente a sua viabilidade. Essa oportunidade surgiu-nos no âmbito do projecto LEPAForM.

A descrição breve desse projecto, que constituíu o caso de estudo que iremos usando para clarificar as várias ideias que apresentamos, será o assunto da próxima subsecção.

1.1 O caso de estudo: TSCvlc

O projecto LEPAForM —em curso desde 1995 entre o nosso grupo (gEPL) e o grupo de trabalho da Dra Leonor Barroca, da *Open University, UK*— tem por motivação o desenvolvimento de linguagens/ambientes visuais para a aplicação pragmática de

métodos formais. Nesse contexto e no quadro da especificação comportamental de sistemas reactivos de tempo-real, pretendia-se a construção de um tradutor de uma linguagem visual, TSCvl (*Timed StateChart Visual Language*) para axiomas em RTL (*Real Time Logic*).

Para dar uma visão do compilador TSCvlc requerido descreveremos a linguagem fonte e a linguagem objecto, bem como se dirá alguma coisa sobre o método de especificação subjacente.

A linguagem visual TSCvl terá de permitir escrever especificações de sistemas reactivos de tempo-real com base no conceito de máquina de estados temporizada — *Timed StateChart*—, uma extensão introduzida por Armstrong e Barroca [AB96] às *Statecharts* propostas originalmente por Harel [Har87, HPSS87, HLNP90]. As especificações baseadas em *Timed StateCharts* são construídas dividindo o problema em subproblemas e descrevendo cada uma das partes através dum diagrama de transição de estados[1]. Desta forma, uma frase escrita na linguagem TSCvl descreve o conjunto de máquinas de estados que compõem determinado sistema; em relação a cada máquina indica-se os estados que a caracterizam e as transições que podem ocorrer, evidenciando os estímulos que as provocam e os efeitos que são desencadeados.

A linguagem objecto RTL pode ser usada, também, para descrever o funcionamento de cada máquina de estados, porém a sua utilização é muito menos intuitiva, sendo exigida uma sólida formação em lógica para que se consiga modelar o comportamento de cada componente do sistema. Para cada máquina deve ser escrito um conjunto de axiomas em RTL, de tal modo que seja possível recorrer a um *provador automático de teoremas* para verificar a integridade da especificação e raciocionar formalmente sobre as propriedades dinâmicas do sistema em estudo.

Graças a um conjunto de regras genéricas de transformação, propostas em [AB96], é viável obter esses axiomas em RTL sistematicamente a partir dos diagramas de transição de estados em TSC. O compilador TSCvlc deve, precisamente, reconhecer a linguagem TSCvl e mecanizar esse processo de transformação para síntese dos axiomas RTL.

2 Linguagens Visuais de Programação

Uma linguagem visual é uma linguagem que usa uma notação predominantemente gráfica, isto é, um alfabeto formado por símbolos gráficos, ou icónicos, sendo as frases construídas pela combinação espacial (a duas dimensões) desses símbolos (e não, apenas, pela sua concatenação).

Existem muitos argumentos a favor das linguagens visuais. Esses argumentos baseiam-se essencialmente no facto de que os humanos processam mais rapidamente figuras do que texto. O raciocínio humano é bastante orientado à imagem; as pessoas adquirem informação a maior velocidade descobrindo relações gráficas em figuras complexas do que lendo texto.

[1]Neste artigo, o termo TSC designará a representação textual dessas especificações diagramáticas.

Um grande número de linguagens visuais têm sido exploradas, sendo frequentemente baseadas num dos seguintes paradigmas (cf. [Per96]): *controlflow* — usa *flowcharts* para descrever o fluxo de controlo do programa, onde as caixas representam operações simples que são interligadas por linhas; *dataflow* — usa caixas para denotar funções e linhas para conectar o resultado de uma função à entrada de outra; *rewriting* — emprega regras de reescrita para descrever a transformação de cada figura; a solução constroi-se procurando fazer a concordância de uma figura, ou parte dela, com a parte esquerda de uma regra, substituindo-a, então, pela parte direita da mesma regra.

Claro que a recurso a linguagens visuais não traz exclusivamente vantagens e o seu sucesso depende do domínio a que aplica. Além disso, existe um problema de escala latente, dado que é muito difícil analisar um programa visual muito grande. Só parcialmente este problema é resolvido com técnicas expeditas de *browsing*.

Sobre estes paradigmas e sobre as linguagens visuais em geral, recomenda-se a leitura de [Roc95], visto ser uma revisão bastante completa destes assuntos.

2.1 Relacionamento espacial

Apesar da definição de linguagens textuais (em especial linguagens independentes de contexto) estar bem estudada, bem como a respectiva geração de processadores, surgem algumas dificuldades quando se tenta transpor os mesmos raciocínios para as linguagens visuais de programação.

Ausência de uma ordem linear dos símbolos

As frases das linguagens textuais são formadas por sequências de caracteres, pelo que existe uma ordem linear subjacente. Os algoritmos de reconhecimento baseiam-se nesse facto, e as representações que constroem reflectem essa característica.

Por outro lado, um desenho é uma disposição de símbolos num espaço a duas dimensões. Por isso, as suas componentes não se podem ordenar linearmente, já que se perde informação sobre a disposição relativa entre elas.

Múltiplos relacionamentos simultâneos

Numa linguagem textual, os símbolos relacionam-se unicamente com outros que lhes sejam adjacentes (anterior e sucessor), pelo que podemos dizer que só é permitido compor símbolos por concatenação.

Nas linguagens visuais isso não se verifica de modo nenhum. Em primeiro lugar, porque existe um número grande de operadores de composição (à esquerda de, abaixo de, dentro de, etc) e não somente a concatenação, e em segundo, porque cada símbolo pode relacionar-se com vários outros, estabelecendo com cada um relacionamentos diferentes.

A estrutura subjacente é um grafo

As árvores têm sido usadas para representar a estrutura das frases de um programa textual. Essas árvores preservam a ordem dos símbolos, já que são uma estrutura ordenada por natureza.

As árvores, no entanto, não são suficientemente expressivas para preservar todos os relacionamentos entre os objectos de um desenho. Provavelmente, símbolos com mais do que um relacionamento apareceriam mais do que uma vez na árvore, pelo que se pode adiantar que a estrutura subjacente a um desenho é um grafo e não uma árvore.

2.2 Caracterização léxica

A análise léxica nas linguagens textuais consiste em agrupar os caracteres, obtendo as palavras reservadas, identificadores, etc, que constituem o alfabeto da linguagem. Nas linguagens visuais essa análise corresponde a obter os vocábulos, por exemplo, setas e quadrados, a partir de primitivas mais básicas, como pontos ou linhas.

Se, para as linguagens textuais, é difícil encontrar a fronteira entre o que é o nível léxico e o nível sintáctico, para as linguagens visuais também o é.

Pela experiência obtida nas especificações de linguagens visuais, convém considerar que objectos como rectângulos, palavras, setas, são elementos básicos, i.é., fazem parte do léxico de qualquer linguagem. Não convém trabalhar a níveis mais baixos, ao nível do pixel ou do segmento de recta.

Terminais da linguagem TSCvl

Na especificação da linguagem TSCvl, as primitivas gráficas usadas são: caixilhos de janelas, traços duplos, círculos, rectângulos, setas duplas e outras figuras compostas por vários elementos básicos. Estas figuras compostas são considerados terminais da linguagem e, por isso, simplificam a análise léxica (por serem fornecidos como *tokens* pelo editor). Para além destes terminais, existe ainda o texto, que funciona como etiqueta e corresponde a nomes de máquinas, de estados, de eventos, de condições e de acções.

Como veremos, associado a cada um dos terminais existem sempre quatro atributos intrínsecos referentes à sua posição no espaço. O terminal *texto* tem também um atributo intrínseco `conteúdo`, cujo valor é a sequência de caracteres que contém.

Estes terminais têm ainda outros atributos, como a existência de setas ou não nas linhas, a espessura do traço, se o traço é a cheio ou a tracejado, cor, etc. O conjunto de terminais e respectivos atributos, que foi considerado na construção de um editor específico para linguagens visuais de programação é apresentado em [Per96].

2.3 Caracterização Sintática

Uma frase é composta por componentes que se formam à custa de outras componentes mais simples.

Como as linguagens visuais são a duas (ou mais) dimensões e por isso as componentes relacionam-se com outras componentes que lhes são adjacentes no espaço, as relações passam a ser feitas em função de atributos dos componentes, geralmente em função daqueles que se referem à posição ocupada no espaço.

Esta característica indroduz a obrigatoriadade de atributos para captar a estrutura sintática, enquanto que nas linguagens textuais basta apenas a ordem dos símbolos para criar a representação da sua estrutura. A necessidade dos atributos deixa de ser só para modelar a semântica, mas também para determinar a estrutura sintática.

As linguagens visuais que estamos interessados em especificar relacionam as suas componentes em função das suas posições. Outras linguagens visuais poderão-se definir em função de outros atributos, como o tamanho ou a forma. Ou ainda, usando atributos como a cor, podem ser estabelecidos outros tipos de composição que não a espacial.

3 Especificação de Linguagens Visuais

3.1 Motivação

Ao começar a nossa investigação na área das linguagens visuais, reparámos que existiam efectivamente muitas linguagens já definidas, e até suportadas por editores mais ou menos sofisticados, mas sem uma definição formal da sua sintaxe. Pela nossa experiência, a existência de uma definição formal da sintaxe trás um conjunto de vantagens, que vão desde funcionar como um elemento de apoio à aprendizagem da própria linguagem por novos utilizadores, até constituir um suporte necessário à posterior definição da semântica. De entre essas vantagens, destacamos as duas que nos interessam mais, no contexto deste artigo:

- Serve como especificação rigorosa do que pode fazer um editor dirigido pela sintaxe da linguagem,
- Serve como fonte de um gerador automático de reconhecedores de linguagens visuais.

Esta razão motivou-nos a desenvolver um formalismo de especificação sintática de linguagens visuais. As características que se exigiam eram uma forte expressividade, rigor, facilidade de leitura e de desenvolvimento. Além disso, inspirado no trabalho desenvolvido por Pedro Henriques, em [Hen92], acrescentamos aos requisitos a modularidade, e o mecanismo de parametrização, fundamental para suportar a modularidade.

Pensamos que muito beneficiaram as linguagens textuais pelo estabelecimento de um formalismo de aceitação comum para a sintaxe das linguagens, o BNF. Da mesma forma, seria interessante dispôr de algo semelhante aplicado às linguagens visuais. Para já, parece-nos difícil haver uma aceitação generalizada de um determinado formalismo. Além desta nossa proposta do formalismo MASOViLa, existem outras,

das quais destacamos *Visual Grammars* [Lak86], de F. Lakin, *Picture layout Grammars* [Gol90], de E. Golin, *Gramáticas de Adjacência* [JP95], de J. Jorge, *Graph Grammars* [Rek94], de J. Rekers e *Constraint Multiset Grammars* [HM91], de K. Marriot.

3.2 Características do MASOViLa

Como se referiu, uma das particularidades das linguagens visuais, é a presença de atributos (espaciais ou outros) para determinar a sintaxe da linguagem. Esta exigência, juntamente com prévias experiências com gramáticas de atributos, levou-nos a adoptar uma abordagem orientada à semântica.

Com esta abordagem resolvemos o problema da definição sintática e em simultâneo o da definição do significado. Além disso, como se verá, é possível especificar a reacção ao reconhecimento, que permite (juntamente com os valores de atributos) incluir qual a acção de interpretação (no caso de um interpretador) ou qual o código a gerar (no caso de um compilador).

Dado que um dos requisitos era a modularidade, atendeu-se ao facto de serem conhecidos melhores resultados em aproximações orientadas à semântica, conseguindo-se separar em diferentes módulos cada uma das tarefas, sendo uma das razões que leva esta abordagem a ser preferida na criação de processadores mais complexos. Esta aproximação está subjacente em ferramentas como o Synthesizer Generator [RT88] ou o Eli [W+92].

A nossa abordagem, contudo, ao nível da modularidade pretende ser mais ambiciosa, optando-se (como se faz em [Hen92]) por orientá-la ao símbolo. Concretamente, o símbolo é a unidade de especificação, o alvo de toda a atenção de quem desenvolve a especificação. Concentrado na especificação de um símbolo de cada vez, o utilizador define todos os seus aspectos: a sintaxe, a semântica (pelo cálculo do significado) e a reacção ao processamento.

Posto isto, como pretendíamos disponibilizar um mecanismo real de reaproveitar, no desenvolvimento de novos processadores, componentes (símbolos) de outros processadores foi necessário incluir dois mecanismos de suporte à especificação modular:

Composição A construção de uma especificação é feita por composição de outras mais pequenas.

Parametrização Este mecanismo é que permite que haja verdadeiramente reutilização, já que a versão parametrizável de uma especificação corresponde a uma classe, que depois pode ser instanciada, assumindo as características concretas necessárias no contexto em que será usada.

No desenho do MASOViLa houve a preocupação de permitir que os diversos elementos intervenientes na definição de um módulo (símbolo) fossem parametrizáveis.

3.3 Definição dos símbolos

Apresentamos o MASOViLa começando pela especificação de um símbolo.

```
Nonterminal:  JANELA {

     Uses:

          Ts:

               CAIXILHO(↑LLx, LLy, URx, URy:  INT)
               NOME_PROB(↑LLx, LLy, URx, URy:  INT ↑NomeP: string)

     Structure:

          Form1:  CAIXILHO NOME_PROB

               where labels(CAIXILHO, NOME_PROB)

     Semantics:

          Attributes:

               Synthesized LLx, LLy, URx, URy:  INT, NomeP: string

          Rules:

               Form1:
                    SYNTHESIZE JANELA FROM CAIXILHO
                    SYNTHESIZE JANELA.NomeP FROM NOME_PROB.NomeP

     Translation:

}
```

O primeiro aspecto que é facilmente identificável na especificação de um símbolo é sua divisão em quatro secções distintas:

- *USES* - declaração de todos os símbolos não-terminais, ou terminais de classe, usados na estrutura do símbolo em causa.
- *STRUCTURE* - descrição das diferentes formas sintáticas do símbolo (produções).
- *SEMANTICS* - esta parte está dividida em duas. A primeira parte constitui a declaração dos atributos herdados e sintetizados do símbolo. A segunda parte é usada para associar as regras semânticas e as condições de contexto a cada produção.
- *TRANSLATION* - esta parte opcional, é usada para associar a cada forma sintática as acções necessárias para processar o símbolo definido.

3.3.1 Operadores pré-definidos

Na definição da estrutura sintática de JANELA, obriga-se a que exista uma determinada relação espacial entre os símbolos NOME_PROB e CAIXILHO.

Essa relação poderia ser expressa somente em função dos atributos espaciais LLx, LLy, URx e URy. Mas para facilitar a legibilidade do formalismo, existiu um trabalho prévio de identificação das relações espaciais mais típicas das linguagens visuais (trabalho inspirado em [Gol90]).

Associado a cada uma das relações identificadas, definiu-se o seguinte conjunto de operadores, divididos em três tipos:

- Operadores que combinam formas geométricas
  - *over* : relaciona verticalmente dois objectos impondo que o primeiro esteja acima do segundo.
  - *left_to* : é semelhante ao anterior, só que rodado 90^o no sentido contrário ao dos ponteiros do relógio.
  - *contains* : compõe dois símbolos, em que o segundo está dentro do primeiro.
  - *adjacent_to* : refere uma composição espacial muito pouco específica: o primeiro símbolo está algures à volta do outro.
- Operadores que combinam segmentos de recta
  - *follows* : combina duas linhas em que a primeira começa onde a segunda acaba.
  - *join* : combina duas linhas em que o ponto final das duas coincide.
  - *fork* : combina duas linhas e obriga a que os pontos iniciais sejam coincidentes.
  - *alternate* : combina duas linhas em que os pontos iniciais e finais coincidem.
  - *reverse* : operador unário que reconhece um segmento e devolve um novo segmento cujo ponto inicial coincide com o ponto final do primeiro e cujo ponto final coincide com o inicial do mesmo.
- Operadores que combinam segmentos com formas geométricas
  - *leaves* ou *points_from* : força que o ponto inicial do segmento coincida com a fronteira da forma geométrica.
  - *arrives* ou *points_to* : força que o ponto final do segmento coincida com a fronteira da forma geométrica.
  - *labels* : serve para reconhecer uma forma geométrica (geralmente um terminal TEXT que serve de etiqueta um objecto). Não é obrigatório que haja coincidência de pontos.
  - *tiling* : é um operador sem restrições e sem regras semânticas associadas. Aceita um ou mais argumentos, sendo estes formas geométricas, ou segmentos de recta. É usado quando não existe nenhuma relação específica entre os símbolos.

3.3.2 O uso de sub-linguagens visuais

Para simplificar a especificação visual de um problema, será proveitoso usar uma noção de sub-gramática (e sub-linguagem) para descrever partes da frase visual que devem ser ocultadas numa fase inicial da especificação, ou seja, permitir que num dado momento se esteja a produzir uma frase (um esquema visual) usando símbolos terminais que representam conceitos complexos e que podem, posteriormente, ser expandidos para sub-frases elaboradas. A gramática TSCvl pode ser considerada como tendo duas sub-gramáticas associadas. Alguns símbolos da gramática inicial serão vistos como terminais (representação visual no diagrama principal) e simultaneamente como não-terminais (raízes das sub-gramáticas). É o caso de:

```
SETA_I (terminal)  e  CLICK_SETA_I (raiz de uma sub-gramatica)
SETA_D (terminal)  e  CLICK_SETA_D (raiz da outra sub-gramatica)
```

Ao terminal do lado esquerdo deve corresponder o não-terminal do lado direito, pretendendo-se que essa associação (expansão) seja realizada através de um *click de rato*. Cada uma das sub-gramáticas está relacionada com um tipo de transição em TSC (imediata ou retardada). Assim, a correspondente sub-linguagem permite representar (visualmente) informação pormenorizada acerca da transição (*Trigger* e *Effect*) em causa.

4 Implementação de Processadores de Linguagens Visuais

Dispondo já de um formalismo para a definição de processadores de linguagens visuais, o óptimo seria ter um gerador automático de processadores para as linguagens especificadas.

Tal ainda não foi possível desenvolver, devido a dimensão e complexidade do problema. Mas precisávamos de disponibilizar um "processador"de TSCvl no âmbito do projecto LEPAForM, como nos tínhamos comprometido.

A solução foi desenvolver um editor gráfico, com a definição sintática da linguagem TSCvl incorporada, editor esse que produz uma representação textual da frase visual para servir de entrada a um compilador tradicional que irá gerar os axiomas RTL.

O editor gráfico foi desenvolvido em Delphi, segundo o paradigma da programação orientada aos objectos combinado com a filosofia de interacção MDI —*Multiple Document Interface*.

Para cada símbolo terminal foi criada uma classe onde se define: a respectiva representação visual (apresentação no écran); um predicado que indica se o símbolo está ou não disponível (para ser inserido numa frase, conforme o contexto sintático (estado da edição); a respectiva representação textual (o que irá servir para obter o texto final). O esquema de tradução não está definido de forma totalmente estática, na medida em que a obtenção do texto correspondente a um símbolo pode requerer a troca de mensagens com outros objectos. Os detalhes da implementação do editor podem ser vistos em [BS97].

Devido ao uso da abordagem MDI, o editor fornece um *form*, aberta logo no início, para construção da frase visual principal, sendo abertos outros *forms* consoante a necessidade de expandir certos símbolos da frase (estados duma máquina nas submáquinas respectivas, etc).

Quanto ao compilador TSC foi gerado recorrendo à ferramenta Eli, a qual produz um *tradutor orientado pela semântica* a partir da gramática de atributos da linguagem fonte e das regras atributivas para geração da saída.

A especificação do compilador, usando esta ferramenta, está dividida em vários ficheiros relativos à descrição de cada componente do processador:

- um contém a descrição das sequências de caracteres que correspondem a cada classe terminal da linguagem;
- outro contém a gramática concreta, independente de contexto, da linguagem TSC;
- um terceiro encerra a gramática de atributos abstracta, definindo os atributos associados a todos os símbolos, descrevendo as regras para o seu cálculo e as condições de contexto a serem validadas na aplicação de cada produção (na realidade, esta componente pode ser dividida em vários módulos, o que permite aumentar visivelmente a legibilidade e facilidade de gestão da especificação);
- um quarto inclui as regras que associam a cada produção, e em função dos atributos que nela ocorrem, o esquema de tradução a usar na produção do texto final.

O Eli[W+92] é baseado num sistema pericial que analisa a descrição dos requisitos do utilizador e inter-relaciona as diversas especificações de modo a seleccionar de entre as suas ferramentas as necessárias para processar cada componente, de forma a alcançar a solução global pretendida.

A tese [Per96] discute as alternativas que podiam ter sido seguidas para o desenvolvimento automático do processador e fornece todos os pormenores da implementação, referindo a exploração que foi levada a cabo no sentido de tirar o máximo partido do gerador escolhido.

5 Conclusões e Trabalho Futuro

Uma linguagem visual é uma linguagem que usa uma notação predominantemente gráfica, isto é, um alfabeto formado por símbolos gráficos, ou icónicos, sendo as frases construídas pela combinação espacial a duas dimensões desses símbolos (e não, apenas, pela sua concatenação sequencial).

O recurso a uma linguagem visual que representa máquinas de transição de estados, TSCvl, para especificar rigorosamente sistemas reactivos de tempo-real, constitui um bom exemplo da importância dessas linguagens.

Nesse quadro, o problema consiste realmente em estabelecer um método para o desenho dessas linguagens, para a sua especificação e para a forma de implementar

processadores que reconheçam e traduzam as ditas linguagens visuais. A caracterização dessas facetas e a proposta de uma abordagem, foi o tema discutido ao longo do presente artigo.

O desenvolvimento de um compilador que produz axiomas RTL[2] a partir de *Timed StateCharts*, foi o caso prático a que aplicamos as ideias aqui expostas. Por isso usámo-lo ao longo desta comunicação para exemplificar os vários tópicos que abordámos. No projecto que realizamos não se discutiu a notação visual concreta que se apresentou pois, claramente, os objectivos do estudo consistiam na passagem sistemática e segura de uma linguagem textual para visual e na aplicação de determinados formalismos de descrição de linguagens visuais.

Uma vez validada pragmaticamente a possibilidade de construir o processador acrescentando um editor gráfico a um compilador tradicional —o editor, desenvolvido em Delphi, produz uma representação textual dos diagramas, a qual serve, então, de entrada para o tradutor tradicional, construído com o gerador de compiladores Eli a partir duma gramática de atributos transformacional. Interessa-nos, no futuro próximo:

- reforçar essa validação, resolvendo outros casos concretos diferentes;
- explorar outras tecnologias para implementação do editor da linguagem visual;
- averiguar da possibilidade de gerar automaticamente o referido editor, a partir da especificação formal da linguagem em MASOViLa;
- gerar processadores cuja linguagem fonte seja o próprio desenho.

Embora não tenha sido, ainda, aplicada no trabalho descrito, já definimos [Dia96] uma versão visual do formalismo MASOViLa. De momento estamos, precisamente, a investigar o desenvolvimento dum processador para visual-MASOViLa.

Agradecimentos

Os autores gostariam de agradecer em primeiro lugar aos revisores do 2º Simpósio Brasileiro de Linguagens de Programação.

O projecto SARA decorre no âmbito do contrato JNICT PBIC/C/TIT/2481/95. O projecto LEPAForM foi possível com o apoio conjunto da JNICT e do British Council.

O trabalho da Maria João Varanda foi em parte suportado por uma bolsa de mestrado no âmbito do programa PRAXIS XXI.

Referências

[AB96] J. Armstrong and L. Barroca. Specification and verification of reactive system behaviour: The railroad crossing example. *Real-Time Systems*, March 1996.

[2] Axiomas esses que vão permitir raciocinar rigorosamente sobre as propriedades comportamentais dos sistemas modelados.

[BS97] Carlos Barbosa and Rui Santos. Tsc graphical editor. Technical report, Dep. de Informática, Universidade do Minho, 1997.

[Dia96] Luis Miguel Silva Dias. Linguagens Visuais de Programação: paradigmas e aplicações, December 1996.

[Gol90] E. J. Golin. *A method for the specification and parsing of visual languages.* PhD thesis, Brown University, 1990.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen92] Pedro R. Henriques. *Atributos e Modularidade na Especificação de Linguagens Formais.* PhD thesis, Universidade do Minho, December 1992.

[HLNP90] D. Harel, H. Lachover, A. Naamad, and A. Pnueli. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[HM91] R. Helm and K. Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2(4):311–332, December 1991.

[HPSS87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, New York, 1987. IEEE.

[JP95] Joaquim Jorge and Ephraim P.Glinert. Online parsing of visual languages using adjacency grammars. Technical report, 1995.

[Lak86] F. Lakin. Spatial parsing for visual languages. In S.-K. Chang, T. Ichikawa, and P. Ligomenides, editors, *Visual Languages*, pages 35–85. Plenum Press, New York, 1986.

[Per96] Maria João Varanda Pereira. Concepção e Especificação de uma Linguagem Visual, September 1996.

[Rek94] J. Rekers. On the use of graph grammars for defining the syntax of graphical languages. Technical Report 94-11, Leiden University, 1994.

[Roc95] Jorge Gustavo Rocha. Especificação de linguagens visuais de programação. Master's thesis, Universidade do Minho, Departamento de Informática, Julho 1995.

[RT88] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator.* Springer-Verlag, 1988.

[W+92] William M. Waite et al. ELI: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.