UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Identificação de Componentes Reutilizáveis através de Especificações Formais

Francisco José Moreira Couto
(Licenciado)

Dissertação para a obtenção do grau de mestre
em
Engenharia Informática e Computadores

Orientador: Doutor Rui Gustavo Nunes Pereira Crespo

Júri
Presidente: Doutor Arlindo Manuel Limede de Oliveira
Vogais : Doutora Maria Antónia Bacelar da Costa Lopes
Doutor Rui Gustavo Nunes Pereira Crespo

Julho de 2001

Resumo

A reutilização de *software* só poderá ser posta em prática de uma forma eficaz se for aplicada de uma forma sistemática. Neste documento, propõe-se como objectivo principal o desenvolvimento de uma aplicação informática que automatiza o processo de reutilização de *software* através de especificações formais.

A funcionalidade dos componentes é expressa através de especificações algébricas de espécie única, que depois são traduzidas, automaticamente pela aplicação, em especificações categoriais.
A aplicação faz a selecção de componentes reutilizáveis através do emparelhamento isomorfo e composicional das especificações categoriais.

Existem publicações cientificas que abordam este tema, mas sempre a um nível conceptual muito teórico. Desta forma, neste documento, os conceitos apresentados nessas publicações são revistos e validados, por forma a possibilitar a sua implementação numa aplicação informática. Muitos deles são modificados, sendo alguns mesmo substituídos por novos conceitos desenvolvidos neste projecto. Extensões a estes conceitos são também produzidas, por forma a eliminar algumas das restrições da linguagem utilizada nas especificações propostas.

Os métodos desenvolvidos são também aqui objecto de uma análise de complexidade, que possibilita a demonstração da sua viabilidade quando aplicados a grandes quantidades de informação.

Palavras-chave: Especificação algébrica, especificação categorial, reutilização de *software*, emparelhamento isomorfo, emparelhamento composicional.

Abstract

Software reuse can only be implemented in an efficient way if it is practiced systematically. The purpose of this document is the development of a computer application that automates the process of software reuse through formal specifications.

The components functionality is expressed through single-sort algebraic specifications, which later are translated, automatically by the application, to categorical specifications.
The application makes the selection of reusable components through isomorphic and compositional matching of the categorical specifications.

There are scientific publications that approach this subject, but always in very theoretical way. In this document, the concepts presented in these publications are reviewed and validated, so that they can be implemented in an actual application. Many of them are modified, or even substituted by new concepts developed in this project. Extensions to these concepts are also produced, to eliminate some of the restrictions from the language used in the proposals specifications.

We also present a complexity analysis of the methods. In this way, it's possible to evaluate its viability when applied to large amounts of information.

Key-words: Algebraic specification, categorical specification, software reuse, isomorphic matching, compositional matching.

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 parte deste documento.

Saliento o excelente ambiente de trabalho que pude usufruir no Departamento de Engenharia Informática e no Departamento de Matemática do Instituto Superior Técnico, onde também me foi disponibilizado meios informáticos importantes para o desenvolvimento deste trabalho.

Para as várias pessoas que me responderam às perguntas que coloquei na *Internet*, sobre problemas que surgiram na utilização das ferramentas informáticas, quero deixar um especial agradecimento pela sua desinteressada ajuda.

Para os meus amigos e colegas quero deixar o meu apreço pelo seu apoio constante.

Por fim, fica o mais especial agradecimento para os meus pais, aos quais dedico todo este meu trabalho.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A reutilização de *software* é uma área do desenvolvimento de aplicações informáticas que tem recebido atenções crescentes, quer na comunidade científica, quer na área comercial. Neste documento, descreve-se o desenvolvimento de uma aplicação informática que automaticamente selecciona componentes reutilizáveis a partir de especificações formais.

1.1 O Problema

Informalmente, a *reutilização de software* é o aproveitamento de recursos, já produzidos, no processo de desenvolvimento de um novo produto de software. Os recursos são produtos criados durante o ciclo de desenvolvimento de um produto informático. Para além de apenas código, estes incluem componentes de *software*, testes de verificação, arquitecturas e documentação.
A reutilização não consiste apenas em copiar os recursos, consiste também na adaptação dos recursos para possibilitar a sua reutilização.
Neste trabalho, o conceito de recurso é limitado a componentes de *software*, abreviados por apenas componentes ao longo deste documento.

O processo de reutilização de *software*, normalmente, baseia-se em três etapas: classificação dos componentes de um sistema, que é responsável por identificar os vários componentes e especificar a sua funcionalidade; selecção dos componentes reutilizáveis, que tem como objectivo identificar componentes com o mesmo comportamento, para assim poderem ser reutilizados; adaptação dos componentes reutilizáveis, onde se poderá introduzir algumas alterações nos componentes seleccionados por forma a poderem ser reutilizados.

Existem várias formas de reutilização, focando-se nesta dissertação as seguintes:

reutilização *isomorfa*, que consiste na selecção de um componente reutilizável, que só por si cumpre os requisitos necessários a ser reutilizado sem necessidade de qualquer tipo de alterações; reutilização *composicional*, que envolve a construção de novos componentes através da composição de dois ou mais componentes reutilizáveis.

O problema, aqui tratado, consiste na identificação de um método de reutilização de software que possa ser aplicado de uma forma sistemática.

1.2 Importância do Problema

Para incrementar a eficácia na produção de *software*, há interesse em saber como e quando, num novo projecto, se pode reutilizar *software* já desenvolvido em projectos anteriores. De facto é muito frequente existir a necessidade de criar novos produtos com componentes comuns a produtos já implementados. É raro para uma empresa de média dimensão criar um produto que seja totalmente inovador, no sentido em que todo o código necessário à sua implementação seja completamente disjunto de todo o código produzido pela empresa até ao momento.
Apesar deste facto, na maior parte dos casos esse código não é reutilizado. Uma das causas deve-se ao facto das empresas não possuírem meios para detectar automaticamente os componentes reutilizáveis. Isto, porque normalmente as empresas têm um mau processo de desenvolvimento de projectos, e também porque ainda não estão disponíveis ferramentas que permitam facilmente o automatismo deste processo. Este é exactamente o objectivo deste trabalho, i.e., pretende-se obter uma aplicação informática que torne o processo de reutilização automático.

Se for bem implementada, a reutilização pode ajudar a alcançar as vantagens descritas em seguida. A descrição desta vantagens são também acompanhadas pelo caso particular dos resultados obtidos pela companhia *Hewlett-Packard* (HP) [Gri93].

- **Aumento da produtividade:** Após um investimento inicial, a reutilização permitirá aos projectos diminuir o seu custo de desenvolvimento e manutenção. A HP obteve aumentos de produtividade na ordem dos 6 a 40%, com projectos reutilizáveis.

- **Diminuição dos prazos de conclusão:** Por reutilizar *software* para diminuir o caminho crítico da entrega de um produto, as organizações ligadas à HP registaram uma redução dos prazos na ordem dos 12 a 42%.

- **Melhor qualidade:** O *software* que é utilizado múltiplas vezes irá ter menos defeitos que o código criado de novo. Os produtos da HP, derivados do processo de reutilização, tiveram uma melhoria de qualidade na ordem dos 24 a 76%.

- **Consistência entre os vários produtos:** No desenvolvimento de vários produtos, a reutilização de componentes irá facilitar a utilização dos produtos, pois os seus comportamentos irão ser consistentes.

- **Validação dos requisitos do utilizador:** Como o uso da reutilização facilita a prototipagem, os requisitos do utilizador podem ser mais facilmente verificados. Esta prototipagem permitirá também a detecção e resolução de defeitos mais cedo, evitando os custos inerentes a resolver estes defeitos em fases posteriores.

- **Redução do risco:** O risco no desenvolvimento de novo *software* é reduzido quando estão disponíveis componentes reutilizáveis que já abrangem a funcionalidade desejada e que podem ser facilmente integrados.

- **Melhor divisão das tarefas:** A reutilização permite que especialistas possam implementar e optimizar os recursos, que irão ser reutilizados por outros especialistas de áreas diferentes.

- **Maior rentabilidade do investimento:** O investimento feito na reutilização é depois recuperado no desenvolvimento de todos os produtos. Isto torna-o mais rentável em relação a qualquer outro investimento feito a pensar num só produto.

1.3 Formas de Resolução

Não existe uma solução para o problema da reutilização que seja suficientemente abrangente para ser viável a sua utilização por todas as entidades de produção de *software*. Neste momento, estão em estudo vários mecanismos de selecção de componentes reutilizáveis, cada um com as suas vantagens e desvantagens.
Um exemplo desses mecanismos consiste em seleccionar componentes reutilizáveis por analogia [SC94, MS92], explorando a semelhança entre os componentes. Outro exemplo, é o da selecção através de tabelas de atributos [PDF93] e métricas [CB91], que é vantajoso em termos do custo, escalabilidade e portabilidade.

1.4 Solução Adoptada

A solução seguida neste trabalho baseia-se no processo 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

sintácticamente mas terem o mesmo comportamento e, portanto, poderem serem equivalentes. 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 facilitar a produção de *software* a nível industrial.

Este projecto baseia-se nos conceitos propostos no artigo [Cre98a][1]. Desta forma, foram escolhidas as especificações algébricas de espécie única para a classificação dos componentes, e as especificações categoriais para a selecção dos componentes reutilizáveis. As razões que levaram a esta escolha foram as seguintes:

- A especificação algébrica é a especificação formal mais frequentemente usada no meio académico e empresarial na classificação de componentes, e que mais facilmente pode ser manipulada pelo ser humano. Isto é importante já que a classificação dos vários componentes de um sistema informático será uma tarefa do utilizador.
- A especificação categorial é a especificação que mais eficazmente pode ser usada em algoritmos de emparelhamento, devido à sua simplicidade sintáctica.
- Ambas as especificações têm uma linguagem independente e poderosa baseada em fundamentos matemáticos.

1.5 Aplicabilidade

A reutilização pode ser aplicada nos mais diversos ramos da produção de *software*. Todas as entidades que produzem *software* estão interessadas em utilizar processos de reutilização no desenvolvimento dos seus produtos.
Mas, a reutilização só é realmente eficaz quando usada sistematicamente, i.e., quando a reutilização de componentes é planeada com processos bem definidos, pessoal especializado, e com incentivos para a produção e uso de recursos reutilizáveis.

Uma das grandes dificuldades na implementação da reutilização reside na forma como são classificados os vários componentes de um sistema. Neste momento, uma grande parte da produção de *software* nem sequer é especificada, e quando o é, normalmente, são utilizadas especificações informais, que deixam um grau de subjectividade muito elevado, tornando impossível tratar essa informação automaticamente. As principais justificações para esta situação são as seguintes:

- A utilização de especificações formais tem custos muito elevados, e na maior parte dos casos as vantagens obtidas não cobrem os custos.

[1] Artigo de autoria do orientador científico deste projecto.

- Não existe uma educação que valorize a especificação dos projectos.
- Os custos têm de ser despendidos no início do projecto e as vantagens só são alcançadas na parte final do projecto, o que para a maior parte das empresas é um factor limitador, visto não terem capacidade de fazer investimentos a longo prazo.

Deste modo, tem-se aqui um típico problema de engenharia, ou seja, para tornar a utilização de especificações formais mais atractiva têm-se duas hipóteses:

- Diminuir os custos.
- Aumentar as vantagens.

No processo de desenvolvimento de *software*, a classificação dos componentes através de especificações formais acarreta custos bastante elevados. A principal razão para esses custos derivam do facto de serem necessários recursos humanos especializados para utilizar este tipo de metodologia. Desta forma, uma solução para diminuir os custos é investir cada vez mais na formação de pessoal especializado neste tipo de metodologia. Mas para que, por parte das empresas, exista esse investimento há que valorizar a importância das vantagens desta metodologia.

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.
No caso de serem utilizadas especificações formais é possível construir a prova da correcção do sistema, que pode ser útil em sistemas críticos, i.e., sistemas que não podem falhar. Para além disso, também é possível, em alguns casos, implementar animações (protótipos) através das especificações, permitindo a sua validação.

O principal objectivo deste trabalho é valorizar ainda mais a utilização de especificações formais. Isso, é alcançado se for possível encontrar uma aplicação que identifique componentes reutilizáveis automaticamente a partir das respectivas especificações formais. Assim, será fácil para as entidades de produção de *software* compreenderem que o balanço entre os custos e vantagens da utilização de especificações formais irá ser muito mais favorável às vantagens.

Em suma, neste momento a solução seguida neste projecto só pode ser aplicada a um número muito limitado de casos. Mas, espera-se que no futuro esse número aumente conforme se vai alcançando resultados relevantes neste domínio.

1.6 Objectivos

O principal resultado deste projecto é a implementação de uma aplicação informática que automatize o processo de reutilização de componentes através das suas especificações formais.

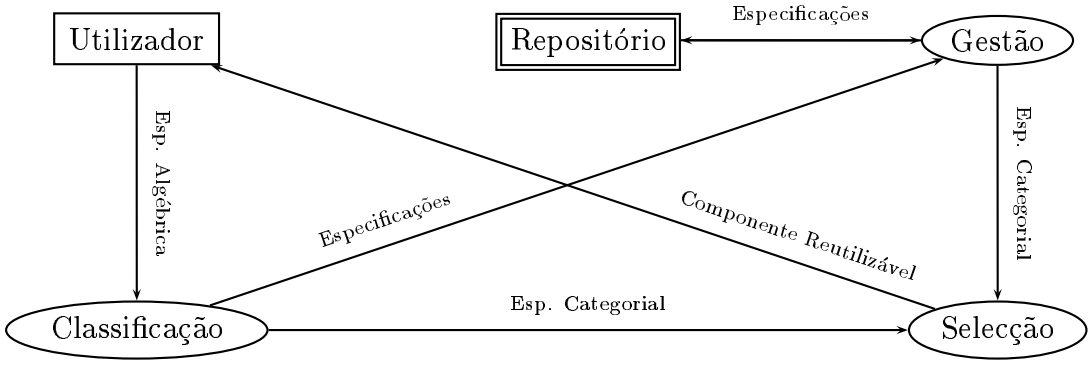

Figura 1.1: Arquitectura da aplicação informática proposta

Tal como demonstra a arquitectura da aplicação descrita na figura 1.1, o processo de reutilização é composto por três sub-processos:

- *Classificação*, que irá tratar da tradução das especificações algébricas em especificações categoriais semanticamente equivalentes.
- *Selecção*, que será responsável pela identificação e possível adaptação de componentes reutilizáveis.
- *Gestão*, que terá a seu cargo a manutenção do repositório das especificações dos componentes.

A descrição da arquitectura dos componentes será uma tarefa atribuída ao utilizador, i.e., o utilizador terá de decompor o sistema nos seus vários componentes e representar o comportamento funcional de cada componente através de especificações algébricas.

Os objectivos deste projecto serão divididos por três fases: Classificação, Selecção e Extensões. As secções seguintes apresentam em traços gerais os principais objectivos a alcançar em cada uma destas fases.

1.6.1 Classificação

Nesta fase, os objectivos serão identificar e implementar eficazmente os mecanismos de tradução das especificações algébricas em especificações categoriais.

O primeiro passo irá consistir na revisão e no estudo de viabilidade dos conceitos teóricos, apresentados nas publicações [Cre98a, Lai76, Gol86], nos aspectos que seguidamente se referem:

- Encontrar as alterações necessárias efectuar às metodologias propostas, por forma a ser possível a utilização das fórmulas em forma normal.
- Estudar a estrutura proposta para as especificações algébricas, e identificar modificações benéficas para a sua implementação.
- Verificar a semântica implícita numa especificação algébrica, por forma a que esta não se perca nas especificações categoriais.
- Estudar a viabilidade da metodologia proposta, i.e., verificar se todos os passos podem ser implementados através de um processo computacional.

O segundo passo consistirá na identificação de representações computacionais eficazes para as especificações algébricas e categoriais. Para cada uma delas serão criados dois tipos de representação, uma para armazenar as especificações em ficheiro (rep. externa) e outra para as manipular em memória (rep. interna). Neste passo serão também desenvolvidos e implementados algoritmos para transformar uma representação externa em interna, e vice versa.

No terceiro passo serão definidos os algoritmos que traduzem os conceitos teóricos, desenvolvidos nos passos anteriores, para um conjunto de regras e instruções que possam ser executadas mecanicamente.

No quarto passo será feita uma análise final da complexidade dos algoritmos desenvolvidos, onde serão identificados os principais factores que influenciam essa complexidade.

O quinto passo irá iniciar-se com a escolha da linguagem de programação que mais se adequa ao tipo de algoritmos identificados nos passos anteriores. Em seguida, será feito um estudo sobre as ferramentas de desenvolvimento disponíveis para a linguagem escolhida, por forma a implementar os algoritmos com as ferramentas mais adequadas.
Por fim, será implementado e testado o processo *Classificação*, em conformidade com o que foi desenvolvido nos passos anteriores.

Por ultimo, o sexto passo terá como principal objectivo o desenvolvimento e implementação do processo *Gestão*, respeitando os conceitos já existentes sobre sistemas de bases de dados [Dat94, BCN92]. Este processo terá de ser simples e eficiente, pois as suas funções vão ser utilizadas com muita frequência pelo processo *Selecção*.

1.6.2 Selecção

O principais objectivos desta fase irão consistir na identificação e implementação de mecanismos de selecção de componentes reutilizáveis, e também no seu estudo de viabilidade.
Esta fase será dividida em duas etapas, que seguidamente se descrevem:

Emparelhamento Isomorfo

Esta primeira etapa consistirá na identificação e implementação do emparelhamento isomorfo entre duas especificações categoriais.

O primeiro passo irá consistir na revisão e no estudo de viabilidade dos conceitos teóricos, propostos no artigo [Cre98a], com maior incidência no seguintes pontos:

- Adaptar os métodos propostos aos novos resultados alcançados na fase da *Classificação*.
- Estudar a viabilidade do método proposto, i.e., verificar se todos os passos podem implementados através de um processo computacional.
- Rever a complexidade do método proposto, por forma a desenvolver técnicas de emparelhamento mais eficazes.
- Alterar e completar os algoritmos de selecção, em consonância com os resultados aqui obtidos.

No segundo passo será feita uma análise final da complexidade dos algoritmos desenvolvidos, onde serão identificados os principais factores que influenciam essa complexidade.

O terceiro passo irá consistir na implementação dos algoritmos desenvolvidos, que são testados através de especificações criadas de propósito para esse efeito.

Emparelhamento Composicional

Esta segunda etapa terá como objectivos a identificação e implementação de um processo de emparelhamento composicional entre uma especificação categorial e uma sequência de especificações categoriais.

Os objectivos desta etapa irão ser divididos em três passos. A divisão será feita de forma análoga à apresentada para o emparelhamento isomorfo.

1.6.3 Extensões

As especificações algébricas, consideradas na secção 1.6.1, contêm algumas restrições importantes, tais como:

- Os quantificadores são aplicados a um só predicado.
- Os conjuntos de suporte são de dimensão finita.
- Os predicados só podem conter um argumento.
- As pré-condições e os antecedentes das pós-condições não podem ser expressas através de implicações.

Como o objectivo deste trabalho é implementar uma metodologia de reutilização viável, então nesta fase terão de ser identificadas formas de resolução destas limitações.
Quando finalmente estas limitações forem todas ultrapassadas, aplicar-se-á a ferramenta desenvolvida a um caso prático, i.e., será criado um conjunto de componentes reutilizáveis sobre uma determinada área, como por exemplo, a manipulação de grafos. Isto, irá permitir obter resultados mais fidedignos sobre as potencialidades e limitações da ferramenta desenvolvida.

1.7 Contribuições

A hipótese de investigação desta dissertação consiste na possibilidade de se automatizar a identificação de componentes reutilizáveis através do uso de especificações formais na descrição da funcionalidade dos todos os componentes.

As principais contribuições resultantes da conclusão dos objectivos apresentados na secção 1.6 irão consistir nos seguintes aspectos:

- Validação e revisão de resultados descritos na bibliografia consultada.
- Identificação de novos e mais eficientes métodos e algoritmos de classificação e selecção dos componentes reutilizáveis.
- Estudo da complexidade dos métodos adoptados.
- Implementação de uma ferramenta que automatiza o processo de classificação e selecção dos componentes reutilizáveis.
- Extensão da linguagem proposta para as especificações algébricas.

- Escrita de publicações científicas que transmitam os resultados alcançados neste projecto.

Capítulo 2

Conceitos Básicos

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

2.1 Lógica de Predicados de Primeira Ordem

Nesta secção são apresentados os constituintes da lógica de predicados de primeira ordem utilizados ao longo deste documento.
Para se obter mais informação sobre a lógica de predicados de primeira ordem sugere-se a consulta de [BM77].

Definição 2.1 (Alfabeto) *A linguagem da lógica de predicados de primeira 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 (Aridade) *Cada predicado tem um número fixo de argumentos, designado por* aridade.

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 2.3 (Termo) *Um* termo *pode ser uma variável ou uma constante.*

Definição 2.4 (Proposição) *Uma* proposição *é um tuplo de termos aplicados como argumentos de um predicado. O número de elementos do tuplo é igual à aridade do predicado.*

Definição 2.5 (Fórmula bem definida) *De seguida apresentam-se as regras para a 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 *é uma variável individual, então* $\forall v\, \phi$ *e* $\exists v\, \phi$ *também são fórmulas.*

Para além da sua linguagem, uma lógica é constituída por duas componentes: o sistema dedutivo e a semântica. O sistema dedutivo permite manipular simbolicamente as fórmulas, enquanto que a semântica permite verificar a sua validade.

2.1.1 Sistema Dedutivo

Um sistema dedutivo é formado a partir de um conjunto de axiomas, e de um conjunto de regras de inferência.

Definição 2.6 (Axiomas) *Os* axiomas *são fórmulas que são aceites sem necessidade de prova.*

Definição 2.7 (Regras de Inferência) *As* regras de inferência *são regras que definem como produzir novas fórmulas a partir de fórmulas já existentes.*

Exemplo 2.2 *Uma regra de inferência muito utilizada, de seu nome* Modus Ponens, *define o seguinte: caso se tenha uma fórmula α e outra fórmula da forma $\alpha \Rightarrow \beta$, então podemos obter β.*

Definição 2.8 (Dedução) *Diz-se que α é* dedutível *de um conjunto de fórmulas Δ ($\Delta \vdash \alpha$) sse existe uma sequência de regras inferência que aplicadas a Δ e aos axiomas produzem α.*

Definição 2.9 (Teoria) *Dado um conjunto de fórmulas Δ, ao conjunto de todas as fórmulas deduzíveis de Δ, chama-se* conjunto de teoremas *de Δ, ou* teoria gerada *por Δ, e escreve-se $Th(\Delta)$. Formalmente:*

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

2.1.2 Semântica

A semântica especifica as condições sobre as quais se decide se as fórmulas são verdadeiras ou falsas. A semântica é baseada na noção de interpretação, um mapeamento entre as entidades da linguagem e as entidades da conceptualização (entidades do mundo no qual estamos interessados em representar).

Definição 2.10 (Interpretação) *Uma* interpretação *é uma associação dos objectos da conceptualização com os objectos da linguagem.*

Definição 2.11 (Consequência Semântica) *Dado um conjunto de fórmulas Δ e uma fórmula α, se não existe nenhuma interpretação tal que faça todas as fórmulas de Δ verdadeiras e α falsa, então α é* consequência semântica *de Δ, e escreve-se*

$$\Delta \models \alpha$$

Definição 2.12 (Equivalência Semântica) *Duas fórmulas são* semanticamente equivalentes *sse para qualquer interpretação se obtém o mesmo valor lógico para ambas as fórmulas.*

2.2 Formas Normais

As formas normais são formas sob as quais as fórmulas são substituídas por outras semanticamente equivalentes, com uma estrutura mais regular, tornando a sua manipulação mais eficiente.
Informação mais detalhada acerca das formas normais pode ser encontrada no livro [Hog90].

Forma Clausal

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

Definição 2.13 (Literal) *Um* literal positivo *é uma proposição. Um* literal negativo *é a negação de uma proposição. Um* literal *é um literal positivo ou negativo.*

Definição 2.14 (Cláusula) *Uma* cláusula *é um conjunto de literais que denota a sua disjunção.*

Definição 2.15 (Afirmação Clausal) *Uma* afirmação clausal *é um conjunto de cláusulas que denota a sua conjunção.*

Uma cláusula pode também ser vista como uma implicação, i.e., uma cláusula pode ser representada na forma:

$$Antecedente \Rightarrow Consequente$$

em que:

- O antecedente é um conjunto de proposições que denota a conjunção dos literais negativos.
- O consequente é um conjunto de proposições que denota a disjunção dos literais positivos.

Neste trabalho as cláusulas são sempre representadas sob a forma de implicação.

2.2.1 Forma Normal Prenex

A forma normal prenex é um caso particular de uma forma normal, que permite a representação de quantificadores.

Definição 2.16 (Cláusula Quantificada) *Uma* cláusula quantificada *é uma fórmula com a seguinte estrutura:*

$$\Delta_1 x_1 \ldots \Delta_n x_n [w]$$

em que:

- $\Delta_1 \ldots \Delta_n$ *são quantificadores;*
- *as variáveis* $x_1, \ldots, x_n$ *são todas distintas;*
- w *é uma fórmula sem quantificadores, logo pode ser representada através de uma afirmação clausal.*

Definição 2.17 (Afirmação Prenex) *Uma* afirmação prenex *é um conjunto de cláusulas quantificadas que denota a sua conjunção.*

Exemplo 2.3 *Considere-se a seguinte fórmula bem definida:*

$$(\forall x\, P_1(x) \vee P_2(x)) \Rightarrow (\exists y\, P_3(y) \wedge P_4(y))$$

A afirmação prenex equivalente à fórmula anterior é a seguinte:

$$\begin{gathered}\{\forall x \exists y\, P_1(x) \Rightarrow P_3(y),\\ \forall x \exists y\, P_1(x) \Rightarrow P_4(y),\\ \forall x \exists y\, P_2(x) \Rightarrow P_3(y),\\ \forall x \exists y\, P_2(x) \Rightarrow P_4(y)\}\end{gathered}$$

Para se melhor entender a relação entre uma fórmula a sua representação em forma normal é apresentado no exemplo 2.3 o resultado da transformação de uma fórmula na sua afirmação prenex.
Uma das características das formas normais, aqui apresentadas, consiste no facto de que para toda a fórmula da lógica de primeira ordem existe uma representação sob a forma normal logicamente equivalente. São conhecidos na literatura algoritmos de transformação de fórmulas da lógica de primeira ordem numa forma normal [Hog90].

2.3 Especificações

As especificações indicam o comportamento dos componentes que constituem um sistema. Desta forma, não é necessário conhecer o código que os implementa para saber qual a sua funcionalidade. 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].

2.3.1 Classes de Especificações

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.

A classe de especificações utilizada neste trabalho é formal. Uma especificação formal não é mais que uma teoria (definição 2.9) formada a partir de um conjunto de fórmulas e de um sistema dedutivo.

Um exemplo de uma especificação formal com estas características é a especificação Z [Spi88]. Esta é baseada em modelos que utilizam a teoria de conjuntos para modelar dados, e a lógica de predicados de primeira ordem para descrever operações sobre os dados. Esta, também, é uma linguagem tipificada, i.e., requer que a cada variável seja associado um tipo.

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

Neste projecto são utilizadas duas linguagens formais de especificação:

- Especificações Algébricas de Espécie Única.
- Especificações Categoriais

2.4 Especificações Algébricas de Espécie Única

As especificações algébricas[1] são definidas através de uma determinada estrutura de fórmulas da lógica de predicados de primeira ordem. O facto de ser de espécie única indica que as variáveis utilizadas nas fórmulas só podem tomar valores pertencentes a um determinado *conjunto suporte*.
Um exemplo do uso de especificações algébricas na representação de componentes reutilizáveis pode ser visto em [Wir88].

De seguida descrevem-se as especificações algébricas usadas neste projecto.

2.4.1 Restrições nas Fórmulas

Por forma a tornar a manipulação de especificações algébricas mais eficiente, neste trabalho representam-se todas as suas fórmulas sob a forma clausal.

Nas especificações algébricas propostas são impostas as seguintes restrições nas suas fórmulas:

- Os quantificadores são aplicados a um só predicado.
- Os predicados só podem conter um argumento.

Desta forma, é necessário alterar o conceito de proposição, dando lugar ao conceito de pseudo-proposição.

Definição 2.18 (Pseudo-Proposição) *O conceito de* pseudo-proposição *é definido da seguinte forma:*

- *A constante* true *é uma pseudo-proposição.*
- *A constante* false *é uma pseudo-proposição.*
- *Seja P um predicado e c uma constante, então $P(c)$ é uma pseudo-proposição.*
- *Seja P um predicado e v uma variável, então $\forall v\, P(v)$ é uma pseudo-proposição.*
- *Seja P um predicado e v uma variável, então $\exists v\, P(v)$ é uma pseudo-proposição.*

[1] Por uma questão de legibilidade, algumas vezes neste documento abrevia-se *especificações algébricas de espécie única* por apenas *especificações algébricas*.

A razão que levou a ser introduzidas as constantes *true* e *false* deriva do facto de estas serem consideradas como os elementos neutros respectivamente da conjunção e da disjunção. Por essa razão, é necessário por vezes introduzir essas constantes para representar uma afirmação clausal, tal como demonstra o exemplo 2.4.

Exemplo 2.4 $[p_1 \wedge \neg p_2] \Leftrightarrow [true \Rightarrow p_1 \ \wedge \ p_2 \Rightarrow false]$

O facto de os predicados nas pseudo-proposições só poderem ser aplicados a constantes é justificado na secção 3.2.1.

Sempre que não existir informação em contrário, neste trabalho as afirmações clausais são constituídas por pseudo-proposições no lugar de proposições.

2.4.2 Estrutura das Especificações

As especificações algébricas de um sistema de *software* são divididas pelos vários componentes que o constituem. Neste trabalho, uma especificação algébrica de um componente foi definida como tendo a seguinte estrutura [2]:

Componente: identificador
Espécie: membros_do_conjunto
Variáveis: (nome_da_variável)+
Invariante: condição
(**Método**: ...)+

O conjunto de suporte é definido no campo *Espécie*, onde são indicadas todas as constantes que pertencem a este conjunto. O conjunto suporte tem de ser um conjunto enumerado e finito.

O nome das variáveis de estado são definidas no campo *Variáveis*. Os valores que estas variáveis poderão tomar são definidos pelo conjunto suporte.

A condição do *Invariante* é uma afirmação clausal, que define as propriedades que a implementação do componente tem que verificar em todos os estados da sua execução.

Um componente pode ser constituído por um ou mais métodos. Cada um desses métodos obedece à seguinte estrutura:

Método: identificador
Interface: (nome_dos_parâmetros)*
Requer: pré-condição
Garante: (pós-condição)+

[2] A estrutura é definida através de expressões algébricas [Cre98b]

Os parâmetros do método são definidos no campo *Interface*. Os nomes dos parâmetros são nomes de variáveis antecedidas dos símbolos ? e !, caso seja, respectivamente, um parâmetro de entrada ou saída.

A pré-condição do campo *Requer* é constituída por um conjunto de disjunções de pseudo-proposições denotando a sua conjunção.
A pré-condição indica as propriedades que que devem ser satisfeitas para que o método possa ser executado. Desta forma, cada disjunção, normalmente, é utilizada para especificar os valores que uma determinada variável poderá ter, de modo a ser possível executar o método.
Como a pré-condição está relacionado com as propriedades iniciais do método, este só pode conter variáveis de estado e de entrada.

Cada pós-condição do campo *Garante* é da forma: $antecedente \rightarrow consequente$.
O antecedente é uma conjunção de pseudo-proposições, e o consequente é uma afirmação clausal.
Cada pós-condição do método garante que se o seu antecedente estiver satisfeito no inicio da execução do método, então o seu consequente será satisfeito no final da execução do método.
Como o antecedente está relacionado com as propriedades iniciais do método, este só pode conter variáveis de estado e de entrada.
Por sua vez, o consequente está relacionado com as propriedades finais do método, logo só pode conter variáveis de estado e de saída.

A especificação algébrica tem que ser determinista, i.e., em cada método os vários antecedentes das pós-condições têm que ser disjuntos entre si.
A especificação algébrica 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 linguagem de especificação não existe a garantia que se uma variável não for referida na parte do consequente, o seu valor não será alterado. Se quisermos garantir isso temos que indicar explicitamente no consequente o valor dessa variável.

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

Exemplo 2.5 *Um exemplo de uma especificação algébrica de um componente com o comportamento de um contador é apresentada na figura 2.1.*
A variável `Estado` *indica o valor que o contador têm registado em cada momento da sua execução.*
O método `Incrementa` *incrementa uma unidade ao valor do contador, desde que ele*

```
Componente:  Contador
   Espécie:  {1, 2, 3}
 Variáveis:  Estado
Invariante:  false ⇒ true

    Método:  Incrementa
 Interface:  !EstadoFinal
    Requer:  Estado=1 ∨ Estado=2
   Garante:  Estado=1 →  true ⇒ Estado=2, true ⇒ !EstadoFinal=2;
             Estado=2 →  true ⇒ Estado=3, true ⇒ !EstadoFinal=3;

    Método:  Inicializa
 Interface:
    Requer:  true ⇒ false
   Garante:  true →  true ⇒ Estado=1;
```

Figura 2.1: Especificação algébrica de um contador

não exceda o valor 3. Na variável `!EstadoFinal` *será devolvido o valor do contador após a execução do método.*
O método `Inicializa` *atribui o valor 1 ao contador.*

Para se ter uma melhor ideia do uso das especificações algébricas, é apresentada a especificação de um contador no exemplo 2.5. Este é um exemplo muito simples de como podem ser utilizadas as especificações algébricas, aqui definidas, para representar o comportamento de um componente.

2.5 Especificações Categoriais

As especificações categoriais baseiam-se nos conceitos da teoria das categorias. A teoria das categorias é a álgebra das funções, cuja principal operação é a composição de funções.
Abordagens mais detalhadas sobre a teoria das categorias podem ser consultadas em [Wal91, Pie93, SS93].

Nesta dissertação, uma *especificação categorial* é constituída por um conjunto de categorias interligadas entre si.

Definição 2.19 (Categoria) *Uma* categoria $\mathcal{C}$ *consiste num conjunto de* objectos $obj(\mathcal{C})$ *e num conjunto de* setas $set(\mathcal{C})$. *Os objectos são denotados por* $A, B, C, \ldots$ *e as setas por* $f, g, h, \ldots$

As propriedades que devem ser verificadas são as seguintes:

- *Cada seta tem uma* origem *e um* destino *em* $obj(\mathcal{C})$. *Quando a origem de* f *é* A *e o destino de* f *é* B *escreve-se* $f : A \to B$.
- *O conjunto* $set_{A,B}(\mathcal{C})$ *representa o conjunto de todas as setas de* $\mathcal{C}$ *com origem no objecto* A *e destino no objecto* B.
- *Dado quaisquer duas setas* $f : A \to B$, $g : B \to C$, *existe uma seta designada* composição, $g \circ f : A \to C$
- *Dado qualquer objecto* A, *existe uma seta designada* identidade, $1_A : A \to A$
- *A composição de setas 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 composição *Se* $f : A \to B$, $g : B \to C$ *e* $h : C \to D$ *então*

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

Exemplo 2.6 *Seja uma categoria* $\mathcal{C}$ *definida da seguinte forma:*

- $obj(\mathcal{C}) = \{A, B\}$
- $set(\mathcal{C}) = \{sucessor, predecessor, 1_A, 1_B\}$

em que:

$$A = \{1, 2\} \quad B = \{2, 3\}$$

$$\begin{aligned} sucessor : A &\to B \\ 1 &\mapsto 2 \\ 2 &\mapsto 3 \end{aligned}$$

$$\begin{aligned} predecessor : B &\to A \\ 2 &\mapsto 1 \\ 3 &\mapsto 2 \end{aligned}$$

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 2.6 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, dentro do domínio definido pelos objectos.

Definição 2.20 (Categoria Magra) *Uma* categoria magra *é uma categoria onde existe no máximo uma seta entre cada par de objectos, i.e., uma categoria $\mathcal{C}$ é magra sse, para cada par de objectos $A, B \in obj(\mathcal{C})$, o conjunto $set_{A,B}(\mathcal{C})$ têm no máximo um só elemento.*
Numa categoria magra $\mathcal{C}$, uma seta com origem e destino, respectivamente, nos objectos $A, B \in obj(\mathcal{C})$ denota-se apenas por $A \to_{\mathcal{C}} B$, visto ser a única com essa origem e destino.

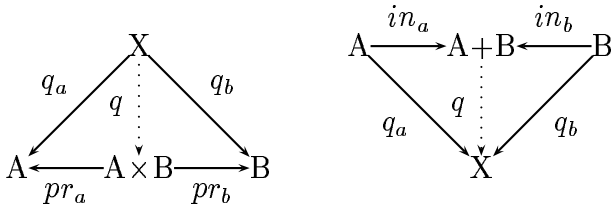

Figura 2.2: Diagrama do Produto(esquerda) e do Co-Produto(direita)

Definição 2.21 (Produto (Co-Produto) Binário) *Sejam $\mathcal{C}$ uma categoria e $A, B \in obj(\mathcal{C})$. Um* produto *(*co-produto*) de A e B é um triplo composto por um objecto $A \times B$ $(A+B)$ de $obj(\mathcal{C})$ e duas setas $pr_a : A \times B \to A$, $pr_b : A \times B \to B$ $(in_a : A \to A+B$, $in_b : B \to A+B)$ de $set(\mathcal{C})$, tais que:*
Quaisquer que sejam o objecto $X \in obj(\mathcal{C})$ e as setas $q_a : X \to A$, $q_b : X \to B$ $(q_a : A \to X$, $q_b : B \to X)$ de $set(\mathcal{C})$ existe um única seta $q : X \to A \times B$ $(q : A+B \to X)$ em $set(\mathcal{C})$ tal que $pr_a \circ q = q_a$ e $pr_b \circ q = q_b$ $(q \circ in_a = q_a$ e $q \circ in_b = q_b)$. Isto é, tal que o respectivo diagrama da figura 2.2 comuta.

Definição 2.22 (Objecto Terminal (Inicial)) *Dada uma categoria $\mathcal{C}$, um objecto 1 (0) é* terminal *(*inicial*) sse por cada objecto* A$\in obj(\mathcal{C})$ *existe uma e uma só seta* A$\to$1 *(*0$\to$A*).*

Definição 2.23 (Functor) *Sejam $\mathcal{C}_1$ e $\mathcal{C}_2$ duas categorias, então um* functor *de $\mathcal{C}_1$ para $\mathcal{C}_2$, $F : \mathcal{C}_1 \to \mathcal{C}_2$, consiste nas funções $F_{obj} : obj(\mathcal{C}_1) \to obj(\mathcal{C}_2)$ e, por cada par de objectos $A_1, A_2 \in obj(\mathcal{C}_1)$, $F_{A_1,A_2} : set_{A_1,A_2}(\mathcal{C}) \to set_{F_{obj}(A_1),F_{obj}(A_2)}(\mathcal{C})$, tal que:*

- $F_{A,A}(1_A) = 1_{F_{obj}(A)}$
- *Para quaisquer duas setas* $f : A \to B$ *e* $g : B \to C$, *tem-se* $F_{A,C}(g \circ f) = F_{B,C}(g) \circ F_{A,B}(f)$

2.6 Cálculo Combinatório

Nesta secção são apresentados os conceitos do cálculo combinatório utilizados ao longo deste documento.
Para se obter mais informação sobre cálculo combinatório sugere-se a consulta de [MR99].

Definição 2.24 (Arranjo com Repetição) *Chama-se* arranjo com repetição *de tamanho k a uma qualquer sequência formada por k elementos de um conjunto dado. Dado um conjunto C então um arranjo com repetição de tamanho k de C denota-se por* $\langle\langle c_1, \ldots, c_k \rangle\rangle \in C$.

Definição 2.25 ($\overline{\mathcal{A}}_p^n$) *Dado um conjunto com n elementos, designa-se por* $\overline{\mathcal{A}}_p^n$ *o número total de arranjos com repetição de tamanho p que é possível formar com p elementos escolhidos entre os n dados.*

Postulado 2.1 $\overline{\mathcal{A}}_p^n = n^p$.

Definição 2.26 (Arranjo) *Chama-se* arranjo *de tamanho k a uma qualquer sequência de k elementos todos distintos, escolhidos de entre os elementos de um conjunto dado. Dado um conjunto C então um arranjo de tamanho k de C denota-se por* $\langle c_1, \ldots, c_k \rangle \in C$.

Definição 2.27 ($\mathcal{A}_p^n$) *Dado um conjunto com n elementos, designa-se por* $\mathcal{A}_p^n$ *o número total de arranjos de tamanho p que é possível formar com p elementos escolhidos entre os n dados.*

Postulado 2.2 $\mathcal{A}_p^n = \frac{n!}{(n-p)!}$

Exemplo 2.7 *Supondo que apenas quatro atletas* $\{Carlos, José, Pedro, João\}$ *se apresentam no final de uma prova olímpica. O número de formas distintas de como pode vir a ser feita a distribuição das três medalhas* $\{ouro, prata, bronze\}$*, pode ser obtido calculando* $\mathcal{A}_3^4 = 24$*. Este resultado confirma-se, porque* 24 *é o tamanho do conjunto que representa todos os possíveis arranjos entre os dois conjuntos dados. Esse conjunto têm a seguinte forma:*

{ {*(ouro, Carlos), (prata, José), (bronze, Pedro)*},
{*(ouro, Carlos), (prata, José), (bronze, João)*},
{*(ouro, Carlos), (prata, Pedro), (bronze, José)*},
{*(ouro, Carlos), (prata, Pedro), (bronze, João)*},
{*(ouro, Carlos), (prata, João), (bronze, Pedro)*},
{*(ouro, Carlos), (prata, João), (bronze, José)*},

{*(ouro, José), (prata, Pedro), (bronze, João)*},
{*(ouro, José), (prata, Pedro), (bronze, Carlos)*},
{*(ouro, José), (prata, João), (bronze, Pedro)*},
{*(ouro, José), (prata, João), (bronze, Carlos)*},
{*(ouro, José), (prata, Carlos), (bronze, João)*},
{*(ouro, José), (prata, Carlos), (bronze, Pedro)*},

{*(ouro, Pedro), (prata, João), (bronze, Carlos)*},
{*(ouro, Pedro), (prata, João), (bronze, José)*},
{*(ouro, Pedro), (prata, Carlos), (bronze, João)*},
{*(ouro, Pedro), (prata, Carlos), (bronze, José)*},
{*(ouro, Pedro), (prata, José), (bronze, Carlos)*},
{*(ouro, Pedro), (prata, José), (bronze, João)*},

{*(ouro, João), (prata, Carlos), (bronze, José)*},
{*(ouro, João), (prata, Carlos), (bronze, Pedro)*},
{*(ouro, João), (prata, José), (bronze, Carlos)*},
{*(ouro, João), (prata, José), (bronze, Pedro)*},
{*(ouro, João), (prata, Pedro), (bronze, José)*},
{*(ouro, João), (prata, Pedro), (bronze, Carlos)*} }

O postulado 2.2 pode ser aplicado quando se quer conhecer o número de combinações entre dois conjuntos, tal como é apresentado no exemplo 2.7.

Capítulo 3

Classificação

Nesta dissertação, a classificação de componentes de *software* consiste na identificação de uma especificação categorial semanticamente equivalente à especificação algébrica do componente introduzido pelo utilizador.

3.1 Revisão da Teoria Proposta

A revisão da teoria de classificação de componentes propostas na literatura resultou no desenvolvimento de um novo método de classificação, do qual se destacam as seguintes características:

- Os quantificadores são representados através de uma nova metodologia, que tem como ideia base a substituição das proposições por asserções. Uma das vantagens desta abordagem consiste no facto de esta estar mais perto de uma solução geral, i.e. permite mais facilmente eliminar a restrição da aplicação dos quantificadores a um único predicado. Outra vantagem, consiste na sua melhor eficácia no emparelhamento, pois não necessita de utilizar uma categoria específica para representação dos predicados.

- A representação das fórmulas em forma normal, por uma razão de eficiência computacional, levou a que se tivesse que alterar alguns aspectos nos métodos propostos.

- Na teoria proposta não estava definida qual a semântica inerente aos predicados utilizados nas especificações algébricas. Neste projecto, chegou-se à conclusão que o significado dos predicados tinham de ser preservados nas especificações categoriais, caso contrário existiria um espaço para ambiguidade nas especificações desenvolvidas.

- As categorias geradas não contêm toda a informação representada de uma forma explícita. Assim, torna-se necessário o desenvolvimento de um mecanismo de inferência, que aplica as regras que nesta fase foram definidas, para se manipular a informação implícita. O motivo para manter alguma informação implícita reside no facto de que guardar toda informação explicitamente é muito ineficaz em termos de complexidade computacional.

- O conjunto de variáveis consideradas na classificação de um método são todas as que são realmente usadas e não as que poderiam ser usadas. Isto permite um emparelhamento de componentes mais eficiente, pois só será necessário emparelhar as variáveis realmente usadas.

3.2 Método de Classificação

Nesta secção vamos descrever detalhadamente o método de classificação desenvolvido neste trabalho.

3.2.1 Semântica dos Predicados

Primeiro vai-se listar as restrições impostas aos predicados.

Aridade do Predicado Igualdade

O predicado referente à igualdade tem aridade dois, mas neste trabalho só temos predicados com aridade um. Os métodos propostos na literatura não indicavam a forma de como se deve tratar este caso. Desta forma desenvolveu-se neste trabalho um método para representar o predicado igualdade. Esse método transforma cada proposição da forma $=(Variável, constante)$ na proposição $Variável = (constante)$. Ou seja, existe um predicado igualdade distinto por cada variável de estado e de interface. Todos estes predicados têm o mesmo significado, que é o de verificar se o valor da variável é igual à da constante.

Em consequência de considerar um predicado igualdade por cada variável, teve de se proibir a aplicação de predicados a variáveis de estado e de interface. Isto, porque no caso de se permitir ter $Var_1 = Var_2$, que corresponde a ter $Var_1 = (Var_2)$ ou $Var_2 = (Var_1)$, teria-se que escolher uma das duas representações. Por exemplo, se fosse escolhido a variável da esquerda para predicado ficava-se com $Var_1 = (Var_2)$. Mas, se noutra especificação tivesse-se $Var_2 = Var_1$ então iria-se escolher $Var_2 = (Var_1)$. E embora $Var_1 = Var_2$ seja equivalente a $Var_2 = Var_1$,

não seria possível emparelhar os predicados $Var_1 = (Var_2)$ e $Var_2 = (Var_1)$, visto serem dois predicados distintos.
Esta limitação não é grave, pois os predicados aplicados a variáveis de estado ou de interface podem ser representados apenas por predicados aplicados a variáveis quantificadas. Tal como demonstra o exemplo 3.1, um predicado aplicado a uma variável de estado ou de interface pode ser substituído pela aplicação do predicado a um variável quantificada existencialmente que toma obrigatoriamente o valor da variável de estado ou de interface. Por isso, falta só ser resolvida a limitação da aplicação dos quantificadores para também eliminar esta restrição.

Exemplo 3.1 $P(Var) \Leftrightarrow \exists x\,[P(x) \wedge Var = (x)]$

Predicados Especiais

Na literatura consultada o predicado igualdade(=) não recebe nenhum tratamento especial em relação aos outros predicados. Mas, o predicado referente à igualdade é um predicado especial, porque tem um significado implicitamente associado, que é idêntico em qualquer especificação. 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(>), etc....
Isto porque todos estes predicados já têm um significado próprio associado. Neste trabalho só se usou o predicado igualdade, mas este trabalho está preparado para, se necessário, acrescentar outros tipos de predicados especiais.

3.2.2 Classificação de um Componente

A classificação de um componente é igual ao conjunto de especificações categoriais dos seus vários métodos. O método de classificação desenvolvido divide-se então em duas partes:

1. Especificação categorial de uma afirmação clausal.

2. Especificação categorial de um método.

A forma como estas especificações categoriais são obtidas é apresentada nas secções seguintes.

3.2.3 Especificação Categorial de Uma Afirmação Clausal

Dado uma afirmação clausal Φ, a especificação categorial de Φ é uma categoria $\mathcal{C}_\Phi$ semanticamente equivalente a $Th(\Phi)$.

O conjunto $obj(\mathcal{C}_\Phi)$ é constituído por cinco tipos de objectos:

- Terminal, designa-se por 1 e representa a constante *true*.
- Inicial, designa-se por 0 e representa a constante *false*.
- Singular, designa-se pelo caracter # seguido de um conjunto com apenas uma proposição. Este tipo de objecto representa a proposição indicada no conjunto.
- Mínimo, designa-se pelo caracter & seguido de um conjunto de duas ou mais proposições. Este tipo de objecto representa a conjunção das proposições que contém no conjunto.
- Máximo, designa-se pelo caracter | seguido de um conjunto de duas ou mais proposições. Este tipo de objecto representa a disjunção das proposições que contém no conjunto.

Definição 3.1 (Objecto Origem(Destino)) *Um* objecto origem(destino) *é um objecto singular ou mínimo(máximo).*

Definição 3.2 (Junção de Objectos) *Sejam A e B dois objectos origem(destino), então $A \uplus B$ é o objecto origem(destino) que representa a conjunção(disjunção) das proposições de A e B.*

O conjunto $set(\mathcal{C}_\Phi)$ é definido pela noção de implicação, i.e., $A \rightarrow_{\mathcal{C}_\Phi} B$ sse $A \Rightarrow B$.

Propriedades de $\mathcal{C}_\Phi$:

- Pela definição de $set(\mathcal{C}_\Phi)$ obtém-se directamente que a categoria $\mathcal{C}_\Phi$ é uma categoria magra.
- O objecto 1 é objecto terminal, pois qualquer fórmula implica *true*. O objecto 0 é objecto inicial, pois qualquer fórmula é implicada por *false*.
- Sejam $A, B \in obj(\mathcal{C}_\Phi)$ dois objectos origem(destino), então existe o objecto $A \uplus B \in obj(\mathcal{C}_\Phi)$ que forma o produto(co-produto) binário de A e B.

A ideia base para a identificação de $\mathcal{C}_\Phi$ consiste em representar os antecedentes e consequentes das cláusulas de Φ como objectos de $\mathcal{C}_\Phi$, e representar através de setas em $\mathcal{C}_\Phi$ a implicação entre esses mesmos antecedentes e consequentes.

Setas Implícitas: Para não sobrecarregar a especificação categorial de objectos e setas, optou-se por não registar explicitamente todos os objectos e setas induzidas por $Th(\Phi)$. Assim, representa-se explicitamente os objectos e setas induzidas por Φ, obtendo-se os outros objectos e setas pelas seguintes regras:

- *Identidade*: Se $A \in obj(\mathcal{C}_\Phi)$, então $A \rightarrow_{\mathcal{C}_\Phi} A$.
- *Composição*: Se $A \rightarrow_{\mathcal{C}_\Phi} B$ e $B \rightarrow_{\mathcal{C}_\Phi} C$, então $A \rightarrow_{\mathcal{C}_\Phi} C$.
- *Produto Finito*: Se $A, B \in obj(\mathcal{C}_\Phi)$ são objectos origem, então $A \uplus B \rightarrow_{\mathcal{C}_\Phi} A$ e $A \uplus B \rightarrow_{\mathcal{C}_\Phi} B$.
- *Co-Produto Finito*: Se $A, B \in obj(\mathcal{C}_\Phi)$ são objectos destino, então $A \rightarrow_{\mathcal{C}_\Phi} A \uplus B$ e $B \rightarrow_{\mathcal{C}_\Phi} A \uplus B$.
- *Terminal*: Se $A \in obj(\mathcal{C}_\Phi)$, então $A \rightarrow_{\mathcal{C}_\Phi} 1$.
- *Inicial*: Se $A \in obj(\mathcal{C}_\Phi)$, então $0 \rightarrow_{\mathcal{C}_\Phi} A$.
- *Regra de Resolução*: Seja $A_1 \rightarrow_{\mathcal{C}_\Phi} B_1$, $A_2 \rightarrow_{\mathcal{C}_\Phi} B_2$, e $q \in [(A_1 \cap B_2) \cup \{true, false\}]$, então $(A_1 \backslash \{q\} \uplus A_2) \rightarrow_{\mathcal{C}_\Phi} (B_1 \uplus B_2 \backslash \{q\})$.

A completude e correcção da regra de resolução [RN95] dá a garantia que não é perdida informação no método de classificação, embora seja necessário utilizar um mecanismo de inferência para manipular a informação implícita numa especificação categorial.
No exemplo 3.2 são apresentadas duas possíveis aplicações da regra resolução.

Exemplo 3.2 *Aplicações da regra de resolução:*

- *Se* $\#\{p_1\} \rightarrow_{\mathcal{C}_\Phi} \#\{p_2\}$ *e* $1 \rightarrow_{\mathcal{C}_\Phi} \#\{p_1\}$ *então* $1 \rightarrow_{\mathcal{C}_\Phi} \#\{p_2\}$.
- *Se* $\&\{p_1, p_2\} \rightarrow_{\mathcal{C}_\Phi} |\{p_3, p_4\}$ *e* $\#\{p_5\} \rightarrow_{\mathcal{C}_\Phi} 1$ *então* $\&\{p_1, p_2, p_5\} \rightarrow_{\mathcal{C}_\Phi} |\{p_3, p_4\}$.

Os antecedentes e consequentes das cláusulas são representados através de conjuntos pseudo-proposições. Por forma, a representar os antecedentes e consequentes em objectos de $\mathcal{C}_\Phi$, tem-se de transformar de alguma forma as pseudo-proposições em apenas proposições.

As constantes *true* e *false* são casos especiais que são imediatamente traduzidas respectivamente no objecto 1 e 0. Não é considerado o caso em que estas constantes pertencem ao conjunto de pseudo-proposições de um objecto máximo ou mínimo, pois estas constantes têm o papel de elemento neutro ou absorvente nesses casos.

O caso das pseudo-proposições, que representam um quantificador aplicado a um predicado, não podem ser transformados em proposições de uma forma independente da cláusula onde se inserem. Desta forma, a transformação é obtida cláusula a cláusula, aplicando os postulados que em seguida se apresentam.

Postulado 3.1 *Sendo P um predicado e $\{c_1, \ldots, c_n\}$ o conjunto suporte, então temos as seguintes tautologias:*

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

Postulado 3.2 *Sendo P um predicado, $\{c_1, \ldots, c_n\}$ o conjunto suporte, α uma conjunção de proposições, e β uma disjunção de proposições, então temos as seguintes tautologias:*

- $[\alpha \wedge (P(c_1) \vee \ldots \vee P(c_n)) \Rightarrow \beta] \Leftrightarrow [(\alpha \wedge P(c_1) \Rightarrow \beta) \wedge \ldots \wedge (\alpha \wedge P(c_n) \Rightarrow \beta)]$
- $[\alpha \Rightarrow \beta \vee (P(c_1) \wedge \ldots \wedge P(c_n))] \Leftrightarrow [(\alpha \Rightarrow \beta \vee P(c_1)) \wedge \ldots \wedge (\alpha \Rightarrow \beta \vee P(c_n))]$

Note-se que as fórmulas do lado direito da equivalência estão sob forma clausal.

Considerando que *quant* e *supt* representam, respectivamente, o número de quantificadores e o tamanho do conjunto de suporte, a aplicação dos postulados 3.1 e 3.2 provoca uma duplicação de proposições a representar da ordem

$$\mathcal{O}(quant \times supt) \tag{3.1}$$

Para evitar essa duplicação definiu-se os conceitos que seguidamente se apresentam, e que permitem guardar implicitamente os resultados da aplicação dos postulados.

Definição 3.3 (Expansão) *Seja α uma conjunção de proposições, β uma disjunção de proposições, P um predicado, e $\{c_1, \ldots, c_n\}$ o conjunto suporte, então designa-se por* expansão *de P o operador $*P$, tal que:*

$$[\alpha \wedge *P \Rightarrow \beta] \;\Leftrightarrow\; [(\alpha \wedge P(c_1) \Rightarrow \beta) \wedge \ldots \wedge (\alpha \wedge P(c_n) \Rightarrow \beta)]$$

$$[\alpha \Rightarrow \beta \vee *P] \;\Leftrightarrow\; [(\alpha \Rightarrow \beta \vee P(c_1)) \wedge \ldots \wedge (\alpha \Rightarrow \beta \vee P(c_n))]$$

O conceito de expansão de um predicado permite representar numa cláusula o conjunto de cláusulas derivadas da duplicação dessa mesma cláusula, pela aplicação do predicado a todos os elementos do conjunto suporte.

Definição 3.4 (Extensão) *Seja α uma conjunção de proposições, β uma disjunção de proposições, P um predicado, e $\{c_1, \ldots, c_n\}$ o conjunto suporte, então designa-se por* extensão *de P o operador $+P$, tal que:*

$$[\alpha \wedge +P \Rightarrow \beta] \;\Leftrightarrow\; [\alpha \wedge P(c_1) \wedge \ldots \wedge P(c_n) \Rightarrow \beta]$$

$$[\alpha \Rightarrow \beta \vee +P] \;\Leftrightarrow\; [\alpha \Rightarrow \beta \vee P(c_1) \vee \ldots \vee P(c_n)]$$

O conceito de extensão de um predicado permite representar a conjunção ou disjunção (conforme o predicado esteja inserido respectivamente no antecedente ou no consequente da cláusula) das aplicações do predicado a todos os elementos do conjunto suporte.

Definição 3.5 (Asserção) *O conceito de* asserção *é definido da seguinte forma:*

- *Seja P um predicado e c uma constante, então $P(c)$ é uma asserção.*
- *Seja P um predicado, então $*P$ (*expansão*) é uma asserção.*
- *Seja P um predicado, então $+P$ (*extensão*) é uma asserção.*

Postulado 3.3 *Seja α uma conjunção de proposições, β uma disjunção de proposições, e P um predicado, então temos os seguintes tautologias:*

- $[\alpha \wedge \forall x\, P(x) \Rightarrow \beta] \Leftrightarrow [\alpha \wedge +P \Rightarrow \beta]$
- $[\alpha \wedge \exists x\, P(x) \Rightarrow \beta] \Leftrightarrow [\alpha \wedge *P \Rightarrow \beta]$
- $[\alpha \Rightarrow \beta \vee \forall x\, P(x)] \Leftrightarrow [\alpha \Rightarrow \beta \vee *P]$
- $[\alpha \Rightarrow \beta \vee \exists x\, P(x)] \Leftrightarrow [\alpha \Rightarrow \beta \vee +P]$

Com a aplicação do postulado 3.3, a complexidade de 3.1 é reduzida para $\mathcal{O}(1)$.

Identificação de $\mathcal{C}_\Phi$: Seja Ψ a afirmação clausal de asserções resultante da aplicação do postulado 3.3 a todas as pseudo-proposições com quantificadores de Φ, então para qualquer $A \Rightarrow B \in \Psi$ tem-se $A, B \in obj(\mathcal{C}_\Phi)$ e $A \rightarrow_{\mathcal{C}_\Phi} B$.

3.2.4 Especificação Categorial de Um Método

As variáveis de estado de um método representam dois valores, o valor inicial da variável antes do método ser executado e o valor final da variável depois do método ser executado. Desta forma, as variáveis de estado são representadas categorialmente por duas variáveis, uma referente ao seu valor inicial e outra ao seu valor final.

Nesta secção, a especificação categorial de uma determinada afirmação clausal é sempre a especificação categorial da conjunção dessa afirmação com o invariante do componente.

A especificação categorial de um método é constituída pelos seguintes elementos:

- Pré-categoria: a pré-categoria é a especificação categorial da afirmação clausal que representa as implicações entre todos os antecedentes das pós-condições e a pré-condição.

- Pós-categorias: para cada consequente de uma pós-condição cria-se uma pós-categoria, que corresponde à especificação categorial da afirmação clausal que representa esse consequente.

- Ligação entre a pré-categoria e as pós-categorias: por cada pós-condição associa-se a sua pós-categoria com o objecto da pré-categoria induzido pelo antecedente dessa mesma pós-condição.

3.3 Representação das Especificações

Este trabalho manipula dois esquemas de representação, algébricos e categoriais. Para ambas é necessário criar uma representação computacional que seja adequada aos objectivos que se pretende alcançar.
Ambas as especificações têm de ser guardadas de alguma forma em memória permanente através de ficheiros, por forma a que o utilizador as possa facilmente manipular e criar fora do âmbito da ferramenta que se pretende implementar. Desta forma, foram criados dois tipos de representação para cada especificação: a representação externa, que é uma cadeia de caracteres respeitando uma determinada sintaxe e que é guardada sob a forma de ficheiro de texto; a representação interna, que é a transformação da sintaxe da representação externa para um conjunto de listas e registos que são manipulados pela aplicação.

A sintaxe das representações externas de cada especificação é apresentado em notação EBNF no apêndice A. Por forma a não prejudicar a portabilidade das

representações externas foram só usados caracteres *ASCII*. Deste modo, nas especificações algébricas os símbolos $\forall, \exists, \wedge, \vee$ são respectivamente representados pelos caracteres `>`, `<`, `&`, `|`.
As estruturas de dados que determinam as representações internas das especificações algébricas e das especificações categoriais estão definidas respectivamente nos ficheiros `data_spc.sml` e `data_ctg.sml` apresentados no apêndice C.

A principal preocupação na definição da sintaxe das representações externas foi a sua legibilidade, isto por forma a disponibilizar ao utilizador especificações legíveis e facilmente manipuláveis a partir de um qualquer editor de texto. Por sua vez, as representações internas foram definidas de maneira a que a sua manipulação pela aplicação fosse o mais eficaz possível.

Por consequência, foi necessário também realizar algoritmos de tradução entre os dois tipos de representação. Este algoritmos foram construídos a partir de técnicas já conhecidas na área de Compiladores [Cre98b], onde se utiliza analisadores sintácticos e léxicos no desenvolvimento de algoritmos deste tipo.
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.

3.4 Algoritmos Desenvolvidos

Depois, de desenvolvidos e definidos os conceitos teóricos inerentes ao processo de classificação, foi necessário desenvolver os algoritmos que traduzem esses conceitos teóricos para um conjunto de regras e instruções que possam, sem dificuldade, ser executadas computacionalmente. Os algoritmos produzidos podem ser descritos da seguinte forma:

Classifica Componente
Entrada: A especificação algébrica de um componente de *software*.
Saída: A especificação categorial equivalente à especificação algébrica dada.

1. Executar o algoritmo *Classifica Método* para cada método do componente.

2. Devolver o conjunto suporte e a lista das especificações categoriais obtidas por cada método.

Classifica Método
Entrada: A especificação algébrica de um método.
Saída: A respectiva especificação categorial do método dado.

1. Executar o algoritmo *Classifica Afirmação Clausal* com o seguinte argumento:
   A afirmação clausal resultante da conjunção das fórmulas da forma $\varepsilon_{ai} \Rightarrow R$ (com i=0, ..., nº de pós-condições), em que ε_{ai} é a conjunção relativa ao antecedente de uma pós-condição e R é a pré-condição.
   A categoria, que é devolvida pelo algoritmo neste passo, designa-se por pré-categoria.

2. Por cada consequente de uma pós-condição executar o algoritmo *Classifica Afirmação Clausal*.
   A cada categoria, que é devolvida pelo algoritmo neste passo, designa-se por pós-categoria.

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

Classifica Afirmação Clausal
Entrada: Φ - Uma afirmação clausal.
Saída: A categoria $\mathcal{C}_\Phi$.

1. Adicionar ao conjunto de cláusulas Φ o conjunto de cláusulas que constituem o invariante do componente.

2. Executar o algoritmo *Classifica Cláusula* por cada cláusula pertencente a Φ.

3. A reunião dos objectos e das setas devolvidas no passo anterior formam a categoria $\mathcal{C}_\Phi$.

Classifica Cláusula
Entrada: ϕ - Uma cláusula constituída por pseudo-proposições.
Saída: O conjunto de objectos e setas induzidas pela cláusula dada.

1. Executar o algoritmo *Retira Quantificadores* para a cláusula ϕ.

2. Por cada cláusula φ de asserções resultante do passo anterior deve-se executar os seguintes passos:

   (a) A cláusula φ pode ser 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.
- Os símbolos $p_1, \ldots, p_n, q_1, \ldots, q_m$ representam pseudo-asserções[1].

(b) Criar os seguintes objectos:

- $A = \begin{cases} \&\{p_1, \ldots, p_n\} & \text{se } n > 1 \\ 1 & \text{se } n = 1 \text{ e } p_1 = true \\ 0 & \text{se } n = 1 \text{ e } p_1 = false \\ \#\{p_1\} & \text{caso contrário} \end{cases}$
- $B = \begin{cases} |\{q_1, \ldots, q_m\} & \text{se } n > 1 \\ 1 & \text{se } n = 1 \text{ e } p_1 = true \\ 0 & \text{se } n = 1 \text{ e } p_1 = false \\ \#\{q_1\} & \text{caso contrário} \end{cases}$

(c) Criar a seta $A \rightarrow_{\mathcal{C}_\Phi} B$.

Retira Quantificadores
Entrada: ϕ - Uma cláusula constituída por pseudo-proposições.
Saída: Uma cláusula constituída por pseudo-asserções equivalente à cláusula dada.

1. A cláusula ϕ pode ser 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.
   - Os símbolos $p_1, \ldots, p_n, q_1, \ldots, q_m$ representam pseudo-proposições.
2. Por cada símbolo do tipo $p_1, \ldots, p_n$ que represente:
   - $\forall x\, P(x)$, substituir por $+P$.
   - $\exists x\, P(x)$, substituir por $*P$.
3. Por cada símbolo do tipo $q_1, \ldots, q_n$ que represente:
   - $\forall x\, P(x)$, substituir por $*P$.
   - $\exists x\, P(x)$, substituir por $+P$.

Exemplo 3.3 *A figura 3.1 apresenta a especificação categorial que se obtém aplicando o algoritmo* Classifica Método *ao método* `Incrementa` *do exemplo 2.5.*

[1]Por pseudo-asserção entenda-se a extensão do conceito de asserção às constantes *true* e *false*.

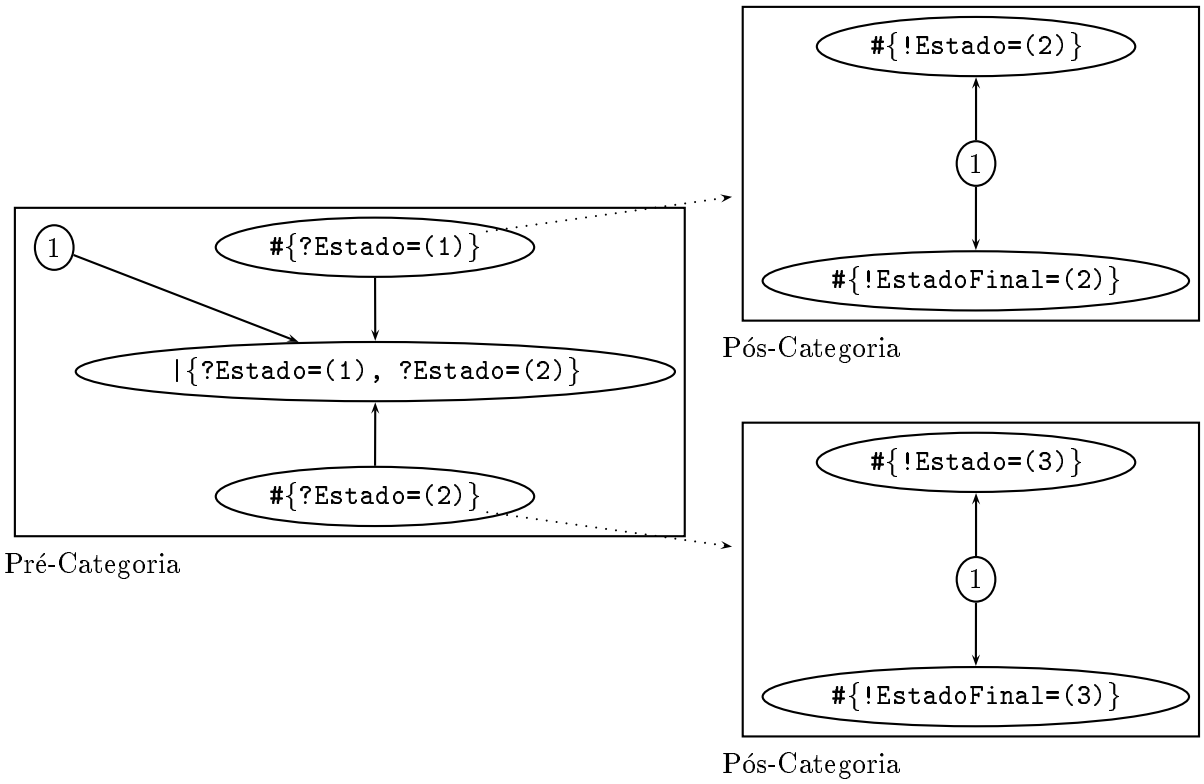

Figura 3.1: Especificação categorial do método `Incrementa`

3.5 Complexidade

Nesta secção é descrita a análise da complexidade temporal e espacial do método de classificação desenvolvido.

3.5.1 Complexidade Temporal

Calcular a complexidade temporal de classificar um método de um componente, é calcular a complexidade dos algoritmos que desempenham essa tarefa. Desta forma, em seguida apresenta-se a análise da complexidade temporal, no pior caso, de cada algoritmo descrito na secção 3.4.

A complexidade aqui apresentada depende das seguintes variáveis:

- *prop*, representa o número máximo de pseudo-proposições numa cláusula.
- *cla*, representa o número máximo de cláusulas numa afirmação clausal.
- *pos*, representa o número de pós-condições.
- *pre*, representa o número de disjunções na pré-condição.

Retira Quantificadores A execução deste algoritmo percorre todas as pseudo-proposições de uma cláusula, transformando-as em asserções, deste modo, a sua complexidade é $\mathcal{O}(prop)$.

Classifica Cláusula Este algoritmo utiliza o *Retira Quantificadores*, e cria os dois objectos correspondentes. Desta forma, a sua complexidade é também $\mathcal{O}(prop)$.

Classifica Afirmação Clausal O algoritmo *Classifica Cláusula* é chamado por este algoritmo por cada cláusula da afirmação clausal, logo, a sua complexidade é $\mathcal{O}(cla \times prop)$.

Classifica Método Este algoritmo chama o algoritmo *Classifica Afirmação Clausal* com o conjunto cláusulas formadas a partir das implicações entre todos os antecedentes das pós-condição e a pré-condição. Assim, o número de cláusulas formado neste passo será $pos \times pre$, e deste modo, a complexidade deste passo é $\mathcal{O}(pos \times pre \times prop)$.
De seguida, este algoritmo chama de novo o algoritmo *Classifica Afirmação Clausal* com os consequentes das pós-condições. Desta forma, a complexidade deste passo é $\mathcal{O}(pos \times cla \times prop)$.
Conclui-se, então, que a complexidade do algoritmo *Classifica Método* é

$$\mathcal{O}(pos \times prop \times (pre + cla))$$

3.5.2 Complexidade Espacial

A análise da complexidade espacial tem como objectivo comparar o espaço de dados da especificação categorial com o espaço de dados da especificação algébrica que lhe deu origem.

No método de classificação temos as seguintes relações entre os elementos das especificações algébricas e os elementos das especificações categoriais:

- A cada pseudo-proposição corresponde uma asserção.
- A cada conjunção ou disjunção de pseudo-proposições corresponde um objecto composto pelas respectivas asserções.
- A cada implicação corresponde uma seta.
- A uma pós-condição corresponde uma pós-categoria.

- A uma pré-condição corresponde uma pré-categoria.

A partir destas relações conclui-se que o espaço de dados de uma especificação categorial é linearmente proporcional ao espaço de dados da especificação algébrica que lhe deu origem.

3.6 Gestão das Especificações

O processo *Gestão* têm como objectivo guardar e manipular as especificações algébricas introduzidas pelo utilizador, e as especificações categoriais resultantes da execução do processo *Classificação*.
As operações disponibilizadas por este processo são as comuns a qualquer sistema de gestão de base de dados. O principal factor de preocupação no desenvolvimento deste processo esteve concentrado no objectivo de implementar as operações de uma forma simples e eficaz, pois estas são um factor muito importante para o cálculo da complexidade do processo selecção, que executa estas operações com muita frequência.

A solução encontrada para a construção do repositório consiste nos seguintes pontos:

- Por cada componente existem dois ficheiros: um ficheiro que contém a sua especificação algébrica, e outro que contém a sua especificação categorial.
- O nome dos ficheiros com as especificações algébricas terminam com a extensão "`.spc`".
- O nome dos ficheiros com as especificações categoriais terminam com a extensão "`.ctg`".
- O prefixo do nome dos ficheiros deverá ser o identificador do componente que a especificação, contida no ficheiro, representa.
- Existe um ficheiro índice de seu nome *index.ctg*. Este ficheiro contém por cada componente o seu identificador e o tamanho do seu conjunto suporte.
- Existe uma directoria que contém todos os ficheiros referidos anteriormente.

O ficheiro de índice contém o tamanho do conjunto de suporte, por forma a possibilitar uma filtragem das especificações categoriais candidatas ao emparelhamento, sem ser necessário consultar os respectivos ficheiros.

Cada vez que é dado ao processo um identificador de um componente que já existe no repositório, este considera que se trata de uma actualização, e desta forma substitui as especificações antigas pelas novas.

3.7 Implementação

A informação relativa à implementação dos métodos descritos neste capítulo pode ser consultada na secção 4.3.

3.8 Extensão das Especificações

Nesta secção são apresentadas técnicas, originais, que permitem eliminar algumas das restrições inerentes à linguagem das especificações utilizadas pela aplicação implementada.
Estas técnicas são extensões dos métodos implementados, i.e., não contrariam o que foi implementado, apenas acrescentam novas capacidades.

3.8.1 Quantificadores

As especificações algébricas utilizadas nos métodos implementados só permitem a aplicação de quantificadores a um só predicado. Nesta secção é apresentada uma técnica de classificação que permite o uso de afirmações prenex, i.e., sem restrições em relação à aplicação de quantificadores.

Definições

Nesta secção são apresentadas algumas definições necessárias para descrever o método aqui proposto. Para essas definições considera-se o conjunto de suporte igual a $\{c_1, \ldots, c_n\}$.

Definição 3.6 (Combinador) *Um* combinador *é uma sequência de conjuntos de proposições e denota-se por* $\langle\{\alpha_1, \ldots, \alpha_n\}\rangle$, *em que* $\alpha_1, \ldots, \alpha_n$ *são conjuntos de proposições, e n um qualquer número natural maior que zero.*

Definição 3.7 (Substituição) $[c/x]\alpha$ *denota o resultado da substituição de todas as ocorrências da variável x pela constante c no conjunto de proposições* α, *ou seja:* $[c/x]\alpha \ =_{def} \ \{P(c) : P(x) \in \alpha\} \cup \{P(y) : P(y) \in \alpha \wedge y \neq x\}$

Definição 3.8 (Multiplicação) *Uma* multiplicação *é uma sequência de pares. Cada um desses pares contém uma variável e uma operação. Existem dois tipos de operação, denotando-se respectivamente por* $+$ *(extensão) e* $*$ *(expansão).*

Definição 3.9 (Seta Múltipla) *Uma* seta múltipla *representa um conjunto de setas e denota-se por* $\gamma \xrightarrow{m} \delta$*, em que* γ *e* δ *são dois conjuntos de proposições ou dois combinadores e* m *uma multiplicação. O conjunto de setas que representa é definido da seguinte forma:*

$$\begin{aligned}
\gamma \xrightarrow{m\cdot(x,+)} \delta \;&=_{def}\; \{[c_1/x]\alpha \to [c_1/x]\beta, \ldots, [c_n/x]\alpha \to [c_n/x]\beta \\
&\qquad\qquad : \; \alpha \to \beta \in \gamma \xrightarrow{m} \delta\} \\
\gamma \xrightarrow{m\cdot(x,*)} \delta \;&=_{def}\; \{[c_1/x]\alpha_1 \cap \ldots \cap [c_n/x]\alpha_n \to [c_1/x]\beta_1 \cup \ldots \cup [c_n/x]\beta_n \\
&\qquad\qquad : \; \langle\langle \alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n \rangle\rangle \in \gamma \xrightarrow{m} \delta\} \\
\gamma \longrightarrow \delta \;&=_{def}\; \{\gamma \to \delta\} \\
\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \longrightarrow \langle\{\beta_1, \ldots, \beta_n\}\rangle \;&=_{def}\; \{\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n\}
\end{aligned}$$

Especificação Categorial de Uma Afirmação Prenex

Nesta secção apresenta-se a forma como se produz a especificação categorial de uma afirmação prenex.

A especificação categorial de uma afirmação prenex consiste numa categoria. O tipo de objectos e setas, dessa categoria, são os já definidos na secção 3.2.3.

Em seguida, define-se a função Υ, que dado uma afirmação prenex devolve as setas múltiplas que a representam categorialmente. Apesar das setas múltiplas poderem ser traduzidas no conjunto de setas que representam, será boa ideia manter essa informação implícita para não aumentar a complexidade espacial do método.

Para um conjunto de cláusulas quantificadas $\Phi_1, \ldots, \Phi_n$ tem-se que:

$$\Upsilon(\Phi_1, \ldots, \Phi_n) \;=_{def}\; \bigcup_{1 \leqslant i \leqslant n} \Upsilon_{cq}(\Phi_i)$$

Para uma cláusula quantificada $\Delta_1 x_1 \ldots \Delta_n x_n[w]$ tem-se que:

$$\Upsilon_{cq}(\Delta_1 x_1 \ldots \Delta_n x_n[w]) \;=_{def}\; \Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n \Upsilon_{cs}(w))$$

Em que:

$$\Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n \forall x\{\alpha_1 \xrightarrow{m} \beta_1, \ldots, \alpha_n \xrightarrow{m} \beta_n\}) =_{def} \Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n\{\alpha_1 \xrightarrow{m\cdot(x,+)} \beta_1, \ldots, \alpha_n \xrightarrow{m\cdot(x,+)} \beta_n\})$$

$$\Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n \forall x\{\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \xrightarrow{m} \langle\{\beta_1, \ldots, \beta_n\}\rangle\}) =_{def} \Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n\{\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \xrightarrow{m\cdot(x,+)} \langle\{\beta_1, \ldots, \beta_n\}\rangle\})$$

$$\Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n \exists x\{\alpha_1 \xrightarrow{m} \beta_1, \ldots, \alpha_n \xrightarrow{m} \beta_n\}) =_{def} \Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n\{\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \xrightarrow{m\cdot(x,*)} \langle\{\beta_1, \ldots, \beta_n\}\rangle\})$$

$$\Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n \exists x\{\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \xrightarrow{m} \langle\{\beta_1, \ldots, \beta_n\}\rangle\}) =_{def} \Upsilon_q(\Delta_1 x_1 \ldots \Delta_n x_n\{\langle\{\alpha_1, \ldots, \alpha_n\}\rangle \xrightarrow{m\cdot(x,*)} \langle\{\beta_1, \ldots, \beta_n\}\rangle\})$$

$$\Upsilon_q(\{s_1, \ldots, s_n\}) =_{def} \{s_1, \ldots, s_n\}$$

Para um conjunto de cláusulas $\phi_1, \ldots, \phi_n$ tem-se que:

$$\Upsilon_{cs}(\phi_1, \ldots, \phi_n) =_{def} \{\Upsilon_c(\phi_i) \ : \ 1 \leqslant i \leqslant n\}$$

Para uma cláusula da forma $N \Rightarrow P$ tem-se que:

$$\Upsilon_c(N \Rightarrow P) =_{def} N \dot{\longrightarrow} P$$

Observações

A ordem dos pares numa multiplicação segue as mesmas regras da ordem de como são apresentados os quantificadores.
Estas regras têm as seguintes propriedades:

- $(x,+)\cdot(y,+) = (y,+)\cdot(x,+)$ tal como $\forall x \forall y = \forall x \forall y$
- $(x,*)\cdot(y,*) = (y,*)\cdot(x,*)$ tal como $\exists x \exists y = \exists x \exists y$
- $(x,+)\cdot(y,*) \neq (y,*)\cdot(x,+)$ tal como $\forall x \exists y \neq \exists x \forall y$

Para melhor se entender a relação entre $(x,*)$ e $\exists x$, basta ver que:

$$\begin{aligned}\exists x[\alpha_1 \Rightarrow \beta_1 \wedge \ldots \wedge \alpha_m \Rightarrow \beta_m] \Leftrightarrow \\ ([c_1/x]\alpha_1 \Rightarrow [c_1/x]\beta_1 \wedge \ldots \wedge [c_1/x]\alpha_m \Rightarrow [c_1/x]\beta_m) \\ \vee \\ \vdots \\ \vee \\ ([c_n/x]\alpha_1 \Rightarrow [c_n/x]\beta_1 \wedge \ldots \wedge [c_n/x]\alpha_m \Rightarrow [c_n/x]\beta_m)\end{aligned}$$

e aplicando as leis de Morgan, fica-se com:

$$\begin{aligned}
&\exists x[\alpha_1 \Rightarrow \beta_1 \wedge \ldots \wedge \alpha_m \Rightarrow \beta_m] \Leftrightarrow \\
&\quad([c_1/x]\alpha_{f(1,1)} \wedge \ldots \wedge [c_n/x]\alpha_{f(1,n)} \Rightarrow [c_1/x]\beta_{f(1,1)} \vee \ldots \vee [c_n/x]\beta_{f(1,n)}) \\
&\quad\wedge \\
&\quad\vdots \\
&\quad\wedge \\
&\quad([c_1/x]\alpha_{f(k,1)} \wedge \ldots \wedge [c_n/x]\alpha_{f(k,n)} \Rightarrow [c_1/x]\beta_{f(k,1)} \vee \ldots \vee [c_n/x]\beta_{f(k,n)})
\end{aligned}$$

Em que f é uma função de re-indexação, de tal modo que os seguintes conjuntos $\{\langle\!\langle \alpha_{f(i,1)}, \ldots, \alpha_{f(i,n)} \rangle\!\rangle : 1 \leqslant i \leqslant k\}$ e $\{\langle\!\langle \beta_{f(i,1)}, \ldots, \beta_{f(i,n)} \rangle\!\rangle : 1 \leqslant i \leqslant k\}$ representam, respectivamente, todos os arranjos com repetição de tamanho n dos conjuntos $\{\alpha_1, \ldots, \alpha_m\}$ e $\{\beta_1, \ldots, \beta_m\}$. O valor de k é determinado por $k = \overline{\mathcal{A}}_n^m = m^n$.

3.8.2 Pré-condição

Nas especificações algébricas, utilizadas nos métodos implementados, as pré-condições e os antecedentes das pós-condições não podem ser expressas por uma qualquer fórmula. As pré-condições só podem ser expressas por uma conjunção de disjunções, e os antecedentes por uma conjunção.

Por forma a eliminar esta restrição, desenhou-se uma nova forma de especificar a pré-condição de um método. Esta nova forma caracteriza-se pelos seguintes aspectos:

- O antecedente de uma pós-condição pode ser expressa de igual forma que o consequente.
- Passa a existir uma pré-categoria por cada pós-condição, que representa o seu antecedente.
- A pré-condição é induzida directamente pela disjunção de todos os antecedentes da pós-condições. Desta forma, a categoria que representa a pré-condição é a simples junção de todas as pré-categorias.

Capítulo 4

Selecção

Neste capítulo são apresentadas as metodologias, desenvolvidas ao longo deste trabalho, que têm por missão encontrar no repositório componentes com a mesma funcionalidade do componente introduzido pelo utilizador. Este processo baseia-se no emparelhamento das especificações categoriais obtidas pelo processo *Classificação*.

Neste trabalho foi considerada uma abordagem mais relaxada do emparelhamento de especificações. A especificação do repositório pode ser uma extensão da especificação dada pelo utilizador, i.e., cada propriedade derivada da especificação do utilizador é também derivada da especificação do repositório [WT87], mas podem existir propriedades da especificação do repositório que não sejam derivadas da especificação do utilizador.

Os métodos de selecção aqui apresentados não manipulam a informação implícita nas especificações categoriais, para não aumentar a complexidade do processo *Selecção*, que nesta dissertação se quer o menos elevada possível.

4.1 Emparelhamento Isomorfo

O emparelhamento isomorfo é efectuado entre a especificação categorial do utilizador e uma especificação categorial do repositório. O emparelhamento isomorfo consiste em encontrar uma renomeação para os elementos do conjunto de suporte e para os predicados, de tal modo que a renomeação induza functores entre as várias categorias da especificação categorial do utilizador e do repositório.

Nesta secção, os elementos da especificação categorial do utilizador(repositório) são denotados com o sufixo u(r).

A teoria proposta no artigo [Cre98a] sobre emparelhamento isomorfo não sofreu

qualquer alteração, mas como não existia nenhuma indicação de como obter os functores, existiu a necessidade de desenvolver um método que identificasse os functores de uma forma realmente eficaz.

Começa-se primeiro com a introdução de alguns dos conceitos utilizados na metodologia desenvolvida.

Definição 4.1 (Renomeação) *Uma* renomeação R *é um par de funções parciais* (R_x, R_p), *em que:*

- R_x, *é uma função injectiva entre os elementos do conjunto suporte da especificação categorial do utilizador e os elementos do conjunto suporte da especificação categorial do repositório.*
- R_p, *é uma função entre os predicados da especificação categorial do utilizador e os predicados da especificação categorial do repositório.*

A função R_x tem de ser injectiva porque é inconsistente emparelhar duas constantes do método do utilizador com uma só constante do método do repositório. Por exemplo, uma função que devolve verdade ou falso nunca pode ser emparelhada com uma função que devolva sempre o mesmo valor.
Devido a este facto, pode-se concluir que o tamanho do conjunto suporte do utilizador tem de ser menor ou igual ao do repositório. Desta forma, no processo de emparelhamento só são considerados os componentes do repositório que verifiquem essa condição.
A função R_p não necessita de ser injectiva, pois pode acontecer que dois predicados do método do utilizador possam ser colapsados num só predicado, se apresentarem um comportamento igual.

Estas duas funções podem ser implementadas cada uma por um conjunto de pares, visto os seus domínios serem finitos. No caso do conjunto de suporte do utilizador ser infinito, a função R_x vai ter de ser definida através de uma regra porque o conjunto de pares é infinito.

Considere-se R uma renomeação e $\mathcal{C}_u$, $\mathcal{C}_r$ duas categorias.

Definição 4.2 (Renomeação de asserções) R_a *é a função que renomeia asserções a partir de* R. *Seja* Ass *o conjunto de asserções definidas na definição 3.5, então* R_a *é definida da seguinte forma:*

$$\begin{aligned} R_a : Ass &\rightarrow Ass \\ P(c) &\mapsto R_p(P)(R_x(c)) \\ \alpha P &\mapsto \alpha R_p(P) \ , \ com\ \alpha \in \{*, +\} \end{aligned}$$

Definição 4.3 (Mapeamento) *Um* mapeamento $F : \mathcal{C}_u \rightarrow \mathcal{C}_r$ *induzido por* R *é definido da seguinte forma:*

$$\begin{array}{rcl} F_{obj} : obj(\mathcal{C}_u) & \rightarrow & obj(\mathcal{C}_r) \\ 0 & \mapsto & 0 \\ 1 & \mapsto & 1 \\ \#\{p\} & \mapsto & \#\{R_a(p)\} \\ |\{p_1, \ldots, p_n\} & \mapsto & |\{R_a(p_1), \ldots, R_a(p_n)\} \\ \&\{p_1, \ldots, p_n\} & \mapsto & \&\{R_a(p_1), \ldots, R_a(p_n)\} \end{array}$$

$$\begin{array}{rcl} F_{set} : set(\mathcal{C}_u) & \rightarrow & set(\mathcal{C}_r) \\ A \rightarrow_{\mathcal{C}} B & \mapsto & F_{obj}(A) \rightarrow_{\mathcal{C}} F_{obj}(B) \end{array}$$

4.1.1 Renomeação por Procura Exaustiva

A forma mais intuitiva de encontrar um emparelhamento isomorfo é considerar todas as hipóteses possíveis de renomeação e verificar se os mapeamentos induzidos constituem functores, entre as categorias das duas especificações categoriais.
O problema desta solução, como facilmente se pode verificar, é a sua complexidade, pois é exponencial ao tamanho do conjunto suporte($supt$) e ao número de predicados($pred$) $\mathcal{O}(2^{supt+pred})$. Este aspecto torna completamente inviável a utilização desta solução, pois mesmo para casos simples a complexidade é muito elevada.

4.1.2 Renomeação por Construção Progressiva

A solução encontrada neste trabalho para construir uma renomeação, com uma complexidade mais reduzida, consiste em ir construindo as suas funções progressivamente. Ou seja, ao mesmo tempo que se verifica se as duas especificações categoriais são isomorfas, constrói-se as duas funções que constituem uma renomeação.

O processo de emparelhamento das duas especificações categoriais é constituído pelos emparelhamentos dos seus elementos. Desta forma, foram definidas as seguintes funções de emparelhamento:

Método: Para emparelhar as especificações categoriais que representam o método do utilizador e do repositório, tem de se emparelhar as suas respectivas pré-categorias e pós-categorias.

Pré-categoria: Emparelhar a pré-categoria do utilizador com a pré-categoria do repositório é emparelhar respectivamente as duas categorias.

Pós-categorias: Por cada par $(obj_u, pcatg_u)$ objecto, pós-categoria, tem-se:

1. Como já foi emparelhada a pré-categoria, então pode-se obter $F_{obj}(obj_u) = obj_r$.

2. Procura-se um par $(obj_r, pcatg_r)$ nos pares objecto, pós-categoria do repositório. Se o par não for encontrado, o emparelhamento falha.

3. Emparelha-se as duas categorias respectivas, $pcatg_u$ e $pcatg_r$.

Categoria: Considere-se $\mathcal{C}_u$ e $\mathcal{C}_r$ as categorias dadas, R a renomeação actual e $F : \mathcal{C}_u \to \mathcal{C}_r$ o mapeamento induzido por R.
Para emparelhar $\mathcal{C}_u$ com $\mathcal{C}_r$, F têm de constituir um functor de $\mathcal{C}_u$ para $\mathcal{C}_r$. Como, à partida, R pode não conter a informação suficiente para induzir um functor, esta vai sendo completada ao mesmo tempo que se verifica se F é um functor. A forma de como R vai ser completada é descrita no seguinte método:

1. Em primeiro lugar emparelha-se os objectos e fica-se com F_{obj} completamente definido.

2. Se F_{obj} estiver totalmente definido, então por definição também F_{set} está. Desta forma, o segundo passo é verificar a consistência de F_{set}, i.e., verificar se F constitui um functor.

Objectos: Como se pode observar, na definição de F_{obj}, não é possível emparelhar dois objectos de tipos diferentes e com um número de asserções diferentes. Logo, o primeiro passo consiste em agrupar os objectos das duas categorias pelo seu tipo e número de asserções que contêm.
Em seguida, emparelha-se cada grupo de objectos formado na categoria do utilizador com o grupo de objectos, emparelháveis com os anteriores, da categoria do repositório.

Grupo de objectos: Considere-se Obj_u e Obj_r o grupo de objectos dados.
Para emparelhar Obj_u com Obj_r tem de se encontrar o seguinte conjunto de funções injectivas $\Delta = \{f : \forall x \in Obj_u \, \exists^1 y \in Obj_r \, f(x) = y\}$.
Em seguida, tem de se verificar se $\exists f \in \Delta$ tal que a relação $\{(x, f(x)) : x \in Obj_u\}$ é emparelhável.

Relação de Objectos: Verificar se uma relação de objectos é emparelhável, é verificar se, para todos os pares de objectos (obj_u, obj_r) pertencentes à relação, é possível emparelhar obj_u com obj_r.

Objecto: Considere-se obj_u e obj_r os objectos dados.
Sendo Ass_u, Ass_r os grupos de asserções respectivamente de obj_u e obj_r, então emparelhar obj_u com obj_r é emparelhar Ass_u com Ass_r.
Para emparelhar Ass_u com Ass_r tem de se encontrar o seguinte conjunto de funções injectivas $\Delta = \{f : \forall x \in Ass_u \, \exists^1 y \in Ass_r \, f(x) = y\}$.
Em seguida, tem de se verificar se $\exists f \in \Delta$ tal que a relação $\{(x, f(x)) : x \in Ass_u\}$ é emparelhável.

Relação de Asserções: Verificar se uma relação de asserções é emparelhável, é verificar se, para todos os pares de asserções (ass_u, ass_r) pertencentes à relação, é possível emparelhar ass_u com ass_r.

Asserção: Considere-se ass_u, ass_r as asserções dadas e R a renomeação actual.
Para emparelhar ass_u com ass_r tem-se verificar se $R_a(ass_u) = ass_r$. No caso de $R_a(ass_u)\uparrow$ (indefinida) altera-se R de tal modo que se fique com $R_a(ass_u) = ass_r$.

Setas: Considere-se Set_u e Set_r o conjunto de setas dadas, R a renomeação actual e F o mapeamento induzido por R.
Emparelhar Set_u com Set_r consiste em verificar se F_{set} está bem definido. Para isso, em primeiro lugar obtém-se o conjunto $Set_t = \{F_{set}(s) : s \in Set_u\}$, o que é possível pois aqui F_{set} está completamente definido. De seguida, tem de se considerar um dos dois casos:

- Se as setas pertencerem a pré-categorias, então verifica-se se $Set_t \subseteq Set_r$.
- Se as setas pertencerem a pós-categorias, então verifica-se se $Set_t \supseteq Set_r$.

Esta distinção entre o caso das pré-categorias e pós-categorias, deve-se ao facto da especificação do repositório poder ser uma extensão da especificação do utilizador. Ou seja, a pré-categoria do utilizador pode ser menos restritiva do que a do repositório e as pós-categorias do utilizador podem ser mais restritivas do que as do repositório. Sabendo que cada seta representa uma restrição, conclui-se imediatamente a necessidade das relações apresentadas anteriormente, entre os dois conjuntos de setas.

Um exemplo de um emparelhamento isomorfo entre duas especificações categoriais pode ser observado na ultima parte do exemplo 4.1.

Representação do Problema

Para representar o processo de emparelhamento, descrito na secção 4.1.2, utiliza-se o conceito dos grafos *And/Or* [Pea85], muito usados na resolução de problemas na área de Inteligência Artificial.
Neste projecto, o conceito é ampliado por forma a ser possível representar o nosso problema. Cada nó representa uma das funções de emparelhamento. O tipo de nó define a relação entre esse nó e os seus sucessores. Desta forma, no nosso caso existem três tipos de nós presentes no grafo:

- O nó *And* indica que é necessário fazer várias chamadas a uma função sucessora, com diferentes argumentos, que têm de ser todas verificadas.
- O nó *Or* indica que é necessário fazer várias chamadas a uma função sucessora, com diferentes argumentos, em que basta uma delas ser verificada.
- O nó *Single* indica que é necessário fazer uma chamada a cada uma das funções sucessoras, que têm de ser verificadas.

Os nós *And* e *Or* são definidos através da função que representam, da sua função sucessora, e da respectiva lista de argumentos para a função sucessora.
Se num nó *And* uma das chamadas à função sucessora falha, o nó também falha. No caso de um nó *Or*, este só falha se todas as chamadas à função sucessora falharem.

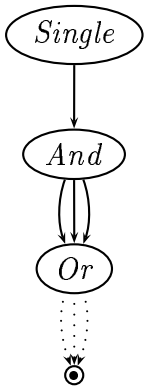

Figura 4.1: Tipos de nós

A figura 4.1 apresenta a forma de como são representados os três tipos de nós. O nó *Single* é representado por uma seta para cada nó sucessor. O nó *And* é representado por um conjunto de setas para o nó sucessor. O nó *Or* é representado por um conjunto de setas a tracejado para o nó sucessor.

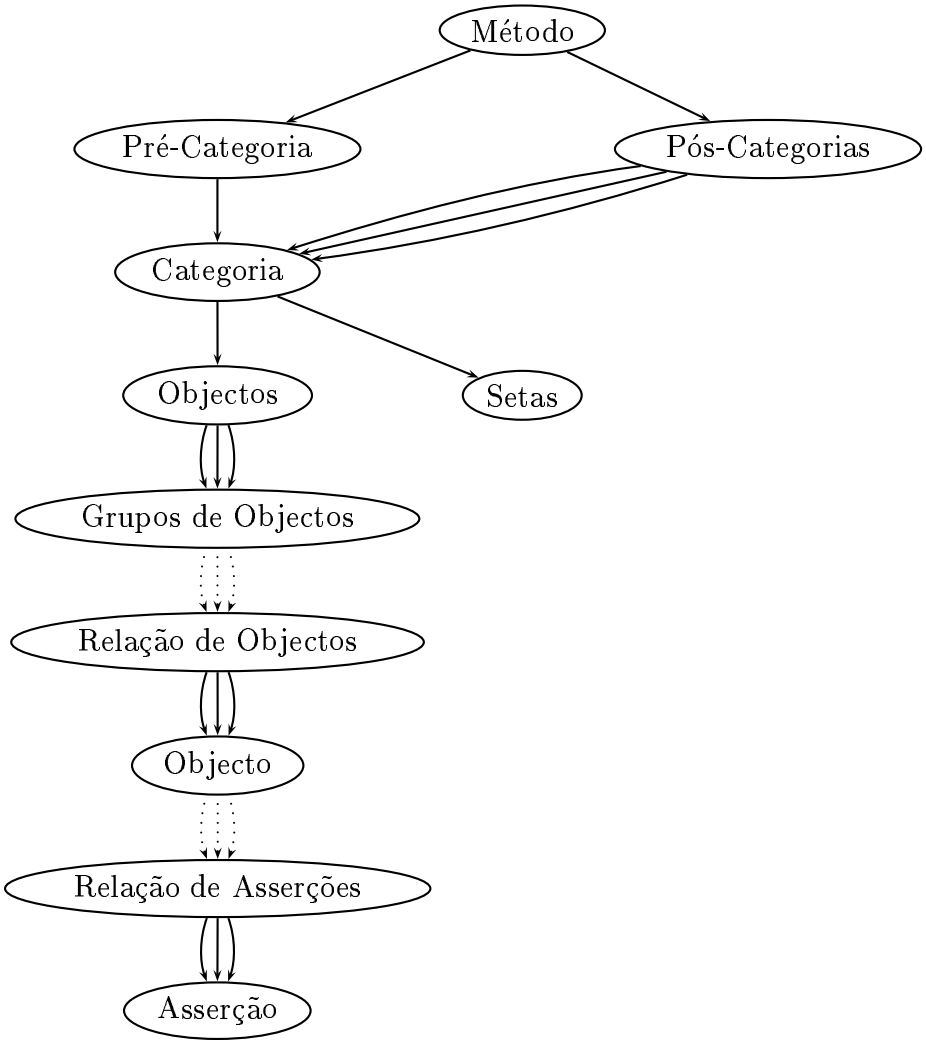

Figura 4.2: Interacção das funções

A interacção entre as funções de emparelhamento, anteriormente descritas, pode ser representada através do grafo da figura 4.2. Como se pode observar nesse grafo, os nós *Or* são criados aquando o emparelhamento de um grupo de objectos e de um objecto. A lista de argumentos são, respectivamente, as possíveis combinações entre os objectos do grupo e as asserções do objecto.
Os nós *And* são criados aquando o emparelhamento das pós-categorias, dos objectos, da relação dos objectos e da relação de asserções. A lista de argumentos são, respectivamente, todas as pós-categorias, todos os grupos de objectos, os objectos da relação e as asserções da relação.

4.1.3 Procura de Uma Solução

A secção 4.1.2 mostra como o conceito dos grafos *And/Or* foi utilizado para a representação do problema. Porém, em relação à procura da solução no grafo, não foi

possível utilizar nenhuma das técnicas de procura já desenvolvidas para este tipo de grafos. Isto, porque os emparelhamentos das várias categorias não são independentes, pois todos tem de respeitar uma renomeação comum. Desta forma, ao longo de toda a procura de uma solução no grafo tem de se ter em conta todas as combinações ainda não exploradas. Ou seja, quando se considera uma combinação e num dos passos seguintes não for possível emparelhar, tem de ser possível revogar todas as decisões tomadas a partir da escolha dessa combinação, escolher outra combinação e reiniciar a procura a partir desse ponto.
A função de renomeação vai sendo completada ao longo da procura através da escolha das combinações e nem sempre essas escolhas são as acertadas. Desta forma, tem de se garantir que é sempre possível retroceder na procura para um qualquer nó *Or*, com combinações ainda por explorar, e eliminar toda informação inserida na renomeação a partir desse nó.
Por exemplo, suponha-se que se emparelhou uma pós-categoria $\mathcal{C}$, mas não se consegue emparelhar uma outra. Nesse caso, tem de se retroceder até um nó *Or* pertencente ao emparelhamento de $\mathcal{C}$, revogar todas as decisões tomadas a partir desse nó, escolher uma outra combinação e reiniciar a procura a partir desse nó. Ou seja, $\mathcal{C}$ vai ser de novo emparelhada tendo em conta outra combinação.

Implementar o retrocesso da procura, anteriormente descrito, unicamente através de recursão era impossível. Isto porque nenhum nó do grafo poder ser definitivamente finalizado, até que toda a procura seja finalizada.
A solução foi criar uma pilha de funções, que indica a cada momento do algoritmo as funções que faltam ser chamadas para a conclusão da procura.
No inicio da procura, a pilha é inicializada vazia e é chamada a função **Método**.
Antes de se terminar uma qualquer função f, é chamada e retirada a primeira função g da pilha.
Se o valor devolvido por g indicar falha no emparelhamento, tem-se duas hipóteses: Se f for representada por um nó *Or* e existir mais combinações a explorar, então considera-se uma nova combinação e chama-se de novo g; Caso contrário, mete-se f na pilha e termina-se a execução de f devolvendo o valor de falha.
Se o valor devolvido por g indicar um emparelhamento, então termina-se f devolvendo o mesmo resultado.
A função **Método** insere na pilha as funções **Pós-categorias** e **Pré-categoria**, nesta mesma ordem. Também, a função **Categoria** insere na pilha as funções **Setas** e **Objectos**, nesta mesma ordem. As funções representadas por nós *And* utilizam a pilha para lá colocarem todas as chamadas que têm de fazer.

4.1.4 Combinações

Pelo que foi dito na secção 4.1.3, pode-se observar que uma combinação não é mais que uma função que relaciona os elementos de um conjunto com outro. Sabendo que os conjuntos são finitos, então essa função pode ser descrita através de um conjunto de pares. Desta forma, dado dois conjuntos $A = \{a_0, \ldots, a_n\}$ e $B = \{b_0, \ldots, b_m\}$, com $m \geqslant n$, pode-se descrever o conjunto das combinações de A com B, como sendo o conjunto resultante de relacionar um arranjo do conjunto A com todos os possíveis arranjos de tamanho n do conjunto B. O conjunto tem a seguinte forma:

$$\{\langle(a_0, b_0), \ldots, (a_n, b_n)\rangle, \ldots, \langle(a_0, b_m), \ldots, (a_n, b_{m-n})\rangle\}$$

Deste modo, o número total de combinações é dado por $\mathcal{A}_n^m = \frac{m!}{(m-n)!}$.
Pode-se então concluir que a complexidade do cálculo das combinações é exponencial a m [Knu73], mais precisamente $\mathcal{O}(m!)$.

Ao longo do processo, descrito na secção 4.1.2, os conjuntos considerados no cálculo de combinações vão ser: o conjunto das asserções de um objecto, e o conjunto dos objectos de um grupo. Assim, o valor máximo que m poderá tomar, é o valor máximo do tamanho desses dois conjuntos. Isto implica que é preciso dar uma especial atenção ao tamanho desses conjuntos.

Atenuação da exponencialidade

Esta secção, apresenta de seguida algumas técnicas desenvolvidas com o objectivo de tornar a complexidade do processo de emparelhamento ainda mais reduzida.

Cada grupo de objectos, formado no emparelhamento de uma categoria, é constituído por objectos com um igual número n_{ass} de asserções. Os grupos de objectos com n_{ass} elevado são geralmente mais pequenos, pois é pouco frequente ter-se implicações entre um grande número de asserções. Assim, como o emparelhamento destes grupos têm um número baixo de combinações a considerar, emparelha-se os grupos por ordem decrescente do seu valor de n_{ass}. Desta forma, quando os grupos com um menor valor de n_{ass} forem emparelhados, já a função de renomeação contêm informação que permite eliminar muitas das combinações.

Como, normalmente, os grupos de objectos com um valor de n_{ass} pequeno são grandes, calcular todas as suas combinações é incomportável, mesmo para uma especificação de média dimensão. A solução encontrada consiste em reduzir o tamanho dos grupos de objectos a emparelhar. Isso, é feito dividindo o grupo de objectos Obj_u nos dois grupos seguintes:

- $Obj_{u1} = \{F_{obj}(x) : x \in Obj_u \wedge F_{obj}(x)\downarrow\}$
- $Obj_{u2} = \{x : x \in Obj_u \wedge F_{obj}(x)\uparrow\}$

Desta forma, basta só verificar se $Obj_{u1} \subseteq Obj_r$ e emparelhar Obj_{u2} com $Obj_r \backslash Obj_{u1}$. A solução anterior foi também aproveitada para o emparelhamento dos conjuntos de asserções. Neste caso, o grupo de asserções Ass_u é divido, análogamente, nos dois grupos seguintes:

- $Ass_{u1} = \{R_a(x) : x \in Ass_u \wedge R_a(x)\downarrow\}$
- $Ass_{u2} = \{x : x \in Ass_u \wedge R_a(x)\uparrow\}$

Em seguida, basta só verificar se $Ass_{u1} \subseteq Ass_r$ e emparelhar Ass_{u2} com $Ass_r \backslash Ass_{u1}$.

Outra técnica usada consiste em só considerar as combinações viáveis, i.e., no cálculo das combinações possíveis usa-se um predicado que indica se dois elementos podem ser emparelhados. Este predicado permite eliminar muitas das combinações, pois no caso dos objectos só se combinam os que tiverem o mesmo tipo de asserções, e no caso das asserções só se combinam as que forem do mesmo tipo e não violem a informação já registada na função de renomeação.

4.1.5 Complexidade

Nesta secção apresenta-se o estudo da complexidade temporal, no pior caso, na identificação de um emparelhamento isomorfo entre duas especificações categoriais.
Para se calcular essa complexidade, tem-se de calcular a complexidade das funções que constituem o método de emparelhamento isomorfo. Deste modo, em seguida apresenta-se a análise da complexidade temporal de cada função descrita na secção 4.1.2:

A complexidade aqui apresentada depende das seguintes variáveis:

- *ass*, representa o número máximo de asserções num objecto.
- *grp*, representa o número máximo de objectos num grupo de objectos.
- *grps*, representa o número máximo de grupos de objectos.
- *obj*, representa o número máximo de objectos numa categoria.
- *set*, representa o número máximo de setas numa categoria.

- *pos*, representa o número máximo de pós-categorias numa especificação categorial.
- *mtds*, representa o número máximo de métodos por componente.
- *cpt*, representa o número máximo de componentes no repositório.

Setas Para se obter o conjunto Set_t tem de se percorrer cada seta de Set_u e traduzir as suas asserções. Deste modo, este passo tem complexidade $\mathcal{O}(set \times 2 \times ass) = \mathcal{O}(set \times ass)$. No passo seguinte, onde se verifica se um dos conjuntos de setas está contido no outro, temos uma complexidade de $\mathcal{O}(set^2)$. Desta forma, a complexidade total desta função é $\mathcal{O}(set \times ass + set^2)$.

Asserção Esta função limita-se a verificar a consistência entre as duas asserções dadas, logo a sua complexidade é $\mathcal{O}(1)$.

Relação de Asserções A função *Asserção* é chamada por cada par de asserções da relação, assim a complexidade desta função é $\mathcal{O}(ass)$.

Objecto Esta função identifica todas as combinações entre as asserções de dois objectos $\mathcal{O}(\mathcal{A}_{ass}^{ass}) = \mathcal{O}(ass!)$, e depois chama a função *Relação de Asserções* para cada uma dessas combinações. Desta forma, a complexidade desta função é $\mathcal{O}(ass! \times ass) = \mathcal{O}((ass+1)!)$.

Relação de Objectos A função *Objecto* é chamada por cada par de objectos da relação, assim a complexidade desta função é $\mathcal{O}(grp \times (ass+1)!)$.

Grupo de Objectos Esta função calcula as combinações entre os objectos dos dois grupos $\mathcal{O}(\mathcal{A}_{grp}^{grp}) = \mathcal{O}(grp!)$, e depois chama a função *Relação de Objectos* para cada uma dessas combinações. Desta forma, a complexidade desta função é $\mathcal{O}(grp! \times grp \times (ass+1)!) = \mathcal{O}((grp+1)! \times (ass+1)!)$.

Objectos Para agrupar os dois conjuntos de objectos temos de percorrer todos os objectos das duas categorias, logo a complexidade deste passo é $\mathcal{O}(2 \times obj) = \mathcal{O}(obj)$. Em seguida, é chamada a função *Grupo de Objectos* por cada par de grupos criado. Deste modo, as complexidade total desta função é $\mathcal{O}(obj + grps \times (grp+1)! \times (ass+1)!)$.

Categoria A função *Categoria* limita-se a chamar as funções *Objectos* e *Setas*, assim a sua complexidade é $\mathcal{O}(set \times ass + set^2 + obj + grps \times (grp+1)! \times (ass+1)!)$.

Pós-Categorias Por cada pós-categoria, no primeiro passo a complexidade de traduzir um objecto é $\mathcal{O}(ass)$, o segundo passo tem complexidade $\mathcal{O}(pos \times ass)$, e por ultimo é chamada a função *Categoria*. Desta forma, a complexidade total da função *Pós-Categorias* é $pos \times \mathcal{O}(ass + pos \times ass + set \times ass + set^2 + obj + grps \times (grp+1)! \times (ass+1)!) = pos \times \mathcal{O}(pos \times ass + set \times ass + set^2 + obj + grps \times (grp+1)! \times (ass+1)!)$.

Pré-Categoria A função *Pré-Categoria* limita-se a chamar a função *Categoria*, logo a complexidade das duas funções é igual.

Método Esta função chama a função *Pré-categoria*, e depois chama a função *Pós-categorias*. Facilmente se conclui que a complexidade de chamar a função *Pré-categoria* é um termo menor da complexidade total. Deste modo, a complexidade de emparelhar dois métodos é

$$\mathcal{O}(Iso) = pos \times \mathcal{O}(pos \times ass + set \times ass + set^2 + obj + grps \times (grp+1)! \times (ass+1)!) \quad (4.1)$$

A complexidade de emparelhar isomorfamente um método do utilizador é a complexidade de o emparelhar com todos os métodos do repositório. Como o número de métodos no repositório é dado por $mtds \times cpt$, temos a seguinte complexidade:

$$\mathcal{O}(Iso) \times mtds \times cpt$$

Pode-se observar que a complexidade só é exponencial em relação a ass e a grp, que são valores quase sempre baixos mesmo para especificações mais complexas, ao contrário dos valores de obj e set que aumentam exponencialmente relativamente à complexidade das especificações.

A aplicação das técnicas descritas na secção 4.1.4 funcionam como heurísticas que minimizam o impacto do valor de ass e grp.

4.2 Emparelhamento Composicional

O emparelhamento composicional consiste no emparelhamento da especificação categorial do utilizador com uma sequência de especificações categoriais do repositório, de tal modo que todas as pós-categorias verificam a pré-categoria da seguinte especificação categorial nessa sequência.

O tamanho das sequências consideradas é irrelevante para a construção do método, mas para a sua utilização é um factor muito importante. Considerar todas as sequências de tamanho arbitrário torna o processo impraticável, devido ao aumento exponencial da sua complexidade temporal. Desta forma, torna-se quase que obrigatório limitar o tamanho da sequência. Neste trabalho o tamanho da sequência está limitada a apenas duas especificações categoriais. primeira especificação categorial da sequência dá-se o nome de inicial e à segunda dá-se o nome de final.

Nesta secção são usados os prefixos ir, fr e u para representar, respectivamente, os elementos da especificação categorial inicial, final e do utilizador.

4.2.1 Revisão da Teoria Proposta

Tal como é referido no artigo [Cre98a], para desenvolver um método de emparelhamento composicional, temos primeiro de decidir onde é aceitável o uso de variáveis de interface.
A solução apresentada nesse artigo propõe a eliminação das variáveis de saída das pós-categorias da especificação categorial inicial, e não considera especificações categoriais finais que contenham variáveis de entrada.
Como neste trabalho, no âmbito de um método, uma variável de estado é representada por uma variável de entrada e outra de saída, então permite-se o uso de variáveis de entrada na especificação categorial final sse esta corresponder a uma variável de saída da especificação categorial inicial.

Mesmo com a limitação de só termos sequências de tamanho dois, se considerarmos todas as combinações das especificações categoriais duas a duas, ficamos de novo com um processo impraticável. A solução aqui apresentada consiste em apenas considerar as sequências de especificações categoriais todas do mesmo componente. Esta solução foi tomada porque diminui em grande escala o número de sequências a considerar, e porque a probabilidade de emparelhamento de sequências com especificações categoriais de diversos componentes ser muito baixa.
Esta solução têm ainda outra vantagem, porque deixa de ser necessário a identificação de functores entre as especificações categoriais de uma sequência. Este facto torna o método muito mais eficaz, sem grande perca de generalidade.

4.2.2 Construção da Sequência

O objectivo na construção de uma sequência de especificações categoriais é obter uma só especificação categorial que representa a composição das especificações categoriais da sequência. A essa especificação categorial dá-se o nome de especificação

categorial *composta*.
A especificação categorial composta é depois utilizada conjuntamente com a especificação categorial do utilizador no método de emparelhamento isomorfo, para verificar se representam comportamento equivalentes.

A ideia base para obter a especificação categorial composta consiste em ligar cada pós-categoria da especificação categorial inicial a um objecto da pré-categoria da especificação categorial final, que represente um antecedente, i.e. que induza uma pós-categoria. Desta forma, todos os objectos da pré-categoria da especificação categorial inicial, que representem antecedentes, ficam ligados a uma pós-categoria da especificação categorial final.

O algoritmo desenvolvido para obter a especificação categorial composta pode ser descrito da seguinte forma:

Composta
Entrada: As especificações categoriais inicial e final.
Saída: A respectiva especificação categorial composta.

1. A pré-categoria da especificação categorial composta é a pré-categoria da especificação categorial inicial.

2. As pós-categorias da especificação categorial composta são as pós-categorias da especificação categorial final.

3. Por cada par $(obj_{ir}, posctg_{ir})$ objecto, pós-categoria, da especificação categorial inicial:

   (a) Procura-se um par $(obj_{fr}, posctg_{fr})$ nos pares objecto, pós-categoria da especificação categorial final, tal que $posctg_{ir}$ *verifique* obj_{fr}. Se o par não for encontrado, a composição falha.

   (b) Cria-se o par $(obj_{ir}, posctg_{fr})$ na especificação categorial composta.

Como se pode observar no algoritmo é introduzido o conceito de verificação. O conceito de verificação de um objecto por uma categoria, consiste em encontrar um morfismo do objecto terminal para o objecto, na categoria em causa. Caso o objecto represente uma conjunção então tem-se de encontrar os morfismos do objecto terminal para os objectos que representam cada uma das asserções do objecto.
Como as especificações são deterministas, cada $posctg_{ir}$ só pode verificar um e só um obj_{fr}, ou seja, não existe ambiguidade na identificação das ligações entre a pré-categoria da especificação categorial inicial e as pós-categorias da especificação categorial final.

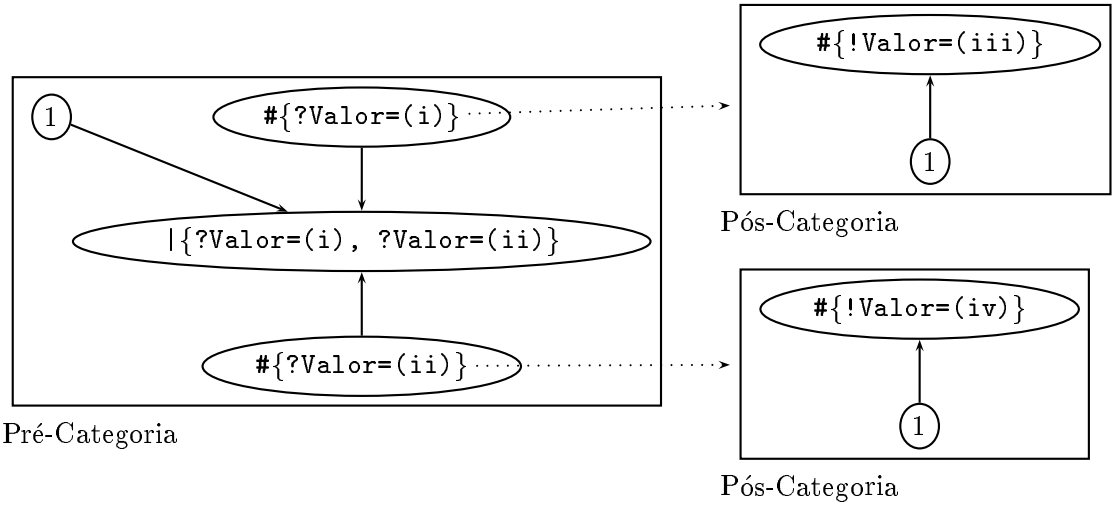

Figura 4.3: Especificação categorial inicial

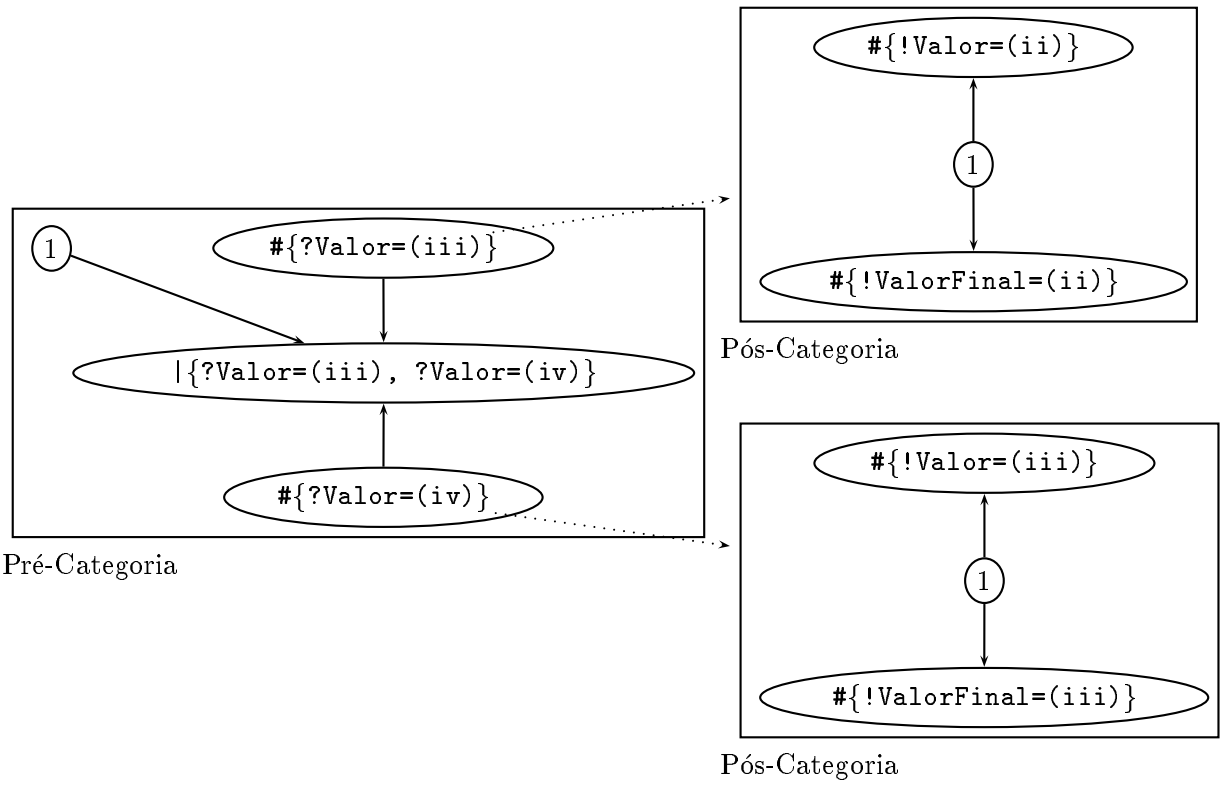

Figura 4.4: Especificação categorial final

Exemplo 4.1 *Neste exemplo é apresentado o caso de um emparelhamento composicional. Para especificação categorial do utilizador escolheu-se a especificação categorial apresentada na figura 3.1, para especificação categorial inicial e final foram escolhidas, respectivamente, as especificações categoriais apresentadas na figura 4.3 e na figura 4.4.*

Como se pode observar as especificações categoriais inicial e final formam uma sequência válida, isto porque todas as pós-categorias da especificação categorial inicial verificam a pré-categoria da especificação categorial final. Esta propriedade é verificada devido aos seguintes factos:

- *O objecto* `#{?Valor=(i)}` *na pré-categoria da especificação categorial inicial induz a pós-categoria superior, que por sua vez contém um morfismo do objecto terminal para o objecto* `#{!Valor=(iii)}`*. Na pré-categoria da especificação categorial final esse objecto induz a pós-categoria superior. Assim, por transitividade o objecto* `#{?Valor=(i)}` *induz a pós-categoria superior da especificação categorial final na especificação categorial composta.*
- *O objecto* `#{?Valor=(ii)}` *na pré-categoria da especificação categorial inicial induz a pós-categoria inferior, que por sua vez contém um morfismo do objecto terminal para o objecto* `#{!Valor=(iv)}`*. Na pré-categoria da especificação categorial final esse objecto induz a pós-categoria inferior. Assim, por transitividade o objecto* `#{?Valor=(ii)}` *induz a pós-categoria inferior da especificação categorial final na especificação categorial composta.*

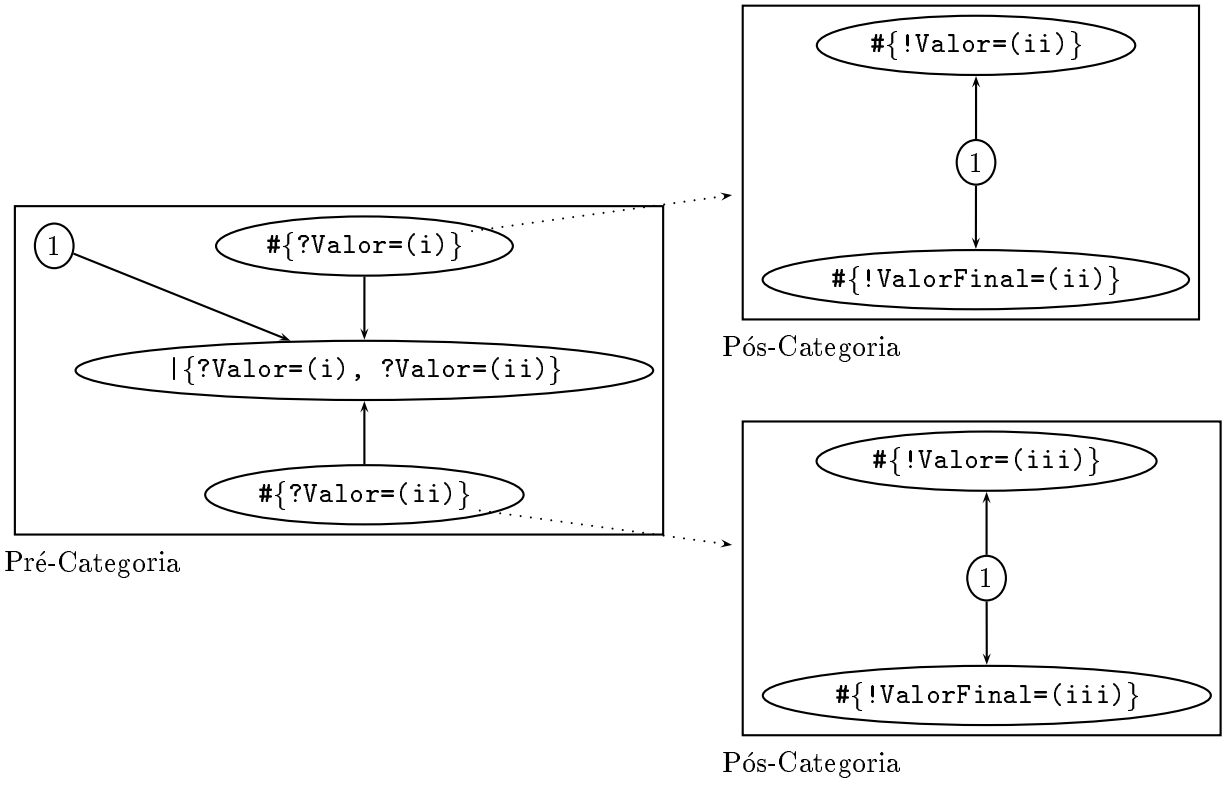

Figura 4.5: Especificação categorial composta

Desta forma, a composição das duas especificações categoriais resulta na especificação categorial composta apresentada na figura 4.5.

Finalmente, a especificação categorial do utilizador pode ser emparelhada isomorfamente com a especificação categorial composta considerando a seguinte renomeação R:

$$\begin{aligned} R_x = &\ \{1 \mapsto \texttt{i}, 2 \mapsto \texttt{ii}, 3 \mapsto \texttt{iii}\} \\ R_p = &\ \{\texttt{?Estado=} \mapsto \texttt{?Valor=}, \texttt{!Estado=} \mapsto \texttt{!Valor=}, \\ &\ \texttt{!EstadoFinal=} \mapsto \texttt{!ValorFinal=}\} \end{aligned}$$

4.2.3 Complexidade

Nesta secção descreve-se a análise da complexidade temporal do método de emparelhamento composicional.

A complexidade aqui apresentada depende das seguintes variáveis:

- ass, representa o número máximo de asserções numa cláusula.
- set, representa o número máximo de setas numa categoria.
- pos, representa o número máximo de pós-categorias numa especificação categorial.
- $mtds$, representa o número máximo de métodos por componente.
- cpt, representa o número máximo de componentes no repositório.

Para construir uma especificação categorial composta é necessário por cada pós-categoria da especificação categorial inicial percorrer as setas da pré-categoria final para encontrar uma que verifique o antecedente da pós-categoria em causa. Desta forma, a complexidade deste passo é $\mathcal{O}(pos \times set \times ass)$.

A complexidade do emparelhamento isomorfo entre dois métodos é $\mathcal{O}(Iso)$ (equação 4.1), e como as sequências de especificações categoriais são de tamanho dois e todas do mesmo componente, ficamos com a seguinte complexidade para o emparelhamento composicional de um método do utilizador: $(\mathcal{O}(pos \times set \times ass) + \mathcal{O}(Iso)) \times mtds^2 \times cpt$,
Observando o valor de $\mathcal{O}(Iso)$ podemos constatar que a complexidade de construir a especificação categorial composta é um termo menor, logo a complexidade do emparelhamento composicional é

$$\mathcal{O}(Iso) \times mtds^2 \times cpt$$

Como se pode verificar a complexidade do emparelhamento composicional não é exponencial a qualquer um dos factores.

4.3 Implementação

4.3.1 Linguagem

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* ou o *C*.
O *SML* utiliza expressões para denotar entidades matemáticas, em vez de definir as transições de uma máquina abstracta. Estes factos são importantes porque neste trabalho são utilizados dois tipos de especificações baseadas em fundamentos matemáticos.

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 podem ser encontradas no seguinte endereço da *Internet*:
`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 da aplicação desenvolvida poder ser utilizado na maioria das máquinas e sistemas operativos, embora esta tenha sido desenvolvida usando o sistema *Linux*.
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.

4.3.2 Testes

Para a verificar o funcionamento da aplicação informática produzida foi necessário identificar diversas especificações algébricas que serviram de dados de entrada para a aplicação.

No apêndice B são apresentadas duas especificações: *Interruptor* e *Triplo Interruptor*.
A especificação *Interruptor* foi retirada do artigo [Cre98a], e define o comportamento das operações básicas de um interruptor.

A especificação *Triplo Interruptor* descreve o funcionamento de três interruptores. As suas operações não são as comuns, isto porque o objectivo deste exemplo foi preencher o vazio criado pela inexistência de especificações algébricas que fizessem uso dos quantificadores de uma forma lógica e abrangente.
No mesmo apêndice apresenta-se também as especificações categoriais obtidas, pela aplicação informática, a partir das especificações descritas anteriormente. O aspecto que mais se destaca ao consultar ambas as especificações, é o facto de as especificações categoriais serem menos compreensíveis. Tal como foi dito, este era um resultado esperado devido à simplicidade sintáctica das especificações categoriais.

Por forma, a testar eficazmente os algoritmos de emparelhamento, a ordem de todas as listas da especificação categorial do utilizador são invertidas. Assim, no caso do emparelhamento entre duas especificações categoriais iguais obriga-se o algoritmo a considerar várias combinações até obter um emparelhamento, i.e., a retroceder várias vezes na procura.

Capítulo 5

Conclusões

Neste trabalho foram revistos e adaptados os métodos, propostos na literatura, para a classificação de especificações algébricas e selecção de especificações categoriais. Alguns desses métodos foram mesmo totalmente substituídos por outros métodos mais eficientes, que foram desenvolvidos ao longo deste projecto.
Da implementação desses métodos resultou uma aplicação informática que automatiza o processo de reutilização de *software*, i.e., que automaticamente classifica e selecciona componentes reutilizáveis.
Foram também identificados novos métodos, que apesar de não terem sido implementados, permitem assegurar a possibilidade de resolução de algumas das limitações da aplicação.

Tal como o previsto, este trabalho foi constituído por três fases. As conclusões obtidas em cada uma destas fases são apresentadas nas seguintes secções.

5.1 Classificação

Na fase de desenvolvimento do processo *Classificação* foram feitas alterações significativas ao conceitos teóricos propostos, das quais se destacam as seguintes:

- Uma nova solução para a representação dos quantificadores, que permite uma maior flexibilidade e eficácia. Nesta solução, as proposições de uma especificação categorial são substituídas por asserções.
- As fórmulas são expressas através de afirmações clausais, o que permite uma maior eficácia do método.
- Identificação de predicados que têm um significado próprio e que não pode ser ignorado.

- Existência de informação explicita e implícita numa especificação categorial, o que permite diminuir o espaço ocupado por uma especificação categorial.
- Na classificação de um método, as variáveis consideradas são todas aquelas que são realmente usadas pelo método. Desta forma, abre-se mais hipóteses para a identificação de componentes reutilizáveis.

O estudo da complexidade dos métodos desenvolvidos revelou ser polinomial, o que faz com que esta abordagem seja viável. Neste estudo, foi calculado a complexidade temporal e espacial, porque é importante conhecer a ordem de grandeza do tempo de execução dos métodos e do espaço de dados da especificação categorial produzida, para concluirmos sobre a escalabilidade dos métodos.

Nesta fase foram também identificadas representações computacionais para as especificações algébricas e para as especificações categoriais. Uma das suas principais características é que podem ser armazenadas através de uma simples cadeia de caracteres, o que aumenta a portabilidade da aplicação e facilita a compreensão das especificações manipuladas.

O repositório de especificações é controlado pelo processo *Gestão*. Este processo foi desenvolvido a partir de um conjunto de regras que definem o seu funcionamento. Estas regras foram definidas com o objectivo de criar um processo simples e eficaz na manipulação das especificações.

5.2 Selecção

O processo *Selecção* foi implementado através de duas metodologias de emparelhamento de especificações categoriais.

5.2.1 Emparelhamento Isomorfo

O objectivo do emparelhamento isomorfo é encontrar uma renomeação entre os elementos do conjunto de suporte e os predicados da especificação categorial do utilizador e de uma especificação categorial do repositório.

Identificar essa renomeação por procura exaustiva não era certamente uma boa solução, devido à exponencialidade da sua complexidade. Por esta razão, foi desenvolvido um método de construção progressiva para identificação de uma renomeação. Este método baseia-se no conceito de grafos *And/Or* para representar o problema, e onde é utilizado um método que foi desenvolvido propositadamente para identificar eficazmente uma possível solução.

Esta metodologia de emparelhamento permite que o conjunto de suporte possa ser infinito, desde que sejam criadas regras que permitam renomear os seus elementos.

A análise da complexidade do emparelhamento isomorfo revelou ser exponencial ao número máximo de asserções num objecto, e ao número máximo de objectos num grupo. Este facto é tolerado porque estes valores tem um factor de crescimento muito pequeno, e também porque foi criado um conjunto de técnicas que permitem reduzir o seu impacto.

5.2.2 Emparelhamento Composicional

O emparelhamento composicional consiste em identificar um emparelhamento isomorfo entre a especificação categorial do utilizador e um par de especificações categoriais do repositório, desde que esse par possa induzir uma especificação categorial composta. As principais diferenças do método desenvolvido para o que era proposto na literatura são as seguintes:

- A especificação categorial final pode conter variáveis de entrada desde que estas correspondam a variáveis de saída da especificação categorial inicial, o que permite uma maior abrangência do método.
- As especificações categoriais inicial e final são obrigatoriamente do mesmo componente, pois, caso contrário, considerar todas as outras combinações seria computacionalmente muito despendioso, e a probabilidade de obter uma especificação categorial composta de duas especificações categoriais de componentes distintos é muito menor.
- É introduzida a noção de verificação que permite obter as ligações entre as especificações categoriais iniciais e finais.

A complexidade deste método está directamente relacionada com a complexidade do emparelhamento isomorfo. Essa relação é polinomial permitindo que este método seja perfeitamente viável.

Implementação

Em relação à linguagem e ferramentas, utilizadas neste trabalho, foi necessário perder bastante tempo na sua aprendizagem. Por exemplo, foi despendido muito tempo a descobrir como se podia utilizar ambos os analisadores léxico e sintáctico no mesmo processo e a cooperarem entre si. A razão para esta demora reside no facto da

documentação das ferramentas ser pouco completa e com poucos exemplos práticos. Apesar disso, o trabalho recuperou todo esse tempo na fase de implementação da aplicação informática, pois a tradução dos algoritmos desenvolvidos para código foi uma tarefa facilitada pelo nível de abstracção da linguagem.

Por forma a testar a correcção da aplicação informática produzida, foram identificadas diversas especificações algébricas que serviram de dados de entrada para a aplicação. De todas elas, destaca-se a especificação que descreve o funcionamento de um triplo interruptor. Esta especificação preencheu o vazio criado pela inexistência de especificações algébricas que fizessem uso dos quantificadores de uma forma lógica e abrangente.

5.3 Extensões

No intuito de confirmar o facto de ser possível resolver as limitações inerentes à linguagem das especificações usadas, foram desenvolvidos algumas técnicas que quando implementadas eliminarão as limitações consideradas.

A primeira dessas técnicas propõe-se a resolver a restrição da aplicação dos quantificadores a um só predicado. Esta técnica têm como ideia base a introdução de setas especiais na especificação categorial dos quantificadores. Por seta especial entenda-se uma seta que têm a capacidade de se multiplicar em diversas setas, todas com uma característica comum.

Outra das técnicas desenvolvidas consiste no uso de uma pré-categoria para representar cada antecedente de uma pós-condição. Isto, permite a que os antecedentes e a pré-condições possam ser expressas da mesma forma que os consequentes das pós-condições.

5.4 Contribuições

De entre as diversas contribuições resultantes deste projecto, destacam-se as seguintes:

- Foram validados e revistos os conceitos teóricos publicados acerca do tema deste projecto.

- Novos e mais eficientes métodos e algoritmos, de classificação e selecção dos componentes reutilizáveis foram desenvolvidos.

- Analisou-se a complexidade dos métodos adoptados, o que permitiu demonstrar a viabilidade da solução seguida.

- Produziu-se uma aplicação informática que automatiza o processo de classificação e selecção dos componentes reutilizáveis, através dos métodos criados neste projecto. Fazem parte desta aplicação um conjunto de exemplos de especificações algébricas que podem ser tratadas pela aplicação. A aplicação está disponível no seguinte endereço de *Internet*:
  `http://www.math.ist.utl.pt/~fcouto/tese`

- Novas técnicas foram desenvolvidas por forma a extender a linguagem das especificações usadas.

- Sobre este projecto foi realizada uma apresentação com o título:
  "The Development of a Computer Application that Identifies Reusable Components through Formal Specifications"
  First International Workshop on Automated Verification of Infinite-State Systems (AVIS'01), co-located with FME 2001[1].
  12 de Março de 2001 Humboldt-Universität zu Berlin Alemanha

5.5 Trabalho Futuro

Os objectivos das próximas fases deste projecto irão englobar os seguintes aspectos:

- Publicação de um artigo científico sobre os resultados alcançados.

- Implementação das técnicas desenvolvidas para a extensão da linguagem das especificações consideradas.

- Desenvolvimento de novas técnicas que eliminem as restantes limitações da linguagem das especificações consideradas. Por exemplo, permitir o uso de predicados com aridade superior a um.

- Aperfeiçoar os métodos de emparelhamento, tornando-os mais abrangentes e eficientes. Isso, pode ser alcançado através: da implementação de mecanismos de inferência que manipulam a informação implícita nas especificações categoriais; da identificação de heurísticas para o processo de emparelhamento; etc....

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

- Aplicação deste projecto a um caso de estudo, onde será possível demonstrar todas as suas potencialidades.

Índice

Bibliografia

[BCN92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Data Base Design: An Entity-Relationship Approach.* Benjamin Cummings, Redwood City, California, 1992.

[BM77] John Bell and Moshé Machover. *A course in Mathematical Logic.* North-Holland, 1977.

[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 retrieval 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.

[Gol86] R. Goldblatt. *Topoi, the categorical analysis of logic*, volume 98. North Holland, 1986.

[Gri93] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[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.

[Knu73] Donald E. Knuth. *Fundamental Algorithms.* Addison-Wesley, 1973.

[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.

[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.

[MR99] Douglas C. Montgomery and George C. Runger. *Applied statistics and Probability for Engineers.* John Wiley & Sons, 1999.

[MS92] N. A. Maiden and A. G. Sutcliffe. Exploiting 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.

[Pea85] Judea Pearl. *Heuristics : intelligent search strategies for computer problem solving.* Addison-Wesley, 1985.

[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.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[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.

[Wir88] M. Wirsing. Algebraic description of reusable software components. Report MIP–8816, Universität Passau, 1988.

[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 Especificações Usadas

A.1 Especificações Algébricas

```
<specification>  ::=  Component <id>
                      Sort:{<constants>}
                      Variables:<vars>
                      Invariant:<clauses>
                      {<method>}
    <constants>  ::=  <constant>{,<constant>}
         <vars>  ::=  <var>{,<var>}
       <method>  ::=  Method:<id>
                      Interface:<interfaces>
                      Requires:<preconditions>
                      Ensures:<postconditions>
   <interfaces>  ::=  [<interface>{,<interface>}]
    <interface>  ::=  ?<var> | !<var>
<preconditions>  ::=  <disjunction>
                      {,<disjunction>}
<postconditions>  ::=  {<postcondition>;}
 <postcondition>  ::=  <antecedent>-><consequent>
   <antecedent>  ::=  <conjunction>
   <consequent>  ::=  <clauses>
      <clauses>  ::=  <clause>
                      {,<clause>}
       <clause>  ::=  <conjunction>=><disjunction>
  <conjunction>  ::=  <pseudo-proposition>
                      {&<pseudo-proposition>}
  <disjunction>  ::=  <pseudo-proposition>
                      {|<pseudo-proposition>}
```

```
<pseudo-proposition>  ::=  >&lt;var&gt;:&lt;id&gt;(&lt;var&gt;) |
                           <<var>:<id>(<var>) |
                           <id>(<constant>) |
                           <variable>=<constant> |
                           true | false
          <variable>  ::=  <var> | ?<var> | !<var>
               <var>  ::=  <id>
          <constant>  ::=  <id>
                <id>  ::=  <character>{<character> | <digit>}
         <character>  ::=  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
             <digit>  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

O caracter % pode ser usado para colocar comentários.

A.2 Especificações Categoriais

```
<categ.rep.>  ::=  Categorical Representation
                   Component <id>
                   Sort:{<constants>}
                   {<method>}
     <method>  ::=  Method:<id>
                   <precategory>
                   {<postcategory>}
 <precategory>  ::=  PreCategory:<category>
<postcategory>  ::=  PostCategory:
                   <object>→<category>
    <category>  ::=  <objects><arrows>
     <objects>  ::=  Objects:{<object>}
      <arrows>  ::=  Arrows:{<arrow>}
       <arrow>  ::=  <object>⇒<object>
      <object>  ::=  0 | 1 | <singular> |
                   <minimum> | <maximum>
    <singular>  ::=  #{<assertion>}
     <minimum>  ::=  &{<assertions>}
     <maximum>  ::=  |{<assertions>}
  <assertions>  ::=  <assertion>{,<assertion>}
   <assertion>  ::=  <predicate>(<constant>) |
                   +<predicate> |
                   *<predicate>
   <predicate>  ::=  <id> | <variable>=
```

<constants>, <variable>, <constant> e <id> estão descritos na secção A.1.

Apêndice B

Exemplos de Especificações

B.1 Interruptor

B.1.1 Especificação Algébrica

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Switch Component
%
% Algebraic Specification
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Component Switch
Sort: {On, Off, Usv}
Variables: Status
Invariant: false => true

Method: Init
Interface: ?InitialStatus
Requires:
        ?InitialStatus=On | ?InitialStatus=Off
Ensures:
?InitialStatus=On ->
   true => Status=On;
?InitialStatus=Off ->
   true => Status=Off;

Method: Toggle
Interface: ?InitialStatus
Requires:
        ?InitialStatus=On | ?InitialStatus=Off
Ensures:
?InitialStatus=On ->
   true => Status=Off;
?InitialStatus=Off ->
   true => Status=On;

Method: SetAndDisplay
Interface: !Value, ?InitialStatus
Requires: true
Ensures:
?InitialStatus=On ->
   true => !Value=On,
   true => Status=On;
?InitialStatus=Off ->
   true => !Value=Off,
   true => Status=Off;
?InitialStatus=Usv ->
   true => !Value=Usv,
   true => Status=Usv;
```

```
Method: SwitchOff
Interface:
Requires: Status=On
Ensures:
Status=On ->
        true => Status=Off;

Method: Discontinue
Interface:
Requires: Status=Off
Ensures:
Status=Off ->
        true => Status=Usv;
```

B.1.2 Especificação Categorial

```
Categorical Representation
Component Switch


Sort: {Usv,Off,On}

Method: Discontinue

        PreCategory:
        Objects:
1
0
#{?Status=(Off)}
        Arrows:
1 => #{?Status=(Off)}
#{?Status=(Off)} => #{?Status=(Off)}
0 => 1

        PostCategory:
#{?Status=(Off)} ->
        Objects:
1
0
#{!Status=(Usv)}
        Arrows:
1 => #{!Status=(Usv)}
0 => 1

Method: SwitchOff

        PreCategory:
        Objects:
1
0
#{?Status=(On)}
        Arrows:
1 => #{?Status=(On)}
#{?Status=(On)} => #{?Status=(On)}
0 => 1

        PostCategory:
#{?Status=(On)} ->
        Objects:
1
0
#{!Status=(Off)}
        Arrows:
1 => #{!Status=(Off)}
0 => 1

Method: SetAndDisplay

        PreCategory:
        Objects:
```

```
1
0
#{?InitialStatus=(On)}
#{?InitialStatus=(Usv)}
#{?InitialStatus=(Off)}
        Arrows:
1 => 1
#{?InitialStatus=(Off)} => 1
#{?InitialStatus=(Usv)} => 1
#{?InitialStatus=(On)} => 1
0 => 1

        PostCategory:
#{?InitialStatus=(Usv)} ->
        Objects:
1
0
#{!Status=(Usv)}
#{!Value=(Usv)}
        Arrows:
1 => #{!Value=(Usv)}
1 => #{!Status=(Usv)}
0 => 1

        PostCategory:
#{?InitialStatus=(Off)} ->
        Objects:
1
0
#{!Status=(Off)}
#{!Value=(Off)}
        Arrows:
1 => #{!Value=(Off)}
1 => #{!Status=(Off)}
0 => 1

        PostCategory:
#{?InitialStatus=(On)} ->
        Objects:
1
0
#{!Status=(On)}
#{!Value=(On)}
        Arrows:
1 => #{!Value=(On)}
1 => #{!Status=(On)}
0 => 1

Method: Toggle

        PreCategory:
        Objects:
1
0
#{?InitialStatus=(On)}
#{?InitialStatus=(Off)}
|{?InitialStatus=(Off),?InitialStatus=(On)}
        Arrows:
1 => |{?InitialStatus=(Off),?InitialStatus=(On)}
#{?InitialStatus=(Off)} => |{?InitialStatus=(Off),?InitialStatus=(On)}
#{?InitialStatus=(On)} => |{?InitialStatus=(Off),?InitialStatus=(On)}
0 => 1

        PostCategory:
#{?InitialStatus=(Off)} ->
        Objects:
1
0
#{!Status=(On)}
        Arrows:
1 => #{!Status=(On)}
```

```
0 => 1

        PostCategory:
#{?InitialStatus=(On)} ->
        Objects:
1
0
#{!Status=(Off)}
        Arrows:
1 => #{!Status=(Off)}
0 => 1

Method: Init

        PreCategory:
        Objects:
1
0
#{?InitialStatus=(On)}
#{?InitialStatus=(Off)}
|{?InitialStatus=(Off),?InitialStatus=(On)}
        Arrows:
1 => |{?InitialStatus=(Off),?InitialStatus=(On)}
#{?InitialStatus=(Off)} => |{?InitialStatus=(Off),?InitialStatus=(On)}
#{?InitialStatus=(On)} => |{?InitialStatus=(Off),?InitialStatus=(On)}
0 => 1

        PostCategory:
#{?InitialStatus=(Off)} ->
        Objects:
1
0
#{!Status=(Off)}
        Arrows:
1 => #{!Status=(Off)}
0 => 1

        PostCategory:
#{?InitialStatus=(On)} ->
        Objects:
1
0
#{!Status=(On)}
        Arrows:
1 => #{!Status=(On)}
0 => 1
```

B.2 Triplo Interruptor

B.2.1 Especificação Algébrica

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Switches Component
%
% Algebraic Specification
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Component Switches
Sort: {On, Off, Usv}
Variables: switch1, switch2, switch3
Invariant:

        % One, tells if exists one
        % switch with the value given.
One(On) => switch1=On | switch2=On | switch3=On,
One(Off) => switch1=Off | switch2=Off | switch3=Off,
One(Usv) => switch1=Usv | switch2=Usv | switch3=Usv,
```

```
        % Definicao do predicado All
        % All, tells if all the
        % switches have the value given.
        % Attention: the proposition All(Usv) isn't
        % defined, so it's false.
switch1=On & switch2=On & switch3=On => All(On),
switch1=Off & switch2=Off & switch3=Off => All(Off)


Method: Init
Interface: ?InitSwitch
Requires:
        ?InitSwitch=On | ?InitSwitch=Off,
        switch2=On
Ensures:
?InitSwitch=On & switch2=On ->
   true => switch1=On,
   true => switch2=On,
   true => switch3=On;
?InitSwitch=Off & switch2=On ->
   true => switch1=Off,
   true => switch2=Off,
   true => switch3=Off;

        % If all the switches have the same value
        % it returns this value; else it returns Usv
        % if all the switches have different values.
Method: DisplayAll
Interface: !value
Requires: <x:All(x) | >x:One(x)
Ensures:
All(On) ->
   true => !value=On;
All(Off) ->
   true => !value=Off;
>x:One(x) ->
   true => !value=Usv;

        % switch2 gets the value from ?s2;
        % and put all the switches with different value;
        % and returns the final value of switch1.
Method: PutAllTypes
Interface: ?s2, !s1
Requires: <x:All(x)
Ensures:
<x:All(x) ->
   switch1=On => !s1=On,
   switch1=Off => !s1=Off,
   switch1=Usv => !s1=Usv,

   ?s2=On => switch2=On,
   ?s2=Off => switch2=Off,
   ?s2=Usv => switch2=Usv,

   true => >x:One(x);
```

B.2.2 Especificação Categorial

```
Categorical Representation
Component Switches


Sort: {Usv,Off,On}

Method: PutAllTypes

        PreCategory:
        Objects:
1
```

```
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{*All}
|{+All}
#{All(On)}
#{One(Off)}
|{?switch3=(On),?switch2=(On),?switch1=(On)}
|{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
|{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)}
        Arrows:
#{One(Off)} => |{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)} => #{All(On)}
1 => |{+All}
#{*All} => |{+All}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
#{One(On)} => |{?switch3=(On),?switch2=(On),?switch1=(On)}

        PostCategory:
#{*All} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{!s1=(Off)}
#{!switch1=(Off)}
#{!switch2=(On)}
#{?s2=(On)}
#{!switch2=(Usv)}
#{?s2=(Usv)}
#{*One}
#{?s2=(Off)}
#{!switch2=(Off)}
#{!switch1=(Usv)}
#{!s1=(Usv)}
#{!switch1=(On)}
#{!s1=(On)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
#{!switch1=(On)} => #{!s1=(On)}
#{!switch1=(Usv)} => #{!s1=(Usv)}
#{?s2=(Off)} => #{!switch2=(Off)}
1 => #{*One}
#{?s2=(Usv)} => #{!switch2=(Usv)}
#{?s2=(On)} => #{!switch2=(On)}
#{!switch1=(Off)} => #{!s1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}

Method: DisplayAll

        PreCategory:
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{All(On)}
```

```
#{One(Off)}
&{+One}
|{*One,+All}
|{?switch3=(On),?switch2=(On),?switch1=(On)}
|{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
|{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)}
        Arrows:
#{One(Off)} => |{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)} => #{All(On)}
1 => |{*One,+All}
#{All(Off)} => |{*One,+All}
&{+One} => |{*One,+All}
#{All(On)} => |{*One,+All}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
#{One(On)} => |{?switch3=(On),?switch2=(On),?switch1=(On)}

        PostCategory:
&{+One} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{!value=(Usv)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
1 => #{!value=(Usv)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}

        PostCategory:
#{All(Off)} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{!value=(Off)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
1 => #{!value=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}

        PostCategory:
#{All(On)} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
```

```
#{All(Off)}
#{!value=(On)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
1 => #{!value=(On)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}

Method: Init

        PreCategory:
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{?switch2=(On)}
#{All(On)}
#{One(Off)}
|{?InitSwitch=(Off),?InitSwitch=(On)}
&{?switch2=(On),?InitSwitch=(Off)}
&{?switch2=(On),?InitSwitch=(On)}
|{?switch3=(On),?switch2=(On),?switch1=(On)}
|{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
|{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)}
        Arrows:
#{One(Off)} => |{?switch3=(Off),?switch2=(Off),?switch1=(Off)}
&{?switch3=(On),?switch2=(On),?switch1=(On)} => #{All(On)}
1 => |{?InitSwitch=(Off),?InitSwitch=(On)}
&{?switch2=(On),?InitSwitch=(On)} => |{?InitSwitch=(Off),?InitSwitch=(On)}
&{?switch2=(On),?InitSwitch=(On)} => #{?switch2=(On)}
&{?switch2=(On),?InitSwitch=(Off)} => #{?switch2=(On)}
&{?switch2=(On),?InitSwitch=(Off)} => |{?InitSwitch=(Off),?InitSwitch=(On)}
1 => #{?switch2=(On)}
&{?switch3=(Off),?switch2=(Off),?switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{?switch3=(Usv),?switch2=(Usv),?switch1=(Usv)}
#{One(On)} => |{?switch3=(On),?switch2=(On),?switch1=(On)}

        PostCategory:
&{?switch2=(On),?InitSwitch=(Off)} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{!switch2=(Off)}
#{!switch3=(Off)}
#{!switch1=(Off)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
1 => #{!switch1=(Off)}
1 => #{!switch3=(Off)}
```

```
1 => #{!switch2=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}

        PostCategory:
&{?switch2=(On),?InitSwitch=(On)} ->
        Objects:
1
#{One(On)}
#{One(Usv)}
#{All(Off)}
#{!switch2=(On)}
#{!switch3=(On)}
#{!switch1=(On)}
#{All(On)}
#{One(Off)}
|{!switch3=(On),!switch2=(On),!switch1=(On)}
|{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
|{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)}
        Arrows:
#{One(Off)} => |{!switch3=(Off),!switch2=(Off),!switch1=(Off)}
&{!switch3=(On),!switch2=(On),!switch1=(On)} => #{All(On)}
1 => #{!switch1=(On)}
1 => #{!switch3=(On)}
1 => #{!switch2=(On)}
&{!switch3=(Off),!switch2=(Off),!switch1=(Off)} => #{All(Off)}
#{One(Usv)} => |{!switch3=(Usv),!switch2=(Usv),!switch1=(Usv)}
#{One(On)} => |{!switch3=(On),!switch2=(On),!switch1=(On)}
```

Apêndice C

Listagem dos Algoritmos Produzidos

C.1 Ficheiro: build.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: syntax.lex
 *
 *  Description: code to build the binaries files
 *
 *****************************************************************************)

CM.make ();

SMLofNJ.exportFn ("../bin/SRTFS", Match.start);
```

C.2 Ficheiro: comp_match.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: comp_match.sml
 *
 *  Description: functions that implement the composite matching
 *
 *****************************************************************************)

structure CompMatch =
    struct

        (* for the debug *)
        structure Err=Err;

        exception not_match

        (* In: usr_post = the user post-categories
         *     lib_post_i = the post-categories from the library initial method
```

```
 *     lib_post_f = the post-categories from the library final method
 *     fct = the incomplete functor
 * Out: the functor or failure
 * Description: for each post-category from usr_post, it finds
 *              the respective post-category from the
 *              composite categorical specification.
 *              Then, it tries to match the two categories. *)
fun match_post_catgs (usr_post:Catg.poscats)
    (lib_post_i:Catg.poscats) (lib_post_f:Catg.poscats) fct=
    let
        (* identifies one antecedent from the lib_post_i that matches
         the given user antecedent. Then, it gives the post-category
         induced by the antecedent identified *)
        fun get_post_i ant =
            case (List.find (fn x =>
                             FunCatg.equal_obj (#1 x)
                             (Fct.Indz.obj fct ant))
                  lib_post_i) of
                SOME (ant,cons) => cons
              | NONE => (debug "get_post_i: fail\n";raise not_match)
        (* identifies the antecedent from the lib_post_f that verifies
         the given lib_post_i post-category.
         Then, it gives the post-category induced
         by the antecedent identified *)
        fun get_post_f cons =
            case (List.find (fn x =>
                             FunCatg.verify cons (#1 x))
                  lib_post_f) of
                SOME (ant,cons) => cons
              | NONE => (debug "get_post_f: fail\n";raise not_match)
        (* get the library post_category
         derived from one user antecedent *)
        fun get_post ant = (get_post_f (get_post_i ant))
        val _ = IsoMatch.t_ctg := IsoMatch.post
    in
        debug "match_post_catgs: start\n";
        (Tree.and_match (List.map (fn (a,c) =>
                                   (c,(get_post a))) usr_post)
         IsoMatch.match_catg fct
         handle not_match => NONE)
    end

(* In: match = message that contains the name of the components
 *     usr_mtd = the user method's categorical spec.
 *     lib_mtd_i = the library initial method's categorical spec.
 *     lib_mtd_f = the library final method's categorical spec.
 * Out: prints the functor if it identifies a match
 * Description: it starts by identifying a isomorphic matching between
 *              the usr_mtd pre-category and the lib_mtd_i pre-category.
 *              Then, it'll match the post-categories. *)
fun match_metd match (usr_mtd:Catg.method)
    (lib_mtd_i:Catg.method) (lib_mtd_f:Catg.method) =
    let
        val fct = Fct.new ()
        val _ = Stack.new ()
        val _ = IsoMatch.t_ctg := IsoMatch.pre
        val _ = Stack.push (match_post_catgs (#3 usr_mtd)
                            (#3 lib_mtd_i) (#3 lib_mtd_f))
        val match = (match^"\t mtd: " ^(#1 usr_mtd)^
                     " <-> "^(#1 lib_mtd_i)^":"^(#1 lib_mtd_f)^"\n");
    in
        case IsoMatch.match_catg (#2 usr_mtd) (#2 lib_mtd_i) fct of
            SOME(f)=> (print (match^"Solution:\n"); Fct.out print f)
          | NONE => ()
    end

(* In: usr_cmp = the user component's categorical spec.
 *     lib_cmp = the library component's categorical spec.
 * Out: nothing
 * Description: it applies the match_metd function to
 *              all the usr_cmp methods combined with
```

```
         *              all possible pairs of methods from lib_cmp. *)
        fun match_comp (usr_cmp:Catg.component) (lib_cmp:Catg.component) =
            let
                val match = ("\nmatching comp: "^
                             (#1 usr_cmp)^" <-> "^(#1 lib_cmp)^"\n");
            in
                List.app (fn u =>
                          (List.app (fn l_i =>
                                     (List.app
                                      (match_metd match u l_i)
                                      (#3 lib_cmp)))
                           (#3 lib_cmp)))
                (#3 usr_cmp)
            end

    end
```

C.3 Ficheiro: data_ctg.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: data_ctg.sml
 *
 *  Description: data structures used in the representations of
 *               categorical specifications
 *
 *****************************************************************************)

structure Catg =
    struct

        structure Spec=Spec;

        (* identifier *)
        type id = Spec.id;

        (* type of variable *)
        datatype tvar =
            in_ of id |
            out of id;

        (* type of predicate *)
        datatype predicate =
            pred of id   (* normal predicate *)
          | comp of tvar;  (* equality predicate *)


        (* type of assertion *)
        datatype assertion =
            prop of predicate*id (* proposition *)
          | exp of predicate  (* predicate expansion *)
          | ext of predicate; (* predicate extension *)


        (* type of object *)
        datatype obj =
            initial | terminal
          | sing of  assertion       (* single object *)
          | max of (assertion list)  (* maximum object *)
          | min of (assertion list); (* minimum object *)

        (* arrow *)
        type 'obj arrow = 'obj*'obj;
```

```
        (* abstract category *)
        type 'obj category = ('obj list*('obj)arrow list);

        (* our category *)
        type catg = obj category;

        (* pre-category *)
        type precat = catg;

        (* post-category linked with the respective
         antecedent from the pre-category *)
        type poscat = obj*catg;

        (* post-categories *)
        type poscats = poscat list;

        (* name as identifier *)
        type name = string;

        (* method *)
        type method = name*precat*poscats;

        (* set of constants which defines sort *)
        type sort =
            id list;

        (* methods *)
        type methods =
            method list;

        (* component *)
        type component = name*sort*methods;

    end;
```

C.4 Ficheiro: data_spc.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: data_ctg.sml
 *
 *  Description: data structures used in the representations of
 *               algebraic specifications
 *
 *****************************************************************************)

structure Spec =
    struct

        (* identifier *)
        type id = string;

        (* constant *)
        type tconst = id;

        (* type of variable *)
        datatype tvar =
            st of id |  (* state variable *)
            in_ of id |
            out of id;

        (* type of term *)
        datatype term =
            const of tconst | (* constant *)
```

```
    normal of id | (* variable quantified *)
    var of tvar; (* variable defined *)

(* type of predicate *)
datatype predicate =
    pred of id   (* normal predicate *)
  | comp of tvar; (* equality predicate *)

(* type of pseudo-proposition *)
datatype pproposition =
    true | false |
    prop of predicate*term | (* term applied to predicate *)
    all of predicate*id |    (* universal quantifier
                                applied to a predicate *)
    one of predicate*id;    (* existential quantifier
                                applied to a predicate *)

(* part of a clause *)
datatype sub_clause =
    conj of pproposition list| (* antecedent *)
    disj of pproposition list; (* consequent *)

(* clause *)
datatype clause =
    prop_impl of sub_clause*sub_clause

(* clausal affirmation *)
type clauses =
    clause list;

(* post-condition *)
type poscondition =
    sub_clause*clauses;

(* name as identifier *)
type name = string;

(* set of constants which defines sort *)
type sort =
    tconst list;

(* state variables *)
type variables =
    tvar list;

(* invariant *)
type invariant =
    clauses;

(* interface variables *)
type interface =
    tvar list;

(* pre-condition *)
type requires =
    sub_clause list;

(* post-conditions *)
type ensures =
    poscondition list;

(* method *)
type method =
    name*interface*requires*ensures;

(* methods *)
type methods =
    method list;

(* component *)
type component =
```

```
            name*sort*variables*invariant*methods;

    end;
```

C.5 Ficheiro: errors.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: errors.sml
 *
 *  Description: functions for treating errors
 *
 *****************************************************************************)

(* functions that help the debug *)
fun debug_ s = print s;
fun debug s =  ();

structure Err =
    struct

        fun write messg =
            TextIO.output(TextIO.stdErr, messg);

        (* type of possible errors *)
        exception non_composable_pair;
        exception invalid_prod;
        exception write_file;
        exception read_file;
        exception data_file;
        exception dir;
        exception bug;
        exception bad_args;

        exception var_st;
        exception var_def;
        exception var_use;
        exception var_itf;

        (* errors messages *)
        fun error_out typ par =
            (write (case typ of
                        non_composable_pair => "Non composable pair"
                      | invalid_prod => "Invalid product"
                      | write_file => "Can't write in the file: "
                      | read_file => "Can't read the file: "
                      | dir => "Can't open the directory: "
                      | data_file => "Bad data in the file: "
                      | bad_args => "Bad args: SRTFS <lib_dir> <name_spec>"
                      | bug => "A internal bug was detected : "
                      | var_st => "Only defined variables at left of ="
                      | var_def => "Only constants are permitted"
                      | var_use => "Isn't used the variable: "
                      | var_itf => "Interface variable not defined"
                      | _ => "Unknown error");
             write (par^"\n");
             raise typ)

        (* treat semantic errors *)
        fun error typ par =
            ( write "Error, ";
             error_out typ par)

        (* treat syntax specification errors *)
```

```
        fun spc_error l typ par =
            (write ("Specification Error, line "^(Int.toString l)^", ");
             error_out typ par)

    end
```

C.6 Ficheiro: fun_ctg.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: fun_ctg.sml
 *
 *  Description: functions that manipulate the categorical specifications
 *
 *****************************************************************************)

structure FunCatg =
    struct

        exception non_composable_pair;
        structure Utils=Utils;
        structure Catg=Catg;

        (* get data from an arrow *)
        fun source((a,_))=a;
        fun target((_,b))=b;

        (* returns the identity arrow of an object *)
        fun ident(a)=(a,a);

        (* returns the terminal(initial) arrow of an object *)
        fun arr_term(a)=(a,Catg.terminal);
        fun arr_ini(a)=(Catg.initial,a);

        (* returns a empty category *)
        fun empty () = ([],[])

        (* classify the type of an assertion *)
        val ast_type =
            fn Catg.prop(_,_) => 1
             | Catg.ext(_) => 2
             | Catg.exp(_) => 3

        (* classify the type of an object *)
        val obj_type =
            fn
            Catg.terminal => 0 | Catg.initial => 0
          | Catg.sing(_) => 1
          | Catg.min(x) => ~(List.length x)
          | Catg.max(x) => (List.length x)

        (* returns the assertions of an object *)
        val assertions =
            fn
            Catg.terminal => [] | Catg.initial => []
          | Catg.sing(x) => [x]
          | Catg.min(x) => x
          | Catg.max(x) => x

        (* classify the type of a predicate *)
        val pred_type =
            fn Catg.pred(_) => 1
             | Catg.comp(_) => 2
```

```
(* returns the identifier of a predicate *)
val pred_id =
    fn
    Catg.pred(id) => id
  | Catg.comp(Catg.in_(id)) => id
  | Catg.comp(Catg.out(id)) => id

(* compare predicates, ignoring the type of variables *)
fun cmp_pred p1 p2 =
    p1=p2 orelse
    ((pred_type p1)=(pred_type p2) andalso (pred_id p1)=(pred_id p2))

(* compare assertions, ignoring the type of variables *)
fun cmp_assertion a1 a2 =
    case a1 of
        Catg.prop(p1,c1) =>
            (case a2 of
                 Catg.prop(p2,c2) => (cmp_pred p1 p2) andalso c1=c2
               | _ => false)
      | _ => a1=a2

(* compare objects *)
fun equal_obj x y =
    (obj_type x = obj_type y) andalso
    Utils.setcont ((assertions x), (assertions y))

(* returns the composition of two arrows *)
fun comp((c,d),(a,b))=
    if equal_obj b c then (a,d)
    else Err.error Err.non_composable_pair "";

(* compare arrows *)
fun equal_arr (xs,xd) (ys,yd) =
    (equal_obj xs ys) andalso (equal_obj xd yd)

(* verifies if one arrow implies the other *)
fun match_arr (xs,xd) (ys,yd) =
    let
        val s = (Utils.setcont_ cmp_assertion ((assertions xs),
                                               (assertions ys)))
            andalso (not((obj_type xs)=0) orelse (equal_obj xs ys))
        val d = (Utils.setcont_ cmp_assertion ((assertions yd),
                                               (assertions xd)))
            andalso (not((obj_type yd)=0) orelse (equal_obj xd yd))
    in
        s andalso d
    end

(* verifies if a category satisfies an object *)
fun verify (catg:Catg.catg) obj =
    let
        val objs = case obj of
            (* if the object represent a conjunction then
             it have to satisfy the objects induced
             by its assertions *)
            Catg.min(x) => (List.map
                            (fn Catg.ext(y) => Catg.sing(Catg.exp(y))
                          | Catg.exp(y) => Catg.max([Catg.ext(y)])
                          | ast => Catg.sing(ast))
                            (assertions obj))
          | _ => [obj]
        val arrs = List.map (fn obj => (Catg.terminal, obj)) objs
    in
        Utils.setcont_ match_arr (arrs, (#2 catg))
    end

(* join two categories *)
fun join (obj1,arr1) (obj2,arr2) =
    (Utils.append obj1 obj2,
     Utils.append arr1 arr2)
```

```
    end;
```

C.7 Ficheiro: functor.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: functor.sml
 *
 *  Description: manipulation of the functor(rename) that links the user
 *               and the library categorical specifications
 *
 *****************************************************************************)

structure Fct =
    struct

        exception bad_insert;

        (* functor *)
        type functor_ =
            (* constants rename *)
            { const : ((Catg.id*Catg.id) list),
              (* predicates rename *)
              pred : ((Catg.predicate*Catg.predicate) list)}

        (* prints the functor given *)
        fun out f (fct:functor_) =
            (f "\t";
             List.app
             (fn (x,y) => f ("("^x^","^y^") "))
             (#const fct);
             f "\n\t";
             (List.app
              (fn (x,y) => f ("("^(Output.pred x)^","^
                              (Output.pred y)^") "))
              (#pred fct));
             f "\n"
             )

        (* returns a empty functor *)
        fun new () = {const = [], pred = []}

        (* inserts the rename of a constant or predicate in
         the list given *)
        fun ins_pair lst (pair as (usr,lib)) =
            case lst of
                [] => [pair]
              | (p as (u,l))::t =>
                    if u=usr then
                        (* if already exists verifies
                         if it has the same value *)
                        (if l=lib then lst
                         else raise bad_insert)
                    else p::(ins_pair t pair)

        (* inserts the rename of a constant in the functor given *)
        fun ins_const (fct:functor_) (pair as (usr,lib)) =
            (* it has to be injective to constants *)
            if (List.exists (fn (x,y) => not(x=usr) andalso y=lib) (#const fct))
                then raise bad_insert
            else {const = (ins_pair (#const fct) pair), pred = (#pred fct)}

        (* inserts the rename of a predicate in the functor given *)
        fun ins_pred (fct:functor_) (pair as (usr,lib)) =
```

```
            {const = (#const fct), pred = (ins_pair (#pred fct) pair)}

        (* finds the rename of a constant in the functor given *)
        fun get_const (fct:functor_) c = (List.find (fn(x,y)=>x=c) (#const fct))

        (* finds the rename of a predicate in the functor given *)
        fun get_pred (fct:functor_) p = (List.find (fn(x,y)=>x=p) (#pred fct))

        (* functions to translate the user categorical specification
         through the functor *)
        structure Indz =
            struct

                exception cant_indz;

                (* translates a predicate *)
                fun pred (fct:functor_) p =
                    case (get_pred fct p) of
                        NONE => raise cant_indz
                      | SOME(_,l) => l

                (* translates a constant *)
                fun const (fct:functor_) c =
                    case (get_const fct c) of
                        NONE => raise cant_indz
                      | SOME(_,l) => l

                (* translates an assertion *)
                fun assertion fct a =
                    case a of
                        Catg.prop(p,c) =>
                            Catg.prop ((pred fct p),
                                       (const fct c))
                      | Catg.ext(p) => Catg.ext (pred fct p)
                      | Catg.exp(p) => Catg.exp (pred fct p)

                (* translates an object *)
                fun obj fct obj =
                    let
                        fun assertions list =
                            List.map (assertion fct) list
                    in
                        case obj of
                            Catg.terminal => Catg.terminal
                          | Catg.initial => Catg.initial
                          | Catg.sing(x) => Catg.sing(assertion fct x)
                          | Catg.max(x) => Catg.max(assertions x)
                          | Catg.min(x) => Catg.min(assertions x)
                    end

                (* translates an arrow *)
                fun arrow fct (obj_source, obj_dest) =
                    (obj fct obj_source, obj fct obj_dest)

                (* translates a category *)
                fun catg fct (objs, arrs) = (obj fct objs, arrow fct arrs)

            end

    end
```

C.8 Ficheiro: index.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
```

```
 *
 *  File: index.sml
 *
 *  Description: management of the specifications repository
 *
 ******************************************************************************)

structure Index =
    struct

        exception BadFile

        val ctg_index = "category.idx";

        (* returns the list of components present in the repository *)
        fun read () =
            let
                val is = TextIO.openIn (ctg_index)
                    handle IO  => Err.error Err.read_file (ctg_index)
                val ret = ref [] : (string*int) list ref;
            in
                while not (TextIO.endOfStream is) do
                    let val line = TextIO.inputLine is
                        (* split the name from the sort length*)
                        val aux = String.tokens
                            (fn #" " => true | _ => false) line;
                        val _ = if (List.length aux) <> 2
                                    then Err.error Err.data_file (ctg_index)
                                else ();
                        val pair =
                            case Int.fromString (hd(tl aux)) of
                                SOME x => (hd aux, x)
                              | _ => Err.error Err.data_file (ctg_index)
                    in
                        (* join one component to the list *)
                        ret := pair::(!ret)
                    end;
                    TextIO.closeIn is;
                    (!ret)
            end;

        (* insert in the repository the name and the sort length
         of a component *)
        fun insert (pair as (name,n)) =
            let
                val actual = read ()
                (* verifies if already exists in the repository *)
                val aux = List.filter (fn x => not(name = (#1 x))) actual
                val is = ((if (List.length aux) = (List.length actual)
                             then TextIO.openAppend (ctg_index)
                           else TextIO.openOut (ctg_index))
                          handle IO  => Err.error Err.write_file (ctg_index))
                fun write (name,n) =
                    TextIO.output (is, name^" "^(Int.toString n)^"\n")
            in
                (* new insert *)
                if (List.length aux) = (List.length actual)
                    then write pair
                (* update *)
                else (List.app
                       (fn pair => write pair)
                       (pair::aux));
                    TextIO.closeOut is
            end;

    end
```

C.9 Ficheiro: iso_match.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: iso_match.sml
 *
 *  Description: functions that implement the isomorphic matching
 *
 *****************************************************************************)

structure IsoMatch =
    struct

        (* for the debug *)
        structure Err=Err;

        exception not_match;

        (* actual type of category match *)
        (* pre-category or post-category *)
        datatype type_ctg = pre | post;
        val t_ctg = ref pre;

        (* In: list of objects, sorted by number of assertions
         *     and type of object
         * Out: a list of objects sub-lists
         * Description: it creates a sub-list, for each set of objects
         *              with the same number assertions and type *)
        val rec ord_obj =
            fn [] => []
             | [x] => [[x]]
             | h::t =>
            case (ord_obj t) of
                r as hr::tr =>
                    if (FunCatg.obj_type (hd hr))=(FunCatg.obj_type h) then
                        (h::hr)::tr
                    else [h]::r
              | _ => Err.error Err.bug "IsoMatch.ord_obj"

        (* In: usr_obj = user object
         *     lib_obj = library object
         *     fct = the actual functor
         * Out: the functor or failure
         * Description: it tries to match the two objects. *)
        fun match_obj usr_obj lib_obj fct =
            let
                (* verifies if it's possible to match the two
                 assertions given *)
                fun comb_p (usr_ast,lib_ast) =
                    case (usr_ast,lib_ast) of
                        (Catg.prop (pu,cu),Catg.prop (pl,cl)) =>
                            (((Fct.Indz.pred fct pu) = pl
                              handle Fct.Indz.cant_indz => true)
                             andalso
                             ((Fct.Indz.const fct cu) = cl
                              handle Fct.Indz.cant_indz => true))
                      | (Catg.exp (pu),Catg.exp (pl)) =>
                            ((Fct.Indz.pred fct pu) = pl
                             handle Fct.Indz.cant_indz => true)
                      | (Catg.ext (pu),Catg.ext (pl)) =>
                            ((Fct.Indz.pred fct pu) = pl
                             handle Fct.Indz.cant_indz => true)
                      | _ => false
                (* updates the functor with the match
                 between the pair of assertions *)
                fun atrib pair fct =
                    case pair of
                        (Catg.prop(pu,cu), Catg.prop(pl,cl)) =>
```

```
                    (Fct.ins_pred
                     (Fct.ins_const fct (cu,cl))
                     (pu,pl)
                     )
              | (Catg.ext(pu), Catg.ext(pl)) =>
                    (Fct.ins_pred fct (pu,pl))
              | (Catg.exp(pu), Catg.exp(pl)) =>
                    (Fct.ins_pred fct (pu,pl))
              | _ => Err.error Err.bug "IsoMatch.atrib"
        (* updates the functor with the match
         between the pairs of assertions *)
        fun match pairs fct =
            ( debug "match_obj: found one possible solution\n";
             Stack.call (Utils.nestlist atrib fct pairs)
             handle Fct.bad_insert => NONE
                 )
    in
        debug "match_obj: start\n";
        debug "\t"; Output.object_ debug usr_obj; debug " : ";
        Output.object_ debug lib_obj;
        debug "\nActual Functor: "; Fct.out debug fct;
        Tree.or_match (FunCatg.assertions usr_obj)
        (FunCatg.assertions lib_obj)
        Fct.Indz.assertion (fn x => fn y => x=y) comb_p
        match fct
    end

(* In: usr_objs = group of user objects
 *     lib_objs = group of library objects
 *     fct = the actual functor
 * Out: the functor or failure
 * Description: it tries to match the two groups of objects. *)
fun match_objs usr_objs lib_objs fct =
    let
        (* verifies if the two objects can be matched,
         comparing the type of their assertions *)
        fun comb_p (usr_obj,lib_obj) =
            let
                fun types obj =
                    (List.map FunCatg.ast_type (FunCatg.assertions obj))
            in
                (* the objects have the same number of assertions *)
                Utils.setcont (types usr_obj, types lib_obj)
            end

    in
        debug ("match objects: start\n");
        debug ("\ttype:"^Int.toString
               (FunCatg.obj_type (hd usr_objs))^"\n");
        debug ("\t\t"^(Int.toString
                        (List.length usr_objs))^":"
               ^(Int.toString(List.length lib_objs))^"\n");
        Tree.or_match usr_objs lib_objs Fct.Indz.obj
        FunCatg.equal_obj comb_p
        (fn c => Tree.and_match c match_obj) fct
    end

(* In: usr_objs = user objects
 *     lib_objs = library objects
 *     fct = the actual functor
 * Out: the functor or failure
 * Description: it tries to match all the objects. *)
fun match_list_objs usr_objs lib_objs fct =
    let
        (* combines each group of user objects with
         the respective group of library object *)
        fun pairs usrs libs =
            case usrs of
                [] => []
              | u as hu::tu =>
                    case libs of
```

```
                        [] => raise not_match
                      | l as hl::tl =>
                            if (FunCatg.obj_type (hd hu))
                                =(FunCatg.obj_type (hd hl))
                                then if (List.length hu)<=(List.length hl)
                                         then (hu,hl)::(pairs tu tl)
                                     else raise not_match
                            else (pairs u tl)

    in
        debug "match_list_objs: start\n";
        debug ("\tnumber of groups: "^(Int.toString
                                        (List.length usr_objs))^":"
              ^(Int.toString(List.length lib_objs))^"\n");
        (Tree.and_match (pairs (List.rev usr_objs) lib_objs)
         match_objs fct
         handle not_match =>
             (debug "match_list_objs: not_match\n";NONE))
    end

(* In: usr_arrs = user arrows
 *     lib_arrs = library arrows
 *     fct = the actual functor
 * Out: the functor or failure
 * Description: verifies if all the arrows can be matched. *)
fun match_arrows usr_arrs lib_arrs fct =
    let
        val _ = (debug
                 ("match_arrows: start\n\tnumber of arrows: "
                  ^(Int.toString(List.length usr_arrs))^":"
                  ^(Int.toString(List.length lib_arrs))^"\n");
                 debug "\tuser arrows(translated): ";
                 (List.app (Output.arrow debug)
                  (List.map (Fct.Indz.arrow fct) usr_arrs));
                 debug "\tlibrary arrows: ";
                 (List.app (Output.arrow debug) lib_arrs);
                 debug "\tActual Functor: ";
                 Fct.out debug fct)
        (* translates the user arrows using the functor *)
        val indz_usr_arrs = List.map (Fct.Indz.arrow fct) usr_arrs
        (* verifies if the arrows are from a
         pre-category or post-category *)
        val comp = if (!t_ctg)=post then (indz_usr_arrs, lib_arrs)
                   else (lib_arrs, indz_usr_arrs)

    in
        if (Utils.setcont_ FunCatg.equal_arr comp)
            then Stack.call fct
        else (debug "match_arrows: fail\n";NONE)
    end

(* In: usr_ctg = user category
 *     lib_ctg = library category
 *     fct = the actual functor
 * Out: the functor or failure
 * Description: it tries to match the two categories. *)
fun match_catg (usr_ctg:Catg.catg) (lib_ctg:Catg.catg) fct =
    let
        val _ = debug "match_category: start\n"
        val usr_obj = ord_obj (#1 usr_ctg)
        val lib_obj = ord_obj (#1 lib_ctg)
        val _ = Stack.push (match_arrows (#2 usr_ctg) (#2 lib_ctg))
        val ret = match_list_objs usr_obj lib_obj fct
    in
        (* cleans the stack *)
        Stack.pop (); Stack.pop ();
        ret
    end

(* In: usr_post = user post-categories
 *     lib_post = library post-categories
```

```
         *     fct = the actual functor
         * Out: the functor or failure
         * Description: it tries to match the post-categories. *)
        fun match_post_catgs (usr_post:Catg.poscats) (lib_post:Catg.poscats) fct =
            let
                val _ = debug "match_post_catgs: start\n"
                (* identifies the library post-category
                 that has an antecedent which matches with the
                 antecedent given *)
                fun get_post ant lib_post =
                    case (List.find (fn x =>
                                     FunCatg.equal_obj (#1 x)
                                     (Fct.Indz.obj fct ant))
                          lib_post) of
                        SOME (ant,cons) => cons
                      | NONE => (debug "match_post_catgs: fail\n";raise not_match)
                val _ = t_ctg := post

            in
                (Tree.and_match (List.map (fn (a,c) =>
                                           (c,get_post a lib_post)) usr_post)
                 match_catg fct
                 handle not_match => NONE)
            end

        (* In: match = message that contains the name of the components
         *     usr_mtd = user method
         *     lib_mtd = library method
         * Out: prints the functor if it identifies a match
         * Description: it tries to match the two methods. *)
        fun match_metd match (usr_mtd:Catg.method) (lib_mtd:Catg.method) =
            let
                val fct = Fct.new ()
                val _ = Stack.new ()
                val _ = t_ctg := pre
                val _ = Stack.push (match_post_catgs (#3 usr_mtd) (#3 lib_mtd))
                val match = (match^"\t mtd: "^
                             (#1 usr_mtd)^" <-> "^(#1 lib_mtd)^"\n")
            in
                case match_catg (#2 usr_mtd) (#2 lib_mtd) fct of
                    SOME(f)=> (print (match^"Solution:\n"); Fct.out print f)
                  | NONE => ()
            end

        (* In: usr_cmp = user component
         *     lib_cmp = library component
         * Out: nothing
         * Description: it tries to match the two component's methods. *)
        fun match_comp (usr_cmp:Catg.component) (lib_cmp:Catg.component) =
            let
                val match = ("\nmatching comp: "^
                             (#1 usr_cmp)^" <-> "^(#1 lib_cmp)^"\n")
            in
                List.app (fn u =>
                          (List.app
                           (match_metd match u)
                           (#3 lib_cmp)))
                (#3 usr_cmp)
            end

    end
```

C.10 Ficheiro: match.sml

```
(*****************************************************************************
 *
```

```
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: match.sml
 *
 *  Description: management of the operations made by the application
 *
 ******************************************************************************)

structure Match =
    struct

        structure Iso = IsoMatch;
        structure Comp = CompMatch;

        (* apply one type of match between the user
         categorical specification and all the library
         categorical specifications *)
        fun match type_match user_ctg lib_ctgs =
            List.app
            (fn (lib_name,_) =>
             (type_match user_ctg (ParserCtg.parse lib_name)
              handle Err.read_file => ()))
            lib_ctgs

        (* In: dir = repository's directory
         *     name = name of the user specification
         * Out: nothing
         * Description: makes all the operations with the user specification. *)
        fun main dir name =
            let
                val _ = (OS.FileSys.chDir dir
                         handle _ => Err.error Err.dir dir)
                (* translates the user algebraic specification *)
                val user_ctg = ParserSpc.parse name
                (* write the user categorical specification *)
                val _ = Output.component name user_ctg
                val _ = print "The categorical representation was created.\n"
                val sort_size = (List.length(#2 user_ctg))
                (* filter the library specifications by their sort size *)
                val lib_ctgs = (List.filter
                                (fn (lib_name,lib_n) =>
                                 not(name=lib_name) andalso (sort_size<=lib_n))
                                (Index.read ()))
            in
                print "Isomorphic matching\n";
                match Iso.match_comp user_ctg lib_ctgs;
                print "\nComposite matching\n";
                match Comp.match_comp user_ctg lib_ctgs;
                OS.Process.success
            end

        (* treats a possible error and the arguments given *)
        fun start (name :string, args) =
            let
                val old_dir = OS.FileSys.getDir ()
                val res =
                    if (List.length args) = 2 then
                        (main (List.hd args) (List.hd (List.tl args))
                         handle _ => OS.Process.failure)
                        else (Err.error Err.bad_args "";
                              OS.Process.failure)
            in

                if res = OS.Process.success then print "The End\n"
                else print "The application was aborted\n";
                    OS.FileSys.chDir old_dir;
                    res
            end;

    end
```

C.11 Ficheiro: out_ctg.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: out_ctg.sml
 *
 *  Description: writes one categorical specification in a file
 *
 *****************************************************************************)

structure Output =
    struct

        structure Catg=Catg;

        (* prints the elements of a list separated by one symbol *)
        fun print_list print_comma print_elem list  =
            let
                val rec plist =
                    fn [x] => print_elem x
                     | x::rest => (print_elem x;
                                   print_comma ();
                                   plist rest)
                     | _ => ()
            in
                plist list
            end

        (* prints one variable *)
        fun var v =
            case v of
                Catg.in_ (id) => "?"^id
              | Catg.out (id) => "!"^id;

        (* prints one term *)
        fun term t =
            case t of
                Spec.const(id) => id
              | _ => Err.error Err.bug "Out.term"


        (* prints one predicate *)
        fun pred p =
            case p of
                Catg.pred (id) => id
              | Catg.comp (v) => (var v)^"="


        (* prints one object, out defines where is printed *)
        fun object_ out obj =
            let
                (* prints one assertion *)
                fun assertion p =
                    case p of
                        Catg.prop(p,t) => out ((pred p)^"("^t^")")
                      | Catg.ext(p) => out ("+"^(pred p))
                      | Catg.exp(p) => out ("*"^(pred p))

            in
                case obj of
                    Catg.terminal => out "1"
                  | Catg.initial => out "0"
                  | Catg.sing (x) => (out("#{");
                                      assertion x;
```

```
                              out("}") )
          | Catg.min (x) => ( out("&{");
                              print_list (fn()=>out(",")) assertion x;
                              out ("}")
                              )
          | Catg.max (x) => ( out("|{");
                              print_list (fn()=>out(",")) assertion x;
                              out ("}")
                              )
    end;

(* prints one object and changes line, out defines where is printed *)
fun object out obj =
    ( object_ out obj;
     out ("\n") )

(* prints one arrow and changes line, out defines where is printed *)
fun arrow out (origin,destine) =
    (
     object_ out origin;
     out(" => ");
         object_ out destine;
         out ("\n")
         )

(* prints one category, out defines where is printed *)
fun category out (objs,arrs) =
    ( out ("\tObjects: \n");
     List.app (object out) objs;
     out ("\tArrows: \n");
     List.app (arrow out) arrs )


(* prints one post-category and its antecedent,
 out defines where is printed *)
fun pos_category out (obj, catg) =
    (
     out ("\n\tPostCategory:\n");
     object_ out obj;
     out (" -> \n");
     category out catg
     )

(* prints one method, out defines where is printed *)
fun method out (name, precat, poscat) =
    ( out ("\nMethod: "^name^"\n");
     out ("\n\tPreCategory:\n");
     category out precat;
     List.app (pos_category out) poscat
     )

(* prints the set sort, out defines where is printed *)
fun sort out (X) =
    ( out ("\n\nSort: {");
     print_list (fn()=>out(",")) (fn x=>out(x)) X;
     out ("}\n")
     )

(* prints one component, out defines where is printed *)
fun component name_fich (name, X, metds) =
    let
        val f = (TextIO.openOut (name_fich^".ctg")
                 handle IO  => Err.error Err.write_file (name_fich^".ctg"))
        fun out s = TextIO.output (f, s);
    in
        out ("Categorical Representation\n");
        out ("Component "^name^"\n");
        sort out X;
        List.app  (method out) metds;
        TextIO.closeOut f;
        Index.insert (name_fich, (List.length X))
```

```
            end;

    end;
```

C.12 Ficheiro: parser_ctg.grm

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: parser_ctg.grm
 *
 *  Description: reads a categorical specification by its syntax
 *
 *****************************************************************************)

structure Catg = Catg;

%%

%eop EOF

(* %pos declares the type of positions for terminals.
   Each symbol has an associated left and right position. *)

%pos int

%term ID of string |
      COMP | CTGREP | SORT | METD | PRECTG | POSCTG |
      CCTG | PCTG | OBJ | ARR |
      SING | MIN | MAX |
      EXP | ESP |
      TERM | INIT | EQUAL |
      RPAREN | LPAREN | RBRACE | LBRACE | COMMA |
      VAR_IN of string | VAR_OUT of string |
      RARROW | DARROW |
      EOF

%nonterm catg of Catg.component |
      list_const of Catg.sort |
      methods of Catg.methods | method of Catg.method |
      precatg of Catg.precat |
      postcatgs of Catg.poscats | postcatg of Catg.poscat |
      category of Catg.catg |
      objs of Catg.obj list | obj of Catg.obj |
      arrs of (Catg.obj) Catg.arrow list | arr of (Catg.obj) Catg.arrow |
      assertions of Catg.assertion list | assertion of Catg.assertion |
      predicate of Catg.predicate | term of Catg.tvar

%name ParserCtg

%noshift EOF

%verbose
%%

  (* returns a categorical specification *)
  catg : CTGREP
         COMP ID
         SORT LBRACE list_const RBRACE
         methods
         ((ID,list_const,methods))

  (* returns a list of constants *)
  list_const : list_const COMMA ID (ID::list_const)
             |  ID ([ID])
```

```
(* returns a list of methods' categorical specifications *)
methods: methods method (method::methods)
       | ([])

(* returns a method categorical specification *)
method: METD ID
        precatg
        postcatgs
         ((ID,precatg,postcatgs))

(* returns the pre-category *)
precatg : PRECTG category (category)

(* returns the post-categories list *)
postcatgs : postcatgs postcatg (postcatg::postcatgs)
          | ([])

(* returns a post-category *)
postcatg : POSCTG obj RARROW category  ((obj,category))

(* returns a category *)
category : objs arrs ((objs,arrs))

(* returns a list of objects *)
objs : objs obj (obj::objs)
     | OBJ ([])

(* returns a list of arrows *)
arrs : arrs arr (arr::arrs)
     | ARR ([])

(* returns an arrow *)
arr : obj DARROW obj ((obj1,obj2))

(* returns an object *)
obj : INIT (Catg.initial)
    | TERM (Catg.terminal)
    | SING LBRACE assertion RBRACE (Catg.sing(assertion))
    | MAX LBRACE assertions RBRACE (Catg.max(assertions))
    | MIN LBRACE assertions RBRACE (Catg.min(assertions))

(* returns a list of assertions *)
assertions : assertions COMMA assertion (assertion::assertions)
           | assertion ([assertion])

(* returns a assertion *)
assertion : predicate LPAREN ID RPAREN (Catg.prop(predicate,ID))
          | EXP predicate (Catg.ext(predicate))
          | ESP predicate (Catg.exp(predicate))

(* returns a predicate *)
predicate : ID (Catg.pred(ID))
          | term EQUAL (Catg.comp(term))

(* returns a term *)
term : VAR_IN (Catg.in_(VAR_IN))
     | VAR_OUT (Catg.out(VAR_OUT))
```

C.13 Ficheiro: parser_ctg.lex

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: parser_ctg.lex
```

```
 *
 *  Description: identifies the special words and symbols from a
 *               categorical specification
 *
 ***************************************************************************)

structure Tokens = Tokens

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult= (svalue,pos) token

val pos = ref 1;

fun eof () = (pos := 1; Tokens.EOF(!pos,!pos))
fun error (e,l : int,_) =
TextIO.output (TextIO.stdOut,
               String.concat[
                            "line ", (Int.toString l),
                            ": ", e, "\n"
               ])

%%
%header (functor ParserCtgLexFun(structure Tokens: ParserCtg_TOKENS));
alpha=[A-Za-z];
digit=[0-9];
id={alpha}({alpha}|{digit})*;
var=[a-z]({alpha}|{digit})*;
cons=[A-Z]({alpha}|{digit})*;
ws = [\ \t];
%%
\n       => (pos := (!pos)+1; lex());
{ws}+    => (lex());

"Component"                 => (Tokens.COMP(!pos,!pos));
"Categorical Representation" => (Tokens.CTGREP(!pos,!pos));
"Sort:"                     => (Tokens.SORT(!pos,!pos));
"Method:"                   => (Tokens.METD(!pos,!pos));
"PreCategory:"              => (Tokens.PRECTG(!pos,!pos));
"PostCategory:"             => (Tokens.POSCTG(!pos,!pos));
"C-Category:"               => (Tokens.CCTG(!pos,!pos));
"P-Category:"               => (Tokens.PCTG(!pos,!pos));
"Objects:"                  => (Tokens.OBJ(!pos,!pos));
"Arrows:"                   => (Tokens.ARR(!pos,!pos));

"#"      => (Tokens.SING(!pos,!pos));
"&"      => (Tokens.MIN(!pos,!pos));
"|"      => (Tokens.MAX(!pos,!pos));

"1"      => (Tokens.TERM(!pos,!pos));
"0"      => (Tokens.INIT(!pos,!pos));

"+"      => (Tokens.EXP(!pos,!pos));
"*"      => (Tokens.ESP(!pos,!pos));

{id}     => (Tokens.ID(yytext,!pos,!pos));

"=>"     => (Tokens.DARROW(!pos,!pos));
"->"     => (Tokens.RARROW(!pos,!pos));
"{"      => (Tokens.LBRACE(!pos,!pos));
"}"      => (Tokens.RBRACE(!pos,!pos));
"("      => (Tokens.LPAREN(!pos,!pos));
")"      => (Tokens.RPAREN(!pos,!pos));
","      => (Tokens.COMMA(!pos,!pos));
"?"{id}  => (Tokens.VAR_IN(String.extract (yytext,1,NONE),!pos,!pos));
"!"{id}  => (Tokens.VAR_OUT(String.extract (yytext,1,NONE),!pos,!pos));
"="      => (Tokens.EQUAL(!pos,!pos));

"%"[^\n]* => (lex());
"."       => (error ("ignoring bad character "^yytext,!pos,!pos);
```

```
            lex());
```

C.14 Ficheiro: parser_ctg.sml

```
(*********************************************************************
 *
 *  Aplicacao Trad
 *
 *  Ficheiro: syntax.sml
 *
 *  Descricao: interface entre o analisador lexico e o
 *             analisador sintactico
 *
 *********************************************************************)


(* BEGIN: codigo retirado da documentacao *)

structure ParserCtgLrVals =
    ParserCtgLrValsFun(structure Token = LrParser.Token)

structure ParserCtgLex =
    ParserCtgLexFun(structure Tokens = ParserCtgLrVals.Tokens);

structure ParserCtgParser =
    Join(structure LrParser = LrParser
         structure ParserData = ParserCtgLrVals.ParserData
         structure Lex = ParserCtgLex);

structure ParserCtg =
    struct

        structure Out = Output;

        val invoke = fn lexstream =>
            let val print_error = fn (s,i:int,_) =>
                TextIO.output(TextIO.stdErr,"Error, line " ^
                              (Int.toString i) ^ ", " ^ s ^ "\n")
            in ParserCtgParser.parse(0,lexstream,print_error,())
            end;

        (*****************************************************************
         * Funcao: trad
         * Entrada: name_fich, nome do ficheiro sem extensao
         * Descricao: Cria a representacao categorial a partir
         *            da especificacao do ficheiro name_fich.spc,
         *            e escreve-a no ficheiro name_fich.ctg
         *****************************************************************)
        val parse = fn (name_fich) =>
            let
                val is = TextIO.openIn (name_fich^".ctg")
                    handle IO  => Err.error Err.read_file (name_fich^".ctg")
                val dummyEOF = ParserCtgLrVals.Tokens.EOF(0,0)
                fun loop lexer =
                    let val (result,lexer) = invoke lexer
                        val (nextToken,lexer) = ParserCtgParser.Stream.get lexer
                    in if ParserCtgParser.sameToken(nextToken,dummyEOF)
                           then result
                       else loop lexer
                    end
                val result =
                    loop (ParserCtgParser.makeLexer
                          (fn _ => TextIO.inputAll is));
            in
                TextIO.closeIn is;
                result
```

```
            end
    end
```

C.15 Ficheiro: parser_spc.grm

```
(*************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: parser_spc.grm
 *
 *  Description: reads a algebraic specification by its syntax
 *
 *************************************************************************)

structure Spec = Spec;

val sort = ref [] : Spec.sort ref;
val vars = ref [] : Spec.variables ref;
val itfs = ref [] : Spec.interface ref;

%%

%eop EOF

(* %pos declares the type of positions for terminals.
   Each symbol has an associated left and right position. *)

%pos int

%left AND OR

%term ID of string |
      COMP | SORT | VARS | METD | ENS | INV | INTF | REQ |
      AND | OR | TRUE | FALSE | ALL | ONE |
      DARROW | RARROW | EQUAL |
      RPAREN | LPAREN | RBRACE | LBRACE | COMMA |
      VAR_IN of string | VAR_OUT of string |
      SEMI | COLON | EOF

%nonterm start of Catg.component |
      conj of Spec.pproposition list |
      disj of Spec.pproposition list |
      pproposition of Spec.pproposition |
      spec of Spec.component |
      method of Spec.method | methods of Spec.methods |
      list_const of Spec.sort| list_const2 of Spec.sort|
      list_vars of Spec.variables | list_vars2 of Spec.variables |
      intf of Spec.tvar |
      intfs of Spec.interface| intfs2 of Spec.interface|
      preconds of Spec.requires |
      posconds of Spec.ensures | poscond of Spec.poscondition|
      clause of Spec.clause | clauses of Spec.clauses |
      term of Spec.term | proposition of Spec.predicate*Spec.term |
      predicate of Spec.predicate

%name ParserSpc

%noshift EOF

%verbose
%%

  (* returns its categorical specification *)
  start : spec (
                Trans.trans_spec(spec)
```

```
            )

(* returns a algebraic specification *)
spec : COMP ID
       SORT LBRACE list_const RBRACE
       VARS list_vars
       INV clauses
       methods
       ((ID,list_const,list_vars,clauses,methods))

(* returns a list of constants, and saves it *)
list_const : list_const2 (sort := list_const2; list_const2)

(* returns a list of constants *)
list_const2 : list_const COMMA ID (ID::list_const)
            |  ID ([ID])

(* returns a list of constants, and saves it *)
list_vars : list_vars2 (vars := list_vars2; list_vars2)

(* returns a list of variables, and saves it *)
list_vars2  : list_vars COMMA ID (Spec.st(ID)::list_vars)
            | ID ([Spec.st(ID)])

(* returns a list of variables *)
methods: methods method (method::methods)
       | ([])

(* returns the algebraic specification of a method *)
method: METD ID INTF intfs REQ preconds ENS posconds
         ((ID,intfs,preconds,posconds))

(* returns the list of interface variables, and saves it *)
intfs : intfs2 (itfs:=intfs2;intfs2)
      | (itfs:=[];[])

(* returns the list of interface variables *)
intfs2 : intfs2 COMMA intf (intf::intfs2)
       | intf ([intf])

(* returns one interface variable *)
intf : VAR_IN (Spec.in_(VAR_IN))
     | VAR_OUT (Spec.out(VAR_OUT))

(* returns the pre-condition *)
preconds : preconds COMMA disj (Spec.disj(disj)::preconds)
         | disj ([Spec.disj(disj)])

(* returns the list of the post-conditions *)
posconds : posconds poscond SEMI (poscond::posconds)
         | ([])

(* returns a post-condition *)
poscond : conj RARROW clauses ((Spec.conj(conj),clauses))

(* returns a list of clauses *)
clauses : clauses COMMA clause (clause::clauses)
        | clause ([clause])

(* returns a clause *)
clause : conj DARROW disj
           (Spec.prop_impl((Spec.conj(conj),Spec.disj(disj))))

(* returns a predicate *)
predicate :
      ID (Spec.pred(ID))
    | term EQUAL (case term of
                    Spec.var(x) => (Spec.comp(x))
                  | _ => Err.spc_error EQUALleft Err.var_st "")

(* returns a list of pseudo-proposition, which denotes its conjunction *)
```

```
conj : ppproposition         ([ppproposition])
    | conj AND ppproposition (ppproposition::conj)

(* returns a list of pseudo-proposition, which denotes its disjunction *)
disj : ppproposition         ([ppproposition])
     | disj OR ppproposition (ppproposition::disj)

(* returns a pseudo-proposition *)
ppproposition :
      ALL ID COLON proposition
           (* ID can't be a state or interface variable *)
           (case #2 proposition of
                Spec.normal(x) =>
                    if x=ID then Spec.all((#1 proposition,ID))
                    else Err.spc_error IDleft Err.var_use ID
              | _ => Err.spc_error IDleft Err.var_use ID)
    | ONE ID COLON proposition
           (* ID can't be a state or interface variable *)
           (case #2 proposition of
                Spec.normal(x) =>
                    if x=ID then Spec.one((#1 proposition,ID))
                    else Err.spc_error IDleft Err.var_use ID
              | _ => Err.spc_error IDleft Err.var_use ID)
    | proposition
           (* the predicate can only be applied to constants *)
           (case #2 proposition of
                Spec.const(_) => Spec.prop(proposition)
              | _ => Err.spc_error propositionleft Err.var_def "")
    | TRUE   (Spec.true)
    | FALSE  (Spec.false)

(* returns a proposition *)
proposition :
      ID LPAREN term RPAREN ((Spec.pred(ID),term))
    | term EQUAL term
           (* the first term must be a variable *)
           (case term1 of
                Spec.var(x) => (Spec.comp(x),term2)
              | _ => Err.spc_error EQUALleft Err.var_st "")

(* returns a term *)
term : ID (* it's a constant or a state or a normal variable *)
          (if (List.exists (fn x => x=ID) (!sort))
               then Spec.const(ID)
           else if (List.exists
                    (fn Spec.st(x) => x=ID | _ => Err.error Err.bug "")
                    (!vars))
                    then Spec.var(Spec.st(ID))
                else Spec.normal(ID))
     | intf (* verifies if it's really a interface variable *)
          (if (List.exists (fn x => x=intf) (!itfs))
               then Spec.var(intf)
           else Err.spc_error intfleft Err.var_itf "")
```

C.16 Ficheiro: parser_spc.lex

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: parser_spc.lex
 *
 *  Description: identifies the special words and symbols from a
 *               algebraic specification
 *
 *****************************************************************************)
```

```
structure Tokens = Tokens

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult= (svalue,pos) token

val pos = ref 1;

fun eof () = (pos := 1; Tokens.EOF(!pos,!pos))
fun error (e,l : int,_) =
TextIO.output (TextIO.stdOut,
               String.concat[
                             "line ", (Int.toString l),
                             ": ", e, "\n"
               ])

%%
%header (functor ParserSpcLexFun(structure Tokens: ParserSpc_TOKENS));
alpha=[A-Za-z];
digit=[0-9];
id={alpha}({alpha}|{digit})*;
var=[a-z]({alpha}|{digit})*;
cons=[A-Z]({alpha}|{digit})*;
ws = [\ \t];
%%
\n       => (pos := (!pos)+1; lex());
{ws}+    => (lex());
"Component"  => (Tokens.COMP(!pos,!pos));
"Sort:"      => (Tokens.SORT(!pos,!pos));
"Variables:" => (Tokens.VARS(!pos,!pos));
"Invariant:" => (Tokens.INV(!pos,!pos));
"Method:"    => (Tokens.METD(!pos,!pos));
"Interface:" => (Tokens.INTF(!pos,!pos));
"Requires:"  => (Tokens.REQ(!pos,!pos));
"Ensures:"   => (Tokens.ENS(!pos,!pos));

";"      => (Tokens.SEMI(!pos,!pos));
":"      => (Tokens.COLON(!pos,!pos));
">"      => (Tokens.ALL(!pos,!pos));
"<"      => (Tokens.ONE(!pos,!pos));
"true"   => (Tokens.TRUE(!pos,!pos));
"false"  => (Tokens.FALSE(!pos,!pos));

{id}     => (Tokens.ID(yytext,!pos,!pos));

"->"     => (Tokens.RARROW(!pos,!pos));
"=>"     => (Tokens.DARROW(!pos,!pos));
"&"      => (Tokens.AND(!pos,!pos));
"|"      => (Tokens.OR(!pos,!pos));
"{"      => (Tokens.LBRACE(!pos,!pos));
"}"      => (Tokens.RBRACE(!pos,!pos));
"("      => (Tokens.LPAREN(!pos,!pos));
")"      => (Tokens.RPAREN(!pos,!pos));
","      => (Tokens.COMMA(!pos,!pos));
"?"{id}  => (Tokens.VAR_IN(String.extract (yytext,1,NONE),!pos,!pos));
"!"{id}  => (Tokens.VAR_OUT(String.extract (yytext,1,NONE),!pos,!pos));
"="      => (Tokens.EQUAL(!pos,!pos));

"%"[^\n]* => (lex());
"."       => (error ("ignoring bad character "^yytext,!pos,!pos);
              lex());
```

C.17 Ficheiro: parser_spc.sml

```
(*************************************************************
 *
 *  Aplicacao Trad
 *
 *  Ficheiro: syntax.sml
 *
 *  Descricao: interface entre o analisador lexico e o
 *             analisador sintactico
 *
 *************************************************************)


(* BEGIN: codigo retirado da documentacao *)

structure ParserSpcLrVals =
    ParserSpcLrValsFun(structure Token = LrParser.Token)

structure ParserSpcLex =
    ParserSpcLexFun(structure Tokens = ParserSpcLrVals.Tokens);

structure ParserSpcParser =
    Join(structure LrParser = LrParser
         structure ParserData = ParserSpcLrVals.ParserData
         structure Lex = ParserSpcLex);


structure ParserSpc =
    struct

        structure Out = Output;

        val invoke = fn lexstream =>
            let val print_error = fn (s,i:int,_) =>
                TextIO.output(TextIO.stdErr,"Error, line " ^
                              (Int.toString i) ^ ", " ^ s ^ "\n")
            in ParserSpcParser.parse(0,lexstream,print_error,())
            end;


        val input_line = TextIO.inputLine;
        (* END: codigo retirado da documentacao *)


        (*************************************************************
         * Funcao: trad
         * Entrada: name_fich, nome do ficheiro sem extensao
         * Descricao: Cria a representacao categorial a partir
         *            da especificacao do ficheiro name_fich.spc,
         *            e escreve-a no ficheiro name_fich.ctg
         *************************************************************)
        val parse = fn (name_fich) =>
            let
                val is = TextIO.openIn (name_fich^".spc")
                    handle IO  => Err.error Err.read_file (name_fich^".spc")
                val dummyEOF = ParserSpcLrVals.Tokens.EOF(0,0)
                fun loop lexer =
                    let val (result,lexer) = invoke lexer
                        val (nextToken,lexer) = ParserSpcParser.Stream.get lexer
                    in if ParserSpcParser.sameToken(nextToken,dummyEOF)
                           then result
                       else loop lexer
                    end
                val result =
                    loop (ParserSpcParser.makeLexer
                          (fn _ => TextIO.inputAll is));
            in
                TextIO.closeIn is;
                result
            end

    end
```

C.18 Ficheiro: search.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: search.sml
 *
 *  Description: implements the functions stack and the search tree
 *               used by the matching algorithms
 *
 *****************************************************************************)

structure Stack =
    struct

        (* for the debug *)
        structure Err=Err;

        (* the stack of functions that receive a functor and
         return a functor or failure *)
        val stack = ref [] : (Fct.functor_->(Fct.functor_ option)) list ref

        (* creates a empty stack *)
        fun new () = stack:=[]

        (* gives the number of functions in the stack *)
        fun stack_size () = (Int.toString(List.length (!stack)))

        (* returns the function that is in the top of the stack and
         removes it from the stack too *)
        fun pop () =
            if (List.length(!stack) > 0)
                then let
                         val ret = (hd (!stack))
                         val _ = stack:=(tl (!stack))
                     in
                         SOME(ret)
                     end
            else NONE

        (* put a function in the top of the stack *)
        fun push f = stack:=(f::(!stack))

        (* calls the function that is in the top of the stack *)
        fun call fct =
            let
                val _ = (debug "call: start\n\tstack size:";
                         debug (stack_size ());
                         debug"\n")
                val fc = pop ()
            in
                case fc of
                    (* there are no more functions in the stack *)
                    NONE => SOME(fct)
                  | SOME(f) => let
                                   val ret = f fct
                               in
                                   (* can't be discarded because
                                    of the backtracking *)
                                   push f;
                                   ret
                               end
            end
```

```
    end


structure Tree =
    struct

        (* for the debug *)
        structure Err=Err;

        exception cant_match;

        (* In: list1 = a list with elements of type x
         *     list2 = a list with elements of type y
         *     pred = predicate that receives a pair of type (x,y)
         *            and tells if they can't be matched
         * Out: a list of lists of pairs of type (x,y)
         * Description: gives all combinations between the elements
         *              from the list1 and list2 that satisfies
         *              the predicate pred *)
        fun comb list1 list2 pred =
            let
                fun comb_ l1 l2 =
                    if (List.length l1) = 0 then [[]]
                    else
                        (* combines the first element of l1 with
                         all elements of l2 *)
                        List.concat
                        (List.map
                         (fn x =>
                          let
                              val pair = ((hd l1),x)
                          in
                              if (pred pair) then
                                  (List.map (fn e=> (pair::e))
                                   (* get the rest of combinations without
                                    the first element of l1 *)
                                   (comb_ (tl l1)
                                    (List.filter (fn y=>not(y=x)) l2)))
                              else []
                          end)
                         l2)
            in
                debug "comb: start\n";
                if (List.length list1) <= (List.length list2) then
                    (comb_ list1 list2)
                else []
            end

        (* In: combs = a list of combinations
         *     func = a matching function
         *     fct = the actual functor
         * Out: the functor or failure
         * Description: tries the function with all the combinations
         *              until it gets one solution *)
        fun comb_match combs func fct =
            let
                val rec match =
                    fn  [] => NONE
                     | comb::t => (case func comb fct of
                                       NONE => match t
                                     (* found a solution *)
                                     | x => x)
            in
                debug ("comb_match: start\n\tcombination size: "
                       ^Int.toString(List.length(combs))^"\n");
                match combs
            end



        (* In: usr = the user list
```

```
 *     lib = the library list
 *     trans = the function that uses the functor to
 *             translate the elements of usr
 *     cmp = the function that compares the usr elements
 *           with the lib elements
 *     fct = the actual functor
 * Out: a triple (pairs,usr_rest,lib_rest)
 *        pairs = a list of pairs already matched by the functor
 *        usr_rest = the usr elements that aren't in pairs
 *         lib_rest = the lib elements that aren't in pairs
 * Description: gets the usr elements that can be already
 *              translated by the actual functor, and links
 *              with the respective lib elements *)
fun split usr lib trans cmp fct =
    let
        fun split_ usr lib =
            case usr of
                (* there are no elements in usr *)
                [] => ([],[],lib)
              | elem_usr::t =>
                    let
                        (* translates the usr element *)
                        val elem_lib_ = (SOME (trans fct elem_usr)
                                         handle Fct.Indz.cant_indz => NONE)
                    in
                        case elem_lib_ of
                            (* if elem_usr can't be translated *)
                            NONE => (case split_ t lib of
                                       (c,u,l) => (c,elem_usr::u,l))
                          | SOME (elem_lib) =>
                                let
                                    (* removes the respective
                                     element from lib *)
                                    val new_lib =
                                        (Utils.filter_rest
                                         (fn y => cmp y elem_lib) lib)
                                in
                                    (* if exists one element in lib
                                     that matches elem_lib *)
                                    if (List.length lib)
                                        > (List.length new_lib) then
                                        case split_ t new_lib of
                                            (c,u,l) =>
                                                ((elem_usr,elem_lib)::c,
                                                 u,l)
                                    else
                                        (* the two lists can be matched,
                                         because exists one usr element
                                         that is translated to a element
                                         that isn't in lib *)
                                        raise cant_match
                                end
                    end
    in
        split_ usr lib
    end

(* In: usr = the user list
 *     lib = the library list
 *     trans = the function that uses the functor to
 *             translate the elements of usr
 *     cmp = the function that compares the usr elements
 *           with the lib elements
 *     pred = a predicate that tells if a usr element can't be
 *            matched with a lib element
 *     func = a matching function
 *     fct = the actual functor
 * Out: the functor or failure
 * Description: tries all the possible combinations between
 *              lib and usr with the function func until
 *              it gets a solution, and it cuts some
```

```
         *              combinations that aren't reasonable *)
        fun or_match usr lib trans cmp pred func fct =
            (debug "or_match: start\n";
             (case (split usr lib trans cmp fct) of
                  (c, u, l) => comb_match
                      (List.map (fn x => c@x) (comb u l pred)) func fct)
                 handle cant_match => (debug "or_match: fail\n";NONE))

        (* In: pairs = a list of pairs of elements to be matched
         *     func = a matching function
         *     fct = the actual functor
         * Out: the functor or failure
         * Description: applies all the pairs to the function func *)
        fun and_match (pairs:('a*'a) list) func fct =
            (debug "and_match: start\n";
             case pairs of
                 [] => Stack.call fct
               | h::t => let
                             (* puts the rest of the pairs in the stack *)
                             val _ = Stack.push (and_match t func);
                             (* call the function with the first pair *)
                             val ret = func (#1 h) (#2 h) fct
                         in
                             (* call the others functions that are in the stack *)
                             Stack.pop ();
                             ret
                         end)

    end
```

C.19 Ficheiro: transl.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: transl.sml
 *
 *  Description: functions that translates an algebraic specification into
 *               a categorical specification
 *
 *****************************************************************************)

structure Trans =
    struct

        exception semantic_error

        (* actual type of category translation *)
        (* pre-category or post-category *)
        datatype type_ctg = pre | post;
        val t_ctg = ref pre;

        (* predicate translation *)
        fun pred p =
            case p of
                Spec.pred (id) => Catg.pred(id)
              (* in the categorical there are no state variables *)
              | Spec.comp (Spec.st(id)) => if (!t_ctg)=pre
                                           then Catg.comp(Catg.in_(id))
                                      else Catg.comp(Catg.out(id))
              | Spec.comp (Spec.in_(id)) => Catg.comp(Catg.in_(id))
              | Spec.comp (Spec.out(id)) => Catg.comp(Catg.out(id));


        (* In: conj = indicates if the pseudo-proposition is part
```

```
 *            of a conjunction or a disjunction
 *     prop = a pseudo-proposition
 * Out: an assertion
 * Description: translates a pseudo-proposition into an assertion *)
fun proposition conj prop =
    case prop of
        Spec.prop (p,term) =>
            (
             (Catg.prop (pred p,Output.term term)))
      | Spec.all (p,x) =>
            (
             if conj then (Catg.ext (pred p))
             else (Catg.exp (pred p))
                 )
      | Spec.one (p,x) =>
            (
             if conj then (Catg.exp (pred p))
             else (Catg.ext (pred p))
                 )
      | _ => Err.error Err.bug
            "Trans.proposition, Terminal or initial out of place"

(* In: sc = the antecedent or the consequent part of a clause
 * Out: an object
 * Description: translates the sub-clause into an object *)
fun sub_clause sc =
    let
        (* conj = indicates if the pseudo-proposition is part
         *        of a conjunction or a disjunction
         * ps = the list of pseudo-propositions from sf *)
        val (conj, ps) =
            case sc of
                Spec.conj (x) => (true,x)
              | Spec.disj (x) => (false,x)
        (* creates the right type of object *)
        fun typ conj =
            (* one single object for each expansion *)
            fn [Catg.exp(x)] => Catg.sing(Catg.exp(x))
             (* only one proposition *)
             | [Catg.prop(p,t)] => Catg.sing(Catg.prop(p,t))
             | x => if conj then Catg.min(x)
                    else Catg.max(x)
        (* treats the special propositions, and
         creates the respective object *)
        fun object conj ps =
            let
                (* list that contains the special propositions and
                 the objects induced by them *)
                val ord = (Spec.false,Catg.initial,
                           Spec.true,Catg.terminal)
                val ord = if conj then ord
                          else (#3 ord,#4 ord,
                                #1 ord,#2 ord)
                (* removes the neutral proposition *)
                val new_ps = (List.filter (fn x => not((#3 ord)=x)) ps)
            in
                (* tries to find the absorbent proposition *)
                if (List.exists (fn x => ((#1 ord)=x)) new_ps) then
                    (#2 ord)
                else if new_ps=[] then
                    (#4 ord)
                     else typ conj (List.map (proposition conj)
                                    new_ps)
            end
    in
        object conj ps
    end

(* In: the antecedent and the consequent part of a clause
 * Out: one category
 * Description: translates the clause into an arrow and
```

```
 *              its objects *)
fun clause (Spec.prop_impl(antc,cons)) =
    let
        val ant=sub_clause antc;
        val con=sub_clause cons;
    in
        ([ant,con],[(ant,con)])
    end

(* In: a set of clauses
 * Out: one category
 * Description: translates all the clauses into one category *)
fun clausal_afirm ca =
    Utils.nestlist
    (fn cl => (FunCatg.join (clause cl)))
    (FunCatg.empty ())
    ca

(* In: list of objects
 * Out: a list of objects sub-lists
 * Description: it creates a sub-list, for each set of objects
 *              with the same number assertions and type *)
fun ord_obj objs =
    let
        (* inserts the object x in the list res on its right place *)
        fun ins x res =
            case res of
                [] => [[x]]
              | h::t => let (* type of objects *)
                            val tx = (FunCatg.obj_type x)
                            val th = (FunCatg.obj_type (hd h))
                        in
                            (* same type *)
                            if (tx=th) then
                                (x::h)::t
                            else
                                if (abs tx)<(abs th) then
                                    [x]::res
                                else h::(ins x t)
                        end
    in
        List.concat (Utils.nestlist ins [] objs)
    end


(* In: a = a list of clauses
 * Out: a category
 * Description: translates the clauses a into a category *)
fun trans_forms a =
    (
     case clausal_afirm a of
         (obj,arr) => (ord_obj obj,arr)
             )

(* In: name_comp = component's name
 *     sort = single sort
 *     variables = states variables
 *     inv = invariant
 *     name_metd = method's name
 *     interface = interface variables
 *     requires = pre-condition
 *     ensures = post-condition
 * Out: the categorical specification
 * Description: translates the algebraic specification of a method
 *              into its categorical specification *)
fun trans_method (name_comp,sort,variables,inv)
    (name_metd,interface,requires,ensures) =
    let
        (* gives the implications between all the post-conditions
         antecedents with the req *)
        fun comb req =
```

```
                List.map
                (fn (ant,cons) => Spec.prop_impl(ant,req))
                ensures
            (* gives the implications between all the post-conditions
             antecedents with the pre-condition objects  *)
            val antecedents =
                List.concat (List.map comb requires)
            (* says that the pre-condition must be true *)
            val clausal_pre =
                List.map (fn r =>
                          Spec.prop_impl(Spec.conj([Spec.true]),r))
                requires
            val _ = t_ctg := pre
            (* gets the pre-category *)
            val precat =
                trans_forms (antecedents@clausal_pre@inv)
            (* gets the post-category *)
            fun poscat (ant,cons) =
                let
                    val _ = t_ctg := pre
                    val ant_obj = (sub_clause ant)
                    val _ = t_ctg := post
                    val cons_ctg = trans_forms (cons@inv)
                in
                    (ant_obj, cons_ctg)
                end
            (* gets all the post-categories *)
            val poscats = List.map poscat ensures
        in
            (* the categorical specification *)
            (name_metd, precat, poscats)
        end;

    (* In: name_comp = component's name
     *     sort = single sort
     *     variables = state variables
     *     inv = invariant
     *     methods = list of methods
     * Out: a list of categorical specifications
     * Description: gives the list with the categorical specification
     *              of each method *)
    fun trans_spec (name,sort,variables,inv,methods)=
        (name,sort,
         List.map
         (trans_method (name,sort,variables,inv))
         methods)

end;
```

C.20 Ficheiro: utils.sml

```
(*****************************************************************************
 *
 *  Application SRTFS (Software Reuse Through Formal Specifications)
 *
 *  File: utils.sml
 *
 *  Description: utilities functions
 *
 *****************************************************************************)

structure Utils =
    struct

        (* In: p = a two arity predicate to compare elements
         *     and two lists
```

```
 * Out: true or false
 * Description: tells if all elements of the first list belong
 *              to the second one *)
fun setcont_ p (h::t, l) =
    List.exists (fn x => (p x h)) l andalso
    setcont_ p (t,l)
  |setcont_ _ ([],_) = true;

(* In: p = a two arity predicate to compare elements
 *     and two lists
 * Out: true or false
 * Description: tells if the lists have the same elements *)
fun seteq_ p (l1,l2) = setcont_ p (l1,l2) andalso setcont_ p (l2,l1);

(* In: two lists
 * Out: true or false
 * Description: tells if all elements of the first list belong
 *              to the second one *)
fun setcont (l1,l2) = setcont_ (fn x => fn y => x=y) (l1, l2)

(* In: two lists
 * Out: true or false
 * Description: tells if the lists have the same elements *)
fun seteq (l1,l2) = setcont  (l1,l2) andalso setcont  (l2,l1);

(* In: f = a two arity function
 *     i = the initial value
 *     and a list
 * Out\Description: the result of apply the function f
 *                  to all the members of the list using
 *                  the previous results too *)
fun nestlist f i [] = i
  | nestlist f i (x::r) =
    nestlist f (f x i) r;


(* In: p = a two arity predicate to compare elements
 *     and two lists
 * Out: a list
 * Description: gives the common elements of the two lists *)
fun inter p l1 l2 =
    nestlist (fn x => fn l =>
              if (List.exists (fn y => (p x y)) l2) then
                  x::l
              else l
                  ) [] l1

(* In: x = a element
 *     list = a list
 * Out: a list
 * Description: insert x in the list without duplication *)
fun insert x list =
    if List.exists (fn elem => x=elem) list then
        list
    else
        x::list;

(* In: two lists
 * Out: a list
 * Description: append the two lists without duplication *)
fun append l1 l2 =
    nestlist insert l1 l2;

(* In: p = a two arity predicate
 * Out: a list
 * Description: gives the rest of the list when it finds one
 *              element that satisfies the predicate *)
fun filter_rest p =
    fn [] => []
     | h::t => if (p h) then t
               else h::(filter_rest p t)
```

```
    (* gives the value of a SOME *)
    val get_some =
        fn NONE => Err.error Err.bug "Utils.get_some"
         | SOME(x) => x

end;
```