L.E.I.C.
Relatório Final do
Trabalho Final de Curso
no Regime Integrado

Representação Categorial de Especificações de Espécie Única

Orientador: Prof. Rui Gustavo Crespo

Francisco José Moreira Couto

Número: 42675

19 de Julho de 2000

Agradecimentos

O trabalho que aqui apresento é fruto de um longo processo onde os contributos de muitas pessoas foram fundamentais.

A exigência, encorajamento, entusiasmo e permanente disponibilidade do Prof. Rui Gustavo Crespo, orientador deste trabalho, seu crítico e inspirador, foram factores decisivos para o desenvolvimento deste trabalho

Quero agradecer também à Prof. Paula Gouveia pelas suas valiosas sugestões sobre a apresentação deste relatório.

Saliento o excelente ambiente de trabalho que pude usufruir na Secção de Análise Numérica e Matemática Aplicada, onde me foi disponibilizado meios informáticos importantes para o desenvolvimento deste trabalho.

Fica também um especial agradecimento para os meus pais, pelo seu apoio constante.

Conteúdo

Lista de Figuras

Sumário

O principal objectivo deste trabalho consiste no estudo de uma solução para o problema da *reutilização* de *Software* através de métodos formais. A solução irá permitir a diminuição do esforço despendido na fase de codificação de um sistema ao mínimo possível, isto às custas de se especificar formalmente esse mesmo sistema.

O objectivo a alcançar no trabalho final de curso consiste em identificar uma metodologia para traduzir uma especificação de espécie única, de um determinado componente de um sistema, numa representação categorial. De seguida será desenvolvida uma aplicação informática que implemente essa metodologia. Em simultâneo é também criado um repositório dos componentes. Por cada componente residente no repositório, guarda-se a seguinte informação:

- A sua especificação de espécie única.
- A sua representação categorial.
- Um identificador que indica os algoritmos que o implementam.

Este trabalho final de curso tem uma evolução no âmbito da dissertação de mestrado, com o objectivo de produzir uma outra aplicação informática[1]. O objectivo desta aplicação informática é identificar automaticamente os componentes do repositório com a mesma funcionalidade de um componente dado pelo utilizador. Para alcançar este objectivo a aplicação pega nas representações categoriais dos componentes do repositório e identifica emparelhamentos com a representação categorial do componente dado pelo utilizador. Caso

[1] O resultado final não será só o código correspondente à aplicação, mas também os resultados teóricos que serão necessários obter para a implementação da aplicação.

sejam identificados emparelhamentos, o utilizador poderá na implementação do componente *reutilizar* alguns dos algoritmos que foram criados para os componentes identificados.

A contribuição deste trabalho consiste na revisão e no estudo da viabilidade dos objectivos descritos no artigo [Cre98a], tendo-se como meta final o desenvolvimento de uma aplicação informática que cumpra esses mesmos objectivos.

Capítulo 1

Introdução

Este documento[1] refere-se ao relatório sobre o que foi desenvolvido no âmbito deste trabalho final de curso de Engª Informática e Computadores, do Instituto Superior Técnico, da Universidade Técnica de Lisboa, Portugal.

Este projecto insere-se também na primeira fase de uma dissertação de mestrado, que tem como objectivos o prolongamento do trabalho efectuado até aqui, e que será composta por mais uma fase a concluir em 2001.

Neste projecto deu-se uma maior importância à investigação do problema do que ao facto de se construir um produto que pudesse ser comercializado. Desta forma, tenta-se aqui atacar o problema através de uma solução aplicável a longo prazo, isto devido à grande dificuldade da comunidade científica em encontrar uma resolução para o problema.

Neste trabalho foi desenvolvida uma aplicação informática que implementa na prática as ideias obtidas para a resolução do problema. A aplicação informática desenvolvida, bem como toda a sua documentação, estão disponíveis na *internet* no seguinte endereço:
http://www.math.ist.utl.pt/ fcouto/tfc/index.html

[1] Este documento foi desenvolvido em LaTeX.

1.1 O Problema

Hoje em dia, qualquer entidade de produção de *software* tem o problema de saber como e quando, num novo projecto, se pode reutilizar *software* já produzido para projectos anteriores.
O problema de *Reutilização de Software* consiste em encontrar componentes de um determinado projecto de *software* que tenham o mesmo comportamento funcional de outros já implementados. Assim, esses componentes não necessitam de ser implementados, pois basta *reutilizar* o código dos seus semelhantes.

A situação ideal era conseguir que a criação de um novo sistema fosse apenas desenvolver a sua especificação e, consequentemente, *reutilizar* os componentes, que dadas as suas características funcionais são necessários para a implementação desse sistema. Por outras palavras, o objectivo seria permitir que o *software* pudesse ser utilizado da mesma forma que o hardware. No que diz respeito, por exemplo, aos componentes de um PC, é possível ir ir comprar um novo componente para este a uma loja, sabendo apenas as suas características funcionais externas. Mas, hoje em dia, este objectivo ainda está muito longe de ser alcançado em engenharia de *software*.
Neste projecto tenta-se criar uma aplicação que embora com algumas limitações permita introduzir este tipo de situação na produção de *software*.

É de realçar que o problema da *reutilização* é muito mais vasto do que o considerado neste trabalho, pois a *reutilização* não se limita apenas ao *software*, mas também a outras áreas. Como exemplo, temos o caso da engª civil, onde muitas vezes se pode *reutilizar* partes de projectos anteriores, com o fim de criar novos projectos.

1.2 Importância do Problema

Numa entidade de produção de *software* é muito frequente produzir projectos que tenham elementos comuns com projectos anteriormente produzidos. Melhor dizendo, é mesmo muito raro uma empresa de média dimensão criar um projecto que seja totalmente inovador, no sentido em que todo o código que é necessário implementar seja completamente disjunto do código produzido pela empresa até ao momento.
Apesar desse facto, na maior parte dos casos esse código não é reutilizado. Uma das causas deve-se à empresa não possuir meios para detectar automaticamente se um componente pode ser reutilizado. A razão para isto consiste normalmente no facto da empresa ter um mau processo de desenvolvimento de projectos e também a uma falta de produtos informáticos que permitam o automatismo deste processo. É de salientar também que por vezes os produtos informáticos disponíveis não são suficientemente completos, por forma tornar o processo viável. Este é exactamente o objectivo deste projecto, ou seja, pretende-se obter uma aplicação informática que torne o processo de reutilização automático.

De seguida, apresentam-se algumas das vantagens de se utilizar um método de desenvolvimento, de produtos informáticos, que utilize o processo de reutilização:

- **Aumento da produtividade de *software*.**
  No caso de existirem componentes que podem ser reutilizados, estes não precisam de ser implementados. Isto traduz-se na diminuição do tempo de desenvolvimento do sistema, e no facto de ser necessário menos recursos para o desenvolvimento do mesmo.

- **Probabilidade de erros nos sistemas é menor.**
  Como na implementação do sistema é criado menos código, a probabilidade de existirem erros no sistema também é menor. Este facto

surge devido a que os componentes reutilizados já foram previamente testados e corrigidos.

1.3 Aplicações

Um exemplo típico de aplicação da *Reutilização de Software*, é o caso da implementação das listas. Isto porque como muitas das aplicações informáticas têm a necessidade de trabalhar com listas, pode-se *reutilizar* as funções que as manipulam já implementadas para sistemas anteriores. Isto acontece sempre que se quer produzir uma nova aplicação que necessita de usar listas.

O problema aqui tratado surge nos mais diversos ramos da produção de *software*. E todas estas entidades que produzem *software* estão interessadas em resolver este problema, visto as vantagens obtidas serem muito importantes.

A grande dificuldade da resolução deste problema reside na forma como são especificados os vários componentes de um sistema. Neste momento uma grande parte da produção de *software* nem sequer é especificada e quando é especificada, normalmente são utilizadas especificações informais, deixando um nível de subjectividade muito elevado, tornando impossível tratar essa informação automaticamente. A justificação para esta situação deve-se ao facto de a utilização de especificações formais ter custos muito elevados, e na maior parte dos casos as vantagens obtidas não cobrem os custos. Outro aspecto relevante reside no facto de os custos terem de ser despendidos no início do projecto e as vantagens só serem alcançadas na parte final do projecto, o que para a maior parte das empresas é um factor limitador visto não terem capacidade de investir a longo prazo.
Desta forma, tem-se aqui um problema de engenharia, ou seja, para tornar a utilização de especificações formais mais atractiva temos duas hipóteses:

- Diminuir os custos.

- Aumentar as vantagens.

Os custos elevados da utilização de especificações formais surgem do facto de serem necessários recursos humanos especializados para implementar este tipo de metodologia. Desta forma, uma solução para diminuir os custos é investir na formação de pessoal especializado neste tipo de metodologia.

Aumentar as vantagens da utilização de especificações formais é o principal objectivo deste trabalho, pois se for possível encontrar uma aplicação que resolva o problema da reutilização de *software* automaticamente a partir de especificações formais, será muito difícil a uma entidade de produção de *software* não compreender que o balanço entre os custos e vantagens da utilização de especificações formais se revela favorável às vantagens.

Pelas razões descritas anteriormente, compreende-se facilmente que apesar de hoje só se poder aplicar a solução seguida neste projecto a um número muito limitado de casos, espera-se que no futuro esse número aumente conforme se vai alcançado resultados relevantes neste domínio.

Em relação à aplicação informática concluída nesta primeira fase, esta pode ser utilizada por um outro projecto que embora com objectivos diferentes necessite de utilizar as funcionalidades desta aplicação informática. Isto porque esta aplicação é completamente independente da aplicação informática a concluir na segunda fase.

1.4 Formas de Resolução

Neste momento estão em estudo vários mecanismos de selecção de componentes que podem ser reutilizados, cada um com as suas vantagens e desvantagens. Isto deve-se à grande dificuldade encontrada pela comunidade científica em desenvolver uma solução para o problema da reutilização de

software que seja o suficientemente abrangente, por forma a ser viável a sua utilização por todas as entidades de produção de *software*.

Um desses mecanismos consiste em encontrar componentes de *software* por analogia [SC94, MS92], explorando a semelhança entre os componentes. Um outro mecanismo é o da selecção de tabelas de atributos [PDF93] e métricas [CB91], que é vantajoso em termos do custo, escalabilidade e portabilidade.

A solução seguida neste trabalho baseia-se no método de emparelhamento de especificações formais, as quais se centram no comportamento do componente em vez de na sua informação descritiva. Desta forma, duas especificações podem ser diferentes sintacticamente mas terem o mesmo comportamento, e portanto poderem ser reutilizadas. O presente projecto enquadra-se numa colecção de trabalhos de investigação [CHJ93, JC95, KRT87, MMM94, PP93, ZW95] que têm como propósito a utilização de métodos formais, para a facilitar a produção de *software* a nível industrial.
Neste projecto considera-se a especificação como o documento de trabalho fundamental no desenvolvimento e reutilização do *software*. Por este motivo, quando se inicia o desenvolvimento de um novo produto de *software* dever-se-á dividir esse produto nos seus vários componentes e criar as respectivas especificações formais que descrevem o funcionamento desses componentes.

1.5 Solução Seguida neste Trabalho

Nesta primeira fase tem-se como meta a tradução de uma especificação de espécie única de um componente dado pelo utilizador para a sua respectiva representação categorial.
A primeira etapa, desta primeira fase, consiste na revisão e no estudo da viabilidade dos objectivos descritos no artigo [Cre98a], nos aspectos que seguidamente se referem:

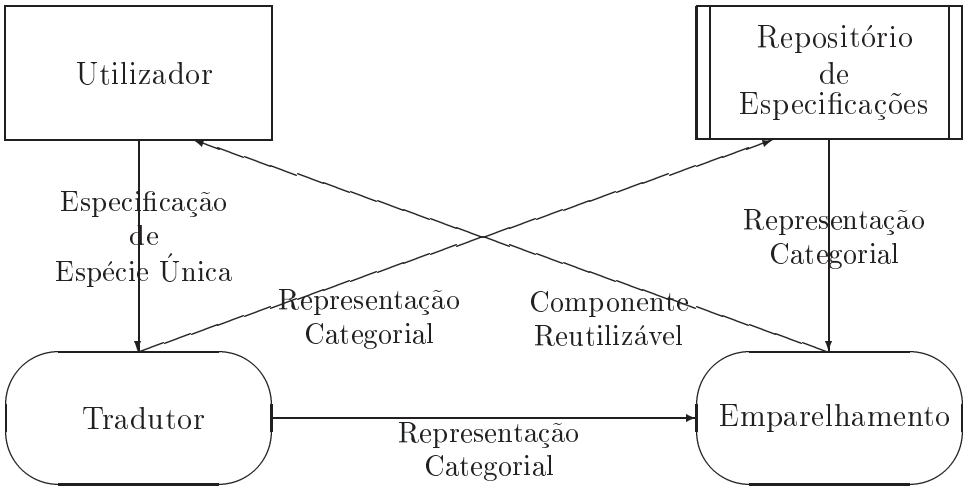

Figura 1.1: DFD da aplicação informática do projecto

- Aprofundar alguns aspectos deixados em aberto, como por exemplo, o significado especial do predicado igualdade.
- Estender a linguagem das fórmulas que podem aparecer na especificação, como por exemplo, ser possível representar implicitamente fórmulas do tipo: $\forall x\,[P_1(x) \Rightarrow P_2(x)]$
- Alterar e criar novos algoritmos de tradução, como por exemplo, trabalhar com fórmulas em forma normal.

A segunda etapa consiste em desenvolver uma aplicação informática que irá implementar a solução obtida na primeira etapa. O DFD[2] genérico desta aplicação pode ser observado na figura 1.1. Esta primeira fase corresponde ao processo Tradutor da figura 1.1.

O objectivo é representar categorialmente a funcionalidade desse componente automaticamente a partir da sua especificação de espécie única. Assim, no final ficamos com duas especificações de um componente. Ambas as especificações representem a funcionalidade desse componente através de duas

[2]DFD é um diagrama de fluxo de dados que modela o funcionamento de uma aplicação [BCN92, capítulo 8].

linguagens diferentes.
A representação categorial foi escolhida porque esta é de todas as linguagens de especificação a que mais se adequa à identificação de emparelhamentos entre especificações, visto esta ser uma linguagem sintacticamente fraca. Assim, nesta linguagem, duas especificações que contenham a mesma informação semântica são necessariamente muito semelhantes sintacticamente.

A segunda fase deste projecto, que constitui o trabalho de investigação conducente a uma tese de mestrado, consiste em utilizar as representações categoriais dos vários componentes para identificar emparelhamentos entre estes. O objectivo é verificar quando uma parte do componente dado pelo utilizador tem uma funcionalidade semelhante a uma parte de um componente já implementado, podendo-se assim reutilizar o código correspondente.

Todos os algoritmos desenvolvidos para este trabalho, foram codificados numa linguagem de programação funcional, mais precisamente *Standard ML*. Estes algoritmos podem ser divididos em três classes:

- Entrada.
- Tradução.
- Saída.

Os algoritmos de Entrada correspondem à análise léxica e sintáctica feita à especificação de entrada. Aqui foram usadas as ferramentas ML-Lex e ML-Yacc, o primeiro é um analisador léxico e o segundo um analisador sintáctico. Estes algoritmos estão descritos na secção 4.2.

Os algoritmos de Tradução correspondem à tradução das especificações, isto é, estes têm como objectivo transformar uma especificação de espécie única na sua respectiva representação categorial.
Estes algoritmos estão descritos na secção 4.4.

Os algoritmos de Saída correspondem à saída da representação categorial, ou seja, estes têm como tarefa pegar na representação categorial interna e produzir a respectiva representação categorial externa.
Estes algoritmos estão descritos na secção 4.5.

Capítulo 2

Conceitos Básicos

Neste capítulo é apresentada uma pequena abordagem aos temas que serviram de suporte ao trabalho desenvolvido.

2.1 Especificações

As especificações indicam o comportamento dos componentes que constituem um sistema, não sendo necessário conhecer o código que os implementa. Logo, a partir das especificações dos vários componentes, pode-se conhecer o funcionamento global do sistema.

Uma descrição mais detalhada sobre as especificações pode ser encontrada em [WT87].

2.1.1 Vantagens

Todos os modelos de processos determinam que se devem criar as especificações antes do respectivo código. Este facto não é nenhuma desvantagem mas sim uma vantagem porque desta forma o código resultante é de melhor qualidade e o tempo despendido na produção das especificações é depois

recuperado na fase de codificação. Por outro lado, a identificação da especificação antes do código gera, normalmente, menos erros na implementação do sistema.

Como no presente caso vão ser utilizadas especificações formais, então é possível construir a prova da correcção do sistema, que pode ser útil em sistemas críticos, isto é, sistemas que não podem falhar. Para além disso, também é possível, em alguns casos, implementar animações (protótipos) através das especificações, permitindo a sua validação.

2.1.2 Linguagens de Especificação

Para produzir uma especificação de um componente é necessário escolher uma linguagem de especificação.

Existem três classes de especificações:

- Informal, orientada ao utilizador e consiste na linguagem natural.
- Semi-Formal, baseada na visualização gráfica do sistema, por exemplo a análise estruturada.
- Formal, baseada na matemática, e orientada ao sistema possuindo a semântica mais rica de entre os três esquemas de representação.

O tipo de especificações utilizadas neste projecto é formal. Uma especificação formal não é mais que uma teoria, formada a partir de um conjunto de fórmulas de um sistema dedutivo e da respectiva semântica. Posteriormente neste capítulo definir-se-á formalmente o que é uma teoria.

Um exemplo de uma linguagem de especificação com estas características é a linguagem Z [Spi88], linguagem baseada em modelos que utiliza a teoria de conjuntos para modelar dados e a lógica de predicados de 1ª ordem para

descrever operações sobre os dados. Esta é uma linguagem tipificada, isto é, requer que a cada variável seja associado um tipo.

Outros exemplos de linguagens de especificação são OBJ [GM92] e CSP [Hoa85]. A primeira trata-se de uma linguagem algébrica e a última tem como objectivo modelar o comportamento dos processos de um sistema.

Neste projecto são utilizadas dois tipos de especificações :

- Especificações de Espécie Única.
- Representação Categorial

As especificações de espécie única podem ser consideradas como uma simplificação das especificações em linguagem Z. Estas foram escolhidas por razões de facilidade de implementação num sistema computacional.
Por uma questão de legibilidade, algumas vezes neste relatório abrevia-se *especificações de espécie única* por apenas *especificação*.

2.1.3 Lógica de Predicados de Primeira Ordem

Definição 2.1 *A linguagem da lógica de predicados de 1ª ordem usada neste projecto, tem um alfabeto constituído por:*

- *Variáveis individuais* $(v_1, v_2 \ldots)$
- *Constantes* $(c_1, c_2 \ldots)$
- *Conectivos de negação, disjunção, conjunção e implicação* $(\neg, \vee, \wedge, \Rightarrow)$
- *Quantificadores universal e existencial* $(\forall, \exists)$
- *Predicados* $(P_1, P_2 \ldots)$.

Definição 2.2 *Um* termo *pode ser uma variável ou uma constante.*

Definição 2.3 *Uma* proposição *é um tuplo de termos aplicados como argumentos de um predicado. O número de elementos do tuplo depende da aridade do predicado.*

Definição 2.4 *De seguida apresentam-se as regras para construção de uma* fórmula *bem definida.*

1. *Uma proposição é uma fórmula.*
2. *Se ϕ e φ são fórmulas, então também $\neg\phi$, $\phi \vee \varphi$, $\phi \wedge \varphi$, $\phi \Rightarrow \varphi$ são fórmulas.*
3. *Se ϕ é uma fórmula e v_i é uma variável individual, então $\forall v_i \phi$ e $\exists v_i \phi$ também são fórmulas.*

O alfabeto não inclui funções, contudo não é uma restrição muito grave, pois as funções podem ser representadas como predicados. Tal como demonstra o Exemplo 2.1, uma função de aridade n pode ser substituída por um predicado com aridade n+1, em que o último argumento representa o resultado da função.

Exemplo 2.1 *A fórmula $>(5, +(1, 2))$ inclui a função + que pode ser substituído pelo predicado plus, obtendo-se a seguinte fórmula equivalente à primeira: $plus(1, 2, resultado) \wedge >(5, resultado)$*

Definição 2.5 *Dado um conjunto de fórmulas Δ, ao conjunto de todas as fórmulas deriváveis de Δ, chama-se* conjunto de teoremas *de Δ, ou* teoria gerada *por Δ, e escreve-se $Th(\Delta)$ [Mar98]. Formalmente:*

$$Th(\Delta) = \{\alpha : \Delta \vdash \alpha\}$$

Dado um conjunto de fórmulas a sua representação categorial será sempre a representação derivada da teoria gerada por esse conjunto.

2.1.4 Estrutura das Especificações Utilizadas

As especificações de um sistema de *software*, são divididas pelos vários componentes que o constituem. A especificação de um componente obedece à seguinte estrutura:

Componente: identificador

Espécie: membros_do_conjunto

Variáveis: (nome_da_variável)+

Invariante: condição

(**Método**: ...)*

O tipo das variáveis pode ser enumerado e definidos pelo utilizador no campo *Espécie*.

A condição do *Invariante* é uma fórmula da lógica de 1ª ordem, que indica o que o sistema tem que verificar em qualquer momento da sua execução.

A parte do Método obedece à seguinte estrutura:

Método: identificador

Interface: (nome_dos_parâmetros)*

Requere: pré-condição

Garante: pós-condições

Os nomes dos parâmetros são nomes de variáveis seguidas dos símbolos ? e !, caso seja, respectivamente, um parâmetro de entrada ou saída.

A pré-condição é uma fórmula da lógica de 1ª ordem que deve ser satisfeita por forma a que o método possa ser executado.

Uma pós-condição é uma fórmula da lógica de 1ª ordem, da forma $Antecedente \Rightarrow Consequente$. O *Antecedente* e o *Consequente* são duas fórmulas. O *Antecedente* escolhe em que situação inicial o *Consequente* tem que ser satisfeito. Por essa razão o *Antecedente* só pode conter variáveis de estado e de entrada. Todas as pós-condições do método são satisfeitas pelo sistema depois de ter sido executado o método.

A especificação tem que ser determinística, i.e., em cada método os vários antecedentes das pós-condições têm que ser disjuntos entre si.
A especificação tem que ser robusta, i.e., em cada método a disjunção dos vários antecedentes das pós-condições tem que ser equivalente à pré-condição.

Na nossa linguagem de especificação não existe a garantia que se uma variável não for referida na parte consequente, o seu valor não será alterado. Se quisermos garantir isso temos que indicar explicitamente no consequente o valor da variável.

Uma variável de estado quando aparece no antecedente refere-se ao valor inicial da variável, antes do método ser executado. Quando esta aparece no consequente refere-se ao valor final da variável depois do método ser executado.

Exemplo 2.2 *Um exemplo de uma especificação de espécie única de um componente de um relógio é apresentada na figura 2.1.*
O método Inicia *inicializa o relógio com um determinado valor. O método* Muda *inverte o estado do relógio entre* Ligado *e* Desligado. *O método* Mostrar *devolve o actual valor do relógio. O método* Desliga *modifica o estado do relógio de* Ligado *para* Desligado. *Por fim o método* Avaria *coloca o relógio no estado* Avariado, *mas só se este não estiver no estado* Ligado.
Trata-se de um exemplo de uma especificação bastante simples, porque neste exemplo não são utilizados quantificadores, e o único predicado usado é a igualdade.
É de ter em atenção que a sintaxe usada neste exemplo não é a mesma que foi utilizada na implementação deste projecto (ver secção 4.1.1).

2.1.5 Formas Normais

As formas normais são formas sob as quais as fórmulas são substituídas por outras equivalentes, com uma estrutura mais regular, tornando a sua

Componente: Relógio

Espécie: {Ligado, Desligado, Avariado}

Variáveis: Estado

Método: Inicia

Interface: EstadoInicial?

Requere:

EstadoInicial?=Ligado $\vee$ EstadoInicial?=Desligado

Garante:

EstadoInicial?=Ligado $\Rightarrow$ Estado=Ligado

EstadoInicial?=Desligado $\Rightarrow$ Estado=Desligado

Método: Muda

Requere:

EstadoInicial?=Ligado $\vee$ EstadoInicial?=Desligado

Garante:

EstadoInicial?=Ligado $\Rightarrow$ Estado=Desligado

EstadoInicial?=Desligado $\Rightarrow$ Estado=Ligado

Método: Mostrar

Interface: Valor!

Garante:

Estado=Ligado $\Rightarrow$ (Valor!=Ligado $\wedge$ Estado=Ligado)

Estado=Desligado $\Rightarrow$ (Valor!=Desligado $\wedge$ Estado=Desligado)

Estado=Avariado $\Rightarrow$ (Valor!=Avariado $\wedge$ Estado=Avariado)

Método: Desliga

Requere: Estado=Ligado

Garante: Estado=Desligado

Método: Avaria

Requere: Estado=Desligado

Garante: Estado=Avariado

Figura 2.1: Especificação de um componente de um relógio

manipulação mais eficiente. Informação mais detalhada acerca das formas normais pode ser encontrada no livro [Hog90].

Representação Clausal

A representação clausal é um caso particular de uma forma normal.

Definição 2.6 *Uma* representação clausal *é um conjunto de cláusulas que denota a sua conjunção.*

Definição 2.7 *Uma* cláusula *é um conjunto de literais que denota a sua disjunção.*

Definição 2.8 *Um* literal positivo *é uma proposição . Um* literal negativo *é a negação de uma proposição . Um* literal *é um literal positivo ou negativo.*

Um conjunto de cláusulas pode ser antecedida por um conjunto de quantificadores.

Exemplo 2.3 *Uma representação clausal com duas cláusulas, contendo três literais cada uma:*
$(\forall X)(\exists Y)\{\{p_2(X), \neg p_1(Y), \neg p_3(X,Y)\}, \{\neg p_3(X,Y), p_1(Y), p_2(X)\}\}$
denota
$(\forall X)(\exists Y)((p_2(X) \vee \neg p_1(Y) \vee \neg p_3(X,Y)) \wedge (\neg p_3(X,Y) \vee p_1(Y) \vee p_2(X)))$
que é equivalente a
$(\forall X)(\exists Y)(((p_1(Y) \wedge p_3(X,Y)) \Rightarrow p_2(X)) \wedge (p_3(X,Y) \Rightarrow (p_1(Y) \vee p_2(X))))$

Como vimos no exemplo 2.3, uma cláusula pode ser vista como denotando uma implicação. Assim, por uma questão de legibilidade, neste projecto, optou-se por representar as cláusulas na forma:

N $\Rightarrow$ P em que

N é uma conjunção de predicados

P é uma disjunção de predicados

Um das características da representação clausal é que para toda a fórmula da lógica de 1ª ordem existe um conjunto de cláusulas logicamente equivalente. São conhecidos na literatura algoritmos de transformação de fórmulas da lógica de 1ª ordem numa representação clausal [Hog90].

2.2 Categorias

A tradução das especificações tem como resultado a sua respectiva representação categorial, isto é, uma representação usando a teoria das categorias, que é outra linguagem de especificação.

Representar o comportamento de qualquer componente de um sistema directamente na teoria das categorias é mais difícil do que através de uma especificação de espécie única. Assim, este tipo de representação torna-se difícil para quem a tem de criar e utilizar. Logo se percebe a utilidade deste projecto em produzir uma aplicação informática que faz automaticamente a conversão entre as duas formas de representação. Desta forma o utilizador não necessita de ter conhecimentos aprofundados sobre a teoria das categorias, para utilizar a aplicação desenvolvida.

A melhor forma de representar a teoria das categorias em computador é através de uma linguagem funcional [RB98], sendo esta a principal razão para a escolha da linguagem de programação usada neste projecto.

2.2.1 Vantagens

A representação categorial é uma linguagem de especificação poderosa, independente e baseada em fundamentos matemáticos, sendo cada vez mais utilizada em computação para modelar sistemas.

2.2.2 Descrição de uma Representação Categorial

Uma *representação categorial* é uma representação matemática usando a teoria das categorias, ou seja, é um conjunto de categorias interligadas entre si.

A *teoria das categorias* é a álgebra das funções, cuja principal operação é a composição de funções.

Uma categoria é uma estrutura abstracta, constituída por uma colecção de objectos interligados por uma colecção de setas. Por exemplo, os objectos podem ser figuras geométricas e as setas podem ser transformações que permitem obter uma figura a partir de outra. Outro exemplo, é os objectos serem dados e as setas programas.

Definição 2.9 *Uma* categoria ***Cat*** *consiste num conjunto de* objectos *(obj* ***Cat****) e num conjunto de* morfismos *ou* setas *(arr* ***Cat****). Os objectos são denotados por* $A, B, C, \ldots$ *e os morfismos por* $f, g, h, \ldots$

As propriedades que devem ser verificadas são as seguintes:

- *Cada morfismo tem um* domínio *e um* codomínio *em obj* ***Cat****. Quando o domínio de* f *é* A *e o codomínio de* f *é* B *escreve-se* $f : A \to B$*.*
- *Dado os morfismos* $f : A \to B$*,* $g : B \to C$*, existe um morfismo designado morfismo* composição*,* $g \circ f : A \to C$
- *Dado qualquer objecto* A*, existe um morfismo designado morfismo* identidade*,* $1_A : A \to A$
- *A composição de morfismos verifica ainda as seguintes leis:*

  Lei da identidade *Se* $f : A \to B$ *então*

  $$1_B \circ f = f \qquad e \qquad f \circ 1_A = f$$

Lei da associatividade *Se* $f : A \to B$, $g : B \to C$ *e* $h : C \to D$ *então*

$$h \circ (g \circ f) = (h \circ g) \circ f : A \to D$$

Definição 2.10 *Um objecto* 0 *é* inicial *numa categoria* C *sse por cada objecto* A *existe um e um só morfismo* 0→A.

Definição 2.11 *Um objecto* 1 *é* terminal *numa categoria* C *sse por cada objecto* A *existe um e um só morfismo* A→1.

Exemplo 2.4 *Seja a categoria* ***Cat*** *definida da seguinte forma:*

- *obj* ***Cat*** $= \{A, B\}$
- *arr* ***Cat*** $= \{sucessor, predecessor, 1_A, 1_B\}$

em que:

$A = \{1, 2\}$ *e* $B = \{2, 3\}$.

$sucessor : A \to B$,

é a função tal que sucessor(1)=2 e sucessor(2)=3.

$predecessor : B \to A$,

é a função tal que predecessor(2)=1 e predecessor(3)=2.

Esta categoria é composta por objectos que são conjuntos e por setas que são funções entre esses conjuntos. A composição é a composição usual de funções e as setas identidade são as funções identidade usuais.

No exemplo 2.4 é apresentado um caso de uma categoria simples, que tem definidas como setas as funções de predecessor e sucessor aplicadas a conjuntos limitados de números inteiros. Logo se pode comprovar a relação próxima entre as categorias e a programação, pois esta categoria não é mais que a especificação das duas funções descritas.

Para uma abordagem mais detalhada, sobre a teoria das categorias, podem ser consultados [Wal91, Pie93, SS93].

2.3 Bases de Dados

Uma base de dados não é mais que um conjunto de dados representados numa forma estruturada, que são guardados digitalmente enquanto necessários. Para se gerir uma base de dados tem que existir um sistema que se encarrega de manipular os dados que estejam contidos na base de dados.

Complementos sobre o que é uma base de dados e de como se cria e gere podem ser consultados em [Dat94, BCN92].

2.3.1 Vantagens

Uma vez que as conversões feitas pela aplicação não são triviais, o respectivo tempo de execução irá ser elevado. Então , por forma a diminuir o número de conversões a executar, cria-se uma base de dados que contenha os resultados obtidos pelos algoritmos até ao momento. Desta forma, sempre que for necessário uma representação categorial de uma especificação que já foi traduzida basta procurar na base de dados, em vez de se executar de novo os algoritmos.

Outra das razões reside no facto das especificações terem de ser guardadas em algum sítio, então porque não aproveitar para guardá-las ordeiramente junto com o aplicação .

Capítulo 3

Técnicas

Neste capítulo descreve-se o conjunto de técnicas e metodologias utilizadas no desenvolvimento deste projecto.

3.1 Ferramentas Usadas

Todos os algoritmos desenvolvidos para este trabalho, foram codificados numa linguagem de programação funcional, mais precisamente *Standard ML (SML)*. A razão para a escolha deste tipo de linguagem reside no facto de esta estar mais perto da notação matemática do que as linguagens imperativas, como o *Pascal* e *C*.
O *SML* utiliza expressões para denotar entidades matemáticas, em vez de definir as transições de uma máquina abstracta. Estes factos adequam-se perfeitamente aos objectivos deste trabalho, pois vamos ter que manipular dois tipos de representações matemáticas na resolução do nosso problema.

O primeiro passo dado na implementação deste trabalho foi a procura das ferramentas desta linguagem que mais se adequassem a este projecto. As ferramentas utilizadas para o desenvolvimento da aplicação fazem parte do conjunto de utilitários que compõem a *Version 110, Patch 6 (110.0.6) of*

Standard ML of New Jersey. Estas estão disponíveis na *internet* no endereço: *http://cm.bell-labs.com/cm/cs/what/smlnj*

As ferramentas utilizadas têm todas a vantagem de ser de domínio público. Outra vantagem de se utilizar estas ferramentas reside no facto do programa desenvolvido poder ser utilizado na maioria das máquinas e sistemas operativos.

A desvantagem de se ter utilizado estas ferramentas foi que estas estão muito longe de terem uma boa interface para o utilizador e uma boa documentação, quando comparado com outras ferramentas de outras linguagens mais comerciais. Isto implicou necessariamente o aumento do tempo dispendido na fase de implementação da aplicação informática produzida, devido à dificuldade na aprendizagem da utilização destas ferramentas.

Mas, o balanço das vantagens e dos inconvenientes revelou-se favorável ao primeiro, o que se veio a confirmar neste relatório.

3.2 Opções Tomadas

Nesta secção são referidas as opções mais relevantes que foram tomadas durante a revisão do artigo[Cre98a] que serviu de base a este trabalho. Os aspectos que são melhorados em relação ao que era proposto no artigo estão referidos seguidamente:

- O predicado de igualdade terá um significado especial.
- Vai ser revista a forma como os quantificadores podem ser aplicados.
- Irá se tentar alterar a implementação do conjunto espécie de modo a poder ser infinito.
- Vai se estender a linguagem das fórmulas que podem aparecer na especificação. Mais precisamente é agora possível representar implicitamente fórmulas do tipo: $\forall x\,[P_1(x) \Rightarrow P_2(x)]$

- Irão ser alterados e criados novos algoritmos de tradução. As fórmulas irão ser representadas todas em representação clausal. O algoritmo de tradução de quantificadores vai ser implementado através de uma nova metodologia desenvolvida neste trabalho.

3.2.1 Predicados só com Um Argumento

Nas especificações de espécie única, utilizadas neste projecto, só se pode utilizar predicados com aridade um, isto é, predicados com um só argumento. A justificação para esta restrição, reside no facto de a utilização de predicados com aridade superior a um torna a relação entre a lógica de predicados de 1ª ordem e a teoria das categorias demasiadamente complexa, para o âmbito deste trabalho.

3.2.2 Quantificadores

Neste trabalho, os quantificadores que aparecem explicitamente nas fórmulas das especificações só podem ser aplicados a um predicado, e não a um conjunto de cláusulas.
A justificação para esta restrição, reside no facto de que a utilização de quantificadores aplicados a várias cláusulas torna a relação entre a lógica de predicados de 1ª ordem e a teoria das categorias demasiadamente complexa, para o âmbito deste trabalho.

3.2.3 Formas Normais

Criou-se uma nova definição de cláusula, a que se dá o nome de pseudo-cláusula, por forma a permitir a representação das nossas especificações em forma normal. Uma pseudo-claáusula tem definição análoga à definição de

cláusula apresentada anteriormente mas para além de literais podem existir também quantificadores aplicados a um literal.

Por forma a tornar a manipulação das especificações de espécie única mais eficiente, estas têm que ser introduzidas pelo utilizador com todas as suas fórmulas representadas através de pseudo-cláusulas.
Isto podia ser evitado se fosse implementado o algoritmo de transformação em pseudo-cláusulas, libertando assim o utilizador dessa tarefa. Mas como já foi dito este projecto não tem ambições comerciais, logo não se deu prioridade à interface da aplicação informática com o utilizador.

Desta forma ficamos com as seguintes características na representação de uma especificação:

- O invariante de uma especificação é representado através de um conjunto de pseudo-cláusulas.
- A pré-condição de um método é representada através de um conjunto de cláusulas que só contenham literais positivos. Uma cláusula só com literais positivos não é mais que uma disjunção de proposições .
- Uma pós-condição de um método é representada através de um par antecedente e consequente, em que o antecedente é uma cláusula de literais positivos e o consequente é um conjunto de pseudo-cláusulas.

3.2.4 Implicação entre Predicados

Ao conjunto de fórmulas possíveis numa especificação acrescentou-se uma nova, a implicação entre predicados.
Ou seja, na definição de uma fórmula, apresentada na página 26, acrescenta-se a regra 4:

4. Se P_1 e P_2 são predicados, então $P_1 \rightarrow P_2$ é também uma fórmula.

O significado desta nova fórmula é o mesmo do que o da fórmula:
$\forall x\,[P_1(x) \Rightarrow P_2(x)]$
Criou-se este novo tipo de fórmula porque no nosso caso não é possivel utilizar explicitamente o quantificador aplicado a uma fórmula. E também porque este é um tipo de fórmula muito usada nas especificações e o seu o significado encaixa-se perfeitamente numa representação categorial.

3.2.5 Igualdade(=) como Predicado Especial

O predicado de igualdade(=) desempenha um papel fundamental nas especificações. O predicado referente à igualdade tem aridade dois, mas como neste trabalho só temos predicados com aridade um, transformamos o predicado =(Variável,Constante) no predicado =Variável(Constante). Ou seja, existe um predicado = distinto por cada variável embora tenham todos o mesmo significado, que é de verificar se o valor da variável é igual à da constante.

Consequentemente, pressupõe-se que nas especificações que se consideram neste trabalho o argumento de um predicado igualdade só pode ser uma constante e não uma variável. Isto é, não se permite ter Var1=Var2, que corresponde a ter =Var1(Var2) ou =Var2(Var1). Se esta opção não fosse tomada tinha-se que escolher um de entre os dois, por exemplo =Var1(Var2). Mas se noutra especificação tivéssemos Var2=Var1 então iriamos escolher =Var2(Var1). E embora Var1=Var2 seja equivalente a Var2=Var1, não seria possível emparelhar os predicados =Var1(Var2) e =Var2(Var1), visto serem dois predicados distintos. Este problema só será resolvido quando tivermos a possibilidade de ter predicados com aridade superior a um.
O predicado referente à igualdade é um predicado especial, porque tem um significado associado e esse significado é igual em qualquer especificação utilizada. Por isso, no emparelhamento dos predicados vai se ter em conta que não se pode emparelhar um predicado de igualdade com um que não seja de igualdade. Tal como existe o predicado especial igualdade podiam existir

outros predicados especiais, tal como menor(<), maior(>), ...
Isto porque todos estes predicados já têm um significado próprio associado a cada um deles. Neste projecto só se usou o predicado igualdade, mas este trabalho está preparado para se acrescentar, se necessário, outros tipos de predicados especiais.

3.2.6 Conjunto Espécie Finito

Neste trabalho, o conjunto de suporte da espécie não pode ser infinito. Isto não é grande limitação se pensarmos que qualquer implementação de um método tem sempre os valores das variáveis limitadas. Por exemplo, um inteiro na linguagem C tem sempre um valor máximo e mínimo.
Durante o planeamento do projecto, foi considerado também o caso de o conjunto de construtores da espécie poder ser infinito. Por isso, se necessário, é possível uma implementação do trabalho com o conjunto *Espécie* infinito.

3.2.7 Quantificadores na Especificação

Uma pré-condição e o antecedente de uma pós-condição podem também conter quantificadores, ou seja, representadas através de pseudo-cláusulas. Esta possibilidade implica que o antecedente da pós-condição possa ser transformado em múltiplos objectos. Isto tem a vantagem de permitir ao utilizador especificar a mesma informação com menos esforço, ou seja com menos linhas de especificação.

Capítulo 4

Resultados

A aplicação informática desenvolvida nesta primeira fase corresponde ao processo Tradutor da figura 1.1, na página 19. Este processo foi decomposto em outros três processos, tal como se indica na figura 4.1. Nesta secção são descritos esses processos, bem como as estruturas de dados que utilizam.

4.1 Estrutura de Dados das Especificações

As especificações de espécie única utilizadas neste projecto foram representadas de duas formas distintas, denominadas por externa e interna.

4.1.1 Representação Externa

A representação externa corresponde à forma como é representada a especificação que serve de argumento de entrada da aplicação .
Pode-se verificar na figura 4.1 que este tipo de representação é a forma usada pelo utilizador para introduzir uma especificação .
A informação é representada através de uma cadeia de caracteres que são escritos num ficheiro, e que representam uma especificação correspondente a um componente. Este ficheiro pode ser criado pelo utilizador através de um

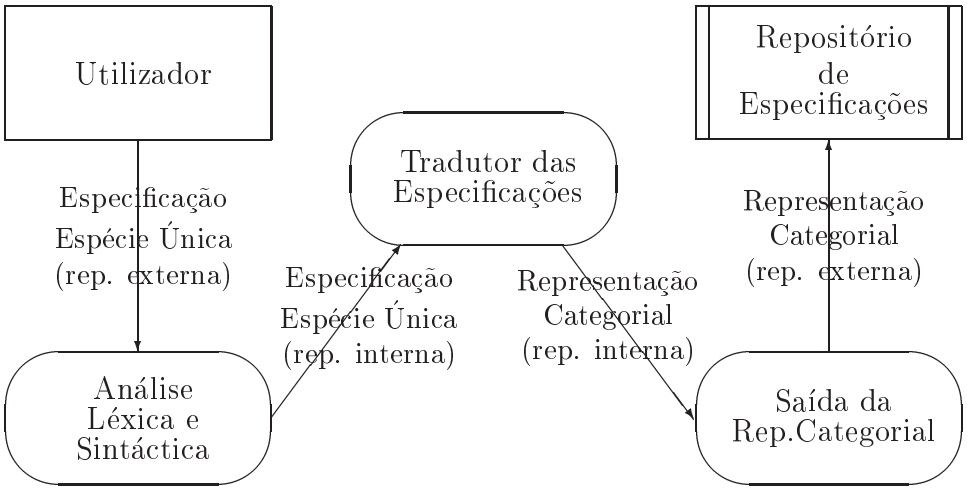

Figura 4.1: DFD da aplicação informática desenvolvida

qualquer editor de texto, tendo em conta que o ficheiro tem de ser gravado só como texto.

As vantagens deste tipo de representação residem no facto da informação ficar guardada permanentemente e também no facto da informação poder ser utilizada facilmente por uma outra qualquer aplicação informática.

A representação da especificação tem que obedecer à sintaxe em notação EBNF[1] apresentada na secção A.1.

No meio de uma representação pode-se sempre colocar um comentário. Para isso coloca-se o caracter % seguido do comentário até ao final da linha. Ou seja, todo o texto que esteja entre o caracter % e o final da linha é ignorado pelo analisador léxico.

É de notar que esta sintaxe é diferente da descrita anteriormente na secção 2.1.4, isto por forma a ser possível guardar a informação num ficheiro de texto *ascii*. Isto evita a necessidade de o utilizador usar um editor especial para criar o ficheiro com a especificação.

[1]EBNF (*Extended Backus Normal Form*) é uma forma de definir uma sintaxe. Mais informação sobre EBNF pode ser obtida em [Cre98b, páginas 34–35].

As alterações feitas na sintaxe de uma representação foram as seguintes:

- O símbolo $\forall$ foi substituído pelo símbolo $>$.
- O símbolo $\exists$ foi substituído pelo símbolo $<$.
- O símbolo $\wedge$ foi substituído pelo símbolo &.
- O símbolo $\vee$ foi substituído pelo símbolo |.
- Os símbolos ? e ! passaram a ser utilizados no inicio da variável.
- As fórmulas da lógica de 1ª ordem usadas apresentam-se em representação clausal, tal como foi dito na secção 3.2.3.
- Os quantificadores só podem estar aplicados a um predicado, tal como foi dito na secção 3.2.2.
- Só se pode ter predicados com aridade um, tal como foi dito na secção 3.2.1.

Embora tenham sido apresentadas duas sintaxes diferentes a semântica manteve-se inalterada para ambas as formas de representação, excepto no caso referido da secção 3.2.4.

4.1.2 Representação Interna

A representação interna corresponde à forma de como neste trabalho representa uma especificação na memória. Uma especificação só está representada deste modo enquanto o programa está activo. A vantagem deste tipo de representação é a maior eficiência na manipulação da informação pelos algoritmos.

A estrutura usada para a representação interna de uma especificação está definida no ficheiro *data_spc.sml*.

Neste tipo de representação uma especificação é representada através de um conjunto de registos e listas. Deste modo pode-se definir a estrutura de dados aqui usada como sendo:

- Um componente é um registo composto por:
  - O seu nome.
  - Uma lista de identificadores que representa o conjunto *Espécie*.
  - Uma lista das variáveis de estado.
  - Uma lista de fórmulas relativas ao invariante.
  - Uma lista dos métodos.
- Um método é um registo composto por:
  - O seu nome.
  - Uma lista de variáveis de interface.
  - Uma lista de subfórmulas referentes à pré-condição.
  - Uma lista de pós-condições.
- Uma pós-condição é um registo composto por:
  - Uma subfórmula referente ao antecedente da pós-condição.
  - Uma lista de fórmulas referente ao consequente da pós-condição.
- Uma fórmula pode ser uma implicação de predicados ou uma implicação de subfórmulas.
- Uma implicação de predicados é um registo composto por:
  - Um predicado referente ao antecedente da implicação.
  - Um predicado referente ao consequente da implicação.
- Uma implicação de subfórmulas é um registo composto por:

  - Uma subfórmula referente ao antecedente da implicação.
  - Uma subfórmula referente ao consequente da implicação.

- Uma subfórmula pode ser uma conjunção ou uma disjunção.
- Uma conjunção é constituída por uma lista de asserções.
- Uma disjunção é constituída por uma lista de asserções.
- Uma asserção pode ser um dos seguintes casos:
  - A constante *true*.
  - A constante *false*.
  - Um predicado aplicado a uma constante.
  - O quantificador universal aplicado a um predicado.
  - O quantificador existencial aplicado a um predicado.
- Um predicado pode ser um dos seguintes casos:
  - Um identificador de um predicado definido pelo utilizador.
  - O predicado igualdade referente a uma variável. Ou seja, o predicado que recebe uma constante e verifica se essa variável tem um valor igual a esta constante.
- Uma variável pode ser um dos seguintes casos:
  - Um identificador de uma variável de estado.
  - Um identificador de uma variável de entrada.
  - Um identificador de uma variável de saída.
- Uma constante é um identificador.
- O nome de um objecto é uma cadeia de caracteres.
- Um identificador é uma cadeia de caracteres.

4.2 Análise Léxica e Sintáctica de uma Especificação

A manipulação de uma especificação representada num ficheiro é muito ineficiente, logo a primeira operação feita pela aplicação é transformar a representação externa da especificação na sua respectiva representação interna. O processo é subdividido nas fases de análise léxica e sintáctica [Cre98b].
A análise léxica constitui a primeira etapa no processamento dos dados de entrada. Para além de agrupar os caracteres da especificação fonte em palavras, o analisador léxico tem por objectivo a eliminação de comentários da especificação.
A análise sintáctica constitui a segunda etapa no processamento dos dados de entrada e tem por objectivo verificar se a sequência de palavras identificada na análise léxica faz parte da linguagem de uma especificação.

Para implementar este processo foram usadas as ferramentas ML-Lex e ML-Yacc, o primeiro é um analisador léxico e o segundo um analisador sintáctico.

O código para o analisador léxico está definido no ficheiro *syntax.lex*, aí são identificados todos os símbolos terminais da gramática usada na representação externa da especificação.

O código para o analisador sintáctico está definido no ficheiro *syntax.grm*. Neste caso foi só usar da representação externa, por forma a criar a respectiva representação interna, através de simples comandos de construção das estruturas.

Era muito útil que as palavras chave detectadas no analisador léxico fossem depois utilizadas pelo analisador sintáctico. Como, por exemplo, a cadeia de caracteres que representa um identificador. Para isso, foi necessário criar algumas funções que estão definidas no ficheiro *syntax.sml*. Estas funções possibilitam a utilização das duas ferramentas em conjunto.

4.3 Estrutura de Dados das Representações Categoriais

As representações categoriais utilizadas neste trabalho foram representadas de duas formas distintas, denominadas por externa e interna.

4.3.1 Representação Externa

A representação externa corresponde à forma como é representada a representação categorial que constitui o argumento de saída da aplicação .
Pode-se verificar na figura 4.1 que este tipo de representação é a forma usada para guardar uma representação categorial na base de dados.
A informação é representada através de uma cadeia de caracteres que são escritos num ficheiro, representando uma especificação correspondente a um componente. Este ficheiro pode ser acedido pelo utilizador através de um qualquer editor de texto, tendo em conta que é um ficheiro de texto normal.

As vantagens deste tipo de representação residem no facto da informação ficar guardada permanentemente e também no facto da informação poder ser utilizada facilmente por uma outra qualquer aplicação informática.

A representação categorial externa segue a sintaxe em notação EBNF apresentada na secção A.2.

Para se entender a semântica desta gramática é obrigatório ler a secção 4.4

4.3.2 Representação Interna

A representação interna corresponde à forma de representar uma representação categorial em memória. Assim, uma especificação só está representada deste modo enquanto o programa está activo.

A vantagem deste tipo de representação é a maior eficiência na manipulação da informação pelos algoritmos.

A estrutura usada para a representação interna de uma especificaçãoestá definida no ficheiro *data_ctg.sml*.

Uma representação categorial interna é representada através de um conjunto de registos e listas. Deste modo pode-se definir a estrutura de dados aqui usada como sendo:

- Um componente é um registo composto por:
  - O seu nome.
  - Uma lista dos seus métodos.
- Um método é um registo composto por:
  - O seu nome.
  - Uma pré-categoria.
  - Uma lista de pares em que o primeiro elemento é um C-objecto e o segundo é uma pós-categoria.
- Uma pré-categoria é um par de categorias.
- Uma pós-categoria é um par de categoria.
- Um par de categorias é um registo composto por:
  - Uma categoria (C,$\rightarrow$).
  - Uma categoria (P,$\rightarrow$).
- Uma categoria (x,$\rightarrow$) (x $\in$ {C,P}) é um registo composto por:
  - Uma lista de objectos x.
  - Uma lista de setas x.

- Uma seta x (x ∈ {C,P}) é um registo composto por:
  - Um objecto x, referente à origem da seta.
  - Um objecto x, referente ao destino da seta.
- Um C-objecto pode ser um dos seguintes casos:
  - Um objecto singular.
  - Um máximo de proposições.
  - Um mínimo de proposições.
- Um objecto singular é constituído por uma proposição.
- Um máximo de proposições é constituído por uma lista de asserções.
- Um mínimo de proposições é constituído por uma lista de asserções.
- Uma asserção[2] pode ser um dos seguintes casos:
  - Uma constante referente ao objecto terminal.
  - Uma constante referente ao objecto inicial.
  - Um predicado aplicado a uma constante.
- Um predicado aplicado a uma constante é um registo composto por:
  - Um predicado.
  - Uma constante.
- Um predicado pode ser um dos seguintes casos:
  - Um identificador de um predicado definido pelo utilizador.
  - O predicado igualdade referente a uma variável. Ou seja, o predicado que recebe uma constante e verifica se essa variável tem um valor igual a esta constante.

[2]Note-se que neste caso tem-se um subconjunto das possíveis asserções anteriormente referidas

- Um P-objecto é um identificador de um predicado definido pelo utilizador.
- Uma variável pode ser um dos seguintes casos:
  - Um identificador de uma variável de estado.
  - Um identificador de uma variável de entrada.
  - Um identificador de uma variável de saída.
- Uma constante é um identificador.
- Um identificador é uma cadeia de caracteres.

4.4 Tradução das Especificações

Como se pode verificar na figura 4.1, a tradução de especificações tem como objectivo transformar uma especificação de espécie única na sua respectiva representação categorial, ambas em representação interna.

O algoritmo de tradução entre os dois tipos de representação aqui descrito, foi desenvolvido tomando como ideia base a relação que se pode obter entre a lógica de 1ª ordem e as categorias [Lai76].
Essa relação consiste no facto de dado um conjunto de fórmulas da lógica de 1ª ordem, estas revelarem uma categoria (C,→) e uma categoria (P,→).
A categoria (P,→) é formada da seguinte forma:

- O conjunto P refere-se a todos os predicados usados nesse conjunto de fórmulas.
- Setas em (P,→) correspondem ás implicações entre predicados presentes nesse conjunto.

A categoria (C,→) é formada da seguinte forma:

- O conjunto C é constituído da seguinte forma:
  - Dado um predicado p e uma constante c, o objecto p(c) corresponde à proposição referente à aplicação da constante c ao predicado p.
  - Dadas as proposições p e q, o objecto $\sqcap${p,q} corresponde à disjunção entre p e q ($p \wedge q$).
  - Dadas as proposições p e q, o objecto $\sqcup${p,q} corresponde à conjunção entre p e q ($p \vee q$).
  - O objecto terminal corresponde à proposição *true*.
  - O objecto inicial corresponde à proposição *false*.
- Setas em (C,$\rightarrow$) correspondem ás implicações presentes no conjunto.

As funções que constituem o algoritmo utilizado por este processo estão definidas no ficheiro *trad.sml*. O algoritmo de tradução de uma especificação de um componente para a respectiva representação categorial, pode ser descrita da seguinte forma:

Tradução de um Componente
Entrada: Uma especificação em representação interna.
Saída: A respectiva representação categorial interna da especificação.

1. Executar o algoritmo *Tradução de um Método* para cada método do componente.
2. Criar uma lista com as representações categoriais obtidas para cada método.

Tradução de um Método
Entrada: Uma especificação de um método de um componente.
Saída: A respectiva representação categorial do método.

1. Executar o algoritmo *Tradução de Cláusulas* com os seguintes argumentos:

   (a) A conjunção das fórmulas $\varepsilon_{ai} \Rightarrow R$ (com i=0, ..., nº de pós-condições), em que ε_{ai} é uma fórmula do antecedente de uma pós-condicão e R é a pré-condição.
   Como este argumento é uma conjunção de fórmulas traduz-se num conjunto de cláusulas.

   (b) O conjunto das variáveis de estado e de entrada.

   Ao par de categorias que é devolvido pelo algoritmo neste passo dá-se o nome de pré-categoria.

2. Por cada consequente de uma pós-condição executar o algoritmo *Tradução de Cláusulas* com os seguintes argumentos:

   (a) O conjunto das pseudo-cláusulas que constituem o consequente.

   (b) O conjunto das variáveis de estado, de entrada e de saída.

   A cada par de categorias que é devolvido pelo algoritmo neste passo, dá-se o nome de pós-categoria.

3. Por cada pós-condição associar a sua pós-categoria ao objecto da pré-categoria que representa o antecedente.

Tradução de Cláusulas

Entrada:

A - Um conjunto de pseudo-cláusulas.

V - Um conjunto de variáveis

Saída: O respectivo par de categorias (C,$\rightarrow$) e (P,$\rightarrow$).

1. Adicionar ao conjunto de pseudo-cláusulas A o conjunto de pseudo-cláusulas que constituem o invariante do componente.

2. Para cada variável $v_i \in V$ acrescenta-se a pseudo-cláusula $true \Rightarrow \exists x\,(v_i = x)$, ao conjunto A.

3. Executar o algoritmo *Tradução de uma Cláusula* por cada pseudo-cláusula pertencente ao conjunto A.

4. Adicionar todos os objectos e setas das categorias devolvidas no passo anterior, por forma a ficar só com o par de categorias (C,$\rightarrow$) e (P,$\rightarrow$).

Tradução de uma Cláusula

Entrada: a - Uma pseudo-cláusula.

Saída: O conjunto de objectos e setas derivadas da pseudo-cláusula.

1. Caso a pseudo-cláusula a seja o caso especial de implicação de predicados. Então a pode-se representar da seguinte forma:
   $p_1 \rightarrow p_2$ (com p_1 e p_2 sendo dois quaisquer predicados)

   (a) Criar os objectos p_1 e p_2, na categoria (P,$\rightarrow$).

   (b) Criar a seta com origem em p_1 e destino em p_2, pertencente à categoria (P,$\rightarrow$).

   (c) Terminar aqui o algoritmo neste caso.

2. Executar o algoritmo *Retira Quantificadores* para a pseudo-cláusula a.

3. Por cada cláusula b resultante do passo anterior fazer:

   (a) A cláusula b é representada da seguinte forma:
   $\{p_1 \wedge \ldots \wedge p_n \Rightarrow q_1 \vee \ldots \vee q_m\}$
   Em que:

   - m e n são dois quaisquer números inteiros maiores que zero.
   - Cada um dos símbolos $p_1, \ldots, p_n, q_1, \ldots, q_m$ pode representar uma das seguintes proposições :
     - Um predicado aplicado a uma constante $p(c)$.

- O símbolo *true*
- O símbolo *false*.

(b) Criar os seguintes objectos pertencentes a C:

- $min = \begin{cases} \sqcap\{p_1, \ldots, p_n\} & \text{se } n > 1 \\ \%p_1 & \text{c.c.} \end{cases}$
- $max = \begin{cases} \sqcup\{q_1, \ldots, q_m\} & \text{se } n > 1 \\ \%q_1 & \text{c.c.} \end{cases}$

O símbolo $\sqcap$ corresponde ao mínimo(intersecção) do conjunto.

O símbolo $\sqcup$ corresponde ao máximo(reunião) do conjunto.

O símbolo $\%$ corresponde ao objecto singular.

(c) Criar a seta na categoria (C,$\rightarrow$) com origem em min e destino em max.

(d) Por cada um dos símbolos $p_1, \ldots, p_n, q_1, \ldots, q_m$ que represente um predicado aplicado a uma constante $p(c)$, adicionar o objecto referente ao predicado p à categoria (P,$\rightarrow$).

Antes de se descrever o algoritmo *Retira Quantificadores*, apresentam-se alguns resultados teóricos necessários para a compreensão desse algoritmo.

Resultado 4.1 *Sendo p um predicado e $\{c_1, \ldots, c_n\}$ o conjunto de possíveis valores para uma variável. Então temos os seguintes resultados:*

- $\forall x\, p(x) \Leftrightarrow p(c_1) \wedge \ldots \wedge p(c_n)$
- $\exists x\, p(x) \Leftrightarrow p(c_1) \vee \ldots \vee p(c_n)$

Resultado 4.2 *Sendo a, b, c, d proposições, então temos as seguintes tautologias:*

- $[a \wedge (b \vee c) \Rightarrow d] \Leftrightarrow [(a \wedge b \Rightarrow d) \wedge (a \wedge c \Rightarrow d)]$
- $[a \Rightarrow b \vee (c \wedge d)] \Leftrightarrow [(a \Rightarrow b \vee c) \wedge (a \Rightarrow b \vee d)]$

Note-se que as fórmulas do lado direito da equivalência estão em representação causal.

Retira Quantificadores

Entrada:

a - Uma pseudo-cláusula.

X - O conjunto *Espécie* que define o tipo de variáveis.

Saída: Um conjunto de cláusulas equivalente à pseudo-cláusula de entrada, ou seja, sem quantificadores.

1. A pseudo-cláusula a é representada da seguinte forma:
   $\{p_1 \wedge \ldots \wedge p_n \Rightarrow q_1 \vee \ldots \vee q_m\}$
   Em que:
   - m e n são dois quaisquer números inteiros maiores que zero.
   - Cada um dos símbolos $p_1, \ldots, p_n, q_1, \ldots, q_m$ pode representar uma das seguintes asserções :
     - Um predicado aplicado a uma constante $p(c)$.
     - Um quantificador aplicado a um predicado $\forall x\, P(x)$ ou $\exists x\, P(x)$.
     - O símbolo *true*
     - O símbolo *false*.
2. O conjunto X é representado por $\{x_1, \ldots, x_r\}$.
   Em que:
   - r é um qualquer número inteiro maior que zero.
   - Cada um dos símbolos $x_1, \ldots, x_r$ representam constantes.
3. Aplicar o resultado 4.1 à pseudo-cláusula a. Assim, por cada símbolo do tipo $p_1, \ldots, p_n, q_1, \ldots, q_m$ que represente:,
   - $\forall x\, P(x)$, substituir por $P(x_1) \wedge \ldots \wedge P(x_r)$.

- $\exists x\, P(x)$, substituir por $P(x_1) \vee \ldots \vee P(x_r)$.

4. Aplicar o resultado 4.2, por forma a transformar a fórmula obtida no passo anterior num conjunto de cláusulas.

Exemplo 4.1 *Devido à complexidade do algoritmo* Tradução de uma Cláusula, *chegou-se à conclusão que talvez a melhor forma de o entender será através de um exemplo.*

Considera-se a seguinte pseudo-cláusula a*:*

$$\forall x\, p_1(x) \wedge \exists x\, p_2(x) \wedge p_3(c_1) \Rightarrow \forall x\, p_4(x) \vee \exists x\, p_5(x) \vee p_6(c_2)$$

Considera-se o conjunto Espécie *do componente igual a* $\{x_1, x_2\}$

Utilizando o resultado 4.1, retiram-se os quantificadores e fica-se com a seguinte fórmula:

$p_1(x_1) \wedge p_1(x_2) \wedge (p_2(x_1) \vee p_2(x_2)) \wedge p_3(c_1) \Rightarrow (p_4(x_1) \wedge p_4(x_2)) \vee p_5(x_1) \vee p_5(x_2) \vee p_6(c_2)\}$

Utilizando o resultado 4.2, transforma-se a fórmula anterior no seguinte conjunto de cláusulas, que é equivalente à pseudo-cláusula a*:*

$\{p_1(x_1) \wedge p_1(x_2) \wedge p_2(x_1) \wedge p_3(c_1) \Rightarrow p_4(x_1) \vee p_5(x_1) \vee p_5(x_2) \vee p_6(c_2),$
$p_1(x_1) \wedge p_1(x_2) \wedge p_2(x_2) \wedge p_3(c_1) \Rightarrow p_4(x_1) \vee p_5(x_1) \vee p_5(x_2) \vee p_6(c_2),$
$p_1(x_1) \wedge p_1(x_2) \wedge p_2(x_1) \wedge p_3(c_1) \Rightarrow p_4(x_2) \vee p_5(x_1) \vee p_5(x_2) \vee p_6(c_2),$
$p_1(x_1) \wedge p_1(x_2) \wedge p_2(x_2) \wedge p_3(c_1) \Rightarrow p_4(x_2) \vee p_5(x_1) \vee p_5(x_2) \vee p_6(c_2)\}$

Seguidamente executam-se os seguintes passos:

1. *Criam-se os seguintes objectos na categoria (C,*$\rightarrow$*):*

   - $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_1), p_3(c_1)\}$
   - $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_2), p_3(c_1)\}$
   - $\sqcup\{p_4(x_1), p_5(x_1), p_5(x_2), p_3(c_1)\}$
   - $\sqcup\{p_4(x_2), p_5(x_1), p_5(x_2), p_3(c_1)\}$

2. *Criam-se as seguintes setas da categoria (C,*$\rightarrow$*):*

- *origem:* $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_1), p_3(c_1)\}$
  destino: $\sqcup\{p_4(x_1), p_5(x_1), p_5(x_2), p_3(c_1)\}$
- *origem:* $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_2), p_3(c_1)\}$
  destino: $\sqcup\{p_4(x_1), p_5(x_1), p_5(x_2), p_3(c_1)\}$
- *origem:* $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_1), p_3(c_1)\}$
  destino: $\sqcup\{p_4(x_2), p_5(x_1), p_5(x_2), p_3(c_1)\}$
- *origem:* $\sqcap\{p_1(x_1), p_1(x_2), p_2(x_2), p_3(c_1)\}$
  destino: $\sqcup\{p_4(x_2), p_5(x_1), p_5(x_2), p_3(c_1)\}$

3. *Cria-se os seguinte objectos pertencentes a P:*

   $p_1, p_2, p_3, p_4, p_5, p_6$

4.5 Saída da Representação Categorial

Tal como foi dito, existe uma base de dados mantida pela aplicação informática produzida neste projecto que corresponde ao repositório de especificações da figura 4.1.
Este repositório contém todas as representações categoriais criadas pela aplicação informática a partir das especificações de espécie única dadas pelo utilizador.
O processo de saída da representação categorial da figura 4.1 tem como tarefa pegar na representação categorial interna e produzir a respectiva representação categorial externa. Por isso por cada especificação de espécie única de um componente, este processo cria a cadeia de caracteres que representa categorialmente os vários métodos do componente. Esta cadeia de caracteres que representa os vários métodos do mesmo componente é escrita num ficheiro.

Por forma a facilitar a manipulação da informação mantida no repositório, os nomes dos ficheiros utilizados pela aplicação informática cumprem a seguinte regra. Os ficheiros que contêm as especificações de espécie única têm

a extensão *esp*, e os ficheiros que contêm as representações categoriais têm a extensão *ctg*. Ou seja, os dois ficheiros que representam a funcionalidade de um componente têm o mesmo nome com diferentes extensões.

Exemplo 4.2 *No caso do componente relógio do exemplo 2.2, ficamos com os seguintes ficheiros:*

relogio.esp

relogio.ctg

As funções usadas por este processo estão definidas no ficheiro *out_ctg.sml*. O algoritmo para obter a representação externa de uma representação categorial a partir da sua representação interna, pode ser descrita da seguinte forma:

Saída da Representação Categorial
Entrada: Uma representação categorial interna.
Saída: Um ficheiro com a respectiva representação categorial externa.

1. Escrever o nome do componente no ficheiro.
2. Por cada método do componente, primeiro escrever o seu nome, depois a sua pré-categoria e em seguida as suas pós-categorias.
   (a) Por cada pré-categoria de um método, escreve-se o par de categorias que a representam.
   (b) Por cada pós-categoria referente a uma pós-condição de um método, escreve-se o objecto que representa o antecedente e depois o par de categorias que representam o consequente.
3. Por cada par de categorias, escreve em primeiro lugar a categoria (C,$\rightarrow$) e depois a categoria (P,$\rightarrow$).
4. Por cada categoria, escreve-se os seus objectos e depois a suas setas.
   (a) Por cada objecto identifica-se o seu tipo e escreve-se a sua representação externa respectiva.

(b) Por cada seta, escreve-se o objecto de origem relacionado com o objecto de destino.

Capítulo 5

Conclusões

A principal conclusão retirada desta primeira fase de desenvolvimento do projecto foi a de que é possível transformar uma especificação de espécie única numa representação categorial através de um processo algorítmico.

O trabalho efectuado nesta primeira fase foi dividido em duas etapas. A primeira etapa consistiu na revisão e no estudo da viabilidade dos objectivos descritos no artigo [Cre98a], nos aspectos que seguidamente se referem:

- Aprofundaram-se alguns aspectos deixados em aberto. Deu-se um significado especial ao predicado igualdade. Foi revista a forma como os quantificadores podiam ser aplicados. Pensou-se como se podia alterar a implementação do conjunto espécie de modo a ser infinito.

- Estendeu-se a linguagem das fórmulas que podem aparecer na especificação. Mais precisamente é agora possível representar implicitamente fórmulas do tipo: $\forall x\,[P_1(x) \Rightarrow P_2(x)]$

- Foram alterados e criados novos algoritmos de tradução. As fórmulas são representadas todas em representação clausal. O algoritmo de tradução de quantificadores foi implementado através de uma nova metodologia desenvolvida durante este trabalho.

Na segunda etapa o resultado prático obtido pelo trabalho foi a implementação de uma aplicação informática. O comportamento desta aplicação identifica-se por receber uma especificação de espécie única, de um determinado componente de um sistema, e devolver a sua respectiva representação categorial.

Por forma a testar a correcção da aplicação informática criada, foram identificadas algumas especificações. Estas serviram de dado de entrada para a aplicação, por forma a serem obtidas as respectivas representações categoriais.

No apêndice B são apresentados as especificações do exemplo 2.2 que foram utilizadas na aplicação informática.
Outro exemplo, apresentado nesse apêndice, é o de uma especificação que descreve o funcionamento de três relógios. Neste exemplo, tentou-se colocar o máximo de casos tipo possíveis, ou seja tentou-se fazer uma quase generalização de uma especificação. Por isso não se deu importância ao aspecto de se obter uma especificação de um relógio com funcionalidades pouco comuns, pois este é um caso de estudo.
No mesmo apêndice apresenta-se também as representações categoriais obtidas, pela aplicação informática, a partir das especificações descritas anteriormente. Um aspecto que se identifica ao consultar ambas as especificações, é o facto de a representação categorial ser muito mais extensa e menos compreensível. Tal como foi dito, este era um resultado esperado apesar de ambas as representações representarem a mesma informação.

A aplicação informática desenvolvida, bem como toda a sua documentação, estão disponíveis na *internet* no seguinte endereço:
http://www.math.ist.utl.pt/ fcouto/tfc/index.html

Em relação à linguagem e ferramentas utilizadas neste trabalho, pode-se dizer que foi necessário perder bastante tempo na sua aprendizagem. Por exemplo, foi dispendido muito tempo a descobrir como se podia utilizar ambos os

analisadores léxico e sintáctico no mesmo processo e a cooperarem entre si.
A razão de se ter demorado muito tempo a solucionar problemas deste tipo, reside no facto da documentação das ferramentas usadas neste trabalho ser pouco completa e com poucos exemplos práticos. Mas, apesar disso, o trabalho recuperou todo esse tempo na fase de implementação da aplicação informática, pois o código produzido ficou bem estruturado e legível. Tal como foi dito anteriormente esta linguagem adequa-se perfeitamente aos objectivos pretendidos.
Outro factor relevante reside no aspecto desta linguagem ser fortemente tipificada, o que influenciou o facto de terem existido muitos poucos erros semânticos para corrigir, ao contrário do que é usual com outros tipos de linguagem.

5.1 Contribuições

Durante o desenvolvimento deste projecto foram identificados novos algoritmos de tradução de especificações de espécie única de um componente para a sua respectiva representação categorial (secção 4.4).
Foram também identificadas formas eficazes de representação dos dois tipos de especificação usadas neste projecto (ver secção 4.1 e secção 4.3).

5.2 Trabalho Futuro

Tal como foi dito anteriormente os quantificadores no contexto deste trabalho só podem ser aplicados a um predicado. Para tentar evitar esta restrição foram feitos alguns estudos durante o desenvolvimento deste projecto sobre a utilização de quantificadores na forma normal. Nestes estudos descobriu-se algumas técnicas para passar essas fórmulas para uma representação categorial, mas o problema surgia quando um quantificador abrangia duas ou

mais cláusulas. Desta forma não se conseguiu obter nesta primeira fase do trabalho um método viável para a transformação de quantificadores aplicados a mais de uma cláusula. Assim numa próxima etapa fica em aberto a tarefa de solucionar este problema, através de um aperfeiçoamento das ideias encontradas, e da sua respectiva implementação.

A linguagem de especificação utilizada neste trabalho só permite utilizar predicados com aridade um, mas a maior parte dos comportamentos dos sistemas reais são praticamente impossíveis de representar só com esse tipo de predicados. Isto verifica-se, por exemplo, no facto de ser impossível representar funções através de predicados com aridade um. Consequentemente, será necessário numa fase posterior estender a linguagem de especificação usada neste projecto, por forma a incluir predicados com aridade superior a um.

Como foi referido, este trabalho final de curso tem uma evolução no âmbito da dissertação de mestrado, com o objectivo de produzir uma aplicação informática que identifica componentes reutilizáveis.

5.3 Ligação com Trabalho de Investigação

Este trabalho representa a parte inicial de uma investigação conducente a uma tese de mestrado e que vai ser submetido ao *FME 2001*[1] através de um artigo que será produzido até ao final de Agosto de 2000.

[1] *FME 2001* é o décimo simpósio organizado por *Formal Methods Europe*, que tem por tema o aumento da produtividade de *software* através de métodos formais.

Bibliografia

[BCN92] Batini, Ceri, and Navathe. *Conceptual Database Design*. Benjamin/Cummings, 1992.

[CB91] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24:61–70, 1991.

[CHJ93] P. S. Chen, R. Hennicker, and M. Jarke. On the retrival of reusable software components. *in 2nd International Workshop on Software Reusability*, pages 99–108, 1993.

[Cre98a] Rui Gustavo Crespo. Matching single-sort algebraic specifications for software reuse. *International Journal of Software an Knowledge Engineering*, 8(3):401–425, 1998.

[Cre98b] Rui Gustavo Crespo. *Processadores de linguagens da concepção à implementação* . IST Press, 1998.

[Dat94] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1994.

[GM92] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall, 1985.

[Hog90] C. J. Hogger. *Essentials of Logic Programming.* Oxford Press, 1990.

[JC95] J. J. Jeng and B. Cheng. Specification matching for software reuse. *in ACM Symposium on Software Reusability*, pages 97–105, 1995.

[KRT87] S. Katz, C. A. Richter, and K. S. The. Paris: A system for reusing partially interpreted schemas. *in 9th International Conference on Software Engineering*, pages 377–385, 1987.

[Lai76] Luis M. Laita. Un estudio de la lógica algebraica desde el punto de vista de la teoría de categorías. *Notre Dame Journal of Formal Logic*, 17(1):89–118, January 1976.

[Mar98] João Pavão Martins. *Knowledge Representation.* IST-UTL, 1998.

[MMM94] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. *in 16th Int. Conference on Software Engineering*, pages 91–100, 1994.

[MS92] N. A. Maiden and A. G. Sutcliffe. Explointing reusable specifications through analogy. *Communications of the ACM*, 35:52–64, 1992.

[PDF93] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE software*, 3:6–16, 1993.

[Pie93] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1993.

[PP93] D. E. Perry and S. S. Popovich. Inquire: Predicate-based use and reuse. *in 8th Knowledge Base and Software Engineering Conference*, pages 144–151, 1993.

[RB98] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory.* Prentice Hall, 1998.

[SC94] G. Spanoudakis and P. Constantopoulos. Similarity for analogical software reuse: A conceptual modeling approach. In *LNCS*, number 685 in 5th Conference on Advanced Information Systems Engineering, pages 483–503. Springer-Verlag, 1994.

[Spi88] J. M. Spivey. *Understanding Z: A Specification language and its formal semantics*. Cambridge University Press, 1988.

[SS93] Amílcar Sernadas and Cristina Sernadas. Teoria da programação, Janeiro 1993. IST-UTL.

[Wal91] R.F.C. Walters. *Categories and Computer Science*. Cambridge University Press, 1991.

[WT87] W.M.Tursky and T.S.E.Maibaun. *The Specification of Computer Programs*. Addison-Wesley, 1987.

[ZW95] A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, pages 146–170, 1995.

Apêndice A

EBNF das Representações Externas

A.1 Especificações de Espécie Única

<especificação> ::= Component <id>
Sort: { <lista de constantes> }
Variables: <lista de variáveis>
Invariant: <fórmulas>
<métodos>

<lista de constantes> ::= <id> { , <id> }

<lista de variáveis> ::= <id> { , <id> }

<métodos> ::= <métodos> <método> |
<método>

<método> ::= Method: <id>
Interface: <interfaces>
Requires: <pré-condições>
Ensures: <pós-condições>

<interfaces> ::= [<interface> { , <interface> }]

<interface> ::= ? <id> |
! <id>

<pré-condições> ::= <disjunção> { , <disjunção> }

<pós-condições> ::= <conjunção> → <fórmulas>

<fórmulas> ::= <fórmulas> { , <fórmula> }

<fórmula> ::= <conjunção> ⇒ <disjunção> |
<id predicado> → <id predicado>

<id predicado> ::= <id> |
<variável> =

<conjunção> ::= { <proposição> & <proposição> }

<disjunção> ::= { <proposição> | <proposição> }

<proposição> ::= > <id> : <predicado> |
< <id> : <predicado> |
<predicado> |
true |
false

<predicado> ::= <id> (<id>) |
<variável> = <id>

<variável> ::= <id> |
? <id> |
! <id>

<letra> ::= A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S
| T | U | V | X | Y | W | Z
| a | b | c | d | e | f | g | i | j
| k | l | m | n | o | p | q | r | s
| t | u | v | x | y | w | z

<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<id> ::= <letra> <letras ou dígitos>

<letras ou dígitos> ::= { <letra> | <dígito> }

A.2 Representações Categoriais

<rep. categorial> ::= Categorial Representation
Component <id>
<métodos>

<métodos> ::= <método> { <método> }

<método> ::= Method: <id>
<pré-categoria>
<pós-categorias>

<pré-categoria> ::= PreCategory : <categorias> |

<pós-categorias> ::= <pós-categoria> { <pós-categoria> }

<pós-categoria> ::= PosCategory : <C-objecto> ⇒ <categories>

<categorias> ::= <categoria (C,→)> <categoria (P,→)>

<categoria (C,→)> ::= Category C: <C-objectos> <setas C>

<C-objectos> ::= Objects : <C-objecto> { <C-objecto> }

<setas C> ::= Arrows : <seta C> { <seta C> }

<seta C> ::= <C-objecto> → <C-objecto>

<C-objecto> ::= <singular> |
<mínimo> |
<máximo>

<singular> ::= # { <proposição> }

<mínimo> ::= & { <proposições> }

<máximo> ::= | { <proposições> }

<proposições> ::= <proposição> { , <proposição> }

<proposição> ::= 0 |
1 |
<predicado> (<id>)

<categoria (P,→)> ::= Category P: <P-objectos> <setas P>

<P-objectos> ::= Objects : <P-objecto> { <P-objecto> }

<setas P> ::= Arrows : <seta P> { <seta P> }

<seta P> ::= <P-objecto> → <P-objecto>

<P-objecto> ::= <predicado> (<id>)

<predicado> ::= <id> |
= <variável>

<variável> ::= <id> |
? <id> |
! <id>

<letra> ::= A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S
| T | U | V | X | Y | W | Z
| a | b | c | d | e | f | g | i | j
| k | l | m | n | o | p | q | r | s
| t | u | v | x | y | w | z

<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<id> ::= <letra> <letras ou dígitos>

<letras ou dígitos> ::= { <letra> | <dígito> }

Apêndice B

Exemplos das Especificações Desenvolvidas

Apêndice C

Listagem dos Algoritmos Produzidos