UM MODELO ARQUITECTÓNICO PARA DESENVOLVIMENTO DE COMPILADORES: APLICAÇÃO À *FRAMEWORK DOLPHIN*

Paulo Jorge Teixeira Matos

Universidade do Minho
Escola de Engenharia

Tese submetida à Escola de Engenharia da Universidade do Minho, para obtenção do grau de Doutor em Informática, especialização em Tecnologias da Programação.

Braga
Portugal
Janeiro, 2005

UM MODELO ARQUITECTÓNICO PARA DESENVOLVIMENTO DE COMPILADORES: APLICAÇÃO À *FRAMEWORK DOLPHIN*

Paulo Jorge Teixeira Matos
Engenheiro

Universidade do Minho
Escola de Engenharia

Orientador: Prof. Dr. **Pedro Rangel Henriques**

Tese submetida à Escola de Engenharia da Universidade do Minho para a obtenção do grau de Doutor em Informática, especialização em Tecnologias da Programação.

Braga
Portugal
Janeiro, 2005

DEDICATÓRIA

Ao David, Sara e Paula.

UM MODELO ARQUITECTÓNICO PARA DESENVOLVIMENTO DE COMPILADORES: APLICAÇÃO À *FRAMEWORK DOLPHIN*

Autor: PAULO JORGE TEIXEIRA MATOS
Orientador: Prof. Dr. PEDRO RANGEL HENRIQUES

RESUMO

A indústria dos microprocessadores é uma área com uma dinâmica ímpar, reconhecida pelo seu avançado estado tecnológico, mas também pela sua constante evolução. Esta dinâmica deve-se em grande parte à pressão que é constantemente exercida no sentido de se obter maior poder computacional. Pressão essa que tem crescido de forma exponencial, e à qual não é alheia a forte expansão da Web e o uso generalizado de conteúdos multimédia.

No entanto, esta evolução só é rentabilizada através da utilização de ferramentas adequadas, como é o caso dos compiladores, que por desempenharem um papel fundamental têm forçosamente de acompanhar a evolução dos microprocessadores. É, como tal, exigida uma resposta rápida por parte de quem desenvolve este tipo de ferramentas e a tecnologia que lhes está inerente. Daí que o desenvolvimento de aplicações que visem a construção de compiladores tenha um papel crucial, quer no sentido de tornar mais acessível a realização de uma tarefa que é complexa, quer no sentido de acelerar o processo de desenvolvimento de forma a acompanhar a evolução dos microprocessadores. Objectivos que aliás estão relacionados, dado que este processo será tão mais rápido, quanto mais acessível for a própria construção dos compiladores.

Este trabalho de doutoramento tem assim por objectivo contribuir para simplificar o processo de construção de compiladores. Pretende-se mostrar que a melhor forma de alcançar este objectivo, passa pela utilização de uma abordagem, centrada no nível intermédio do processo de compilação, que tem por base uma *framework*. No sentido de suportar esta abordagem, propõe-se um modelo para a representação de código, que introduz novos conceitos e funcionalidades, que visam facilitar o desenvolvimento dos componentes e contribuir para que estes sejam mais eficientes.

A grande contribuição deste trabalho de doutoramento é, no entanto, numa arquitectura que estabelece o comportamento e o relacionamento entre os elementos da representação de código e os componentes que sobre estes trabalham. Esta arquitectura, para além de garantir a eficiência do processo de compilação, simplifica a reutilização dos componentes, poupando o utilizador à implementação de vários mecanismos relacionados com a consistência e a gestão de dependências entre componentes

Do trabalho realizado ao longo deste doutoramento, resultou também uma plataforma para desenvolvimento de compiladores, a *framework Dolphin*, cuja implementação teve por objectivo testar a viabilidade do modelo de representação de código e da arquitectura propostas nesta dissertação.

AN ARCHITECTONIC MODEL FOR COMPILERS DEVELOPMENT: APPLICATION TO *DOLPHIN FRAMEWORK*

Author: PAULO JORGE TEIXEIRA MATOS
Advisor: Prof. Dr. PEDRO RANGEL HENRIQUES

ABSTRACT

The microprocessors industry is a remarkable, dynamic, and recognized area not only by its technological progress, but also by its constant evolution. This dynamic is the result of a systematic pressure to obtain more computational power, which has been raised exponentially with the expansion of the Web and with the generalization of multimedia contents. However this evolution can only be profitable with the usage of appropriated tools, like compilers. They play a fundamental role to take advantage of the whole features supplied by the microprocessors. As consequence, compilers must follow the evolution of the microprocessors, and their developers should have an adequate methodological and technological support.

In this context, compiler development tools give an essential contribution, making the execution of a complex task less difficult, as well as reducing the necessary time to accomplish it. Notice that both goals are related, a simple construction process, implies a shorter development time.

In this sense, the goal established for this Ph.D. project is to simplify the compilers construction process. To achieve this goal, the usage of a data centred approach based on a framework is proposed, relying upon the two main contributions of this thesis. The first one is a code representation model that introduces new concepts and features, to simplify the construction of compiler components and to make them more efficient. The second and main contribution is an architecture that establishes the behaviour and the relationship between the code representation elements and the compilers components. This architecture makes the compilation process more efficient and simplifies the reuse of the components by hiding many implementation details, namely related with components consistency and dependencies.

Another important outcome of this work is a platform for compilers development, the *Dolphin framework*, that was implemented to test and validate the code representation model and the proposed architecture.

AGRADECIMENTOS

Pela confiança que sempre demonstrou na realização deste trabalho, pela abertura que sempre teve para discutir as minhas divagações científicas, pela compreensão e paciência que sempre evidenciou e acima de tudo, pela forma como sempre esteve disponível (mesmo nos momentos que não lhe eram os mais propícios), os meus mais sinceros agradecimentos ao Prof. Doutor Pedro Rangel Henriques.

Agradeço também ao Instituto Politécnico de Bragança, em especial à Escola Superior de Tecnologia e de Gestão e aos seus dirigentes, pelas condições que asseguraram e pelo apoio prestado na preparação dos trabalhos conducentes a esta dissertação.

A minha gratidão a todos os meus colegas que de alguma forma contribuíram para a realização deste doutoramento, em especial: ao Albano Alves, que com as suas sugestões, conhecimentos e experiência contribuiu para que este trabalho fosse realizado num ambiente bem mais eficiente e aprazível; e à Maria João Varanda, pela compreensão que teve e por todo apoio prestado.

Com muito afecto, agradeço aos meus pais pela forma como sempre me apoiaram e pelos sacrifícios que fizeram para que eu pudesse chegar até aqui.

Pelas privações que tiveram que suportar, pelo facto de nem sempre ter estado presente, por terem sido o sustentáculo psicológico e a fonte de toda a minha motivação, é com muita ternura que agradeço ao David, à Sara e à Paula. Fica a promessa de que tentarei compensá-los.

Agradeço também ao programa PRODEP III, pela bolsa de doutoramento que me concedeu no âmbito do concurso número 4, Acção 5.3 - Programa de Formação Avançada de Docentes do Ensino Superior - Bolsas de Doutoramento.

ORGANIGRAMA

Esta dissertação foi elaborada segundo uma perspectiva integrada que visa descrever as soluções desenvolvidas para a resolução de um determinado problema. O leitor que deseje perceber o contexto desta tese, os problemas a resolver, as soluções propostas e averiguar o impacto das mesmas, deverá ler integralmente a dissertação, seguindo a sequência dos capítulos.

Para entender o contexto deste doutoramento recomenda-se a leitura dos Capítulos 3 e 5. Se for necessário ter uma perspectiva do estado da arte, recomenda-se também a leitura do Capítulo 2.

Para quem desejar ter uma ideia geral, mas contextualizada do trabalho desenvolvido, recomenda-se a leitura dos Capítulos 3, 4, 5 e 7.

O leitor que apenas estiver interessado na representação intermédia de código, poderá restringir a leitura aos Capítulos 3 e 4. Já o leitor que apenas estiver interessado na arquitectura, deverá ler os Capítulos 3, 5, 7 e 8. Para mais detalhes sobre a arquitectura, aconselha-se no entanto a leitura do Capítulo 6 e dos Apêndices A e B.

Para o leitor que está interessado em perceber como utilizar as soluções propostas, recomenda-se a leitura dos Capítulos 3, 4, 5, 7 e 8.

Informação sobre o *Sistema Dolphin* encontra-se no Capítulo 9.

ÍNDICE

Página

LISTA DE FIGURAS

Página

LISTA DE TABELAS

Página

CAPÍTULO 1

Introdução

Índice

A compilação de código é um processo cada vez mais complexo, que envolve um número considerável de tarefas, cuja implementação tem por base técnicas bastante elaboradas e de difícil compreensão. Tais técnicas são no entanto fundamentais para concretizar algumas funcionalidades disponibilizadas pelas linguagens de programação mais evoluídas e que fazem uso de paradigmas computacionalmente mais complexos; mas principalmente para gerar código de maior qualidade, que retire todo o proveito dos recursos disponibilizados pelos microprocessadores, dando origem a aplicações mais rápidas e eficientes.

Os compiladores desempenham assim um papel fundamental numa indústria que envolve milhões de dólares/ano, a dos microprocessadores. É apenas com uma actualização contínua dos compiladores, que se torna viável tirar o máximo proveito dos recursos disponibilizados pelos novos microprocessadores. Não é portanto de admirar que uma parte muito significativa do investimento feito no desenvolvimento de novos microprocessadores, seja gasta na actualização e investigação de novas técnicas para compiladores.

No entanto, a construção de ferramentas tão sofisticadas, requer um conhecimento muito especializado e muita experiência, não só porque o processo de compilação é bastante

complexo, mas também porque envolve um grande número de tarefas, cada uma com as suas próprias especificidades. Pode-se mesmo dizer que a construção de compiladores, nomeadamente quando envolve o desenvolvimento da própria linguagem fonte, requer peritos em diversas áreas, por exemplo: peritos em desenvolvimento de analisadores sintácticos e semânticos, cuja implementação está mais dependente da linguagem fonte; ou peritos na concepção e desenvolvimento de processos de análise e optimização de código; ou ainda peritos na construção de geradores de código final, cuja implementação está claramente ligada às características do microprocessador. Mesmo dentro de cada uma destas áreas, há sub áreas de especialização, como por exemplo: concepção e desenvolvimento de optimizações de código de médio ou baixo nível; e construção de geradores de código para arquitecturas com um único microprocessador ou para arquitecturas paralelas.

Pelo facto de a construção de compiladores ser um processo complicado, que requer um conhecimento altamente especializado, mas também pela necessidade de acelerar o seu desenvolvimento, permitindo dar uma resposta tão imediata quanto possível à própria evolução dos microprocessadores, tornou-se imprescindível a criação de novas ferramentas que facilitem a construção dos compiladores. Essas ferramentas, apesar de fazerem uso de soluções bastante diferenciadas, visam essencialmente automatizar o processo de construção dos compiladores, reduzindo o tempo de desenvolvimento, a necessidade de mão-de-obra altamente especializada e garantindo simultaneamente soluções mais normalizadas, menos propensas a erros e que contribuem para assegurar maior compatibilidade entre os componentes do compilador.

É importante realçar e clarificar a relação entre este tipo de ferramentas e os próprios compiladores. Considerando que um compilador é uma ferramenta para desenvolvimento de software, então as ferramentas para desenvolvimento de compiladores podem ser classificadas como meta-ferramentas. Esta relação entre ferramentas para desenvolvimento de compiladores, compiladores e software convencional, encontra-se ilustrada na Figura 1.1. Na parte inferior direita, representado sobre um fundo lilás, está o processo de utilização de software. Contém uma aplicação que funciona sob uma arquitectura de computação (sistema operativo+processador), a qual processa os dados provenientes dos dispositivos de entrada (teclado, dispositivos de memória, etc), computando os resultados e colocando-os nos dispositivos de saída (monitor, dispositivos de memória, etc). Algures a meio da Figura 1.1, sob um fundo verde, encontra-se representado o processo de desenvolvimento de software envolvendo um ou mais programadores que, fazendo uso de uma linguagem de programação, descrevem o código fonte. O qual é submetido a um compilador/interpretador que produz o código final (tipicamente código máquina), que posteriormente e através de um processo de *linkage*, dá origem ao código executável. Do lado esquerdo da Figura 1.1, sob um fundo amarelo, está representado o processo de desenvolvimento de compiladores, que mais não é do que um caso especial do processo de desenvolvimento de software, mas que tem a particularidade de desenvolver as próprias ferramentas utilizadas pelos programadores. Os compiladores são construídos com código desenvolvido directamente pelos programadores (construtores de compiladores), ou reutilizando código previamente implementado (bibliotecas), ou ainda fazendo uso de ferramentas especialmente desenvolvidas para este fim. Essas ferramentas permitem gerar, de forma mais ou menos automática, as rotinas necessárias à implementação de determinadas tarefas de compilação, requerendo para tal uma especificação com a informação necessária à *parametrização* de tais rotinas.

O desenvolvimento de ferramentas para construção de compiladores surge assim como um quarto processo, sobre o qual incide este trabalho de doutoramento. É mais ou menos intuitivo que a concepção e o desenvolvimento de tais ferramentas requerem, por parte de quem as desenvolve, um conhecimento completo das técnicas e soluções utilizadas na implementação das tarefas de compilação, o qual deverá ser substanciado pela experiência de

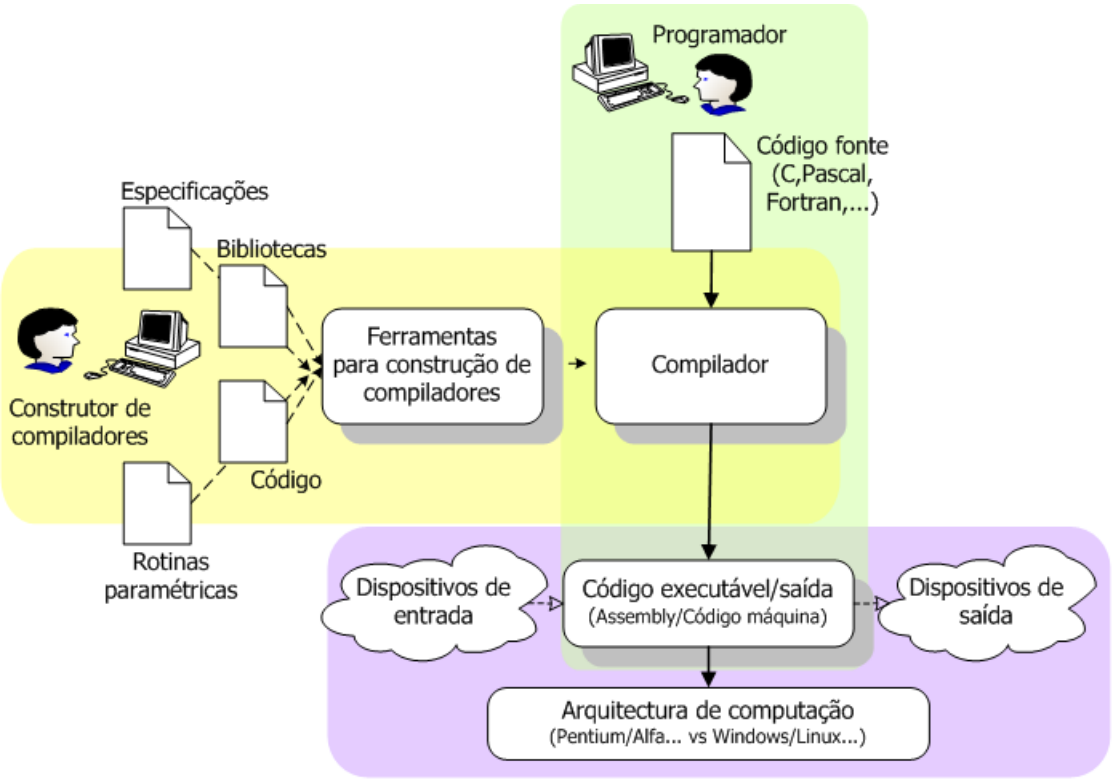


Figura 1.1: Representação do processo de desenvolvimento e utilização de compiladores.

implementação. É ainda importante acrescentar que a maioria destas ferramentas apenas suporta o desenvolvimento de algumas tarefas. Um exemplo típico, que ilustra esta relação e que é bem conhecido, é o caso dos analisadores sintácticos, também designados por *parsers*, e do Yacc [Joh79], uma das várias ferramentas para geração de analisadores sintácticos.

A implementação destas ferramentas trouxe novos conceitos e soluções para a área da construção de compiladores. Por exemplo, o conceito de gramática e tabelas de *parsing*, foi provavelmente introduzido com a utilização dos geradores de analisadores sintácticos (vulgarmente designados por geradores de *parsers*).

Para determinadas tarefas, existem já ferramentas eficientes e de utilização bastante satisfatória, como acontece para o caso da análise léxica, sintáctica e semântica. Tarefas estas que têm a particularidade de fazer uso de soluções bastante normalizadas. Há no entanto outras tarefas, para as quais não é simples conceber soluções normalizadas, dada a sua heterogeneidade. É o caso de algumas análises e optimizações de código de baixo nível, cuja implementação é muito dependente da arquitectura de computação. O que significa que também não é simples desenvolver ferramentas para a construção dessas tarefas, restando assim implementá-las directamente (sem recurso a ferramentas).

É como tal normal, que a construção de um compilador se faça com recurso a ferramentas, mas também através da implementação directa das tarefas. O que coloca normalmente problemas de compatibilidade e de integração entre as tarefas construídas directamente e as tarefas construídas com recurso a ferramentas. Mesmo entre estas últimas, há por vezes problemas de compatibilidade pelo simples facto de serem construídas com ferramentas diferentes. A resolução destes problemas nem sempre é simples, resultando por vezes em soluções pouco eficientes que afectam a qualidade do processo de compilação.

Existem no entanto algumas ferramentas que evitam este e outros problemas, supor-

tando o desenvolvimento de forma integrada das várias tarefas (por vezes de todo o compilador). Na maior parte dos casos, tal é conseguido através da integração de ferramentas mais específicas com outras soluções. Daí que o mais correcto é designar estas soluções por sistemas. Estes visam fornecer uma solução integrada, que suporte completamente (ou em grande parte) o desenvolvimento dos compiladores. A sua concepção e desenvolvimento é no entanto um enorme desafio, substancialmente mais complexo que o desenvolvimento de ferramentas para tarefas específicas, e muito mais complicado que o desenvolvimento dos próprios compiladores.

São variadíssimas as soluções que podem ser utilizadas no desenvolvimento destes sistemas, que podem incluir bibliotecas, rotinas paramétricas, geradores de componentes, etc.

O trabalho realizado na preparação desta dissertação incide sobre *frameworks*. São sistemas que têm por base um determinado modelo, neste caso de compilação, compostos por várias interfaces/componentes que podem ser utilizadas para desenvolver novos componentes ou até mesmo compiladores completos. A Figura 1.2 ilustra a utilização deste tipo de sistemas na construção de compiladores. O desenvolvimento do compilador faz-se criando instâncias dos componentes, criando novos componentes a partir das interfaces da *framework* e, eventualmente, recorrendo a rotinas implementadas directamente pelo programador ou implementadas através de outras ferramentas.

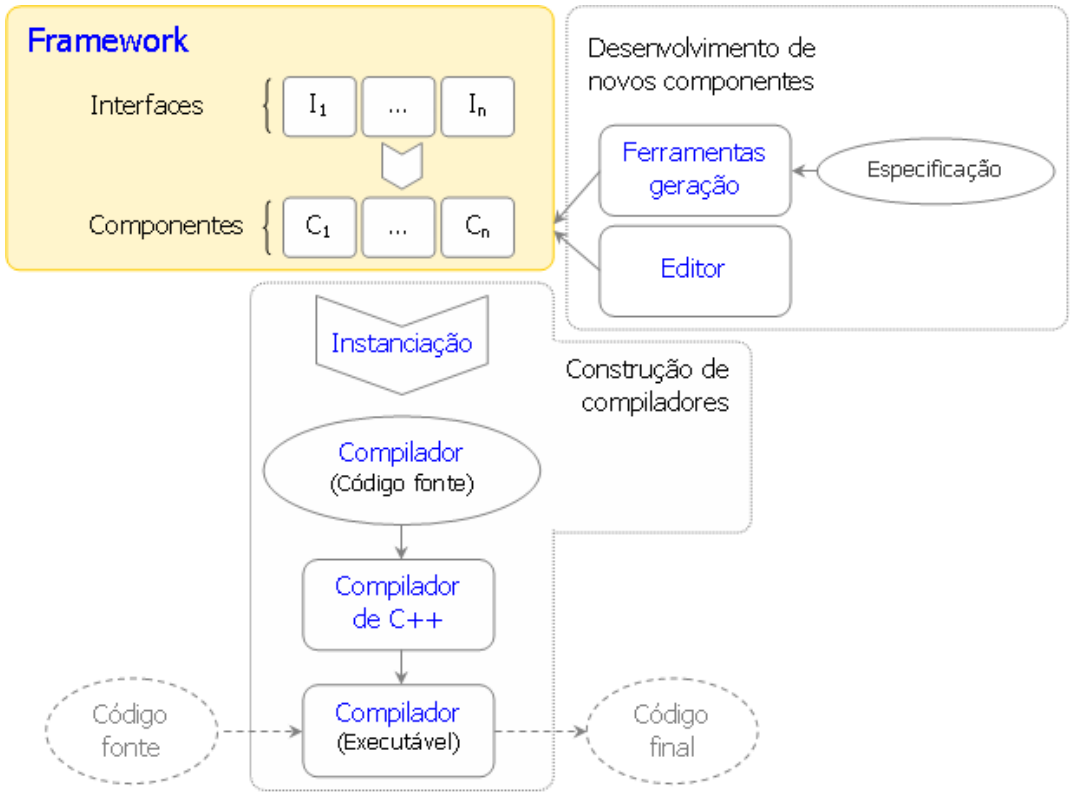


Figura 1.2: Desenvolvimento de compiladores com base numa *framework*.

Pela sua modularidade, organização e concepção, as *frameworks* são também soluções bastante atractivas para efectuar a integração de ferramentas de geração. A aplicação de *frameworks* surge no contexto deste doutoramento, como um passo intermédio no processo de desenvolvimento de compiladores, algures entre as ferramentas de geração e o compilador propriamente dito. O que permite uma abordagem realista para se obter uma solução abrangente capaz de suportar o desenvolvimento de um grande número de tarefas de compilação

ou mesmo de todo o compilador, em oposição a abordagens mais utópicas, como as que visam suportar o desenvolvimento de todas as tarefas de compilação com base numa única ferramenta de geração.

A principal contribuição deste trabalho de doutoramento, consiste na proposta de uma arquitectura para este tipo de sistemas, que sem abrir mão das vantagens que lhe são inerentes, contribui para:

- Simplificar o desenvolvimento de compiladores, potenciando assim menores custos de desenvolvimento;
- Construir compiladores mais rápidos e eficientes, nomeadamente quando comparado com as soluções convencionais de reutilização e de geração de componentes;
- Tornar a expansão e adaptação dos próprios sistemas, no sentido de dotá-los de novos recursos, mais acessível.

Outras vantagens há, que são apresentadas ao longo desta tese, mas cuja origem não está directamente relacionada com a arquitectura que é proposta.

1.1 Motivação

A motivação inicial para o desenvolvimento deste doutoramento, provém da experiência anteriormente adquirida com a implementação do Back-End Development System-BEDS [Mat99], concebido para suportar o desenvolvimento das tarefas de *back-end* do processo de compilação, nomeadamente: a selecção de instruções, a atribuição de registos, a geração do código *assembly* e ainda a implementação de algumas optimizações de código de baixo nível (*peephole optimizations*). A investigação e o trabalho realizado neste projecto contribuíram para adquirir um *know-how* e uma experiência considerável e bastante sólida, quer sobre a construção de compiladores, quer sobre o desenvolvimento de ferramentas para construção de compiladores. Foi atendendo a estes factos, que potenciavam uma situação privilegiada para desenvolver um trabalho inovador, capaz de contribuir para o progresso desta área científica, que se formulou a proposta para este trabalho de doutoramento.

A experiência adquirida com o BEDS foi ainda fundamental para definir os objectivos deste doutoramento. Por exemplo, aquando da concepção e desenvolvimento do BEDS foi possível constatar que realmente havia ferramentas bastante poderosas e eficientes, mas que também existiam tarefas para as quais não havia ferramentas. Ou mesmo quando as havia, a sua utilização era tão difícil ou tão pouco eficiente, que tornava pouco vantajosa a sua utilização. Permitiu também identificar claramente os problemas para os quais havia o interesse, os conhecimentos e as condições necessárias para a realização de um trabalho de investigação, que potenciasse a obtenção de resultados inovadores, capazes de suportar uma dissertação de doutoramento.

Um desses problemas, que continua a ser um desafio interessante e cuja resolução dá continuidade ao trabalho já iniciado com o BEDS, é a concepção e desenvolvimento, ou simplesmente a melhoria, das ferramentas para a construção de compiladores. Partindo desta ideia, pensou-se inicialmente em concentrar os esforços nas tarefas de compilação, cujo desenvolvimento não era então suportado por qualquer tipo de ferramenta. Pretendia-se especialmente abordar as tarefas cujas soluções são pouco normalizadas e que contêm muitas variantes, como é o caso de algumas optimizações e análises de código de baixo nível.

Conceber uma solução e implementá-la sob a forma de uma ferramenta, que auxilie o desenvolvimento de determinadas tarefas para as quais ainda não existe este tipo de suporte,

é sem dúvida um contributo importante e como tal um objectivo válido para um trabalho de doutoramento. No entanto, no passado recente foram poucos os resultados apresentados nesta área. O que é talvez a consequência do elevado investimento feito há já alguns anos atrás, por parte de várias universidades e de grandes colossos industriais, como a IBM e a Intel. Investimento esse que resultou em diversas ferramentas e sistemas que são ainda hoje trabalhos de referência. Pelo que este assunto parece estar actualmente esgotado, não por falta de oportunidades, mas por falta de novas soluções.

Sem abandonar a alternativa até aqui formulada, pensou-se numa segunda possibilidade, que consistia em abordar o problema pelo todo, nomeadamente na concepção de soluções que permitissem uma melhor integração das ferramentas actualmente existentes, no sentido de disponibilizar um sistema integrado que suportasse o desenvolvimento de todas as tarefas de compilação. Apesar deste ser um assunto já estudado, para o qual existem já algumas soluções, continuava a ser muito pertinente e mantinha em aberto muitas oportunidades. É um assunto que não se prende tanto com as soluções utilizadas na construção das várias ferramentas, mas mais com as soluções utilizadas na sua integração, e acima de tudo na forma como toda esta tecnologia é disponibilizada a quem constrói compiladores.

O denominador comum destas duas alternativas e a outras que então se formularam, está no objectivo de facultar soluções, que tornem mais simples a construção de compiladores ou componentes para compiladores. Sem que isso signifique diminuir a qualidade dos mesmos, mas que claramente contribua para reduzir o tempo e o esforço de desenvolvimento. Esta ideia, por si só, sintetiza grande parte da motivação que levou à proposta e realização deste doutoramento.

1.2 Objectivos

O objectivo principal deste doutoramento é facultar novas ou melhores soluções que simplifiquem a construção de compiladores. É evidente que este é um objectivo bastante difuso, mas que não deixa de ser válido, nomeadamente como ponto de partida para a definição de objectivos mais concretos.

É de salientar que melhorar o processo de construção de compiladores, não significa melhorar o processo de compilação e muito menos melhorar a qualidade do código a gerar pelos compiladores. Como, no entanto, são tópicos que estão relacionados, a optimização de um deles pode estar condicionada pelos outros dois. É assim importante que as melhorias que se obtenham no processo de construção de compiladores sejam alcançadas sem deteriorar o processo de compilação e muito menos a qualidade do código produzido pelos compiladores. De preferência deverão mesmo potenciar a melhoria destes dois tópicos, passando assim de limitações a objectivos, que apesar de não serem fundamentais são pertinentes para a afirmação dos resultados obtidos neste doutoramento.

É claro que melhorar o processo de construção de compiladores não passa necessariamente por conceber uma solução completa que suporte o desenvolvimento de todas as tarefas de compilação. Aliás, desde o início que se tentou estabelecer objectivos realistas e exequíveis, que tivessem em conta o tempo e a mão-de-obra disponíveis e que fossem adequados à dimensão de um trabalho de doutoramento. Optou-se assim por apostar na concepção de uma estrutura ou base de trabalho, que fosse modular, expansível, fácil de utilizar e que servisse de base à integração de ferramentas ou de componentes previamente implementados. O mais importante é que resultasse numa clara mais valia para quem apostasse na sua utilização.

Estabeleceu-se assim, como objectivo inicial deste trabalho de doutoramento, fazer o levantamento dos sistemas e ferramentas de desenvolvimento de compiladores mais relevantes,

não só para se apurar o estado da arte, mas também para identificar as soluções utilizadas, as suas características, vantagens e desvantagens. A concretização deste estudo inicial, seria fundamental para alcançar o segundo objectivo: determinar o tipo de solução a conceber e as tecnologias a utilizar. Estes foram os únicos objectivos inicialmente propostos. Esperava-se então que a sua concretização, permitisse identificar claramente o caminho a seguir e os resultados a alcançar, e assim definir os objectivos seguintes.

Felizmente assim aconteceu, e a escolha recaiu numa solução que teve por base uma *framework*, formada por diversos componentes cuja utilização permitiria a construção integral de compiladores. A característica mais proeminente desta solução está na utilização de um modelo comum de representação de código, que ao servir de base à implementação de todos os componentes, permite resolver grande parte dos problemas relacionados com a sua integração e compatibilização.

Partindo deste ponto, foi então possível definir o seguinte objectivo: conceber e implementar um modelo para representação de código, que fosse versátil, expansível, flexível e que acima de tudo, simplificasse a construção de novos componentes. Este último ponto é particularmente importante, dado que o desenvolvimento de aplicações com base numa *framework*, passa na maior parte dos casos por expandir componentes já existentes, ou por desenvolver novos componentes (fazendo uso das interfaces definidas pela *framework*). Assim, para tornar mais simples a construção de compiladores, a *framework* deverá facultar mecanismos que facilitem a expansão e a construção de componentes.

Numa fase bastante mais adiantada do trabalho, quando já se estava a dar uma utilização mais intensiva ao sistema até então desenvolvido, constatou-se que a forma como os componentes eram reutilizados, apesar de ser simples, era no entanto bastante penalizadora para o processo de compilação. Constatou-se também, que o mesmo ocorria em todos os outros sistemas que foram entretanto analisados. Em alguns destes sistemas e também na solução até então desenvolvida, era possível não afectar o processo de compilação, mas implicava tornar a construção dos compiladores bastante mais complexa. Isto levou a estabelecer um novo objectivo para este trabalho de doutoramento: conceber uma solução que simplificasse a reutilização dos componentes e, simultaneamente, garantisse a eficiência do processo de compilação.

O que passou pelo desenho e concepção de uma arquitectura para a *framework*, que define o comportamento e a relação entre componentes e entre estes e a representação de código. Arquitectura essa que é o principal contributo deste doutoramento e que apesar de visar a resolução dos problemas desta *framework*, foi desenvolvida de forma independente desta, permitindo assim que seja aplicada, sem grandes alterações, noutros sistemas que padeçam dos mesmos problemas.

A seguinte lista sintetiza os objectivos que foram estabelecidos, quer inicialmente, quer ao longo da realização deste trabalho de doutoramento:

- Caracterização do problema e definição do domínio de trabalho;
- Escolha do tipo de solução a utilizar (que potencie o maior número de oportunidades para melhorar ou desenvolver soluções que simplifiquem o processo de construção de compiladores);
- Elaboração da solução escolhida no sentido de alcançar o objectivo central desta tese:
  - Desenho e concepção de um modelo para a representação do código;
  - Desenho e concepção de uma arquitectura para a *framework*;
- Aferição dos resultados e dos objectivos alcançados.

1.3 Tese e contribuições

Atendendo ao objectivo deste trabalho de doutoramento, que é contribuir para simplificar o processo de construção de compiladores, definiu-se como tese: *demonstrar que uma forma eficaz de satisfazer estes objectivos passa pela utilização de uma abordagem centrada no nível intermédio do processo de compilação que potencia a reutilização de código.* Abordagem essa que tem por base uma *framework* desenvolvida segundo uma determinada arquitectura. Seguindo esta abordagem, propõe-se:

- Um modelo de representação de código, que não só é genérico, flexível, extensível e consistente, como disponibiliza algumas soluções que visam simplificar a construção dos componentes e contribuir para que estes sejam eficientes;
- Uma arquitectura que ao definir o comportamento, o relacionamento e a estrutura base dos componentes, contribui para simplificar a sua reutilização e, simultaneamente, optimizar o processo de compilação.

Ao longo desta dissertação, mostra-se também que a arquitectura proposta é suficientemente genérica para poder ser utilizada em outros sistemas que abordem a construção de compiladores, ou de outras aplicações, de forma semelhante. Comprovando que as vantagens enunciadas para esta arquitectura, não se restringem ao sistema sob o qual foi desenvolvida.

Do trabalho realizado ao longo da preparação desta dissertação, resultou também um sistema para desenvolvimento de compiladores, a *framework Dolphin*, cuja implementação teve por objectivo testar a viabilidade do modelo de representação de código e da arquitectura propostos nesta dissertação.

Estas contribuições deverão resultar num conjunto de mais valias, que afectam positivamente o processo de construção de compiladores, a própria manutenção e desenvolvimento da *framework*; e ainda contribuem para optimizar o processo de compilação, tornando-o mais rápido e eficiente. Isto tudo sem provocar alterações de fundo, nomeadamente no que diz respeito às próprias ferramentas para construção de compiladores e às soluções utilizadas na implementação das diversas rotinas de compilação.

É ainda de realçar que, se o modelo de representação de código proposto nesta dissertação pode ser considerado algo complexo (independentemente das mais valias que advêm dessa complexidade), o mesmo não se pode dizer da arquitectura que é simples e aberta, potenciando uma série de melhorias e a integração com outras soluções.

1.4 Trabalho relacionado

Foram vários os trabalhos identificados que estão de alguma forma relacionados com o tema desta tese. São no entanto poucos os que partilham da mesma abordagem e que têm objectivos semelhantes. Mesmo esses, na sua grande maioria, foram ou estão a ser desenvolvidos no âmbito de projectos com uma dimensão que ultrapassa, em muito, a preparação de uma "simples" dissertação de doutoramento.

O SUIF Compiler System [ADH+00a] (ver Secção 2.8) é sem dúvida um dos sistemas cujas soluções mais se aproximam do trabalho realizado na preparação desta dissertação. Apesar de não partilhar os mesmos objectivos (pelo menos não os considera como objectivos prioritários) e de ser um projecto com outro tipo de dimensão, a comparação é perfeitamente justificável, nomeadamente quando feita com o sistema que resultou do trabalho realizado neste doutoramento. Ambos têm por base uma *framework*, utilizam modelos de representação de código que assentam nos mesmos princípios e no mesmo tipo de implementação (classes de

C++). Até mesmo algumas das ferramentas/experiências desenvolvidas têm o mesmo tipo de finalidade (apesar de utilizarem abordagens e terem sido implementadas com objectivos diferentes), como é o caso da aplicação desenvolvida para visualizar o código intermédio do processo de compilação.

É no entanto de realçar que o objectivo último deste doutoramento nunca foi o sistema que daí resultou (*framework*). Este funcionou essencialmente como uma *test-bed*, através da qual se desenvolveram e testaram as contribuições que são propostas nesta dissertação. O que não é definitivamente o caso do SUIF System, que é um sistema maduro, que resolve problemas reais, e que fornece um número considerável de componentes e de ferramentas de suporte. O que no entanto não torna menos relevante o trabalho realizado na preparação desta dissertação, antes pelo contrário, reforça a importância do mesmo, nomeadamente quando se sabe que no SUIF System estão envolvidos alguns dos mais importantes investigadores desta área, que tem uma comunidade activa bastante grande, e que foi em tempos um projecto co-financiado pela DARPA, pela NSF e por várias empresas privadas de renome mundial.

Um outro sistema cujo trabalho está claramente relacionado com o que foi efectuado neste doutoramento, até porque esteve na sua base, é o RTL System [JML91] (ver Secção 2.7). A estrutura da solução é em tudo semelhante, isto é, uma *framework* cujo funcionamento assenta num modelo comum de representação de código. O próprio modelo proposto nesta dissertação foi desenvolvido a partir do modelo utilizado no RTL System e até a dimensão dos projectos é bastante semelhante. Pode-se mesmo dizer que este doutoramento foi uma continuação não oficial do trabalho desenvolvido no RTL System, mas com outro tipo de objectivos. Aliás, é interessante acrescentar que o RTL System surge no seio de um grupo com fortes raízes na área da Engenharia de Software[1]. Considerando os vários projectos desenvolvidos por este grupo, tudo leva a crer que o objectivo terá sido demonstrar que a utilização de *frameworks* também é uma excelente solução para o desenvolvimento de ferramentas de construção de compiladores. No entanto não foi possível confirmar esta opinião, mas em muito terá contribuído para ela, o facto deste grupo ter continuado a apostar em outros projectos que visavam a aplicação de *frameworks*, mas tanto quanto foi possível apurar não deu continuidade ao RTL System. A confirmar-se que realmente foi este o objectivo, pode-se dizer que o modelo de representação de código, mas também a arquitectura proposta nesta dissertação, cuja concepção foi essencialmente um trabalho de engenharia de software, é claramente uma continuação do trabalho desenvolvido no RTL System.

Há ainda outros projectos que não podendo ser directamente comparados ou cuja comparação é mais difícil, são no entanto fundamentais para perceber determinadas opções realizadas na elaboração deste trabalho de doutoramento. É o caso do Zephyr [ADR98] (ver Secção 2.6), que apesar de também fazer uso de um modelo comum para a representação de código, aborda o desenvolvimento de ferramentas para construção de compiladores de forma bastante distinta das demais soluções (incluindo as propostas nesta dissertação). Consiste numa *framework* que integra algumas ferramentas e disponibiliza várias soluções já implementadas, mas difere dos demais sistemas por permitir utilizar rotinas externas, independentemente do tipo de representação sobre a qual trabalham. O que é feito através de mecanismos que convertem a representação utilizada por essas rotinas na representação utilizada pelo Zephyr. Isto por um lado, dá uma enorme flexibilidade a este sistema, mas por outro, ao requerer a inserção dos mecanismos de conversão para compatibilizar as rotinas externas com as do sistema, afecta a estrutura dos compiladores, ficando esta mais complexa e pesada, o que resulta em tempos de compilação mais elevados.

[1] Um dos responsáveis pelo RTL System, Ralph E. Johnson, é também um dos elementos do GoF - Gang of Four - autores do famoso livro "*Design Patterns - Elements of reusable object-orient software*".

Outros trabalhos que foram analisados, mas que acabaram por não ter uma grande influência nos resultados finais deste doutoramento, foram: o GENTLE [Sch97] (ver Secção 2.2), que também visa simplificar o processo de construção de compiladores e que faz uso de uma abordagem bastante interessante, mas que é no entanto uma solução muito restrita; o Program Analyzer Generator-PAG [Mar98] e o OPTIMIX [Aßm95], que foram alvo de um estudo aprofundado, que por falta de tempo não chegou a produzir resultados, mas que visava analisar a possibilidade e as mais valias de integrar soluções de geração, na arquitectura proposta e mais especificamente na *framework* entretanto desenvolvida.

É também importante focar alguns trabalhos que tiveram na base do BEDS, e que muito contribuíram para a experiência que foi necessária à realização deste trabalho de doutoramento, é o caso do Back-End Generator-BEG [ESL89], do Bottom-Up Rewrite Generator-BURG [FHP91] e do New Jersey Machine Code Toolkit-NJMCT [RF95].

Na perspectiva da engenharia de software, o trabalho encontrado que está mais relacionado com este doutoramento foi, como já se disse, o RTL System. É no entanto possível que existam trabalhos em outras áreas científicas, que façam uso de soluções semelhantes, mas por questões de disponibilidade, optou-se por não pesquisar e estudar trabalhos cujo assunto não estivesse relacionado com o tema desta dissertação. Esta opção não deverá ter implicações graves, nomeadamente no que diz respeito à originalidade do trabalho, dado que a contribuição que é feita em termos de engenharia de software, é propor uma solução para um problema em concreto (construção de compiladores), com a finalidade de satisfazer um determinado conjunto de objectivos. Neste sentido, não se encontrou qualquer outra referência para lá daquelas que já foram apresentadas e que são apresentadas no Capítulo 2.

Convém no entanto acrescentar que foi realizada toda a pesquisa necessária à concepção e implementação da arquitectura, nomeadamente sobre a tecnologia a utilizar, que incluiu documentação sobre *frameworks*, padrões de desenho[2], desenho de aplicações orientado por objectos, polimorfismo paramétrico, etc.

1.5 Estrutura da dissertação

Este é um doutoramento que envolveu a pesquisa de assuntos bastante distintos, que vão desde a análise das inúmeras técnicas utilizadas na implementação das várias tarefas de compilação, até ao estudo de casos e soluções para reutilização de software. Pelo meio ficam tópicos como: o estudo de soluções para a concepção de ferramentas de compilação; análise de modelos de representação de código; desenho de arquitecturas de software; e até mesmo tópicos que à primeira vista não estariam relacionados com este doutoramento, nomeadamente sobre tecnologias Web, como é o caso do XML, do XML Schema, do desenvolvimento de aplicações em JScript e em Macromedia Flash.

Houve também a necessidade de testar e experimentar um número considerável de aplicações, técnicas e soluções. É claro que nem todo este trabalho produziu resultados palpáveis e também houve muito trabalho feito, cujos resultados extravasam o tema deste doutoramento.

A estrutura desta dissertação, que se encontra dividida em dez capítulos, reflecte apenas o trabalho realizado no sentido de satisfazer os objectivos propostos para este doutoramento. O presente capítulo, que inclui esta secção sobre a estrutura da dissertação, faz também a introdução ao tema deste doutoramento e apresenta: os objectivos, as contribuições, a tese a defender e os trabalhos relacionados.

O Capítulo 2 introduz alguns conceitos sobre ferramentas para desenvolvimento de compiladores e faz o levantamento do estado da arte, focando detalhadamente a abordagem,

[2]Tradução do termo Inglês *design patterns*.

as vantagens e as desvantagens de algumas das soluções mais relevantes.

O Capítulo 3 apresenta a *Framework Dolphin*, que é antes de mais um exemplo da abordagem escolhida para o desenvolvimento de ferramentas para construção de compiladores. Esta *framework* serviu para testar as ideias propostas nesta dissertação, pelo que é um exemplo de aplicação das mesmas, mas também: uma ferramenta perfeitamente autónoma, capaz de suportar o desenvolvimento integral de compiladores; e uma solução bastante atractiva para a integração de ferramentas de geração, que está actualmente na base de um projecto bem maior, o *Sistema Dolphin*.

O Capítulo 4 é dedicado à representação do código intermédio do processo de compilação. Começa por descrever os requisitos e características dos modelos de representação de código, faz de seguida o levantamento das soluções mais relevantes, e conclui apresentando a Dolphin Internal Representation-DIR (modelo de representação de código proposto nesta dissertação), designadamente as características, vantagens e desvantagens deste modelo.

A primeira parte do Capítulo 5, identifica alguns problemas existentes na versão original da *framework Dolphin*. Problemas esses que são comuns a este tipo de solução, que a troco de simplificar o processo de construção, não salvaguardam o desempenho dos compiladores, designadamente no que diz respeito ao tempo e recursos necessários ao processo de compilação. A segunda parte deste capítulo apresenta, para cada um dos problemas identificados, as soluções desenvolvidas e a forma como estas foram sendo integradas para produzir aquela que é, a grosso modo, a principal contribuição deste doutoramento, a arquitectura.

O Capítulo 6 apresenta um conjunto de optimizações para a arquitectura proposta nesta dissertação, que apesar de não serem fundamentais para o seu funcionamento, permitem que em determinadas circunstâncias se optimize de forma muito significativa o tempo de compilação.

O Capítulo 7 apresenta a arquitectura, numa perspectiva integrada que relaciona as diversas entidades que a compõem, com o modelo de compilação que está inerente à *framework Dolphin*. Ilustra também, através de vários diagramas sequenciais e para vários procedimentos, as operações que são efectuadas, quer pelas entidades que compõe a arquitectura, quer pelos utilizadores dos componentes. Este capítulo inclui também a explicação de como é que a arquitectura proposta está adaptada à *framework Dolphin*.

O Capítulo 8 consiste num exemplo que ilustra como a solução no seu todo funciona, desde a implementação de um componente, que aproveita as facilidades disponibilizadas pelo modelo de representação de código e pela arquitectura; até à utilização dos componentes, explicando como é que o processo de compilação se desenrola e identificando onde é que estão as mais valias desta solução. Este capítulo inclui ainda a avaliação das soluções propostas nesta dissertação.

O Capítulo 9 introduz o Sistema Dolphin, um projecto que se encontra ainda em fase de concepção, que tem por base a *framework Dolphin*, e que visa disponibilizar via Web um conjunto abrangente de recursos para o desenvolvimento de compiladores e componentes para compiladores. Pretende-se assim fomentar a utilização e o desenvolvimento da *framework Dolphin*, numa relação de simbiose, que permitirá aos utilizadores usufruir de um ambiente integrado e cooperativo para desenvolvimento de compiladores (mas não só). Servirá também para testar os componentes nativos e a própria solução e arquitectura da *framework Dolphin* e para promover o desenvolvimento de novos componentes por parte de elementos externos ao projecto. Convém ainda acrescentar que foi através do *Sistema Dolphin* que se detectaram os problemas e lacunas da versão original da *framework Dolphin* e que são comuns a outros sistemas, o que levou posteriormente à concepção e ao desenvolvimento da arquitectura que é proposta nesta dissertação.

O Capítulo 10 sintetiza os resultados alcançados, faz a discussão dos mesmos e do

trabalho realizado, apresenta as conclusões, e deixa algumas sugestões para o trabalho futuro.

Esta dissertação inclui também vários apêndices que visam complementar a descrição do texto principal. Os quais, apesar de longos, permitem fazer com que este documento inclua toda a informação que é essencial sobre o trabalho desenvolvido, funcionando assim como um documento de referência.

CAPÍTULO 2

Ferramentas para desenvolvimento de compiladores

Índice

O desenvolvimento de compiladores, nomeadamente de compiladores que visam produzir código altamente optimizado, é um processo elaborado e extremamente exigente, que: envolve tarefas que são implementadas através de abordagens muito distintas; contém muitas variantes; requer conhecimentos altamente especializados de áreas muito diferenciadas; e, necessariamente, tem que acompanhar uma área tecnológica com uma dinâmica ímpar, que é a do desenvolvimento de microprocessadores.

Com o objectivo de atenuar as dificuldades inerentes ao processo de desenvolvimento de compiladores e simultaneamente torná-lo mais eficiente, tem sido feita uma forte aposta na procura de soluções que permitam de alguma forma automatizar este processo. Soluções essas que resultam em ferramentas/sistemas que visam auxiliar a construção de determinadas tarefas do processo de compilação e por vezes de todo o compilador.

No entanto estas ferramentas/sistemas "sofrem" da heterogeneidade que está inerente ao processo de compilação, o que é particularmente evidente na disparidade de soluções que são utilizadas na sua implementação, que vão desde: as bibliotecas, passando pelos tipos paramétricos e pelas *frameworks*, e acabando nas ferramentas de geração. Soluções estas que, directa ou indirectamente, serão abordadas ao longo deste e do próximo capítulo.

Conhecer à priori as diversas ferramentas/sistemas, o tipo de abordagens e tecnologias que utilizam e, principalmente, analisar como funcionam na perspectiva do utilizador, é um passo importante para se alcançarem os objectivos deste doutoramento. Só assim é admissível propor soluções que contribuam para tornar a construção de compiladores mais simples. Conhecer as diversas ferramentas/sistemas é também importante para a análise posterior dos resultados. É que avaliar uma solução que vise tornar mais simples a construção de compiladores, passa por aplicá-la ou utilizá-la numa ferramenta/sistema deste tipo.

Neste sentido, pretende-se apresentar ao longo deste capítulo, aquelas que são as ferramentas e sistemas mais importantes para a construção de compiladores, focando as vantagens e desvantagens de cada solução.

É de realçar que apenas são incluídas as ferramentas/sistemas que estão de alguma forma relacionadas com o trabalho desenvolvido na preparação desta dissertação. É assim dada uma atenção especial às ferramentas/sistemas que visam o desenvolvimento integral de compiladores, nomeadamente as que assentam a sua implementação em *frameworks*, e às ferramentas que, apesar de suportarem apenas o desenvolvimento de algumas tarefas, influenciaram de alguma forma o trabalho realizado. Ficam assim de fora algumas ferramentas, que apesar de muito meritórias, não satisfazem estes requisitos. É, por exemplo, o caso do Lex [Les75], Flex [Pax88], Yacc [Joh79], Bison [Pro88], Ox [Bis92] e muitas outras.

2.1 Eli

O Eli [GHK+90] é um sistema para desenvolvimento de processadores de linguagens, que também pode ser utilizado na construção de determinadas tarefas do processo de compilação, nomeadamente de análise e síntese. Para tal, o Eli requer uma especificação, que depois de processada, dá origem ao código necessário à implementação do processador de texto.

É um sistema que integra várias ferramentas de geração, nomeadamente geradores de analisadores léxicos [Gro98a], de analisadores sintácticos [Gro98b] e de analisadores semânticos com base no cálculo de atributos [Kas97], que funcionam de forma integrada ao ponto de serem completamente transparentes para o utilizador. Contém ainda outras ferramentas mais específicas, que visam suportar o desenvolvimento de determinadas tarefas menos comuns e auxiliar o próprio processo de construção dos tradutores.

O Eli surge no contexto deste doutoramento, não tanto como um sistema que está relacionado com o assunto que é aqui abordado, mas mais como uma solução que deu um importante contributo na construção de algumas rotinas que integram a *framework* que resultou do trabalho efectuado ao longo deste doutoramento.

2.2 GENTLE

GENTLE [Sch97] é um sistema para desenvolvimento de compiladores, concebido e implementado pelo mesmo grupo que criou o BEG [ESL89] e o Cocktail Toolbox (ver Secção 2.4). O que por si só já diz muito acerca da qualidade deste sistema, mas também do tipo de abordagem utilizada.

Trata-se de um sistema de geração que permite a construção integral de compiladores. Para tal, faz uso de uma linguagem de alto nível com a qual se especificam os compiladores. O GENTLE utiliza essa especificação para gerar todas as rotinas necessárias à construção do compilador.

Esta solução, que tem por base um processo de reconhecimento e reescrita de padrões e que funciona de forma integrada com o Lex e o Yacc, permite (com algumas limitações) construir as principais tarefas do processo de compilação, nomeadamente de síntese, transformação, optimização e geração de código.

A mestria desta solução reside na uniformidade com que encara o problema da construção de compiladores. É terrivelmente simples a abordagem utilizada (o que de forma alguma significa que a implementação do GENTLE tenha sido trivial), e permite especificar todas as operações com base numa única linguagem.

É no entanto na metodologia utilizada, que garante a uniformidade e a simplicidade da solução, que está a sua principal desvantagem. No sentido de simplificar a própria especificação e tornar mais simples a construção dos compiladores, as técnicas de reconhecimento e reescrita de expressões são normalmente aplicadas em contextos muito restritos. O que até pode ser adequado para simplificar a implementação de determinadas tarefas, mas não para a implementação das muitas rotinas de análise e de optimização de código de médio e alto nível, que normalmente requerem um contexto de execução mais alargado. Mesmo permitindo integrar soluções implementadas directamente em linguagem C, que recorram a outras metodologias, as limitações do GENTLE mantêm-se dada a forma como os compiladores são especificados.

Não deixa no entanto de ser uma ferramenta interessante e especialmente adequada ao desenvolvimento de compiladores para arquitecturas específicas, como as DSP. O GENTLE surge assim no contexto deste doutoramento como prova do potencial das soluções de geração, nomeadamente quando utilizadas na construção dos geradores de código e de algumas optimizações. Mas também como prova das limitações que esta abordagem tem para desenvolver ferramentas/sistemas que visem a construção integral de compiladores.

2.3 CoSy

O CoSy [Exp03c] é uma referência incontornável no universo das ferramentas/sistemas para desenvolvimento de compiladores. Talvez represente mesmo o que de melhor se faz nesta área a nível europeu. O seu desenvolvimento, que foi feito com financiamento de vários programas comunitários, envolveu uma quantidade de recursos invulgar. Segundo documentação do próprio sistema, estima-se que mais de 120 pessoas/ano estiveram envolvidas neste projecto desde 1994. E envolveu algumas das mais sonantes instituições de investigação europeias que fazem trabalho nesta área.

O CoSy é uma *framework* que contém um vasto conjunto de soluções pré-implementadas, que funcionam sob um modelo comum de representação de código - o CCMIR [Exp03b], e integra várias ferramentas que por si só são trabalhos de referência. É o caso do PAG [Mar98], uma *framework* para desenvolvimento de rotinas de análise de fluxo de dados; do OPTIMIX [Aßm00], um sistema que, através de técnicas de reescrita de grafos, permite a construção de rotinas de análise e de optimização de código; e do Back-End Generator-BEG [ESL89, Exp03a], uma ferramenta para geração das rotinas de selecção de instruções e de atribuição de registos.

O CoSy pelo tipo de solução que utiliza, que faz uso de uma *framework* cujos componentes funcionam sob uma representação comum de código, vulgarmente designada por Representação Intermédia do Código (RIC), seria à primeira vista o sistema que maiores

semelhanças apresenta com a solução utilizada na elaboração desta dissertação e, como tal, o trabalho a utilizar como referência. Mas esse facto não se verifica por vários motivos. Um dos quais é que as semelhanças ficam-se por aqui. O CoSy aborda o problema da construção de ferramentas de compilação de forma única. É uma *framework* no verdadeiro sentido da palavra, só que esta define um modelo de cooperação entre componentes, que se caracteriza por interacções que nada têm a ver com o modelo convencional (em que os componentes são executados sequencialmente). As várias formas de cooperação são asseguradas através dos seguintes mecanismos (componentes dedicados):

- *Pipeline*: mecanismos que permitem a execução sequencial dos componentes (como vulgarmente acontece em outras soluções);
- *Data-parallel*: mecanismos que permitem colocar vários componentes do mesmo tipo, a operar sobre agregações de elementos da RIC;
- *Fork*: mecanismos que permitem colocar componentes de tipos distintos a processar concorrentemente um determinado elemento ou conjunto de elementos da RIC;
- *Loop*: mecanismos que permitem pôr um ou mais componentes a iterar sobre um elemento (ou conjunto de elementos) da RIC, até que se satisfaça determinada condição;
- *Speculative*: mecanismos que permitem colocar vários componentes de tipos distintos a operarem concorrentemente sobre um elemento (ou conjunto de elementos) da RIC, mas em que apenas será utilizado o resultado apurado por um dos componentes (os resultados apurados pelos demais componentes são desprezados);
- *Optimistic*: mecanismos que através de processos heurísticos permitem escolher, de entre vários componentes, aquele que potencialmente produzirá os melhores resultados.

Para além dos mecanismo anteriormente descritos, dos mecanismos convencionais (tarefas convencionais de compilação), e dos mecanismos específicos para a comunicação entre componentes, o CoSy contém ainda um mecanismo especial, o supervisor, ao qual cabe efectuar a gestão e garantir a utilização mais adequada dos restantes mecanismos.

Para que este modelo de compilação funcione, há que utilizar um modelo de representação de código que assegure a execução dos vários tipos de mecanismos, atendendo para tal que há mecanismos que concorrem pelos recursos (RIC), outros que cooperam para efectuar determinadas operações, outros que funcionam em "simultâneo" sobre a RIC, etc. O que requer que o modelo de representação de código possua soluções de controlo de acesso à RIC, de partilha de memória, de replicação, etc. Soluções estas que não são utilizadas em mais nenhum modelo de representação de código. Mas cuja implementação no CoSy, apesar de acarretar alguma complexidade extraordinária, contribui para fazer deste um sistema extremamente versátil, capaz de explorar como nenhum outro a aplicação dos componentes. O que faz com que seja especialmente adequado para construir compiladores para arquitecturas mais exigentes, que envolvam memória partilhada, computação paralela ou concorrencial, processamento vectorial, etc.

A versão actual do CoSy, que é comercial, é utilizada por várias empresas ligadas à construção de computadores, de microprocessadores e de ferramentas de compilação. O que por si só prova a qualidade deste sistema. Foi já utilizado ou encontra-se a ser utilizado no desenvolvimento de compiladores para vários tipos de arquitectura: CISC, RISC, DSP, NPU e VLIW (ver glossário).

Para o seu sucesso contribui, não só a abordagem utilizada, como também a versatilidade da solução e toda a logística envolvente, que faz com que este sistema disponibilize uma quantidade invulgar de recursos, dos quais se destacam os seguintes:

- Vários *front-ends*, nomeadamente para C/C++, DSP-C, Java, Fortran95 e HPF;
- Um leque considerável de rotinas de optimização de código, nomeadamente para arquitecturas mais específicas como DSP e VLIW[1];
- Várias formas de análise, incluindo algumas formas avançadas de análise de *aliases*;
- Suporte para depuração[2] de código [Exp03d];
- Um pacote dedicado ao desenvolvimento de compiladores para arquitecturas DSP;
- Versões para vários sistemas operativos (Windows, SunOS, Solaris e Linux);
- E aparentemente, documentação e suporte ao nível de tudo o resto.

Acontece no entanto que pelo facto do CoSy ter uma abordagem bastante diferente das demais soluções, por ser um sistema comercial e como tal fechado, e por haver pouca informação de índole técnico-científico, mesmo referente as ferramentas e soluções desenvolvidas no âmbito dos projectos de investigação[3], o CoSy acaba por se excluir (ou por ser excluído) dos sistemas/ferramentas que são utilizados como referência nesta área.

2.4 Cocktail - Toolbox for Compiler Construction

O Cocktail [GE90b] é um outro sistema que tem o mesmo berço que o GENTLE (Secção 2.2) e que o CoSy (Secção 2.3), isto é, a Universidade de Karlsruhe. Integra um conjunto substancial e variado de ferramentas, que permitem o desenvolvimento de quase todas as tarefas do processo de compilação.

Foi concebido para ser prático, para reduzir consideravelmente o esforço de desenvolvimento e permitir a construção de compiladores eficientes e fiáveis. A linguagem de programação nativa do Cocktail é o Modula-2, mas também permite gerar rotinas de código em C, através de uma aplicação designada por Mtc (conversor de Modula-2 para C). Contém um conjunto bastante diversificado de ferramentas, muitas das quais partilhadas com o GENTLE, mas especialmente com o CoSy, que inclui geradores de: analisadores léxicos (Rex [Gro00c]), sintácticos (Lalr e o Ell [Gro92a]), de calculadores de atributos (Ag [Gro02a]), de árvores de sintaxe abstracta (Ast [Gro02b]), de transformadores de árvores de sintaxe abstracta (Estra [Vie89]) e de *back-ends* (BEG [ESL89]). Isto para além de uma biblioteca de rotinas reutilizáveis (Reuse [Gro00b, Gro00a]).

Estranhamente, no sentido de manter a flexibilidade e o carácter prático da solução, não houve qualquer esforço de integrar as ferramentas sob uma única interface, como acontece por exemplo no GENTLE. Como a construção de um compilador requer a utilização de várias ferramentas, o utilizador é assim obrigado a fazer uso de várias linguagens de especificação (uma por cada uma das ferramentas utilizadas).

[1] O CoSy herda muitas soluções dos vários projectos que estão na sua origem, como é o caso do COMPARE (COMPilers for Parallel ARchitecturEs) e do PREPARE (PRogramming Environment for Parallel ARchitectures), que como as próprias designações indicam, visavam o desenvolvimento de compiladores e de ambientes de programação para arquitecturas paralelas.

[2] Tradução do termo Inglês *debugging*.

[3] Encontra-se alguma documentação sobre o CoSy em projectos que têm um passado comum, como é o caso do GENTLE ou do Cocktail Compiler Toolbox. No entanto não foi possível confirmar se essa informação continua válida para a versão actual do CoSy. Para complicar, uma parte significativa da documentação está escrita em Alemão e, para a grande maioria das ferramentas, apenas existe o manual de utilização.

Ainda no sentido de manter a flexibilidade e o carácter prático da solução, não foi feito qualquer esforço (salvo raros casos) de garantir que as rotinas geradas pelas várias ferramentas fizessem uso de uma representação de código pré-definida, ou que assegurassem um protocolo mínimo que garantisse a integração entre elas. Cabe assim ao utilizador assegurar que, através da especificação que faz para cada uma das ferramentas, as rotinas geradas são "compatíveis" entre si e que poderão ser integradas sem grandes esforços adicionais.

O Cocktail consiste assim, e como o próprio nome deixa antever, numa caixa de ferramentas independentes que no seu conjunto suportam o desenvolvimento de uma grande parte das tarefas de compilação. Ficam de fora a maioria das análise e optimização de código de médio e alto nível, o que faz do Cocktail adequado à construção do mesmo tipo de compiladores abrangidos pelo GENTLE, se bem que permita alguma flexibilidade adicional (à custa de um esforço extra por parte dos utilizadores).

Este sistema surge no contexto deste doutoramento, como um excelente exemplo do tipo de solução que se optou por não seguir. Enquanto a solução que foi utilizada e desenvolvida na preparação desta dissertação aposta na reutilização, o Cocktail aposta como nenhum outro sistema em soluções de geração. O que de forma alguma significa que o Cocktail é uma má solução, antes pelo contrário. Esta diferença de abordagens deve-se mais à discrepância dos objectivos. O Cocktail foi no entanto fundamental como referência em termos de soluções de geração, pois permitiu perceber até que ponto a utilização deste tipo de tecnologia serve para satisfazer os objectivos deste doutoramento.

2.5 GNU Compiler Collection

O GNU Compilers Collection-GCC [Sta00] é um projecto que visa, não a construção de ferramentas para desenvolvimento de compiladores, mas a construção dos próprios compiladores. Trata-se de um projecto, que tem a sigla da Free Software Foundation (FSF), que visa a construção de compiladores para várias linguagens e arquitecturas de computação, reaproveitando e partilhando as soluções entre si.

Em termos mais concretos, isto significa que para lá das tarefas de *front-end* que são dependentes da linguagem fonte (análise léxica, sintáctica e semântica), e das tarefas de *back-end* que são dependentes da arquitectura de computação (selecção de instruções, atribuição de registos e geração de código assembly/binário), tudo o resto é comum aos vários compiladores, nomeadamente a arquitectura e as muitas rotinas cuja a implementação não depende nem da linguagem fonte, nem da arquitectura de computação.

Apesar de ser um compilador, ou melhor dizendo vários compiladores, o GCC não deve ser menosprezado como solução para desenvolvimento de novos compiladores. Até porque existe uma comunidade bastante grande e activa que o utiliza neste sentido e são vários os motivos para que tal aconteça, a saber:

- São muitas e diversificadas as rotinas (de análise e optimização) que já se encontram implementadas, de tal forma que o GCC é um dos mais eficientes compiladores que actualmente existe;
- Como há uma grande comunidade de utilizadores, que faz uso do GCC para desenvolver compiladores, é fácil obter suporte e ter com quem trocar experiências;
- O GCC comporta alguns mecanismos que visam facilitar a sua adaptação a novas arquitecturas de computação;
- O GCC suporta oficialmente o desenvolvimento de *front-ends* para várias linguagens,

como é o caso do C/C++, Objective-C, Java, Fortran e Ada. Mas existem *front-ends* não oficiais para outras linguagens, como é o caso do PASCAL, Mercury e do COBOL;

- São vários os *back-ends* já desenvolvidos para o GCC, que permitem gerar código para várias famílias de processadores (Pentium, Alfa, MIPS, HPPA, PowerPc, Sparc, etc), incluindo arquitecturas DSP e arquitecturas de 64 bits;
- O GCC é também perfeitamente funcional como *cross-compiler*, isto é, tem a capacidade de gerar código para outras arquitecturas que não aquela em que está instalado. O que permite, por exemplo, compilar programas para sistemas integrados (*embedded systems*), nos quais nem sempre é possível executar um compilador;
- Existem versões do GCC para muitas plataformas (Windows, Linux, Solaris, Ultrix, Irix, HPUX, etc);
- E finalmente, mas de forma alguma menos importante, o GCC é distribuído sob a GNU General Public License (GGPL), que de forma resumida se pode traduzir como sendo software livre.

No que diz respeito a este doutoramento, o GCC é ainda mais importante. Não é a primeira solução que introduz o conceito de *medium-level*, isto é, da utilização de uma camada que separa o *front-end* do *back-end*, que contém todas as tarefas que são independentes da linguagem fonte e da arquitectura de computação, mas é o descendente que maior sucesso tem do sistema que introduziu este conceito, o Peephole Optimizer - PO [FD80].

A introdução do *medium-level* é fundamental para garantir a reutilização das rotinas de código, sem que para tal seja necessário efectuar qualquer tipo de alteração. O que no entanto requer que, a este nível, seja utilizado uma representação de código comum aos vários compiladores, a que se designa genericamente por RIC. A estrutura de um compilador do tipo GCC, que inclui o *medium-level*, encontra-se representada na Figura 2.1.

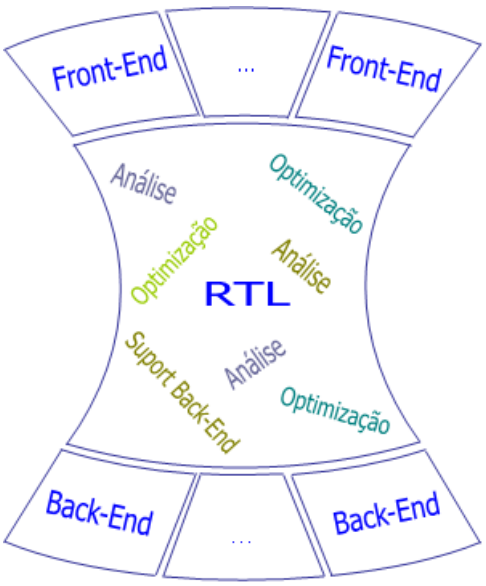


Figura 2.1: Estrutura dos compiladores GCC.

O modelo de RIC utilizado pelo GCC tem a sua origem na Register Transfer List (RTL), que foi introduzida pelo PO. Mais do que um simples modelo, a RTL introduz um conjunto de princípios que visam garantir a *portabilidade* das rotinas de compilação, a saber:

- É uma representação próxima do código final (assembly), mas independente de qualquer arquitectura de computação;
- Faz uso apenas dos principais modos de endereçamento e contém apenas operadores genéricos, que podem ser facilmente "mapeados" para qualquer tipo de microprocessador;
- Abstrai-se de todas as restrições de hardware;
- Considera que as operações são executadas, tanto quanto possível, sobre pseudo-registos (conceito abstracto de registo sem qualquer tipo de restrição).

A introdução dos conceitos de *medium-level* e de RIC, reestruturaram o modelo de compilação até aí existente, permitindo por exemplo fazer uma utilização mais intensiva do conceito de biblioteca. De notar, que mesmo uma rotina cuja implementação não dependa das características da linguagem fonte ou da arquitectura de computação, ao ser implementada no *front-end*/*back-end* pode ficar comprometida com essas características, colocando sérias entraves à sua reutilização.

Na realidade o GCC só não pode ser visto como uma grande biblioteca, porque não existe uma separação clara de tarefas, o que dificulta claramente a reutilização individual das rotinas. Esta forma intrincada de desenvolvimento, que é propositada, foi uma das soluções encontradas para evitar a "transferência" de tecnologia para fins que não visem o desenvolvimento de software livre. Pretenderam assim promover o desenvolvimento do GCC dentro do próprio projecto que é o GCC. É de relembrar, que o GCC não só é *open-source*, como é software livre, permitindo inclusive aos utilizadores alterá-lo para os fins que entenderem (desde que respeitem as condições definidas na GNU General Public License).

Não fosse este indesejável detalhe, de possuir uma estrutura intrincada, nomeadamente para quem apenas pretende utilizar o GCC para fins de investigação, e este funcionaria certamente como uma excelente biblioteca para o desenvolvimento de compiladores. O mais certo seria mesmo evoluir para uma solução do tipo *framework*, como a que foi utilizada na preparação desta dissertação.

Apesar de não ser propriamente uma solução para desenvolvimento de compiladores, o GCC é uma referência obrigatória para este doutoramento, dado que: é utilizado na construção de compiladores por uma comunidade bastante significativa; é uma ferramenta consistente com uma forte componente de optimização; possui um vasto conjunto de tarefas já implementadas; e que faz uso de um modelo de compilação muito semelhante ao que é utilizado pelas soluções proposta nesta dissertação.

Convém apenas acrescentar que são muitos os programadores, incluindo aqueles que constroem compiladores, que utilizam no seu dia a dia o GCC como compilador.

Há também que fazer justiça ao contributo do PO, pelos conceitos que introduziu, que são hoje em dia utilizados na maioria dos modelos de RIC e que estão na base do trabalho apresentado nesta dissertação, mas também em outros trabalhos bem mais sonantes, como o Zephyr (Secção 2.6) ou o RTL System (Secção 2.7).

2.6 Zephyr

O Zephyr [ADR98] é um sistema que tem por base a RTL e várias ferramentas com provas dadas, como é o caso do Very Portable Optimizer-VPO [BD88, BD94] e do New Jersey Machine Code Toolkit-NJMCT [RF95]. Visa disponibilizar uma solução realista e eficiente para desenvolvimento de compiladores.

É um sistema que ainda está numa fase inicial de desenvolvimento, mas que parte de uma base muito consistente, quanto mais não seja por envolver alguns dos nomes mais sonantes desta área, como é o caso de: Andrew Appel, um dos responsáveis pelo SML/NJ [AM91, SA94]; Jack Davidson, responsável pelo VPO, Architectural Neutral Distribution Format-ANDF [P5296, G5096] e um sem número de outras contribuições; David Hanson, um dos autores do Light C Compiler-LCC [FH95]; Norman Ramsey, um dos autores do NJMCT; e muitos outros.

O que distingue o Zephyr dos demais sistemas é a abordagem utilizada na sua construção. Pela descrição até aqui efectuada de alguns desses sistemas, é possível perceber que uma das abordagens que apresenta maiores vantagens é a da reutilização. Isto é, conceber e maximizar o número de rotinas que possam ser directamente reaproveitadas. O que passa pela utilização de um *medium-level* na estrutura dos compiladores, que por sua vez implica estabelecer um modelo comum para a RIC. O problema desta abordagem, é que não é consensual a utilização de um único modelo. Cada ferramenta/sistema faz uso do seu próprio modelo, ou até mesmo de vários modelos (consoante as tarefas a executar).

De notar que este problema também existe quando a abordagem utilizada passa pela geração das rotinas de compilação. Isto porque grande parte das ferramentas de geração visa a construção de tarefas específicas. Assim, para construir integralmente um compilador é necessário utilizar várias ferramentas, como por exemplo acontece no Cocktail Compiler Toolbox (ver Secção 2.4). Significa isto, que apesar de não ser necessário haver uma RIC comum, é de todo recomendável que exista um protocolo sobre a representação de código a utilizar entre cada duas tarefas consecutivas. Ou seja, as rotinas geradas pelas várias ferramentas têm que funcionar sobre representações de código que as tornem compatíveis entre si. O que pressupõe um convénio sobre essas representações (que como é óbvio também não existe).

O Zephyr, cuja solução caminha claramente no sentido de desenvolver uma *framework*, contempla a reutilização de rotinas, mas também a integração de ferramentas de geração. A sua abordagem difere dos demais sistemas, nomeadamente da solução sobre a qual se desenvolveu o trabalho desta dissertação, no sentido em que não exige que as ferramentas e rotinas, quer sejam do sistema ou externas a este, tenham de ser modificadas para funcionarem em conjunto.

A ideia é deixar cada utilizador fazer uso da RIC que melhor se adequa às suas necessidades. No entanto, para integrar as suas rotinas no Zephyr ou para tirar proveito das funcionalidades disponibilizadas por este sistema, o utilizador deverá especificar o modelo de RIC que utiliza. Para tal, o Zephyr disponibiliza uma linguagem própria, a Abstract Syntax Description Language (ASDL), que serve para definir modelos de representação de código cuja estrutura assente em árvores.

Com base na especificação descrita com o ASDL, o Zephyr (mais concretamente o *asdlGen*) gera uma solução para converter a RIC, do modelo do utilizador para a versão RTL do Zephyr. A solução que é gerada, e fazendo uso da terminologia utilizada na documentação do Zephyr, permite "colar" as rotinas e é automaticamente aplicada, sempre que é necessário passar de um modelo de RIC para o outro.

Esta abordagem faz do Zephyr um sistema bastante flexível, capaz de aproveitar muitas das soluções já implementadas em outros sistemas, mas também capaz de integrar sem grandes dificuldades outras ferramentas (que visem a construção de tarefas específicas). A prova de que isto é verdade, é que já existe uma especificação ASDL do Stanford University Intermediate Format - SUIF [ADH+00c], que torna possível reaproveitar muita da tecnologia já existente no SUIF Compiler System (ver Secção 2.8).

É claro que os resultados obtidos vão depender da correcta especificação das linguagens

utilizadas pelos recursos externos e pela geração de soluções que garantam a compatibilidade total dos modelos. Isto só por si levanta algumas dificuldades, dado que existe uma diversidade muito grande de modelos de RIC e que os mais recentes, como é o caso do SUIF, ou da RIC utilizada no RTL System (ver Secção 2.7), ou ainda do modelo proposto nesta dissertação (ver Capítulo 4), são semanticamente mais ricos do que a RTL original (proposta no PO).

Este problema foi contornado, alterando significativamente a estrutura e os conceitos inerentes à versão original do RTL. O que resultou num modelo semanticamente mais rico (RTL Zephry), mas que é substancialmente mais complexo que a RTL original, quer no tipo de conceitos que utiliza, quer na implementação. É no entanto importante realçar que, segundo a abordagem utilizada neste sistema, a RIC é apenas para consumo interno, isto é, apenas é utilizada pelas rotinas e ferramentas que integram o Zephyr. O utilizador não deverá ter necessidade de lidar directamente com a RTL Zephyr, mas apenas com o seu próprio modelo de RIC.

São também bastante interessantes as abordagens utilizadas na construção de *back-ends*[4]. O Zephyr disponibiliza um conjunto de linguagens declarativas, as Computer Systems Description Languages, que permitem especificar integralmente um *back-end*. Por fazer uso de várias linguagens, cada uma dedicada a descrever um determinado tipo de propriedade da arquitectura de computação, e pelo facto de serem linguagens declarativas, que descrevem propriedades e não as operações a executar, deixa por si só antever alguma da originalidade deste trabalho, que está ainda em desenvolvimento.

Na prática são quatro as linguagens que compõem a CSDL: a Specification for Encoding and Decoding-SLED [RM97] (sucessor do NJMCT), que especifica o formato das instruções *assembly* e do código binário; a λ-RTL [RD98], que especifica a semântica das RTLs e é utilizada para construir os *parsers* para a RIC; a Call Conventions Language-CCL [BD95], que serve para especificar a passagem dos parâmetros e a devolução do valor de retorno (das funções/procedimentos); e por fim, a PipeLine Unifying Notation: Graphs and Expressions (PLUNGE), que é uma notação com base em grafos que permite especificar as estruturas de pipeline dos processadores. Essa especificação serve para gerar os vectores de recursos, que são utilizados no escalonamento[5] das instruções.

A principal desvantagem do Zephyr resulta do impacto que a abordagem utilizada tem na estrutura dos compiladores. Isto porque as soluções que realizam a conversão entre modelos de RIC, vão integrar o compilador, chegando mesmo a ser utilizadas em vários pontos do processo de compilação. Significa isto, que vai haver uma deterioração do tempo de compilação e que a estrutura do compilador vai perder em uniformidade, o que se poderá traduzir em maiores custos de manutenção e de implementação.

Há ainda o perigo potencial, que resulta do facto do Zephyr ainda estar em fase de concepção e de desenvolvimento, que não dá garantias ou pelo menos confiança de que esta abordagem seja 100% funcional. Mas sem dúvida nenhuma que é interessante e que está a explorar determinadas opções que, por não serem muito vulgares, deixam muitas expectativas em aberto.

2.7 RTL System

O RTL System [JML91, MRS90a] surge com o objectivo de optimizar o código gerado pelo Typed Smalltalk [JGZ88] (um compilador para Smalltalk). Mais tarde evolui para uma

[4] É de realçar que este projecto envolve investigadores que foram responsáveis por algumas das mais importantes contribuições científicas desta área (PO [FD80], IBURG [FHP92], NJMCT [RF95], lcc [FH95], etc).

[5] Tradução da palavra inglesa *scheduling*

solução independente que visa suportar o desenvolvimento de compiladores, mas especialmente das tarefas relacionadas com a optimização de código.

Desde início que o RTL System abordou o problema do desenvolvimento de compiladores de forma substancialmente distinta do que até aí era feito. A grande maioria dos sistemas pautava-se pelo uso de técnicas de geração e por reduzir a estrutura dos compiladores às tarefas de *front-end* e de *back-end*, não potenciando assim a simples reutilização do código. As poucas excepções resultavam do trabalho desenvolvido à volta do PO (e mais tarde do VPO). O RTL System parte desta base para se impor como solução claramente orientada à reutilização de código, vinculando-se como nenhum outro sistema o tinha feito até aí aos conceitos de *medium-level* e de RIC. Aproveita também os conceitos inerentes à RTL, nomeadamente a utilização de operadores e modos de endereçamento genéricos, fáceis de "mapear" em instruções de uma qualquer arquitectura de computação; mas também o conceito de pseudo-registo, como forma abstracta e sem restrições de representar um qualquer registo físico do microprocessador.

A principal inovação do RTL System está na forma como implementa a RTL, que faz uso de uma linguagem orientada por objectos (o Smalltalk). Significa isto, que os elementos inerentes à RTL se encontram definidos como classes. É através da instanciação dessas classes que se constrói a RIC. As próprias rotinas que fazem parte das várias tarefas do processo de compilação são implementadas como classes/métodos.

A utilização de um paradigma de programação orientado por objectos altera de forma drástica e permanente o conceito de RIC, dando-lhe uma importância nunca até aí conferida. Os atributos que classificam um modelo de RIC, como é o caso da flexibilidade e da capacidade semântica da linguagem, passam a ter outra dimensão e novos atributos são introduzidos.

Em termos de flexibilidade é possível fazer tudo o que se faz com modelos e implementações mais convencionais. Passa no entanto a ser possível: especializar o tipo de operadores através de uma simples derivação de classes; e acrescentar novos operadores através da definição de novas classes. É também mais acessível: associar ou acrescentar informação aos elementos da RIC; e introduzir elementos abstractos, quer para descrever operações fictícias (puramente para consumo do próprio processo de compilação), quer para facilitar o acesso/visibilidade a entidades que estão implicitamente caracterizadas na RIC. Por exemplo, o Grafo de Fluxo de Controlo (GFC) é uma entidade abstracta que está implicitamente representada nos modelos mais convencionais de RIC e que é suficiente para a implementação/execução de muitas rotinas de análise e de optimização de código. Aceder ao GFC num modelo de RIC mais convencional, como no caso das listas de tuplos ou de árvores de expressões, nem sempre é uma operação trivial. O RTL System facilita esta tarefa fazendo do GFC uma entidade explícita da RIC. Para tal, define um conjunto de classes, os *RTLFlowNodes*, que servem para descrever o GFC.

O contributo mais importante é que, através da implementação de métodos, a RIC deixa de ser uma estrutura de dados estática e adquire dinâmica própria. Isto é aproveitado no RTL System para associar aos elementos da RIC mecanismos que permitem, por exemplo, optimizar os próprios elementos ou manter a consistência entre os diferentes níveis de abstracção.

O RTL System resulta assim numa *framework* que, para lá de disponibilizar um conjunto representativo de rotinas de compilação, nomeadamente de optimização de código, define um modelo de RIC (que tem por base a RTL) e implicitamente um modelo de compilação. A utilização desta *framework* faz-se segundo dois princípios: derivação e instanciação. A simples utilização das rotinas ou das classes que definem o modelo de RIC, apenas requer a instanciação das classes. Já a implementação de novas rotinas ou a definição de novos elementos para o modelo de RIC, requer a implementação de raiz de novas classes ou a derivação

a partir das que já existem.

O RTL System é também uma das mais importantes referências para o trabalho desenvolvido na preparação desta dissertação, o qual faz uso de uma abordagem semelhante (reutilização com base numa *framework*). O próprio modelo de RIC que é proposto (ver Capítulo 4) descende do modelo utilizado no RTL System, que para lá das diferenças de implementação e de estrutura, difere essencialmente na maturidade da solução e na clarificação de conceitos.

O RTL System é também uma das primeiras e poucas soluções que abordam o desenvolvimento de ferramentas para construção de compiladores numa perspectiva de engenharia de software. O que reforça a sua importância como referência, dado que a grande contribuição deste doutoramento é, essencialmente, uma solução de engenharia.

2.8 SUIF Compiler System

O SUIF Compiler System [ADH+00a, WFW+94] é uma referência incontornável para este doutoramento. Mais do que um sistema para desenvolvimento de compiladores, o SUIF System é um verdadeiro laboratório através do qual se pode pôr em prática e testar novas ideias. Aliás, o seu principal objectivo é disponibilizar uma infra-estrutura para investigação de técnicas de compilação para arquitecturas de elevado desempenho.

Os recursos que envolve são também muito significativos. Foi um projecto financiado pela DARPA, pela NSF e por vários organismos privados, incluindo algumas empresas de renome, como é o caso da IBM e Intel.

É também um projecto com uma dinâmica ímpar, que envolve alguns dos mais importantes investigadores desta área científica, como é o caso do John Hennessy e Steve Tjiang (do projecto Sharlit [TH92]), Monica Lam (com imenso trabalho realizado em técnicas de compilação para arquitecturas de elevado desempenho), e muitos outros.

Possui também uma comunidade de utilizadores invejável e extremamente activa, havendo bons motivos para que tal aconteça:

- Em primeiro lugar, e apesar de não ser um sistema para todo o tipo de utilizadores (como aliás assumem os autores deste sistema), disponibiliza um conjunto muito válido de recursos para investigação e desenvolvimento de novas soluções de compilação, desde as transformações de alto nível até às convencionais optimizações de fluxo de dados;
- Em segundo lugar, porque o SUIF System disponibiliza um número considerável de soluções, que incluem:
  - *Front-ends* para Fortran, C, C++ e Java (os dois últimos ainda em fase de desenvolvimento);
  - *Back-ends* para processadores da família Alpha, x86, MIPS e para linguagem C;
  - Um conjunto bastante diversificado e numeroso de soluções de análise e optimização de código para arquitecturas escaláveis, incluindo várias optimizações intra-procedimentais;
  - Soluções para paralelizar código, que incluem diversas formas de: análise que visam facilitar a detecção de paralelismo; e transformações que visam potenciar o número de situações em que é possível paralelizar o código;
  - Um modelo de RIC bastante poderoso e flexível, que contempla variantes para a implementação de optimizações de baixo nível (o MachSUIF) e para optimizações que sejam especificas de paradigmas orientados por objectos (o OSUIF);

– Ferramentas de verificação de consistência, visualização e de navegação da RIC (que facilitam a detecção de erros de origem semântica);
– Um *kernel* e um *driver* que visam facilitar a gestão, reutilização e partilha das soluções.

Para lá dos aspectos quantitativos, o SUIF System caracteriza-se por ser uma *framework* que faz uso de um modelo central de RIC (o SUIF), que define um conjunto base de primitivas para manusear a RIC e que, ao contrário do que acontece no RTL System (ver Secção 2.7), faz uma separação clara entre RIC e rotinas que implementam as tarefas de compilação.

O modelo de RIC está também implementado com uma linguagem orientada por objectos (o C++). Difere no entanto do RTL System e da solução utilizada neste doutoramento, na forma como o faz. O SUIF é uma representação muito parecida com uma árvore de sintaxe, que é no entanto independente da linguagem fonte. Esta diferença faz-se notar, por exemplo, na representação das estruturas de controlo: é que enquanto no SUIF é possível representar explicitamente estruturas de alto nível, como ciclos e expressões condicionais; já no RTL System (e também no sistema utilizado na preparação desta dissertação) este tipo de estruturas estão implicitamente representadas no GFC[6]. Isto coloca o SUIF, em termos de capacidade semântica, acima do modelo utilizado no RTL System (e também do modelo proposto nesta dissertação), que como é demonstrado no Capítulo 4 tem as suas vantagens, mas também algumas desvantagens.

É também importante assinalar que o SUIF difere do modelo convencional de uma árvore de sintaxe, no sentido que disponibiliza uma série de funcionalidades que advêm do facto de ser implementado em C++. Por exemplo, define: iteradores[7], visitantes[8] e mecanismos de replicação[9].

Um outro aspecto em que SUIF System difere de todos os outros sistemas, está na forma como as tarefas são implementadas e reutilizadas. Cada tarefa é implementada como se fosse um programa isolado, a que designaremos por componente[10], que obedece a uma API definida pela *framework*. Inicialmente, a RIC transitava entre componentes através de ficheiros. Apesar desta solução ter sido desenvolvida conscientemente, a verdade é que se traduzia em tempos de compilação excessivamente elevados. Actualmente, com a segunda versão do SUIF System, já é possível fazer com que a RIC transite entre componentes através de estruturas de dados em memória.

No entanto, a única forma prevista para que um componente possa transmitir informação aos demais, é actualizando a RIC ou acrescentando-lhe anotações (tipicamente em formato texto). Não há outros mecanismos de comunicação entre componentes e é mesmo aceite, com alguma naturalidade e indiferença, a recomputação integral de informação que já tenha sido apurada em componentes anteriores.

A versão actual do SUIF System (segunda versão) é também um excelente trabalho de engenharia de software. Os componentes são implementados através de DLLs e obedecem, conforme já foi dito, a uma API. O SUIF System disponibiliza uma shell, designada

[6]No modelo utilizado pelo RTL System e no modelo proposto nesta dissertação, as estruturas de controlo são representadas implicitamente no GFC através de três tipos de elementos: nodos condicionais, que terminam com uma operação condicional de salto; nodos simples, que terminam com uma operação incondicional de salto; e nodos de retorno, que terminam com uma operação de retorno, normalmente utilizada para representar fim de procedimento.

[7]Tradução da palavra inglesa *iterators*.

[8]Tradução da palavra inglesa *visitors*.

[9]Tradução da palavra inglesa *cloning*.

[10]O termo utilizado na literatura original do SUIF System é “pass”.

genericamente por *driver* (que corresponde à aplicação *suifdriver*), que permite carregar dinamicamente as DLL, colocando os componentes disponíveis para que possam ser utilizados como comando explícitos (da *shell*) ou para que possam ser integrados em compiladores autónomos.

Na realidade os próprios comandos nativos da *shell* são definidos como componentes que, como tal, implementam a API e que são carregados automaticamente com a activação da *shell*. Estes comandos permitem, por exemplo, incluir ou remover dinamicamente outros componentes, que tenham sido desenvolvidos pelo utilizador ou façam parte do próprio SUIF System; ou ainda carregar/gravar a RIC de/para ficheiro; ou visualizar o estado da RIC; etc.

2.9 Resumo do capítulo

Conclui-se assim a descrição das ferramentas/sistemas de maior relevo para o trabalho desenvolvido neste doutoramento, cuja principal contribuição é uma arquitectura para sistemas implementados através de *frameworks*. Nesta perspectiva, as ferramentas/sistemas mais significativos são aqueles que de forma assumida fazem uso de uma solução deste tipo e que são comparáveis/compatíveis com a solução que é aqui proposta. É o caso do RTL System e do SUIF System, que servem de referência para a avaliação das soluções apresentadas.

O gráfico da Figura 2.2 quantifica a adequação de cada um dos sistemas analisados, no desenvolvimento das tarefas de *front-end*, *medium-level* e *back-end*. Os três primeiros sistemas (Eli, GENTLE e Cocktail) são representativos dos sistemas de geração, caracterizados pela sua adequação ao desenvolvimento de *front-ends*, mas também de *back-ends*. O GCC é o sistema que mais se aproxima de uma biblioteca, adequando-se mais à construção do *medium-level* do processo de compilação. O RTL System é o único sistema cuja implementação assenta unicamente numa *framework*, que como tal só permite suportar o desenvolvimento das tarefas de *medium-level*. Os três últimos sistemas (Zephyr, SUIF System e CoSy) são representativos da aplicação de ferramentas de geração com *frameworks*, que de uma forma geral permitem suportar o desenvolvimento do *medium-level* e do *back-end*, mas por vezes também do *front-end*.

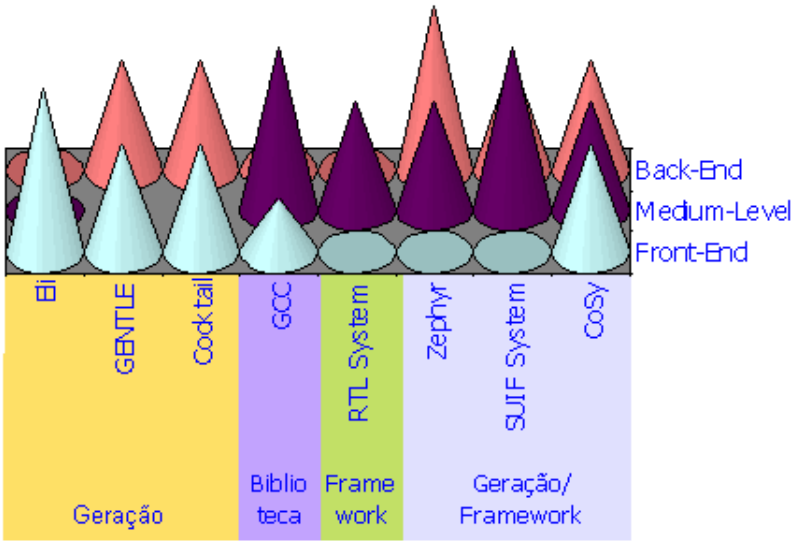


Figura 2.2: Gráfico comparativo entre os vários sistemas.

O levantamento do estado da arte continua, numa perspectiva mais especializada, no

Capítulo 4 no qual se apresentam os principais modelos de RIC e as suas características.

CAPÍTULO 3

Framework Dolphin: Versão original

Índice

No sentido de satisfazer o principal objectivo deste doutoramento, isto é, de tornar mais simples a construção de compiladores, entendeu-se partir de uma solução que claramente apostasse na reutilização, sem que no entanto impossibilitasse a utilização de ferramentas de geração.

Eram vários os motivos que levavam a apostar na reutilização, em detrimento de soluções de geração: em primeiro lugar, porque reutilizar significa pegar e fazer uso de algo que já existe, sem ter que ajustar, modificar ou alterar; em segundo lugar, porque se é totalmente verdade que as ferramentas de geração contribuem para simplificar a construção de determinadas tarefas, também é verdade que especificar essas tarefas nem sempre é simples ou acessível para a grande maioria dos utilizadores. Por exemplo, ninguém põe em dúvida a utilidade de ferramentas como o Lex e o Yacc, ou mesmo de ferramentas mais evoluídas como o Eli, mas quantas pessoas neste mundo podem afirmar ter conseguido desenvolver um *front-end* para C++ ou até mesmo para C? E ferramentas como o Improved Bottom-Up Rewrite Generator-IBURG ou o BEG que permitem construir boa parte das tarefas de *back-*

end, que são sem dúvida grandes auxiliares para quem constrói compiladores e que nem são difíceis de utilizar. Mas quantas pessoas podem afirmar ter especificado um *back-end* que permita aproveitar razoavelmente os recursos disponibilizados por um processador do tipo Pentium? A verdade é que reutilizar é muito mais simples que especificar, seja para tarefas de *front-end*, *back-end* ou de *medium-level*.

Estas questões são de tal forma relevantes, que a grande maioria dos sistemas apresentados no Capítulo 2, valem-se da reutilização para assim disponibilizarem aos seus utilizadores *front-ends* e *back-ends* para as principais linguagens de programação e arquitecturas de computação.

Não significa isto que se pretende excluir as ferramentas de geração, antes pelo contrário, pretende-se poder fazer uso dessas ferramentas, mas apenas quando não é possível reutilizar ou quando os componentes[1] que se desejaria utilizar não existem.

Praticamente todos os sistemas apresentados no Capítulo 2 permitem a reutilização e a aplicação de ferramentas de geração, mas nem todos favorecem a reutilização em detrimento da geração e nem todos disponibilizam uma solução que permita integrar ambas opções.

Para potenciar a reutilização há que apostar em força no *medium-level*, tentando deslocar o maior número de componentes para este nível do processo de compilação, e se possível disponibilizar também ferramentas que gerem componentes para este nível, permitindo assim reutilizar os componentes gerados.

O simples facto de se fazer uso de um *medium-level* por si só, já traz grandes benefícios. Nos compiladores em que este nível não existe, as tarefas têm que ser implementadas no *front-end* ou no *back-end*, mesmo que sejam integralmente ou parcialmente independentes da linguagem fonte e da arquitectura de computação. No entanto, como estão implementadas sobre um modelo de RIC que é dependente destas características (árvore de sintaxe abstracta/assembly), não poderão ser tão facilmente reutilizáveis.

Acresce ainda que nem sempre os modelos de RIC utilizados nos *front-ends* e nos *back-ends* são os mais adequados para implementar as tarefas. Por exemplo, a atribuição global de registos, feita com base no algoritmo proposto por Chaitin [CAC+81, Cha82], requer o que se designa por *inference graph*, uma estrutura de dados que caracteriza a sobreposição do período de vida útil das variáveis do programa. A computação desta estrutura, que não é uma tarefa trivial, requer informação que nem sempre está facilmente acessível nos últimos níveis do processo de compilação. Acresce ainda que, apesar de suportar uma tarefa que é completamente dependente das características do microprocessador (a atribuição de registos), a sua implementação é em grande parte independente do tipo de arquitectura de computação. Pelo que se for implementada no *medium-level*, poderá ser reutilizada sem ser necessário qualquer tipo de alteração.

É já possível tirar algumas conclusões sobre as características que uma ferramenta para construção de compiladores deve satisfazer para potenciar a reutilização de componentes. A primeira é que o modelo de compilação a utilizar deverá conter três níveis: *front-end*, *medium-level* e *back-end*. A segunda conclusão é que as tarefas implementadas sobre o *medium-level* devem utilizar um modelo de RIC comum. A terceira conclusão, é que começa a fazer sentido a utilização de uma solução do tipo *framework* em detrimento de ferramentas de geração.

Verifica-se que a utilização de *frameworks* apresenta várias vantagens. É uma solução orientada à reutilização, que tem a vantagem adicional de permitir aplicar conceitos mais abstractos, nomeadamente de carácter estrutural e comportamental, como por exemplo: definir um modelo de compilação, o tipo de entidades que utiliza e até mesmo a forma como estas se relacionam. É também uma excelente solução para integrar ferramentas de geração

[1] O termo *componente* é utilizado para designar uma rotina ou conjunto de rotinas que visem executar uma ou mais tarefas de compilação.

ou inclusive outras *frameworks*.

Inverte-se assim o papel da *framework* no contexto das ferramentas/sistemas para construção de compiladores, passando agora a ser a base de todo o sistema. Esta solução acaba mesmo por relegar as ferramentas de geração para um segundo plano, em que funcionam como simples "acessórios". Isto para além dos benefícios inerentes a qualquer *framework*, que:

- Pelo simples facto de disponibilizarem componentes já implementados, reduzem os custos de implementação, teste e avaliação;
- Para além dos componentes, disponibilizam um desenho tipo para as aplicações (modelo funcional);
- Mas também de certa forma, o conhecimento e a experiência de desenvolvimento e de implementação;
- Pelo facto das aplicações partilharem o mesmo desenho e componentes, são mais normalizados, tendencialmente mais consistentes e com menores custos de actualização e manutenção;
- Contribuem para uma prototipagem rápida dos componentes, interfaces, protocolos, e das próprias aplicações;
- Facilitam o desenvolvimento cooperativo da *framework*;
- Funcionam como um repositório de experiência.

A utilização de *frameworks* segundo este modelo, encontra-se representada na Figura 1.2.

Dentro deste contexto, apenas dois dos sistemas apresentados no Capítulo 2 satisfazem as condições requeridas. É o caso do SUIF Compiler System e do RTL System. O CoSy fica de fora por falta de informação de índole científica e por não ser um sistema aberto.

O SUIF System, apesar de ser um sistema que visa criar as condições necessárias para a experimentação e investigação de técnicas de compilação, é uma solução menos flexível no que diz respeito à realização de experiências com o próprio sistema, isto devido à sua dimensão e complexidade. O que levou a considerar o RTL System como a solução mais adequada para se desenvolver os trabalhos inerentes a este doutoramento.

O RTL System apresenta no entanto um pequeno senão: está implementado em Smalltalk, quando as linguagens de eleição utilizadas nesta área são o C e o C++. A utilização do Smalltalk coloca logo à partida problemas de compatibilidade. De notar que enquanto numa ferramenta de geração, a linguagem utilizada na sua implementação não tem que ser a mesma das rotinas geradas por essa ferramenta, já numa solução do tipo *framework* não há esta diferença. Por exemplo, muitas ferramentas do Cocktail estão implementadas em Modula-2, no entanto podem gerar código em linguagem C. Isto permite facilmente integrar esse código com soluções provenientes de outros sistemas, ou até mesmo integrar estas ferramentas noutros sistemas, não obstante o facto de estarem implementadas em Modula-2. No entanto, no caso das *frameworks* a linguagem que é utilizada na sua implementação é necessariamente a mesma dos seus componentes, isto porque são os componentes que formam a *framework*. Significa isto que implementar *frameworks* com linguagens que não as de eleição, cria sérias dificuldades à integração dos componentes com soluções provenientes de outros sistemas, mas também à utilização dessas *frameworks* como plataformas de integração de ferramentas de geração.

Estes inconvenientes enfatizaram o trabalho já efectuado no âmbito do projecto Back-End Development System - BEDS [Mat99], o qual poderia ser útil para alcançar os objectivos propostos e que, para além de estar implementado em C++, já continha uma implementação parcial do modelo de RIC utilizado pelo RTL System. Atendendo a que já havia algum *know-how* e mesmo algum código que poderia ser reaproveitado, ponderou-se e optou-se pela implementação de uma solução própria.

Contribuiu ainda para esta opção, o facto de uma solução própria ser mais maleável, menos propensa a restrições e mais incidente nos problemas que este doutoramento visa solucionar. Por um lado, isto significa que não é obrigatório utilizar soluções que não são relevantes para este doutoramento ou das quais se discorda; e por outro lado, é mais acessível fazer uso de soluções que tenham sido desenvolvidas propositadamente ou reaproveitadas de outros sistemas.

Do desenvolvimento desta solução resultou a *framework Dolphin*, que apesar de não fazer parte dos objectivos deste doutoramento, serviu *à priori* como *test-bed* para efectuar as experiências que conduziram aos resultados aqui apresentados, e a posteriori como prova que esses resultados não só são funcionais, como satisfazem em grande parte os objectivos propostos.

Este capítulo visa apresentar o que se designou por *framework Dolphin - Versão Original* (*framework Dolphin-VO*), que foi desenvolvida à imagem de outros sistemas, mas em especial do RTL System, e que foi utilizada para colocar em prática e avaliar, as soluções idealizadas neste doutoramento.

É de realçar que a implementação desta *framework*, não tinha por objectivo desenvolver uma ferramenta/sistema para construção de compiladores, previa-se apenas utilizá-la como *test-bed*. Resultou no entanto numa solução bastante aprazível, com características e uma filosofia própria. O que levou a ponderar utilizá-la efectivamente (para lá do contexto deste doutoramento) como uma ferramenta/sistema para construção de compiladores, que faz mesmo parte de um projecto maior, o *Sistema Dolphin* (ver Capítulo 9). Houve assim necessidade de lhe dar um nome (inicialmente designada por *framework Dolphin*), e posteriormente de distinguir a versão utilizada como base para o trabalho desenvolvido neste doutoramento (*framework Dolphin -VO*), da versão que resulta da aplicação das soluções aqui propostas (a qual se designa, simplesmente, por *framework Dolphin*[2]).

Para simplificar a compreensão do texto, utilizar-se-á o termo "*framework Dolphin-VO*" apenas para referir aspectos específicos da versão original deste sistema. Quando o contexto visar a versão definitiva ou características comuns entre ambas versões, será utilizado simplesmente o termo "*framework Dolphin*".

3.1 Modelo de compilação

Conceptualmente uma *framework* difere de uma biblioteca, porque pode incluir rotinas reutilizáveis, mas principalmente porque deve definir: o modelo da aplicação (a desenvolver através da *framework*); as entidades que compõem esse modelo; e a forma como essas entidades se relacionam. A *framework* reflecte assim a estrutura das aplicações a desenvolver, que neste caso são compiladores.

Já foi dito que: os compiladores para este tipo de solução são formados por um *front-end*, um *medium-level* e um *back-end*; e que as tarefas do *medium-level* funcionam sobre um modelo comum de RIC. Apesar de já dizer muito acerca do modelo de compilação, há outras

[2]Na realidade o trabalho desenvolvido neste doutoramento, permite que a versão original integre e coabite com a versão definitiva da *framework Dolphin*, de forma extremamente simples e eficiente e sem qualquer tipo de redundância (ver Secção 3.4).

características além destas que também são relevantes.

A *framework Dolphin-VO*, à semelhança do que acontece no SUIF System, separa claramente os componentes da RIC, conforme ilustra a Figura 3.1. Significa isto que nenhum elemento da RIC subentende a existência de qualquer componente (o contrário é que já não é verdade).

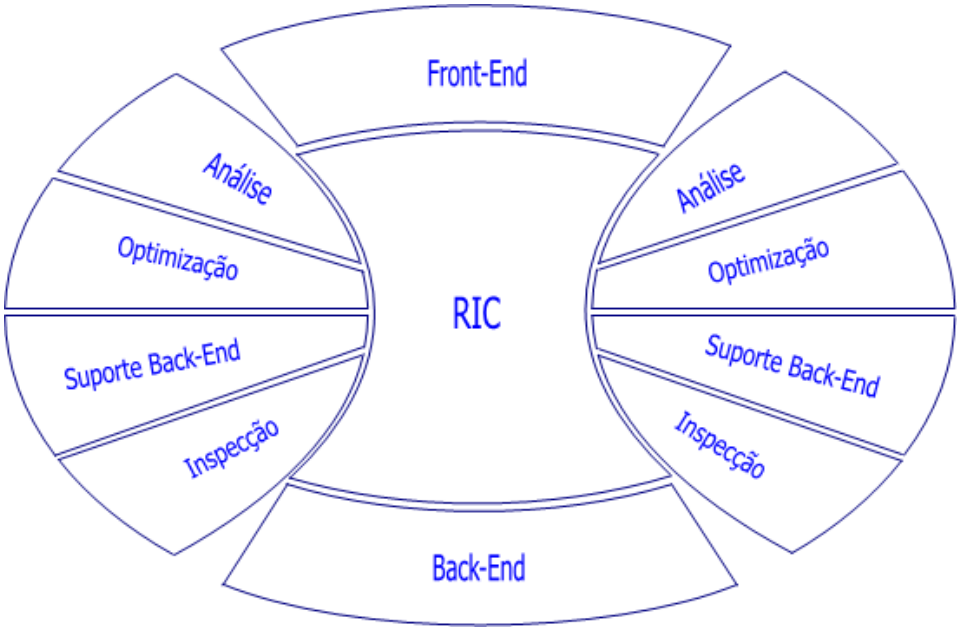


Figura 3.1: Modelo de compilação.

O conceito de "componente" aqui utilizado, identifica um módulo integrado de rotinas, que visa a realização de uma ou mais tarefas de compilação e que funciona sobre a RIC. Em termos de estrutura este é um modelo *data-centered*, em que todos os componentes funcionam sobre a RIC. Mas numa perspectiva mais formal e dado que os componentes são operados sequencialmente, o processo de compilação resulta numa composição/encadeamento de componentes, em que a RIC representa a informação que transita de componente para componente.

3.2 Componentes

Nem todos os componentes se comportam da mesma forma, ou visam realizar o mesmo tipo de operações. Para fazer a distinção, convencionou-se classificar os componentes da seguinte maneira:

- *Front-end*: componente que a partir do código fonte constrói a RIC. É um componente de presença obrigatória na estrutura de um compilador e o primeiro a ser executado. Cabe normalmente ao *front-end* efectuar a análise léxica, sintáctica e semântica e a geração da RIC.

- *Back-End*: componente que converte a RIC para um outro formato, como por exemplo: código binário, assembly ou C. Cada compilador deverá conter pelo menos um *back-end* (não é obrigatório, mas é lógico que assim seja). Um *back-end* convencional, que vise gerar código assembly ou binário, inclui pelo menos as seguintes tarefas: selecção

de instruções, atribuição de registos e geração do código final. É no entanto cada vez mais comum, conter optimizações de código, cuja implementação é dependente da arquitectura de computação.

- *Optimização*: componente cuja tarefa é optimizar a representação intermédia do programa fonte, com o objectivo de optimizar o código a gerar pelo compilador. O que passa por efectuar alterações à RIC, não em termos de modelo de representação, mas em termos de conteúdo. De notar que este tipo de optimização é independente da arquitectura de computação e da linguagem fonte.

- *Análise*: componente que, a partir da RIC, apura determinada informação sobre o programa fonte (na sua representação intermédia) submetido ao compilador. Informação que é utilizada por componentes que são executados posteriormente, nomeadamente pelas optimizações de código acima descritas. A execução deste tipo de componente não deverá provocar alterações à RIC; apenas cria novas estruturas de dados auxiliares.

- *Suporte ao Back-End*: componente específico de *Análise*, que apura informação que visa suportar a execução de determinadas tarefas de *back-end* (que não funcionam sobre a RIC), como é o caso das rotinas de atribuição global de registos.

- *Inspecção*: componente que visa apurar determinados parâmetros acerca da RIC, que permitem inferir da qualidade do processo de compilação. A utilização deste tipo de componente visa essencialmente avaliar a execução de outros componentes, nomeadamente de optimização, ou aquando da construção de compiladores, apurar a melhor sequência pela qual os componentes devem ser executados.

A representação gráfica da *framework Dolphin*, uma especialização do modelo genérico da Figura 3.1, contendo referência aos vários tipos de componentes, encontra-se na Figura 3.2

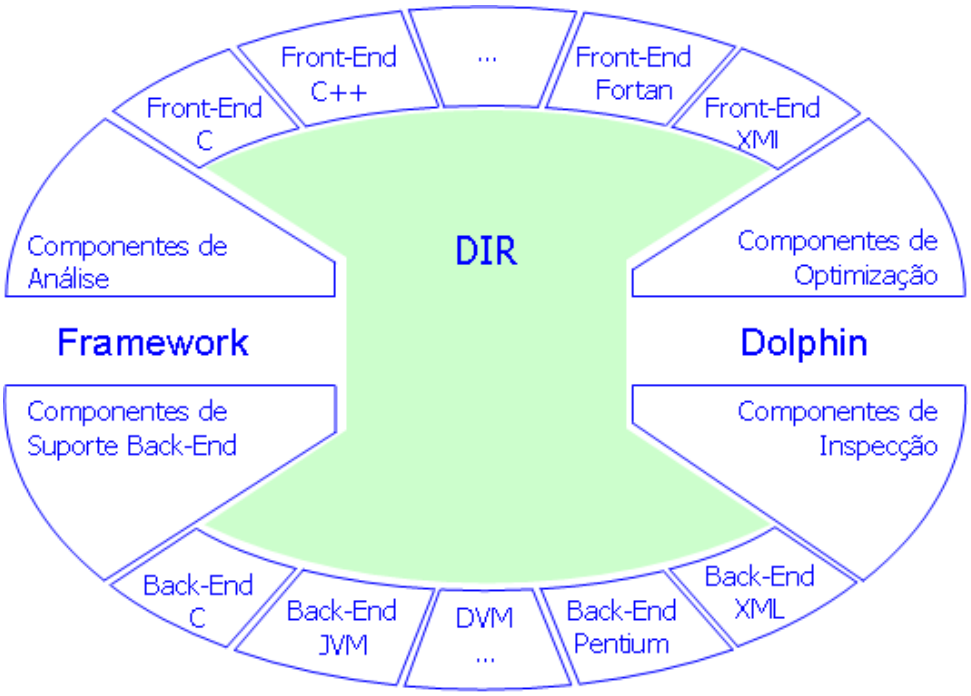


Figura 3.2: Representação gráfica da *framework Dolphin*.

Convém explicar que, à semelhança do que acontece em outros sistemas implementados através de linguagens orientadas por objectos (como é o caso do RTL System), os componentes de análise da *framework Dolphin*, retêm a informação dentro do próprio componente. Conforme é explicado no Capítulo 5 esta solução apresenta sérias vantagens comparativamente com outras soluções.

É também conveniente realçar que, à semelhança do que acontece noutros sistemas, os componentes de análise da *framework Dolphin -VO* são normalmente utilizados dentro do contexto dos componentes activos, isto é, dos componentes que alteram em conteúdo, ou em modelo a RIC (optimizações e *back-ends*). Pelo que por vezes o utilizador poderá nem sequer se aperceber da existência deste tipo de componente. Em qualquer dos casos faz sentido falar em componentes de análise quanto mais não seja em termos de implementação.

É claro que é sempre possível utilizar os componentes de análise num contexto global (como variáveis globais/externas). O que permite não só reutilizar o componente, como a informação computada por este. Mas como se demonstra no Capítulo 5 esta solução pode complicar substancialmente a construção dos compiladores.

A *framework Dolphin* é um sistema recente, que foi em grande parte desenvolvido no âmbito deste doutoramento, o que é o oposto do que acontece nos restantes sistemas aqui descritos, cujo desenvolvimento envolveu/envolve vários projectos de doutoramento. Não será assim de estranhar que o número de componentes que a *framework Dolphin* disponibiliza não seja muito grande, nomeadamente quando comparado com outros sistemas. Na realidade a maior parte dos componentes apenas foram desenvolvidos para testar as diversas soluções, nomeadamente a própria *framework*. Facto que não é grave para os resultados deste doutoramento, dado que o seu primeiro objectivo não era, como já se explicou, o desenvolvimento de um sistema para construção de compiladores. Até porque a construção de componentes nestes moldes, que não visam a investigação de técnicas de compilação, não resulta em mais valias científicas, apesar de consumir bastante tempo.

Pelo que a *framework Dolphin* contém essencialmente os componentes que foram sendo necessários para testar as ideias e as soluções apresentadas, designadamente: alguns *front-ends*, *back-ends* e várias formas de análise e de optimização de código (ver Apêndice C). A *framework Dolphin* disponibiliza também uma biblioteca de classes de uso genérico para utilização de estruturas abstractas de dados, implementadas através de *templates*.

3.3 Modelo da Representação Intermédia do Código

A *framework Dolphin* não é só formada por componentes, define também um modelo de RIC, o qual começou por ser apenas uma implementação em C++ do modelo que é utilizado no RTL System. No entanto as alterações inseridas para satisfazer os objectivos deste doutoramento, nomeadamente em termos de conceitos e de implementação, resultaram num modelo com diferenças significativas, com objectivos mais ambiciosos e com algumas inovações. Esse modelo, que é a segunda maior contribuição deste doutoramento, é apresentado no Capítulo 4 e designa-se por Dolphin Internal Representation (DIR).

Por enquanto basta saber que a DIR consiste num conjunto de classes em C++, através das quais se constrói a RIC. Às instâncias dessas classes, que são os objectos que compõem a RIC, dá-se o nome de *elementos* da RIC (ou simplesmente *elementos*).

A *framework Dolphin* é como tal composta por classes que implementam os componentes e classes que definem os objectos da RIC. Por forma a simplificar o texto, nomeadamente quando se faz referência à *framework*, convencionou-se utilizar o termo *entidade* para designar de forma indiscriminada qualquer uma destas classes (componentes+classes da DIR). Quando o contexto está relacionado com compiladores ou com o processo de compilação, o

mesmo termo serve para designar as instâncias derivadas dessas classes, isto é, os objectos que derivam dos componentes e das classes da DIR.

3.4 Interfaces

O modelo de compilação, as entidades envolvidas e a forma como se relacionam, são definidos através de quatro interfaces (classes abstractas de C++): *FObject*, *DObject*, *Protocol* e *Component*. A hierarquia destas classes encontra-se representada na Figura 3.3.

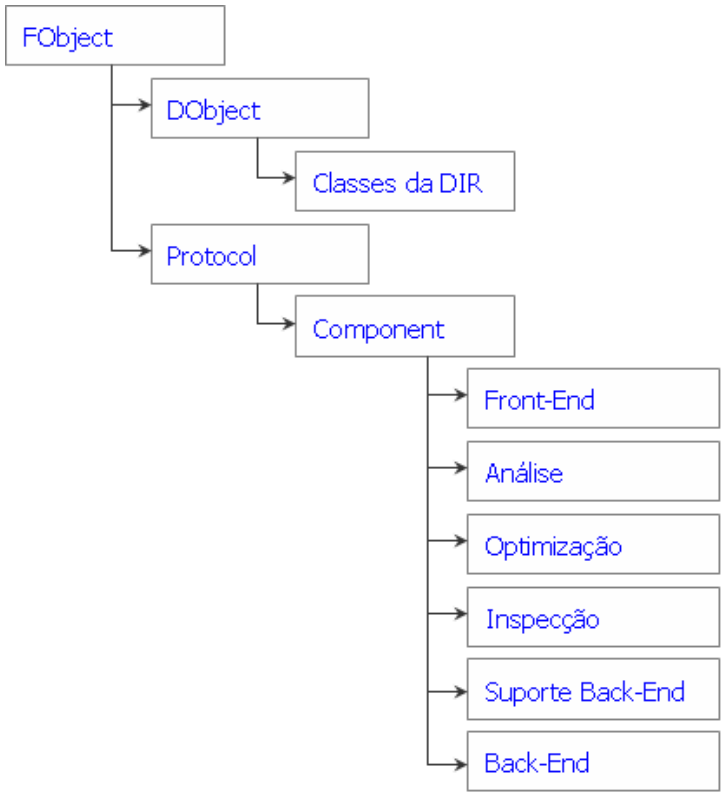


Figura 3.3: Hierarquia das interfaces da *framework Dolphin-VO*.

A interface *FObject* é herdada por todas as classes da *framework Dolphin* e como tal representa uma qualquer entidade. Requer apenas a redefinição de um método virtual que permite identificar uma qualquer classe da *framework Dolphin*. A Figura 3.4 representa a interface *FObject* e a implementação de uma hipotética classe que deriva de *FObject*.

A interface *DObject*, cuja designação provém de "*DIR Object*", deriva de *FObject* e serve para representar uma qualquer classe da DIR. É como tal um dos *hot-spots* desta *framework*, isto é, uma das interfaces que poderá ter que ser expandida se for necessário acrescentar novos elementos ao modelo de RIC.

A interface *Component*, cuja implementação está parcialmente representada na Figura 3.5, é o segundo *hot-spot* desta *framework* e aquele que deve ser utilizado na implementação de novos componentes. Contém duas variáveis: o *__cptype*, que identifica o tipo de componente de acordo com os critérios apresentados na Secção 3.2, e o *__elem* que identifica o elemento (objecto da RIC) através do qual o componente acede à RIC.

As interfaces até aqui apresentadas servem essencialmente para representar as entidades existentes na *framework Dolphin* e, indirectamente, o modelo de compilação. Mas para lá do

```
(1)     class FObject{
(2)        protected:
(3)           FObject(){}
(4)        public :
(5)           virtual ~FObject(){}
(6)           virtual const char *getClass()=0;
(7)     };
(8)
(9)     class X : public FObject{
(10)       private :
(11)          // Variáveis de classe
(12)          static const char *id;
(13)       public :
(14)          // Implementação das interfaces
(15)          // Interface FObject
(16)          const char *getClass(){return id;}
(17)    };
(18)    const char *X::id="X";
```

Figura 3.4: Interface *FObject* e exemplo de implementação.

registo dos elementos nos componentes, efectuado através de *Component::setElem(DObject*)*, não definem, nem impõem qualquer condição à forma como os componentes interactuam com os elementos da RIC, à semelhança do que acontece nos demais sistemas analisados. O que aliás vai de encontro aos objectivos para os quais a *framework Dolphin - VO* foi desenvolvida, isto é, tornam-na mais flexível e adaptável à realização das experiências e testes.

No entanto o relacionamento entre componentes e elementos da RIC, prometia ser uma área com potencial para se obter resultados que fossem de encontro aos objectivos deste doutoramento (o que aliás se confirmou), nomeadamente para facilitar a reutilização dos componentes, garantindo a qualidade do processo de compilação e do código gerado pelos compiladores. Até porque era um assunto pouco explorado e não havia nenhum sistema que apresentasse resultados neste sentido (muito provavelmente por considerarem esta questão pouco relevante). Sem querer perder flexibilidade e tornar a *framework Dolphin - VO* menos adaptável, entendeu-se que era importante disponibilizar um conjunto de funcionalidades mínimas que permitissem explorar este assunto.

Introduziu-se assim a interface *Protocol*, que se encontra representada na hierarquia da Figura 3.3 e que se encontra definida na Figura 3.6. Deriva de *FObject* e é herdada exclusivamente por *Component*. O objectivo desta interface é gerir a forma como os componentes interagem com os elementos da RIC. Cada solução encontrada corresponderá a um protocolo, o qual deverá ser implementado pelos componentes reescrevendo os métodos *bool* $execute_i()$. Por exemplo, o protocolo $P0$ (que corresponde ao comportamento por omissão), está associado ao método *bool* $execute_0()$. O componente é executado através da evocação do método *bool execute*(), que por sua vez evoca o método *bool* $execute_i()$, correspondente ao protocolo em uso, isto é, ao protocolo definido pela variável *__protocol*.

Esta solução não só foi útil para experimentar muitas das soluções idealizadas, como acabou por permitir que a versão da *framework Dolphin* apresentada neste capítulo (*fra-*

```
(1)     class Component:public Protocol{
(2)       protected:
(3)          enum {UNDEFCOMP,FRONTEND,ANALYSES,OPTIMIZATION,
(4)                BACKEND,SUPPORT,INSPECTION} ___cptype;
(5)          // Variáveis do objecto
(6)          DObject *___elem;
(7)          // Métodos do objecto
(8)          void setType(COMPTYPE);
(9)          // Construtores
(10)         Component();
(11)         Component(DObject*);
(12)      public :
(13)         // Destrutor
(14)         virtual ~Component();
(15)         // Métodos do objecto
(16)         void setElem(DObject*);
(17)         DObject *getElem();
(18)         COMPTYPE getType();
(19)    };
```

Figura 3.5: Interface *Component*.

mework Dolphin-VO), coabite com a versão que resultou da aplicação das soluções propostas nesta dissertação, permitindo utilizar uma ou outra mediante a simples alteração da variável *__protocol*[3].

De notar que esta solução não implica a duplicação dos componentes, ou mesmo dos métodos. Apenas da parte que está estritamente ligada ao protocolo, o que requer a reescrita dos métodos *bool* $execute_i()$ e do método *void* $setElem(DObject*)$.

3.5 Exemplo de utilização

A construção de compiladores através da *framework Dolphin* deverá passar em grande parte pela reutilização dos componentes, que é a opção desejável. No entanto quando um determinado componente não existir, o utilizador terá que optar pela opção menos desejável, que consiste em implementar o componente segundo o modelo de compilação e as interfaces definidas pela *framework*. Como o objectivo deste doutoramento é tornar a construção de compiladores mais simples, implica fazer com que a reutilização, mas também a implementação de componentes, sejam ambas tarefas simples e fáceis de realizar.

Em virtude de se estar a utilizar uma *framework*, a reutilização é uma tarefa bastante simples. Consiste em instanciar o componente desejado, efectuar o registo da RIC no componente e executá-lo, conforme ilustra o exemplo da Figura 3.7 (*Program* é uma das classes da DIR).

A reutilização dos *front-ends* é ligeiramente diferente. Nestes casos as operações a realizar são as que se encontram representadas na Figura 3.8:

[3] A versão definitiva da *framework Dolphin* encontra-se implementada através do protocolo P1.

```
(1)    class Protocol:virtual public FObject{
(2)      protected:
(3)        // Variáveis do objecto
(4)        PROTOCOL ___protocol;
(5)        // Métodos do objecto
(6)        virtual bool execute0(){return false;}   // Protocolo P0
(7)        virtual bool execute1(){return false;}   // Protocolo P1
(8)        virtual bool execute2(){return false;}   // Protocolo P2
(9)        ...
(10)       virtual void setProtocol(PROTOCOL);
(11)       // Construtor
(12)       Protocol();
(13)     public :
(14)       // Destrutor
(15)       virtual ~Protocol();
(16)       // Métodos do objecto
(17)       virtual bool execute();
(18)       virtual PROTOCOL getProtocol();
(19)   };
```

Figura 3.6: Interface *Protocol*.

```
(1)    // Opção 1                    // Opção 2
(2)    Program *p=...;               Program *p=...;
(3)    elimComSubExpr ecse(p);       elimComSubExpr ecse();
(4)    ecse.execute();               ecse.setElem(p);
(5)                                  ecse.execute();
```

Figura 3.7: Reutilização de um componente.

A Figura 3.9 contém a especificação integral de um compilador que possui: um *front-end* (littleC); um *back-end* (*genX86*), que é aplicado duas vezes; e algumas optimizações. Alguns destes componentes integram outros componentes, nomeadamente do tipo Análise e de Suporte ao Back-End.

Esta simplicidade tem no entanto um preço bastante elevado, nomeadamente em termos da eficiência do processo de compilação. A principal contribuição deste trabalho de doutoramento é uma arquitectura que permite manter a simplicidade, sem no entanto deteriorar a eficiência do processo de compilação (ver Capítulo 5).

Já no que diz respeito à implementação de componentes, a solução proposta nesta dissertação passa pela concepção de um modelo de RIC (a DIR), cuja concepção foi feita no sentido de facilitar o desenvolvimento de novos componentes, conforme é explicado no Capítulo 4.

```
(1)     Program *p;
(2)     littleC fe(...);
(3)     fe.execute();
(4)     p=fe.getElem();
```

Figura 3.8: Reutilização de um *front-end*.

```
(1)     int main(int argc,char *argv[]){
(2)       littleC fe(argc,argv);
(3)       fe.execute();
(4)       Program *p=fe.getElem();
(5)       if(p){
(6)         elimUnreachCode euc(p);
(7)         euc.execute();
(8)         elimJumpChains ejc(p);
(9)         ejc.execute();
(10)        genX86 gx(p);
(11)        gx.setFile("output1");
(12)        gx.execute();
(13)        elimLoads el(p);
(14)        el.execute();
(15)        gx.setFile("output2");
(16)        gx.execute();
(17)      }
(18)      return 0;
(19)    }
```

Figura 3.9: Exemplo da especificação de um compilador.

3.6 Resumo do capítulo

Conclui-se assim este capítulo que serviu para apresentar a *framework Dolphin-VO*, sobre a qual se desenvolveram as soluções propostas nestas dissertação. É no entanto de destacar que, apesar deste ser o ponto de partida para o trabalho que é relatado neste documento, a realidade é que boa parte deste sistema foi desenvolvido no âmbito deste doutoramento.

Apesar da sua simplicidade, é um sistema que apresenta algumas mais valias em termos científicos. Por exemplo, a relação entre componentes e RIC, que aparentemente não tem muito que se lhe diga, difere no entanto da solução que é utilizada no SUIF System. Neste sistema os componentes são associados à RIC, enquanto que na *framework Dolphin* os componentes são associados a elementos da RIC. Esta pequena diferença permite aos utilizadores ter um controlo mais acurado sobre os componentes, garantindo maior flexibilidade na especificação dos compiladores. Mas é também fundamental para garantir a eficiência do

processo de compilação, conforme se explica no Capítulo 5.

CAPÍTULO 4

Framework Dolphin: Modelo de Representação Intermédia do Código

Índice

Conforme explicado no Capítulo 3, a utilização de *frameworks* como sistemas para construção de compiladores apresenta várias vantagens comparativamente com outro tipo de soluções. No entanto, o sucesso deste tipo de solução está dependente da utilização de um *medium-level* na estrutura dos compiladores, que seja independente da linguagem fonte e da arquitectura de computação. É através deste *medium-level*, que uma *framework* potencia e

disponibiliza as condições necessárias à implementação de componentes reutilizáveis, isto é, que podem ser utilizados sem necessitarem de qualquer tipo de alteração. Mas para que tal seja possível, é de todo conveniente que os componentes que operam a este nível façam uso de um modelo comum de Representação Intermédia do Código (RIC).

O modelo de RIC é assim uma peça chave para o sucesso desta abordagem, pelo que a sua escolha é uma questão delicada e que deve ser bem ponderada. Isto porque tem de servir para representar código proveniente das mais diversas linguagens de programação (correspondentes a diferentes paradigmas), tem de suportar um conjunto bastante grande e diversificado de componentes de análise, optimização e tradução, e tem de ser adequado às necessidades e exigências das tarefas de *back-end*.

Acresce a tudo isto, o facto de se pretender contribuir com este trabalho para tornar a construção de compiladores mais simples, o que se pretende conseguir pela conjugação de duas estratégias distintas: uma que visa facilitar a utilização dos componentes, salvaguardando a qualidade do processo de compilação e do código a gerar pelos compiladores; e outra que vai no sentido de assegurar as condições mais adequadas à implementação de novos componentes. Esta última está inevitavelmente relacionada com o modelo de RIC que é utilizado.

Este capítulo começa por descrever o tipo de características requeridas a um modelo de RIC. De seguida faz um pequeno levantamento do estado da arte, no que a este assunto diz respeito, e depois apresenta a Dolphin Internal Representation (DIR), o modelo proposto nesta dissertação. Conclui com um pequeno resumo no qual se faz a comparação dos vários modelos, incluindo o modelo proposto.

4.1 Modelos de RIC: Características

É cada vez mais perceptível, que o modelo de RIC tem um papel fundamental no processo de construção de compiladores e como tal, no desenvolvimento de sistemas para construção de compiladores que tenham por base *frameworks*. Mas a verdade é que são poucos os estudos feitos que visem este assunto em concreto, nomeadamente no que concerne ao desenho, implementação e avaliação de modelos de RIC. Surge na maioria da bibliografia como um assunto secundário ou subentendido, não tendo mesmo sido encontrado um único artigo em que este fosse o tema principal.

É por exemplo notória a falta de requisitos que permitam avaliar um modelo de RIC. As poucas excepções encontradas, provêm da bibliografia referente ao Register Transfer List-RTL, que apresenta alguns requisitos, que apesar de se manterem válidos são insuficientes para caracterizar os modelos de RIC mais recentes.

Atendendo à relevância que o modelo de RIC tem para o tipo de solução utilizada na preparação desta dissertação, foi fundamental definir requisitos que ajudassem a classificar e a escolher o modelo mais adequado. Neste sentido, entendeu-se que um modelo de RIC deve ser:

Genérico: Capacidade para representar código proveniente de uma qualquer linguagem fonte, sem estar condicionado a detalhes relacionados com a arquitectura de computação.

Flexível: Capacidade para suportar novos operadores, novos modos de endereçamento ou outros detalhes inevitáveis que estão relacionados com a linguagem fonte (como é o caso da tabela de identificadores), sem que isso comprometa a utilização dos componentes já desenvolvidos.

Extensível: Capacidade de albergar e manter informação que não pertencendo estritamente

à RIC, está de alguma forma relacionada com esta ou com alguns dos seus elementos.

Fácil de utilizar: Adequação do modelo à construção da RIC e à implementação de novos componentes.

E deve também disponibilizar:

Mecanismos de abstracção: Que permitam aceder, integralmente ou parcialmente, à RIC segundo diversos níveis de abstracção. Por exemplo, aceder à RIC como sendo apenas um conjunto de funções/procedimentos, ou como sendo um grafo de fluxo de controlo. Esta característica é particularmente útil para simplificar a implementação de novos componentes, dado que permite ao utilizador ignorar determinados detalhes acerca da RIC.

Mecanismos de consistência: Que permitam assegurar a consistência entre os diversos níveis de abstracção e/ou entre RIC e estruturas anexas de dados. Este requisito só tem razão de ser, se o modelo de RIC admitir diversos níveis de abstracção e/ou se for extensível.

É fácil compreender que a simplicidade de um modelo, contribui certamente para que seja fácil de utilizar. Mas o mesmo também acontece se o modelo disponibilizar mecanismos de abstracção e de consistência. No entanto a introdução destes mecanismos pode retirar simplicidade ao modelo. O objectivo é assim encontrar uma solução que combine da melhor forma possível os vários requisitos.

É de realçar a posição antagónica que aqui se assume em relação ao RTL, no que diz respeito à flexibilidade do modelo de RIC. Ao contrário do que acontece no RTL, em que se fomenta a utilização de um número reduzido e genérico de operadores e de modos de endereçamento (fáceis de "mapear" numa qualquer arquitectura de computação). Aqui valoriza-se o facto do modelo permitir introduzir novos operadores e modos de endereçamento, desde que isso não inutilize os componentes de *medium-level*. O objectivo é assim enriquecer semanticamente o modelo de RIC. Até porque a opção feita no RTL estava relacionada com as dificuldades então existentes em "mapear" a RIC em instruções assembly/código máquina. Operação que quando mal efectuada degrada consideravelmente a qualidade do código gerado. Mas actualmente, com soluções como o Bottom-Up Rewrite Generator-BURG [FHP91], o Back-End Generator-BEG [ESL89] e o Improved Bottom-Up Rewrite Generator-IBURG [FHP92], a selecção das instruções não só é mais acessível, como é possível assegurar a solução óptima segundo valores atribuídos à priori ou calculados aquando do processo de selecção.

4.2 Modelos de RIC: Estado da arte

Ao longo do processo de compilação o código é representado de diversas formas, desde uma simples sequência de caracteres, até ao assembly ou código binário. Após a execução das tarefas base do *front-end*, o código encontra-se normalmente sob a forma de uma árvore de derivação, eventualmente decorada com atributos. Ambas as formas são no entanto dependentes da linguagem fonte, o que não permite que sejam utilizadas no *medium-level* como RIC. É assim necessário procurar alternativas que satisfaçam os critérios anteriormente definidos.

4.2.1 Tuplos e árvores de expressões

Os modelos mais convencionais têm por base tuplos ou árvores de expressões. Os tuplos são uma forma de representação de muito baixo nível, próximo do assembly, mas contudo

genérica o suficiente para ser independente de qualquer arquitectura. Cada tuplo representa um operador e zero ou mais operandos (tipicamente não mais do que três). Utilizando apenas um conjunto genérico de operadores, é possível garantir que cada tuplo terá correspondência mais ou menos directa a uma instrução ou conjunto de instruções em assembly/código máquina.

É claro que de todas os requisitos anteriormente definidos, apenas dois fazem sentido para este modelo: é genérico e é flexível. Por ser uma representação de muito baixo nível (em termos de RIC), não possui determinado tipo de informação que é fundamental para a execução de algumas formas de análise e de optimização de código. De forma geral, não é dos modelos mais fáceis de utilizar na implementação das diversas tarefas de compilação, excepto para as optimizações de baixo nível (*peephole optimizations*). Por outro lado, e pela proximidade que tem com o código final, é relativamente fácil converter esta forma de representação para assembly ou código binário.

Convém apenas acrescentar que a RTL original (Register Transfer List), isto é o modelo introduzido pelo PO, utiliza tuplos. Acrescenta no entanto alguns novos conceitos/restrições, dos quais se destacam os seguintes: as operações são efectuadas exclusivamente sobre pseudo-registos (excepto as de *load* e de *store*); os pseudo-registos são uma representação abstracta de registo que se caracteriza por não ter qualquer restrição de hardware, por exemplo, considera-se que o número de pseudo-registos é ilimitado; e faz apenas uso de operadores e modos de endereçamento genéricos e fáceis de "mapear" para uma qualquer arquitectura de computação.

As árvores de expressões são árvores binárias, em que os nodos terminais representam os operandos e os nodos intermédios, os operadores. A geração da RIC (lista de árvores de expressões) é relativamente acessível de efectuar a partir da árvore de derivação. Cada nodo intermédio tem uma correspondência mais ou menos directa a uma instrução de assembly/código binário. A passagem de valores está implícita na própria estrutura das árvores de expressões (não existe um conceito explícito de pseudo-registo). São uma alternativa aos tuplos, ligeiramente mais abstracta, com maior flexibilidade e mais fácil de utilizar. Existem algoritmos muito eficientes para efectuar o reconhecimento e reescrita de padrões sobre árvores, e que são particularmente adequados à implementação de algumas optimizações de código e à selecção de instruções. Não é por acaso que alguns dos modelos de RIC mais recentes, aplicam os conceitos da RTL sobre árvores (GCC, Zephyr, etc).

4.2.2 Modelo do GNU Compiler Collection

O GNU Compilers Collection (GCC) que, conforme foi explicado na Secção 2.5, tem boa parte da sua estrutura assente no *medium-level*, faz uso de um modelo de RIC que tem por base a RTL. É no entanto implementado sobre árvores de expressões.

Pelo seu carácter prático, eficiente e por já ter provas dadas, é inevitável a referência a este modelo. Mas apesar de funcionar muito bem no contexto em que se insere, não satisfaz os critérios necessários a um modelo de RIC. Por exemplo: os operadores existentes já se encontram definidos e não há como os alterar ou modificar (sem que isso signifique ter que reescrever grande parte das rotinas de código); as próprias estruturas utilizadas na representação das expressões não são homogéneas (diferem de operador para operador); e há informação de carácter muito específico associada a cada expressão. Apesar de estes detalhes/características fazerem todo o sentido no GCC, desvirtuam completamente este modelo de RIC como solução genérica.

4.2.3 Modelo do Zephyr

Uma referência que também é importante no que diz respeito a modelos de RIC, é a utilizada pelo Zephyr. É certamente o sucessor mais legítimo da linha de descendência que provém da RTL utilizada no PO e que teve continuação no VPO. Até porque os três projectos têm um ponto em comum, o seu autor que é Jack Davidson.

A versão original da RTL foi proposta no âmbito do PO, uma solução para a implementação de rotinas de optimização de baixo nível (*peephole optimizations*), de forma independente da arquitectura de computação. Funciona como uma espécie de *retargetable optimizer* e tudo graças ao facto da RTL apenas fazer uso de operadores e modos de endereçamento genéricos, que são facilmente convertidos em instruções de um qualquer processador.

O Zephyr tem no entanto objectivos mais ambiciosos do que o PO, que exigem um modelo de RIC com maior expressividade semântica. Neste sentido, desenvolveram o que aqui designaremos por Zephyr RTL, que tem por base o RTL original, mas que difere em alguns detalhes de implementação, em grande parte relacionados com o tipo dos operadores e dos operandos. Por exemplo:

- Utiliza árvores de expressões;
- Não existe ambiguidade entre operadores. O que significa que cada operador contém um identificador e um tipo bem definido (tamanho, deslocamento, ordem dos bits, etc);
- Cada variável ou espaço de memória é "visualizado" de forma unívoca em relação ao seu tipo. O que significa que o mesmo endereço não pode corresponder a dois tipos diferentes;
- Todas as operações de *fetch* e de conversão de tipos são explícitas;
- Todas as operações de *store* encontram-se anotadas com o tipo do operando;
- As operações de conversão entre tipos especificam a forma como a conversão se efectua, nomeadamente em relação à ordem e deslocamento dos bits.

A introdução destes detalhes, que visam enriquecer semanticamente a RIC, é justificada como sendo fundamental para se desenvolver o que poderemos designar por *retargetable applications*. Isto é, aplicações que apesar de serem dependentes da arquitectura de computação, possam ser facilmente "mapeáveis" em novas arquitecturas. A ideia é aplicar o mesmo tipo de solução utilizada no *retargetable optimizer* (PO), para desenvolver simuladores, *linkers*, depuradores (*debuggers*) e outras ferramentas, que sejam fáceis de adaptar a novas arquitecturas.

Em termos práticos, este modelo é semanticamente mais rico que o RTL original, mas abre mão de alguns princípios base deste último, nomeadamente no que diz respeito à simplicidade. O que não é grave, até porque o Zephyr RTL é, conforme explicado na Secção 2.6, utilizado apenas internamente pelas ferramentas do Zephyr. Não está como tal, visível para os utilizadores, dado que estes fazem uso do seu próprio modelo de RIC.

Em termos dos requisitos anteriormente definidos, pode-se dizer que o Zephyr RTL, à semelhança do que acontece no RTL original, é um modelo suficientemente genérico. Mas também é um modelo interno que, como tal, não tem que satisfazer este tipo de requisito, isto é, pode assumir determinados compromissos que são razoáveis dentro do sistema Zephyr, mas que não sejam aceitáveis para outros sistemas.

Também não é flexível, porque tal não é desejado (vai contra o princípio de utilizar operadores e modos de endereçamento genéricos). Quanto ao resto, e apesar de ser semanticamente mais rico que os outros modelos entretanto apresentados, não acresce mais valias.

Quando muito perdeu alguma facilidade de utilização que, conforme já foi explicado, não é grave.

Ao contrário do que acontece com outras abordagens igualmente recentes, nomeadamente em relação ao modelo proposto nesta dissertação, o Zephyr mantém a aposta num modelo de baixo nível, tão próximo quanto possível do código final, sem no entanto se comprometer com detalhes relacionados com a arquitectura de computação. O que cria algumas expectativas em relação a este modelo. É mesmo caso para dizer que o Zephyr acaba por ser ousado, por não ousar.

4.2.4 Modelo do RTL System

O modelo de RIC utilizado pelo RTL System, que será aqui designado por RTLS, é uma das referências incontornáveis para este doutoramento. Até porque serviu de base ao modelo proposto. Também aplica os princípios definidos na versão original do RTL, no entanto difere substancialmente na forma como implementa o modelo.

A principal diferença advém do facto de estar implementado através de uma linguagem orientada por objectos, mais concretamente Smalltalk. Por este facto, permite dotar a RIC de funcionalidades que seriam difíceis de implementar sem o recurso a uma linguagem deste tipo.

O modelo é como tal definido através de um conjunto de classes, cuja instanciação permite construir a RIC. Essas classes correspondem a entidades genéricas que podem ser encontradas numa qualquer linguagem do tipo imperativo. Há essencialmente dois tipos de classes, a saber:

***FlowNode*:** Família de classes que representam as estruturas que controlam o fluxo de execução do programa. Cada objecto desta classe corresponde a um elemento do Grafo de Fluxo de Controlo (GFC);

***RTLExpression*:** Família de classes que representam as expressões do programa (à excepção das expressões que definem as estruturas de fluxo de controlo).

Estas duas famílias de classes permitem três níveis de abstracção. O nível mais alto é obtido através da classe *RTLProgram*, que é utilizada para representar o programa submetido ao compilador. Esta classe deriva de *FlowNode* e mais especificamente de *IntervalNode*, que é uma classe capaz de conter outros objectos do tipo *FlowNode*. No caso do RTLProgram acresce ainda alguns objectos, como o *MachineDescription* que caracteriza a arquitectura de computação.

O segundo nível de abstracção é disponibilizado também através de *RTLProgram*, mas restrito ao GFC. Com este nível de abstracção é possível aplicar e desenvolver rotinas de análise e de optimização de fluxo de controlo, sem ter que lidar com outros detalhes da RIC.

O nível mais baixo de abstracção, é disponibilizado através dos objectos derivados de *RTLExpression*. Convém no entanto explicar que o conceito de níveis de abstracção surge no RTLS de forma implícita. A bibliografia nada refere a este tipo de funcionalidade. O que é particularmente notório neste nível mais baixo de abstracção em que, ao contrário do que deveria acontecer, o utilizador apenas tem uma visão localizada da RIC. Isto porque o acesso às *RTLExpressions* faz-se através dos *FlowNodes*, pelo que a visão que o utilizador tem da RIC está restrita ao contexto do *FlowNode* que está a utilizar para aceder às *RTLExpressions*.

Conforme já se disse, as estruturas de controlo do programa são representadas exclusivamente através de objectos de classes derivadas de *FlowNode*. Classes essas que correspondem aos elementos base de um GFC e através das quais se constroem estruturas mais

elaboradas, nomeadamente estruturas condicionais do tipo *if ... then ...* e estruturas cíclicas, do tipo *while ... do ...*. Essas classes são: *RTLReturnNode*, nodo do GFC que termina com uma operação de retorno; *RTLJumpNode*, nodo que termina com uma operação de salto incondicional; *RTLConditionalNode*, nodo que termina com uma operação de salto condicional; e (estranhamente) *RTLCallNode*, nodo que termina com uma invocação de uma função/procedimento, seguido de uma operação de salto.

A família de classes que deriva de *FlowNode* encontra-se hierarquicamente representada na Figura 4.1(a). É de salientar que *FlowNode*, enquanto classe base que representa um qualquer nodo do GFC, mantém as referências dos nodos antecessores; e que determinadas classes que derivam de *FlowNode*, como é o caso do *RTLJumpNode* e do *RTLConditionalNode*, mantêm também as referências dos nodos sucessores.

```
FlowNode
   RTLFlowNode
      RTLJumpNode
         RTLCallNode
         RTLConditionalNode
      RTLReturnNode
   RTLIntervalNode
      RTLProgram
```

(a) Hierarquia das classes derivadas de *FlowNode*.

```
RTLExpression
   Storage
      Register
         SpecialRegister
         BitRegister
      Memory
   MemoryAccess
   AbstractUpdate
      RegisterUpdate
      MemoryUpdate
   UnaryExpression
   BinaryExpression
   Constant
```

(b) Hierarquia das classes derivadas de *RTLExpression*.

Figura 4.1: Classes da família *FlowNode* e *RTLExpression*.

À excepção da parte que diz respeito às estruturas de controlo, o resto da RIC é formada por árvores de expressões. Essas árvores são construídas a partir das classes derivadas de *RTLExpression*, das quais se destacam as seguintes: *Storage*, que serve para representar registos e variáveis; *Constant*, que serve para representar constantes; *UnaryExpression*, que serve para representar operações unárias; e *BinaryExpression*, que serve para representar operações binárias. As classes derivadas de *RTLExpression* encontra-se representada hierarquicamente na Figura 4.1(b).

O RTLS introduz também os *RegisterTransfers*, uma família de classes que visa principalmente caracterizar as dependências entre dados, mas que permite também ter uma perspectiva linear da RIC e assinalar as operações que controlam o fluxo de execução do programa. O *RegisterTransfer* é a classe principal, que entre outras variáveis, possui dois conjuntos: *flowDependents*, que contém os *RegisterTransfers* que fazem uso dos valores apurados pelo actual; e *flowSupporters*, que contém os *RegisterTransfers* que apuram os valores utilizados no actual *RegisterTransfer*. Na prática, a utilização dos *RegisterTransfers* permite manter, como parte da RIC, o Grafo de Dependências de Dados (GDD).

Os *FlowNodes*, *RegisterTransfers* e *RTLExpressions* estão relacionados da seguinte forma: cada *RTLExpression* está associado a um *RegisterTransfer*; cada *RegisterTransfer* pertence a uma lista que faz parte de um *FlowNode*. Esta relação está ilustrada na Figura 4.2, em que *RTLJumpNode* é o *FlowNode*, *Assignment* é o *RegisterTransfer*, e *Mem*, *Add*, *Cnst*

e *Neg* são *RTLExpressions*.

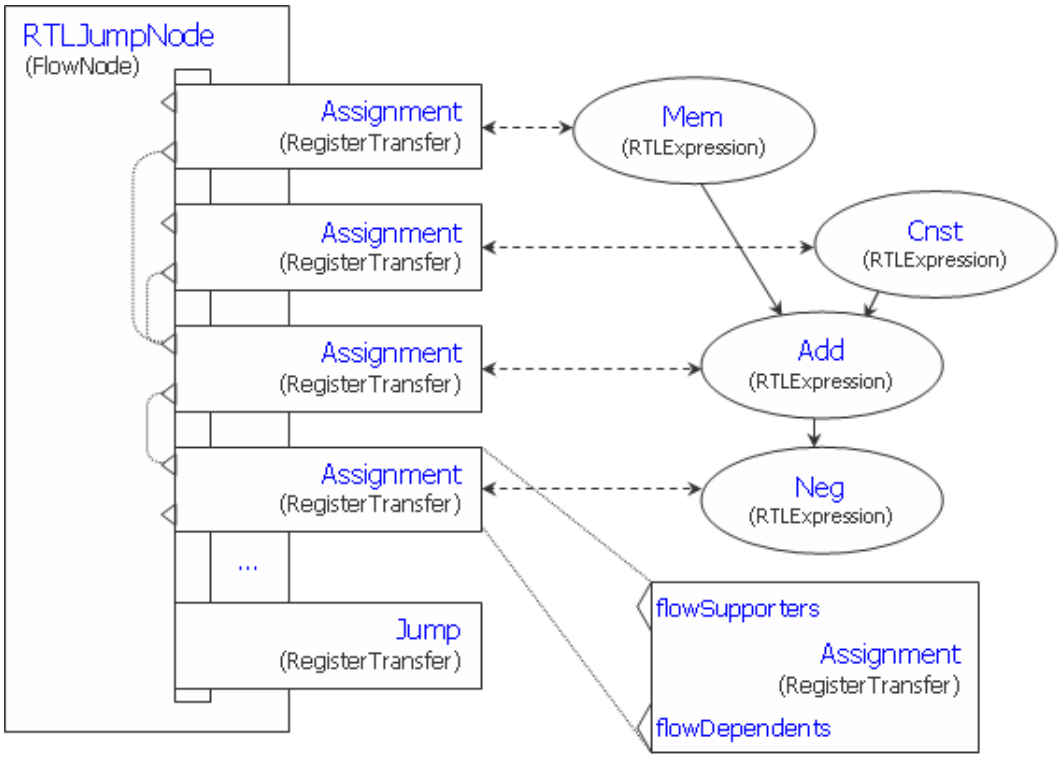


Figura 4.2: Relação entre *FlowNode*, *RegisterTransfer* e *RTLExpression*.

Os *RegisterTransfers* servem também para identificar as operações de fluxo de controlo. Para tal existem as seguintes classes: *Jump*, que representa uma operação de salto incondicional e é o último *RegisterTransfer* dos objectos do tipo *RTLJumpNode*; *CondJump*, que representa uma operação de salto condicional e é o último *RegisterTransfer* dos objectos do tipo *RTLConditionalNode*; *Return*, que representa uma operação de retorno (de uma função) e é o último *RegisterTransfer* dos objectos do tipo *RTLReturnNode*; e o *Call*, que representa uma invocação de uma função/procedimento e é o penúltimo *RegisterTransfer* dos objectos do tipo *RTLCallNode*.

Já vimos que o RTLS é genérico, é flexível (bastando para tal derivar novas classes) e possui alguma capacidade de abstracção. Mas também é extensível, aliás é extremamente extensível. O que se deve, não tanto ao modelo de RIC em si, mas mais à própria filosofia de implementação do RTL System. É que as rotinas de análise e optimização (e outras), são implementadas como métodos das classes do RTLS. Nesta perspectiva é sempre possível acrescentar um novo método ou derivar uma nova classe que contenha novos métodos, incluindo algumas formas de análise que podem computar informação, que não pertencendo directamente à RIC está com esta relacionada. É lógico que o que está errado aqui, não é tanto o modelo de RIC, mas mais a própria filosofia de desenvolvimento do RTL System, que não separa o que é a RIC, do que deveriam ser os componentes que operam sobre a RIC. Isto requer modificar as classes que são utilizadas para construir a RIC, sempre que se pretenda acrescentar ao sistema novas rotinas de compilação.

Este é o primeiro modelo apresentado que, para além de conter mais do que um nível de abstracção (*FlowNodes* e *RTLExpressions*), disponibiliza também informação que normalmente não está integrada na RIC, como é o caso do GDD. É como tal importante que garanta a consistência entre os diversos níveis de abstracção e restantes estruturas de dados

integradas na RIC. Pois se tal não acontecer, caberá então ao utilizador assegurar essa consistência, o que resulta numa sobrecarga de trabalhos que é obviamente indesejável. É claro que o RTLS possui tais mecanismos de consistência, até porque a sua implementação não é difícil quando se faz uso de uma linguagem orientada por objectos, como é o caso. Esses mecanismos permitem, por exemplo, a actualização automática dos *RegisterTransfer* aquando da remoção de uma expressão; ou a actualização dos *FlowNodes*, aquando da remoção ou inserção de ligações entre nodos.

Para concluir falta determinar se a RTLS é ou não fácil de utilizar, o que é um critério que tem sempre alguma subjectividade. É claro que construir uma lista de tuplos ou uma lista de árvores de expressões é uma tarefa mais homogénea, podendo como tal ser mais simples do que construir a RIC através do RTLS. Isto porque, por um lado, há mais do que um nível de abstracção (a RIC é construída neste caso a dois níveis diferentes); por outro lado existe todo um conjunto de informação que é preciso apurar e preencher, que normalmente não existe nos modelos mais convencionais. É difícil negar que a construção da RIC não é mais trabalhosa, mas este é o custo a pagar para que a implementação das rotinas/componentes, que não os *front-ends*, seja mais acessível. Isto porque a RIC é semanticamente mais poderosa, disponibiliza outros recursos (nomeadamente diferentes níveis de abstracção), é flexível e genérica. Acresce ainda que potencia a implementação de rotinas/componentes que são mais rápidos a executar, dado que determinado tipo de informação que em modelos mais convencionais teria que ser calculada, encontra-se neste caso já disponível na própria RIC (como é o caso do GFC e do GDD).

4.2.5 Stanford University Intermediate Format

Uma das referências mais importantes em termos de modelos de RIC, é o Stanford University Intermediate Format (SUIF), que se encontra também implementado através de uma linguagem orientada por objectos, neste caso C++, e é utilizado num dos mais conhecidos sistemas para construção de compiladores, o SUIF Compiler System (ver Secção 2.8).

O SUIF tem um conjunto de características muito próprias. Mas talvez a mais relevante advenha do facto de ser composta por conjunto de elementos organizados segundo uma estrutura em árvore (com algumas raras ocorrências de referências cruzadas), que faz uso de elementos com diferentes níveis de abstracção. Por exemplo, tem a capacidade de representar explicitamente estruturas condicionais e estruturas cíclicas, e em simultâneo (num nível de abstracção inferior) construtores do tipo salto condicional.

É interessante observar que a RIC acaba por se assemelhar muito a uma árvore de sintaxe (que não obedece a uma sintaxe fixa), mas com níveis extra de detalhe. Esta solução, que à primeira vista poderá parecer banal, é na realidade muito relevante, isto porque a "sintaxe" utilizada é independente da linguagem fonte, ou seja, é genérica.

Esta ideia é extremamente interessante e é em parte antagónica à do RTL original. Este último visava estar tão próximo quanto possível da linguagem final, para assim facilitar o "mapeamento" em instruções assembly/código máquina. O SUIF posiciona-se no outro extremo, isto é, perto da linguagem fonte. O que resulta numa RIC semanticamente mais rica, que é mais adequada à implementação de rotinas de análise e de optimização de alto nível (que é um dos objectivos principais do SUIF System). A comparação entre estas duas soluções serve também para constatar que há uma certa tendência para subir o nível de abstracção dos modelos de RIC.

É de realçar que, pelo facto da RIC se assemelhar a uma árvore de sintaxe, cada nodo intermédio pode ser visto como um nível diferente de abstracção. Este conceito de nível de abstracção é no entanto bastante redutor. Isto porque a abstracção é muito localizada e de forma alguma abrange todos os elementos do mesmo tipo. Por exemplo, apesar de ser

possível aceder a um objecto que represente um procedimento (*ProcedureDefinition*), não há um nível de abstracção que permita de forma conjunta ter uma perspectiva de todos os procedimentos do programa, nem mesmo acedendo ao elemento ascendente, que corresponde a *ScopedObject*.

No entanto, o SUIF consegue disponibilizar, no verdadeiro sentido do termo, alguns níveis de abstracção, fazendo para tal uso de iteradores[1]. Incluindo níveis que não fazem directamente parte da RIC. Por exemplo, existe um iterador que permite aceder às *labels* dos blocos de código, permitindo assim ter uma perspectiva da RIC ao nível do GFC. Há ainda um iterador que permite percorrer as árvores de expressões (devolve a raiz de cada árvore), e outro que permite aceder às referências das variáveis que ocorrem na RIC.

O SUIF é também bastante flexibilidade. Não só permite criar novos elementos derivando as classes já existentes, à semelhança do que acontece no RLTS, como disponibiliza uma solução que visa facilitar esta operação. A qual é composta por uma linguagem de especificação, designada por *hoof*, e por uma ferramenta de geração, designada por Smgn, através da qual é possível obter todo o código necessário à implementação e utilização dos elementos especificados através da *hoof*. É no entanto de alertar que, no sentido de maximizar a capacidade de reutilização dos componentes (aqui designados por passos), a definição de novos elementos deve ser evitada. O que aliás também é verdade para qualquer outro modelo de RIC.

Tanto quanto foi possível apurar, o SUIF também possui mecanismos para manter a consistência da RIC. Esses mecanismos servem de intermediários entre elementos de níveis adjacentes da RIC. De tal forma, que quando um elemento é alterado, se activa o mecanismo que vai actualizar os descendentes imediatos. A esses mecanismos designam-se por *dismantlers* e na realidade não actualizam, mas recomputam os elementos descendentes. Tanto quanto foi possível apurar, os *dismantlers* podem ser gerados através do Smgn e funcionam essencialmente no sentido descendente. É por este motivo que o SUIF System dá prioridade de execução aos componentes que operam sobre elementos de maior abstracção.

O SUIF assegura a extensibilidade permitindo a anexação de anotações à RIC. Essas anotações são representadas através de objectos do tipo *AnnotableObject* ou de classes derivadas desta. Por omissão, o SUIF permite que uma anotação consista num inteiro, numa *string* ou em outros objectos do SUIF. Mas o utilizador pode sempre optar por definir novas classes para as anotações.

Acontece no entanto que as anotações são entidades autónomas, isto é, uma vez depositada a informação na RIC, não há qualquer vínculo entre essa informação e o componente que foi responsável pela sua computação. Significa isto que não há mecanismos de consistência entre RIC e informação anexa, ou seja, se a RIC for entretanto modificada, a informação contida nas anotações poderá ficar inutilizada. Com a agravante de que isto pode nem sequer ser detectado.

O problema só não é grave, porque o SUIF System faz uso das anotações essencialmente entre componentes que são executados consecutivamente, isto é, o primeiro componente apura a informação e coloca-a na RIC, para ser utilizada exclusivamente pelo componente seguinte.

Não foi possível apurar com precisão, até que ponto é fácil construir a RIC com este modelo. Mas dada a existência de diversos níveis de abstracção, que são semanticamente redundantes e que tudo indica obrigatórios, é de crer que esta seja uma tarefa complicada.

Em termos de implementação de novos componentes, o facto do SUIF disponibilizar variadíssimos níveis de abstracção, consistentes entre si, e de ser um modelo genérico e flexível, contribui certamente para simplificar este tipo de tarefa.

Não sendo necessariamente uma desvantagem, o SUIF segue no entanto a máxima

[1] Da palavra inglesa *iterators*.

de restringir a RIC ao que é estritamente RIC. Por exemplo, em relação ao RTLS não mantém qualquer controlo das dependências entre dados. Neste sentido, a implementação dos componentes poderá não ser tão acessível e a sua execução será definitivamente menos eficiente.

4.3 *Dolphin* Internal Representation

O modelo de RIC desempenha um papel importante na estrutura de um compilador. Mas é ainda mais importante para sistemas como a *framework Dolphin*, que tem por objectivo simplificar o processo de construção de compiladores. O que passa por simplificar a reutilização e a implementação dos componentes. É neste último ponto, que o modelo de RIC pode dar um importante contributo e que faz com que seja crucial para o trabalho desenvolvido na preparação desta dissertação.

Sem dúvida que grande parte dos modelos anteriormente apresentados satisfazem as condições mínimas requeridas. Em que os melhores exemplos são o RTLS e o SUIF, em especial este último. No entanto, ambos apresentam alguns inconvenientes, por exemplo: o RTLS faz uso de uma linguagem, o Smalltalk, que coloca problemas de integração com soluções externas (ferramentas ou rotinas); não separa a RIC dos componentes; a RIC está fragmentada por dois níveis distintos de abstracção (*FlowNodes* e *RTLExpressions*); e a solução utilizada para estender a RIC não é satisfatória. Em contra-partida, é um modelo: genérico; flexível; possui mecanismos intrínsecos de consistência; não está estritamente limitado aos elementos típicos da RIC, possui mesmo outras estruturas de utilidade indiscutível, como é o caso do GDD; e dispõe de um nível de abstracção muito útil e completo, que corresponde ao GFC.

Já o SUIF é um modelo bastante mais consistente, que separa claramente a RIC dos componentes (passos) e é igualmente genérico e flexível. No entanto, a solução utilizada para a extensibilidade deixa um pouco a desejar; é de acreditar que é possível fazer melhor.

Os próprios mecanismos de consistência, têm a enorme vantagem de fazerem uso de uma solução que aparentemente é uniforme e que tem um papel claramente definido, o que não acontece em mais nenhum outro modelo (nem mesmo no modelo proposto nesta dissertação). Não foi no entanto possível apurar até que ponto estes mecanismos conseguem manter a consistência da RIC sem alguma intervenção pontual do utilizador ou de quem constrói o modelo de RIC. Principalmente, quando estão envolvidos elementos definidos pelo utilizador. Não será por acaso que o SUIF System disponibiliza uma série de componentes de limpeza, como é o caso do *recycle_trash*, do *gc_symbol_table*, e vários outros.

O SUIF ao conter elementos de grande abstracção, impossibilita que a RIC assente no nível mais baixo, uma vez que este nível carece da expressividade semântica necessária a suster elementos de maior abstracção. Significa isto que os elementos de maior abstracção não são apurados a partir dos elementos de menor abstracção, e como a relação inversa também não é possível obter, então a construção da RIC tem que necessariamente contemplar todos os níveis de abstracção supostamente definidos no SUIF (nenhum nível é inferido a partir dos demais). O que definitivamente faz com que a construção da RIC seja bastante mais complexa do que em qualquer outro modelo, requerendo os tais *dismantlers*, para manter a consistência entre os diversos níveis de abstracção.

Falta ainda saber se faz sentido utilizar elementos de abstracção tão elevados. Por falta de experiência na implementação de optimizações e análises de alto nível, para as quais eventualmente se justifica tais elementos, não posso realmente afirmar que sejam dispensáveis. Mas a verdade é que há outros níveis de abstracção que provavelmente são tão ou mais úteis e que dificilmente podem ser disponibilizados numa estrutura deste tipo, que se assemelha a uma árvore de sintaxe. Por exemplo, não há como representar explicitamente o grafo de

fluxo de controlo. Esta falha é colmatada através de um iterador que permite percorrer os nodos que estão implícitos na RIC, mas que não caracteriza de forma alguma o GFC, por exemplo, falta informação sobre os nodos antecessores e sucessores.

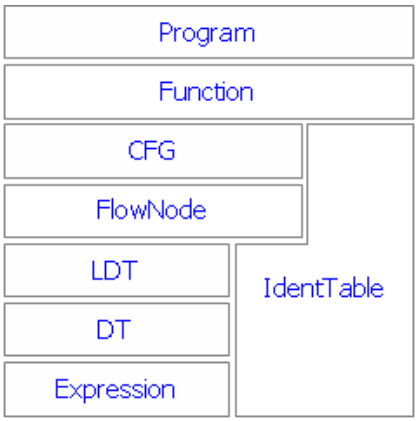
A própria organização dos elementos, que se assemelha a uma árvore de sintaxe, mas com vários níveis de abstracção, pode complicar substancialmente a implementação de determinados algoritmos. Com a agravante de que determinadas optimizações de baixo nível podem destruir os elementos de maior abstracção, forçando o utilizador a ter que intervir nesses níveis, podendo mesmo não ser possível repor a consistência da RIC sem misturar elementos de níveis de abstracção diferentes.

Atendendo à relevância que o modelo de RIC tem para este doutoramento, aos inconvenientes apresentados para os dois modelos mais válidos (RTLS e SUIF) e ao facto de se pretender fazer uso de algumas soluções próprias, chegou-se à concepção e implementação do modelo de RIC que é proposto nesta dissertação, a Dolphin Internal Representation (DIR).

A DIR resulta assim da união de algumas ideias próprias, com o que se considerou ser as boas soluções dos vários modelos analisados. Tem no entanto uma afinidade bastante grande com o RTLS, que serviu de base ao modelo proposto.

4.3.1 Estrutura e classes da DIR

A DIR é um modelo cuja parte executável da RIC, assenta em árvores de expressões. Mas disponibiliza também várias classes para representar elementos mais abstractos. As principais famílias de classes, a sua relação e a especificação formal, encontra-se representada na Figura 4.3.



(a) Organização das classes da DIR.

```
Program      ::   Function*.
Function     ::   IdentTable CFG
                  Function*.
IdentTable   ::   (string,CellTable)*.
CFG          ::   FlowNode*.
FlowNode     ::   LDT FlowNode*
                  [IdentTable].
LDT          ::   DT DT* DT.
DT           ::   Expression.
```

(b) Especificação gramatical da DIR.

Figura 4.3: Classes principais da DIR.

Todas as classes apresentadas são de utilização obrigatória. O que significa que a RIC é construída através dos diversos níveis de abstracção, como acontece no RTLS, mas ao contrário deste modelo, a parte executável da DIR está toda ela representada no nível inferior, composto por árvores de expressões.

A descrição que se faz em seguida das principais classes, descreve apenas as variáveis e aspectos mais relevantes de cada uma.

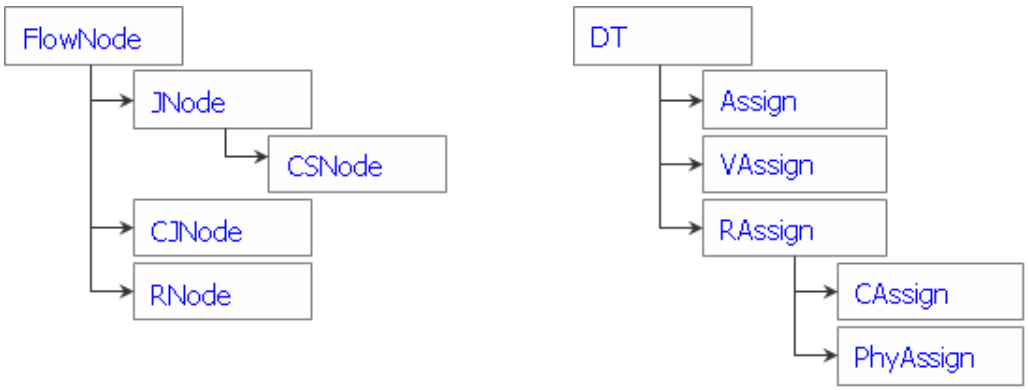
A classe *Program* representa o programa submetido ao compilador. É essencialmente composta por uma lista de apontadores para objectos do tipo *Function*. Dependendo da sintaxe da linguagem fonte, *Program* pode conter um ou mais objectos deste tipo.

A classe *Function* é utilizada para representar funções, procedimentos, instruções compostas ou simples declarações/definições (a que designaremos genericamente por *funções*). Para além de uma *string* com o identificador da função, cujo preenchimento é opcional, esta classe contém um apontador para um objecto do tipo *IdentTable*, outro para um objecto do tipo *CFG* e ainda uma lista de apontadores para objectos do tipo *Function*.

A classe *IdentTable* é uma tabela de hash implementada através de uma *template* que permite construir as tabelas de identificadores. As chaves são do tipo *string* e os valores associados às chaves são definidos através do parâmetro da *template*, o qual deverá obedecer a um protocolo mínimo para assim poder funcionar em conjunto com *IdentTable*. Esse protocolo está definido através da interface *CellTable*. Por omissão existem duas famílias de classes derivadas de *CellTable*, que servem para representar respectivamente tipos (*cellType*) e símbolos (*cellSymbol*). Essas classes foram implementadas com base na solução utilizada no SUIF.

A classe *CFG* representa explicitamente o Grafo de Fluxo de Controlo (GFC). Consiste essencialmente num conjunto de apontadores para objectos do tipo *FlowNode*.

FlowNode é a classe principal de um conjunto de classes que visam representar os nodos (vértices) do GFC. Associado a cada *FlowNode* está um apontador para um objecto do tipo *LDT* e um conjunto de apontadores para objectos do tipo *FlowNode*, que representam os antecessores do presente nodo. Derivam de *FlowNode* três classes: o *JNode*, que representa um nodo que termina com uma expressão de salto incondicional; o *CJNode*, que representa um nodo que termina com uma expressão de salto condicional; e o *RNode*, que representa um nodo que termina com uma expressão de retorno (de função). Existe ainda uma quarta classe, que deriva de *JNode* e que contém um apontador para um objecto do tipo *Function*. Esta classe, que é designada por *CSNode*, visa representar partes do GFC que possuam identificadores próprios (salvaguardados na *IdentTable* de *Function*) e funciona como um nodo que contém mais nodos (através do *CFG* de *Function*). A hierarquia da família de classes FlowNode, encontra-se representada na Figura 4.4(a).



(a) Classes da família *FlowNode*. (b) Classes da família *DT*.

Figura 4.4: Classes *FlowNode* e *DT*.

A classe *LDT* é essencialmente uma lista de *DTs*, cada um deles associado a uma expressão. Esta classe, para além da referida lista, contém mais duas variáveis do tipo *DT*: uma que está associada uma expressão do tipo *Label*, que identifica o *LDT* e implicitamente o *FlowNode*; e outra que está associada à última expressão do *LDT*, e implicitamente à última expressão do *FlowNode*. Expressão essa que serve para representar uma expressão de salto incondicional (*Jump*), uma expressão de salto condicional (*CJump*), ou então uma expressão

de retorno (*Return*).

A classe *DT* é semelhante ao *RegisterTransfers* do RTLS. É utilizada essencialmente com dois objectivos: manter uma estrutura com as dependências entre expressões; e manter uma representação sequencial das expressões (e não apenas das árvores de expressões). De notar que determinadas optimizações, nomeadamente de baixo nível, têm por objectivo reordenar as expressões de forma a rentabilizarem: os *pipelines*, a utilização dos registos e da memória *cache*, ou outros recursos da arquitectura de computação. A utilização de simples árvores de expressões não permite manter a ordem pela qual as expressões devem ser processadas, daí a necessidade dos *DTs*.

Conforme já foi dito, cada *DT* está associado a uma expressão, através de um apontador. As classes derivadas de *DT* são substancialmente diferentes das utilizadas pelo RTLS. Até porque determinados elementos da RIC, nomeadamente os que definem o fluxo de controlo do programa, são no RTLS considerados como *RegisterTransfers*, enquanto na DIR são tratados como expressões comuns. Existem assim três tipos distintos de *DTs*, a saber: *RAssign*, que está associado a expressões que produzem resultados intermédios (que correspondem nos outros modelos aos pseudo-registos); *Assign*, que está sempre associado a uma expressão do tipo *AsgnExpr* e que representa uma atribuição explícita para uma variável ou qualquer outra referência de memória; *VAssign*, que está associado a expressões das quais não resulta qualquer tipo de valor, como por exemplo, a expressões de salto (*Jump*/*CJump*), a *labels* (*Label*), etc. Para além destes *DTs* existem ainda mais dois derivados de *RAssign*, para casos relativamente específicos: o *CAssign*, que está sempre associado a uma expressão de invocação de procedimento/função (*Call*); e o PhyAssign, que está sempre associado a uma expressão do tipo *PhyFunction* ou *PhyTFunction* (utilizadas na representação Static Single Assignment). A hierarquia da família de classes *DT* encontra-se representada na Figura 4.4(b).

Expression é uma classe abstracta que serve para representar uma qualquer expressão, e da qual derivam as seguintes classes: *Terminal*, conjunto de classes que na sua maioria servem para representar elementos terminais das árvores de expressões; *JmpExpr*, classes que implementam as expressões de fluxo de controlo; *UnaryExpr*, classes que representam operações unárias; e *BinExpr*, que deriva de *UnaryExpr* e que serve para representar operadores binários. Para além destas classes, que são todas elas abstractas, existem várias classes concretas, que se encontram representadas na Figura 4.5.

Apesar de não ser muito vulgar, a DIR admite a utilização de operadores com mais do que dois operandos, para tal disponibiliza o *ArgExpr* e o *LExpr*. O primeiro permite decompor os operadores em expressões binárias; e o segundo faz uso de uma lista de operandos, todos eles do tipo apontador para *Expression*. Dois exemplos que ilustram a utilização destas classes encontram-se representados na Figura 4.6(a) e 4.6(b).

Convém acrescentar que, directa ou indirectamente, todas as classes da DIR derivam da classe abstracta *DObject*, que na realidade nada acresce de novo, mas permite representar uma qualquer classe da DIR.

É também importante realçar que quase todas as classes anteriormente definidas, contêm um apontador para o objecto que as inclui, permitindo assim navegar na RIC no sentido ascendente e descendente (subentende-se que a RIC corresponde a um grafo acíclico). Por exemplo, cada objecto do tipo CFG contém um apontador para o objecto *Function* no qual está incluído.

A RIC passa de componente para componente através do objecto *Program*. Em alternativa é possível utilizar a classe *DIR* (que tem a mesma designação do modelo), que para além do objecto do tipo *Program*, contém informação das opções de compilação.

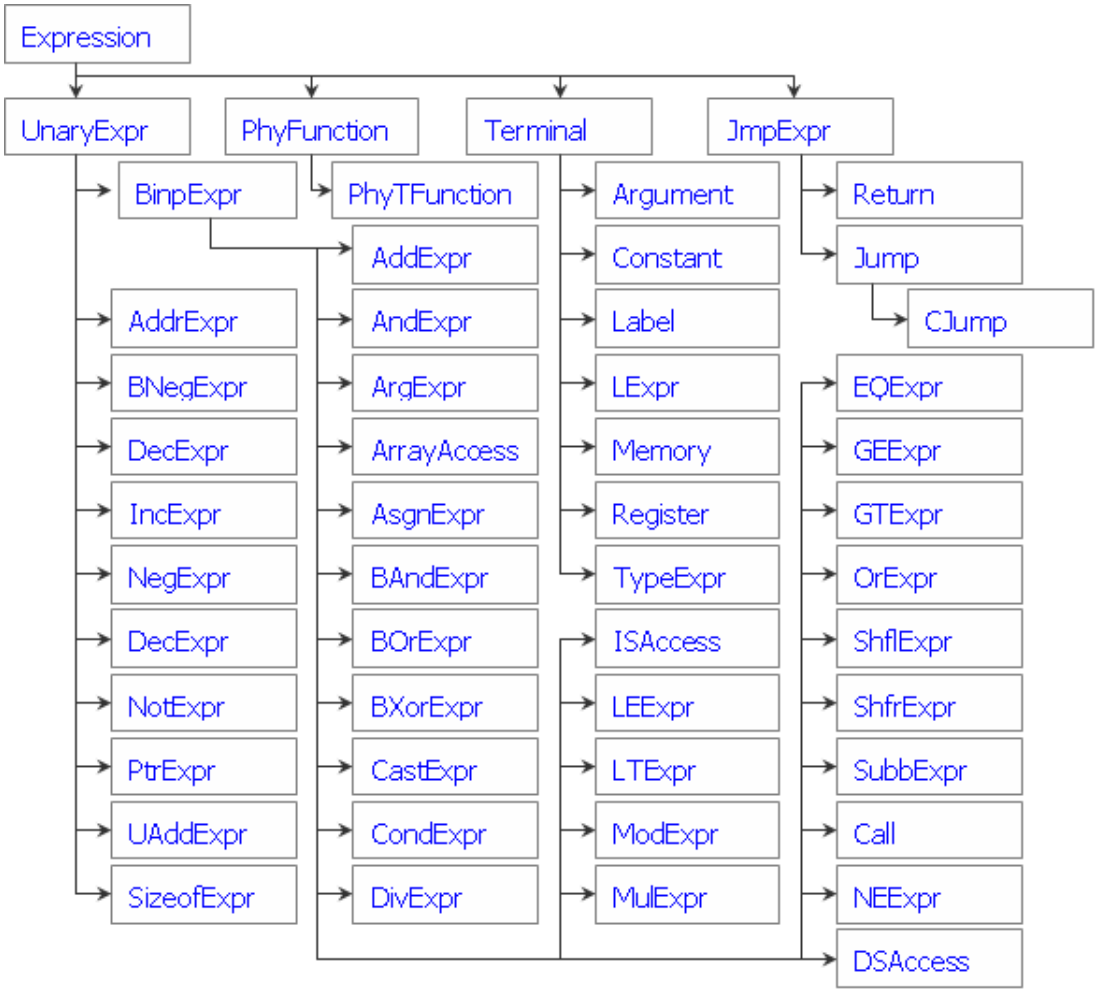


Figura 4.5: Hierarquia da família de classes *Expression*.

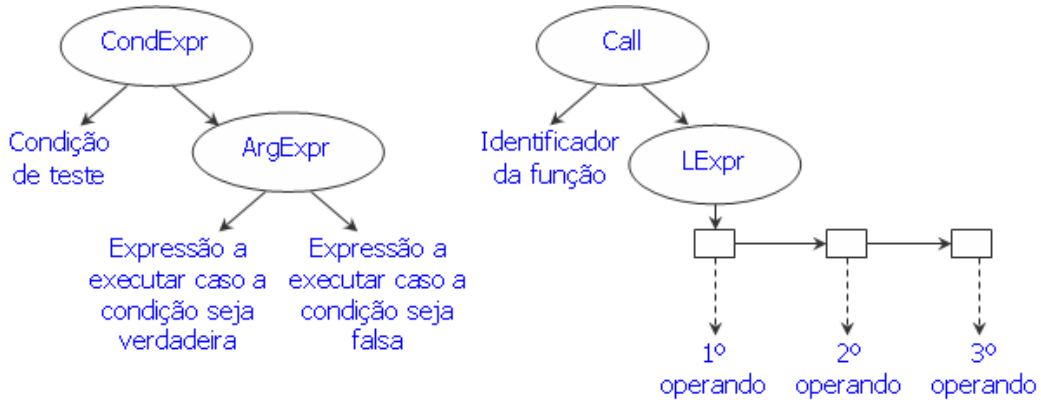
4.3.2 Aplicação da DIR

Pretende-se nesta secção ilustrar como utilizar a DIR na construção da RIC, esperando assim ajudar a compreender o papel das diversas classes e da própria estrutura deste modelo. O programa de Figura 4.7, que está implementado em linguagem C, é utilizado como exemplo para a construção da RIC.

A construção da RIC desenrola-se normalmente em quatro fases, a saber: instanciação dos objectos de *Program* e de *Function*, os dois níveis mais altos de abstracção; instanciação e preenchimento das tabelas de identificadores (*IdentTable*); construção do GFC, o que passa pela instanciação dos *FlowNodes*; e por fim, a instanciação dos *Expressions*, que correspondem ao nível mais baixo de abstracção.

Assim, a primeira classe a instanciar é *Program*, pois será esse objecto que vai conter toda a RIC. Posteriormente, e dependendo do tipo de linguagem fonte, ter-se-á que instanciar um ou mais objectos do tipo *Function*. Neste caso tratando-se de um programa escrito em linguagem C, considera-se que todas as funções estão ao mesmo nível (não há encadeamento de funções), assim pode-se logo à partida instanciar dois objectos: um para a função *swap(...)* e outro para a função *test(...)*, conforme ilustra a Figura 4.8.

A Figura 4.9 ilustra as operações a efectuar para se inserir a informação sobre a função



(a) Utilização de ArgExpr. (b) Utilização de LExpr.

Figura 4.6: Utilização de operadores com mais do que dois operandos.

```
(1)    void swap(int *a,int *b){...}
(2)    int test(int x,int y){
(3)       if(x>y){
(4)          swap(&x,&y);
(5)          return 1;
(6)       }else return 0;
(7)    }
```

Figura 4.7: Programa utilizado como exemplo para construir a RIC.

test(...) na tabela de identificadores. A primeira operação a ser efectuada consiste em definir o tipo da função, o que é feito através de *CTCProcedure*, que requer a definição do tipo de retorno e dos tipos dos parâmetros. O tipo da função, que é representado por *tf2*, é depois utilizado para definir o símbolo que vai representar a função *test(...)* (variável *sp2*). O último procedimento a executar é inserir o símbolo na tabela de identificadores.

Poderá ainda ser necessário preencher as tabelas de identificadores das funções. Este procedimento encontra-se ilustrado na Figura 4.10 para a função *test(...)*, em que se insere na tabela de identificadores informação sobre os parâmetros.

O passo seguinte consiste em construir o GFC. Para tal utiliza-se o objecto da classe *CFG*, que como é automaticamente criado aquando da instanciação de *Function*, apenas tem

```
(1)   Program *p=new Program();
(2)   Function *f1=new Function(p,"swap",FUNCTION,...);
(3)   Function *f2=new Function(p,"test",FUNCTION,...);
```

Figura 4.8: Instanciação de *Program* e de *Function*.

```
(1)  ...
(2)  IdentTable<CellTable*> *tid=p->getIdentTable(); // Acesso à tab.  de
(3)                                                  // identificadores global
(4)  CTCProcedure *tf2=new CTCProcedure();           // Definição do tipo
(5)                                                  // para a função test(...)
(6)  tf2->setType(new CTInt());                      // Atribuição do tipo
(7)                                                  // de retorno de tf2
(8)  tf2->add(new CTInt());                          // Atribuição do tipo
(9)                                                  // do 1º parâmetro
(10) tf2->add(new CTInt());                          // Atribuição do tipo
(11)                                                 // do 2º parâmetro
(12) CSProcedure *sp2=new CSProcedure(tf2);          // Instanciação do símbolo
(13) tid->ins("test",sp2);                           // Inserção de sp2 na
(14)                                                 // tab.  de identificadores
```

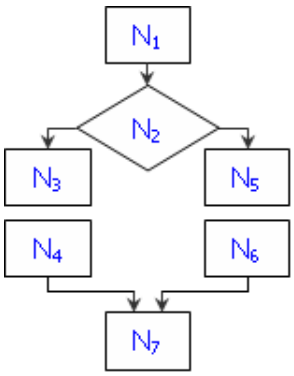
Figura 4.9: Preenchimento da tabela de identificadores global.

```
(1)  ...
(2)  tid=f2->getIdentTable();                        // Acesso à tab.  de
(3)                                                  // identificadores de f2
(4)  CTInt *tpx=new CTInt();                         // Definição do tipo
(5)                                                  // do 1º parâmetro
(6)  CSParameter *spx=new CSParameter(tpx)           // Instanciação do símbolo
(7)                                                  // para o 1º parâmetro;
(8)  tid->ins("x",spx);                              // Inserção do 1º parâmetro
(9)                                                  // na tab.  de identificadores
(10) CTInt *tpy=new CTInt();                         // Definição do tipo
(11)                                                 // do 2º parâmetro
(12) CSParameter *spy=new CSParameter(tpy);          // Instanciação do símbolo
(13)                                                 // para o 2º parâmetro;
(14) tid->ins("y",spy);                              // Inserção do 2º parâmetro
(15)                                                 // na tab.  de identificadores
```

Figura 4.10: Preenchimento da tabela de identificadores local.

que ser preenchido (com objectos do tipo *FlowNode*). A Figura 4.11(b) ilustra a construção parcial do GFC para a função *test(...)*, cujo fluxograma se encontra representado na Figura 4.11(a). Cada *FlowNode* assegura automaticamente a instanciação da respectiva *Label* e expressão de fluxo de controlo.

Para concluir, falta construir as árvore de expressões, o que passa por instanciar objectos das classes derivadas de *Expression*. A Figura 4.12 ilustra esta operação para a função *test(...)*. É de realçar que a instanciação de cada *Expression*, se faz passando o nodo ao qual a expressão pertence. O que serve para registar a expressão no *LDT* associado ao nodo. É



(a) Fluxograma da função *test(...)*.

```
(1)   ...
(2)   CFG *g2=f2->getCFG();
(3)   JNode *n1=new JNode(g2);
(4)   CJNode *n2=new CJNode(g2);
(5)   RNode *n3=new RNode(g2);
(6)   RNode *n5=new RNode(g2);
(7)   JNode *n4=new JNode(g2);
(8)   JNode *n6=new JNode(g2);
(9)   RNode *n7=new RNode(g2);
(10)  n1->setOut(n2);
(11)  n2->setOut(n3);
(12)  n2->setCOut(n5);
(13)  n4->setOut(n7);
(14)  n6->setOut(n7);
```

(b) Instanciação das classes.

Figura 4.11: Construção do GFC da função *test(...)*.

ainda importante realçar que os objectos do tipo *DT* são implicitamente criados aquando das *Expressions*, processo que é completamente transparente para o utilizador. Há apenas que ter em atenção que a ordem pela qual se criam as *Expressions*, vai influenciar a ordem dos *DTs* em *LDT*. Por exemplo, não é conveniente substituir as linhas 7 e 10 da Figura 4.12, pela seguinte expressão: *le→ins(0,new Memory(n3,"x"))*. Isto porque neste caso, o objecto *Memory* é criado após o objecto *LExpr*, pelo que na lista de *DTs* surge primeiro o *DT* de *LExpr* e só posteriormente é que surge o *DT* de *Memory*, o que poderá não fazer muito sentido.

A partir deste ponto é já possível utilizar a RIC, podendo esta ser acedida no seu todo através da variável *p* (apontador para *Program*), ou parcialmente, a partir de *p* ou directamente das variáveis utilizadas na instanciação dos objectos.

Poderá ainda ser útil encapsular *Program* na classe *DIR*, o que permite juntar num único objecto, a RIC e as opções de compilação. Para tal basta efectuar as operações da Figura 4.13. As opções de compilação são armazenadas num dicionário cuja chave e o valor associado à chave, são ambos do tipo *string*.

É também possível preencher automaticamente o dicionário com os parâmetros de compilação, bastando registar no objecto *DIR* os argumentos utilizados na linha de comando da invocação do compilador. É também possível passar este objecto, directamente ao *front-end*, que após construir a RIC, tratará de efectuar automaticamente a anexação de *Program*. Ambas as operações encontram-se representadas na Figura 4.14.

4.3.3 Características da DIR

Uma vez apresentada a DIR, nomeadamente as suas classes e a sua estrutura, é chegada altura de avaliar até que ponto este modelo de RIC satisfaz os critérios anteriormente definidos, e até que ponto é que o faz tão bem ou melhor do que os demais modelos.

Para começar convém assinalar que à semelhança do que acontece no SUIF, a DIR também é implementada em C++, separa claramente a RIC dos componentes, e utiliza a RIC como meio privilegiado de passar informação entre componentes. Mas ao contrário do

```
(1)    ...
(2)    // Expressions  para:  x > y
(3)    Memory *x=new Memory(n2,"x");
(4)    Memory *y=new Memory(n2,"y");
(5)    n2->setCondExpr(new GTExpr(n2,x,y));
(6)    // Expressions  para:  swap(&x,&y)
(7)    AddrExpr *ax=AddrExpr(n3,new Memory(n3,"x"));
(8)    AddrExpr *ay=AddrExpr(n3,new Memory(n3,"y"));
(9)    LExpr *le=new LExpr(n3);
(10)   le->ins(0,ax);
(11)   le->ins(1,ay);
(12)   Call *cl=new Call(n3,new Memory(n3,"swap"),le);
(13)   // Expressions  para:  return 1
(14)   n3->setRetExpr(new Constant(n3,1));
(15)   // Expressions  para:  return 0
(16)   n5->setRetExpr(new Constant(n5,0));
```

Figura 4.12: Construção das árvores de expressões.

```
(1)    ...
(2)    DIR dir(p);
(3)    dir.add("silent");
(4)    dir.add("hnw","3");
(5)    ...
```

Figura 4.13: Instanciação da classe DIR.

SUIF, não há qualquer pudor em fazer com que a informação, que não pertença estritamente à RIC, transite em moldes próprios, directamente de componente para componente.

Genérico

A DIR é um modelo tão genérico como qualquer outro, até porque o núcleo da RIC tem por base árvores de expressões, cujos operadores podem ser tão genéricos quanto o utilizador desejar. É de salientar que apesar das inúmeras classes representadas na Figura 4.5, nem todas são fundamentais para a construção da RIC. É aliás perfeitamente possível construir integralmente a RIC utilizando apenas as classes do tipo *JmpExpr*, algumas classes do tipo *Terminal* e as classes *UnaryExpr* e *BinExpr*. Isto porque a especialização destas classes, na sua grande maioria, apenas se limita a definir o tipo de operador/operando. Por exemplo, a diferença que existe entre *AddExpr* e *BinExpr* reside essencialmente no facto do primeiro se registar como sendo o operador OPADD. É mesmo possível criar um *AddExpr* directamente a partir de *BinExpr*, bastando posteriormente registar o tipo de operador, conforme ilustra a Figura 4.15:

```
(1)    int main(int argc,char *argv[]){
(2)       DIR d(argc,argv);           // Instanciação e registo dos
(3)                                   // argumentos no objecto DIR
(4)       FrontEnd fe(&d);            // Instanciação do front-end com
(5)                                   // registo do objecto DIR
(6)       fe.execute();               // Execução do front-end
(7)       ...
(8)       Comp_A ca(&d);              // Utilização directa do objecto
(9)                                   // DIR no registo de componentes
(10)      Comp_B cb(d.getProgram());  // Utilização de Program via
(11)                                  // DIR no registo de componentes
(12)      ...
(13)      return 0;
(14)   }
```

Figura 4.14: Utilização da classe DIR para gerir os argumentos do compilador.

```
(1)    ...
(2)    BinExpr *exp=new BinExpr(n);
(3)    exp->setOP(OPADD);
(4)    ...
```

Figura 4.15: Criação de operadores a partir das classes abstractas.

Em relação aos restantes níveis de abstracção, a DIR é suficientemente genérica para representar a grande maioria das linguagens imperativas (que não sejam orientadas por objectos). A prova de que tal é verdade, é que as classes que define representam apenas elementos comuns e que não impõem qualquer restrição sintáctica. Os elementos são suficientemente abstractos para que possam ser compostos conforme as exigências da linguagem fonte, sem que isso signifique que a RIC deixe de ser genérica.

É provável que haja outros paradigmas cuja representação não é possível ou pelo menos não é tão acessível ou directa, mas o mesmo acontece com os demais modelos. Aliás é até em parte incorrecto pensar que um único modelo pode representar convenientemente todos os paradigmas/linguagens. A solução que aparentemente é mais adequada, consiste em disponibilizar modelos específicos para esses paradigmas, que complementam os modelos convencionais. Isto é, para determinado tipo de linguagens é, numa primeira fase, utilizado o modelo específico dessas linguagens, que posteriormente é convertido para o modelo convencional, prosseguindo assim o processo de compilação. No fundo é criar *medium-levels* que, não sendo específicos de uma linguagem, são no entanto específicos de um conjunto de linguagens/paradigmas. Por exemplo, o SUIF System faz uso deste tipo de solução, disponibilizando um modelo de RIC específico para linguagens orientadas por objectos (o OSUIF), que funciona como complemento ao modelo convencional (o SUIF).

Flexível

A DIR é tão flexível como os melhores modelos, o que resulta do facto de estar implementada com uma linguagem orientada por objectos, que facilita a criação de novos operadores e operandos, quer por derivação de novas classes, quer pela composição das classes já existentes (como acontece nos exemplos da Figura 4.6). Esta flexibilidade é acrescida pelo facto da DIR fazer uso intensivo de tipos paramétricos (*templates*), conjuntamente com interfaces/classes abstractas. Como aliás acontece com a tabela de identificadores (*IdentTable*), implementada como uma tabela de hash que foi construída com base numa *template*. É o parâmetro requerido por esta *template*, que define o tipo dos valores que estão associados às chaves (as chaves são definidas como sendo do tipo *string*). Parâmetros esse que deverá representar uma classe derivada de *CellTable*, que é uma classe abstracta que estabelece os métodos a implementar (e algumas implementações por omissão) necessários ao correcto funcionamento da *template IdentTable*.

A DIR só perde em termos de flexibilidade em relação ao SUIF que, conforme já foi dito, disponibiliza uma solução específica para definir novos elementos (Hoof + Smgn).

Extensível

O modelo até aqui apresentado da DIR, por si só, não é extensível. Convém no entanto explicar que, neste modelo como em qualquer outro, é sempre possível manter estruturas de dados à parte da RIC. Inclusive, contendo informação relacionada com a RIC ou com alguns elementos da RIC. No entanto, se essa informação estiver relacionada com a RIC, vai estar eventualmente correcta no instante em que é calculada, mas nada garante que após a execução de outras tarefas se mantenha consistente com a RIC. Portanto, quando aqui se fala de extensibilidade, não é apenas no sentido de anexar informação à RIC, é sim no sentido de anexar e manter a consistência dessa informação.

Entendeu-se assim que não fazia muito sentido anexar informação sem que houvesse um protocolo que garantisse a sua consistência com a RIC. Neste sentido, a versão até aqui apresentada da DIR não é extensível. Esta característica é no entanto assegurada aplicando a arquitectura proposta no Capítulo 5 às classes da DIR, e de forma mais genérica à *framework Dolphin*.

A solução que é apresentada no Capítulo 5 permite vincular aos elementos da RIC, os componentes responsáveis pela computação da informação que se pretende anexar. O que possibilita fazer com que sejam os próprios componentes a manter a consistência da informação que apuram.

Dos modelos analisados, o RTLS é o único que reúne as condições necessárias para ser considerado como extensível (segundo a definição aqui utilizada). Isto porque, quer as rotinas que apuram informação sobre a RIC (Análises), quer as rotinas que alteram a RIC (Optimizações), são implementadas como métodos dos objectos que formam a RIC. Pelo que há sempre a possibilidade de sincronizar ambos tipos de rotinas, garantindo assim a extensibilidade do modelo de RIC.

Convém no entanto salientar que a bibliografia do RTLS não contém qualquer referência sobre a utilização deste tipo de solução. Há ainda a agravante de que a condição que permite considerar o RTLS extensível, isto é, o facto das rotinas estarem implementadas como métodos das classes que compõem o RTLS, é por si só uma das maiores desvantagens deste modelo. Isto porque a implementação de novas rotinas implica necessariamente modificar as classes que compõem o modelo de RIC.

Conforme também já foi explicado, o SUIF propõe através dos *AnnotableObjects* uma solução para anexar informação à RIC. No entanto não disponibiliza, nem garante qualquer

solução, que vise manter a consistência entre essa informação e a RIC.

Mecanismos de abstracção

Como já foi demonstrado, a DIR disponibiliza diversos níveis de abstracção, todos eles genéricos ao ponto de poderem ser utilizados na grande maioria das linguagens, nomeadamente do tipo imperativo.

Os níveis de abstracção disponibilizados pela DIR, permitem manter o modelo genérico e, simultaneamente, semanticamente rico. Subir o nível de abstracção, no sentido de aumentar a capacidade semântica da linguagem, como aliás acontece no SUIF, acarreta custos demasiado elevados. Isto porque:

- Ou o modelo se afasta dos níveis de menor abstracção, o que inviabiliza a implementação de rotinas de baixo nível, nomeadamente relacionadas com o escalonamento das instruções (*scheduling*);
- Ou o modelo mantém, simultaneamente, elementos de grande abstracção com elementos de baixa abstracção, o que aumenta:
  - Significativamente a dificuldade de construir a RIC;
  - A dificuldade de manter a RIC, nomeadamente no que diz respeito à consistência entre diferentes níveis de abstracção;
  - A incompatibilidade dos componentes, tornando-os menos reutilizáveis.

Não será por acaso, que os responsáveis pelo SUIF System se viram obrigados a disponibilizar um modelo de RIC para os níveis mais baixos (o MachSUIF).

Também não há necessidade de descer o nível de abstracção, dado que a DIR utiliza na sua base árvores de expressões e DTs. Em que estes últimos, para além de terem outras funcionalidades, desempenham um papel semelhante ao dos tuplos.

Convém esclarecer que, à semelhança do que acontece no RTLS, a construção da RIC com base na DIR requer todos os níveis de abstracção definidos. Difere no entanto do RTLS, no seguinte aspecto: a parte executável da RIC está integralmente representada no nível de menor abstracção, que é composto por árvores de expressões. Significa isto, que alguns dos níveis de abstracção disponibilizados funcionam como interfaces para os níveis inferiores.

Mecanismos de consistência

A DIR garante a consistência da RIC, mas apenas no sentido descendente, pelo que quando uma classe de maior abstracção sofre alterações, as classes de menor abstracção acompanham essas alterações. No sentido ascendente, há alguns casos em que a consistência não é garantida, por exemplo, introduzir uma *Label* nas árvores de expressões não força a criação de um novo nodo no GFC e muito menos estabelece ligações com os nodos já existentes. Esta opção, de não garantir a consistência da RIC no sentido ascendente para todas as situações, deve-se ao simples facto de que para tal seria necessário introduzir mecanismos relativamente complicados, que iriam tornar a utilização das classes da DIR mais pesada e menos flexível.

A consistência entre RIC e informação anexa, é contemplada através da arquitectura proposta no Capítulo 5. A consistência é garantida desde que os componentes e as classes da DIR implementem essa arquitectura.

Facilidade de utilização

Conforme já foi dito, a facilidade de utilização de um modelo de RIC deve ser ponderada em termos de adequação à construção da RIC e em termos de adequação à implementação de componentes. O facto de um modelo ser flexível, admitir diversos níveis de abstracção e garantir a consistência entre esses níveis, contribui claramente para fazer com que seja mais fácil de utilizar, o que é o caso da DIR.

No entanto, a satisfação de alguns desses critérios, pode complicar a construção da RIC. Por exemplo, o facto de uma linguagem possuir diversos níveis de abstracção pode contribuir para complicar a construção da RIC, nomeadamente se esses níveis são de alguma forma redundantes. Mas também nesta perspectiva, a DIR foi desenvolvida para conter apenas os níveis de abstracção essenciais, em que cada um tem um papel bem definido no processo de construção da RIC, de forma a minimizar redundâncias.

Claro que os puristas podem sempre dizer que a construção da RIC é mais acessível fazendo uso de modelos mais simples, como as árvores de expressões. Como o objectivo deste doutoramento é fazer com que a RIC não seja um obstáculo, antes pelo contrário, que contribua para facilitar o desenvolvimento dos componentes, sejam eles *front-ends* ou de outro tipo qualquer, entendeu-se disponibilizar um nível especial de abstracção, a que se designou por *vistas* (*views*).

Uma vista pode consistir numa ou mais classes que funcionam sobre o modelo até aqui apresentado, a que designaremos por *núcleo da DIR*, e cujo objectivo é permitir lidar integralmente ou parcialmente com a RIC, segundo uma estrutura/organização diferente daquela que é utilizada pelo núcleo de classes da DIR. Integrado neste conceito de *vista* estão também alguns iteradores.

Uma das vistas mais útil, que está actualmente em fase de testes, é o *ExprLst*, que permite obter dinamicamente uma lista com todas as árvore de expressões da RIC. Mas *ExprLst* não é um simples iterador, isto porque também permite construir grande parte da RIC, utilizando apenas *Expressions*. O utilizador terá apenas que complementar a RIC produzida através de *ExprLst*, inserindo a informação nas tabelas de identificadores. Evita assim utilizar as classes *Program*, *Function*, *CFG*, *LDT* e *DT*.

Esta vista, para além da classe base que está associada a um objecto do tipo *Program*, introduz algumas novas classes do tipo *Expression*, que são necessárias para se poder lidar, através das expressões, com elementos de maior abstracção.

O *ExprLst* tem a vantagem de manter a consistência em ambos sentidos, entre as árvores de expressões que disponibiliza e a RIC formada pelo núcleo de classes da DIR. É importante realçar que o *ExprLst* não duplica as árvores de expressões, mantém apenas algumas estruturas que permitem linearizar a RIC.

A Figura 4.16 ilustra a utilização desta vista, na construção da RIC da função *test(...)* da Figura 4.7. É de realçar que *ExprLst* tem dois construtores: um sem parâmetros que força a criação de um objecto do tipo *Program*; e outro que aceita como parâmetro um apontador para um objecto do tipo *Program*. A utilização do segundo construtor, força a actualização do estado de *ExprLst*. Por omissão, as expressões vão sendo acrescentadas sequencialmente a *ExprLst*. É no entanto possível remover expressões; navegar na lista de expressões, inclusivamente saltando de função em função ou de nodo em nodo; iterar ao longo da lista; etc.

É ainda de realçar a forma como *ExprLst* gere e constrói o GFC. A introdução de um nodo novo faz-se inserindo uma *Label*. Sempre que tal acontece, *ExprLst* cria um objecto do tipo *FlowNode*. O tipo de nodo que é efectivamente utilizado, apenas é definido quando surge uma expressão de fluxo de controlo, tipo *Jump*, *CJump* ou *Return*. De notar que isto não só é possível, como é fácil de efectuar, dado que o que difere entre *JNode*, *CJNode* e

```
(1)   ExprLst l;
(2)   l.add<FunctExpr>();
(3)   l.add<Label>();
(4)   Jump *j=l.add<Jump>();
(5)   Label *lb=l.add<Label>();
(6)   j->setOut(lb);
(7)   PtrExpr *px=l.add<PtrExpr,Memory*>(l.add<Memory,char*>("x"));
(8)   PtrExpr *py=l.add<PtrExpr,Memory*>(l.add<Memory,char*>("y"));
(9)   CJump *cj=l.add<CJump>(l.add<GTExpr,PtrExpr*,PtrExpr*>(px,py));
(10)  lb=l.add<Label>();
(11)  cj->setOut(lb);
(12)  Memory *x=l.add<Memory>("x");
(13)  Memory *y=l.add<Memory>("y");
(14)  LExpr *le=l.add<LExpr>();
(15)  le->ins(0,x);
(16)  le->ins(1,y);
(17)  Call *cl=l.add<Call>(l.add<Memory>("swap"),le);
(18)  l.add<Return>(l.add<Constant>(1));
(19)  lb=l.add<Label>();
(20)  cj->setCOut(lb);
(21)  l.add<Return>(l.add<Constant>(0));
(22)  l.add<Label>();
(23)  Jump *j1=l.add<Jump>();
(24)  l.add<Label>();
(25)  Jump *j2=l.add<Jump>();
(26)  lb=l.add<Label>();
(27)  j1->setOut(lb);
(28)  j2->setOut(lb);
(29)  l.add<Return>();
```

Figura 4.16: Exemplo da utilização de *ExprLst*.

RNode é apenas a expressão que está associada ao último *DT*.

Uma vista mais simples, que consiste essencialmente num iterador, é o *FunctionLst*. Em que dado um objecto do tipo *Program*, permite aceder às funções da RIC segundo diversas modalidades (em profundidade ou em largura). É também importante realçar que *FunctionList* é utilizado pelo *ExprLst*.

A implementação de vistas é um mecanismo inédito em termos de modelos de RIC, cuja implementação obedece aos seguintes princípios:

- Não deve replicar os elementos da RIC, excepto que a utilização da vista se traduza na destruição da RIC;
- Pode conter referências aos elementos da RIC;
- Pode inclusive operar os elementos da RIC, funcionando como uma interface para inserir ou remover elementos;

- Pode conter informação apurada a partir da RIC, no entanto a actualização dessa informação é exclusivamente da responsabilidade da própria vista. O que pode ser feito de duas formas:
  - A informação é apurada dinamicamente a partir dos elementos da RIC (como é o caso de *ExprLst*);
  - A informação é mantida consistente com a RIC, vinculando a vista aos elementos da RIC dos quais a informação depende.

Este último mecanismo, que permite manter a consistência entre vista e o resto da RIC, é implementado com recurso à arquitectura proposta nesta dissertação. Na qual se assume que uma vista é uma forma especial de componente.

A utilização das vistas resulta assim num conjunto adicional de vantagens, dado que permitem:

- Reformular a perspectiva que os utilizadores têm do núcleo da RIC. Permitindo, por exemplo, aproximar a DIR em termos de utilização a outros modelos;
- Filtrar a RIC, disponibilizando perspectivas parciais, contribuindo assim para simplificar a utilização da RIC na implementação de novos componentes;
- Acrescentar novos níveis de abstracção, mais adequados às necessidades que os utilizadores possam vir a ter.

É de realçar que as vistas não são só uma forma de tornar a utilização da RIC mais simples. Permitem também aproximar este modelo aos demais, abrindo assim caminho para a reutilização de componentes provenientes de outros sistemas. Um pouco à semelhança do que acontece no Zephyr.

4.4 Resumo do capítulo

De forma a resumir este capítulo, apresenta-se uma comparação entre todos os modelos de RIC apresentados, incluindo a DIR. Convém no entanto explicar que é bastante complicado obter uma avaliação objectiva, até porque dificilmente haverá utilizadores que tenham trabalhado com todos estes modelos e a ponderação de cada critério varia muito de utilizador para utilizador. Resta assim apresentar uma avaliação própria, que certamente peca por alguma falta de imparcialidade e até mesmo por falta de algum conhecimento mais aprofundado sobre determinados modelos, nomeadamente na perspectiva de utilizador experimentado.

Os critérios utilizados na avaliação são os mesmos apresentados na Secção 4.1, isto é: se o modelo é genérico; se é flexível; se é extensível; se disponibiliza vários níveis de abstracção; se garante a consistência entre os diversos níveis de abstracção, e entre a RIC e a informação anexa; se facilita a construção da RIC e de componentes. Os resultados encontram-se representados na Tabela 4.1.

O seguintes comentários ajudam a interpretar a Tabela 4.1:

- Critério: Genérico
  - Todos os modelos foram considerados genéricos para representar linguagens do tipo imperativo (com excepção de linguagens orientadas por objectos);
- Critério: Flexível

Modelos	**Tuplos**	**Árvores**	**GCC**	**Zephyr**	**RTLS**	**SUIF**	**DIR**
Genérico	+++	+++	+++	+++	+++	+++	+++
Flexível	++	++	+	+	+++	+++	+++
Extensível	x	x	x	x	++	++	+++
Mecanismos de abstracção	x	x	x	x	++	+++	++
Mecanismos de consistência	x	x	x	x	++	++	++
Facilidade de construção da RIC	++	++	++	++	+++	+	+++
Facilidade de implementação de componentes	+	+	+	+	++	+++	++

Tabela 4.1: Resultados comparativos dos vários modelos de RIC analisados.

  - Apenas o modelo do GCC e do Zephyr foram considerados pouco flexíveis, o que é perfeitamente natural atendendo que este critério não é relevante para os sistemas que integram;
  - Considerou-se que os tuplos e as árvores de expressões são suficientemente flexíveis;
  - E que o RTLS, SUIF e DIR se destacam por permitir derivar e compor novas classes;

- Critério: Extensível
  - Entendeu-se que este critério não se aplica aos tuplos, árvores de expressões, GCC e Zephyr;
  - Considerou-se o RTLS e o SUIF são ambos extensíveis, mas que a solução utilizada no primeiro não é aceitável e a utilizada no segundo, de pouco serve dado que não garante a manutenção das estruturas de dados (a consistência em relação à RIC);
  - A DIR foi avaliada considerando que implementa a arquitectura proposta no Capítulo 5 e como tal, é a única solução que realmente garante a extensibilidade;

- Critério: Mecanismos de abstracção
  - Considerou-se que apenas o RTLS, o SUIF e a DIR dispõem de vários níveis de abstracção, destacando-se o SUIF em relação aos outros dois modelos;

- Critério: Mecanismos de consistência
  - Entendeu-se que este critério não se aplica aos tuplos, árvores de expressões, GCC e Zephyr;

  - Considerou-se que o RTLS, o SUIF e a DIR garantem a consistência entre níveis distintos de abstracção;

- Critério: Facilidade de construção da RIC
  - Pela sua simplicidade, considerou-se que os tuplos, as árvores de expressões, o GCC e o Zephyr são relativamente fáceis de utilizar;
  - Considerou-se também que o SUIF, pelos seus níveis de abstracção redundantes, penaliza o processo de construção da RIC;
  - E que o RTLS e a DIR, por decomporem a RIC nos níveis essenciais de abstracção, facilitam a construção da RIC;

- Critério: Facilidade de implementação de componentes
  - Considerou-se que por disponibilizarem um único nível de abstracção, com a agravante de ser de baixo nível, os tuplos, árvores de expressões, GCC e Zephyr são menos fáceis de utilizar na implementação de componentes;
  - E que o RTLS e a DIR, por disporem de alguns níveis de abstracção, são fáceis de utilizar na construção de novos componentes;
  - Mas entendeu-se distinguir o SUIF neste critério, por dispor de mais níveis de abstracção.

Em termos de conclusão, pode-se dizer que a DIR não é um modelo completamente inovador, até porque foi construído tendo por base as soluções utilizadas nos demais modelos. No entanto, o seu desenho foi efectuado atendendo a determinados detalhes que apenas os modelos mais evoluídos contemplam. Pode-se mesmo considerar que a DIR é a solução que melhor combina as vantagens dos modelos de baixo nível de abstracção, como os tuplos e as árvores de expressões, com algumas funcionalidades que apenas existem nos modelos de mais alto nível, como no caso do RTLS e do SUIF.

Os níveis de abstracção que define resultam também num dos melhores compromissos entre a facilidade de construção da RIC e a utilização desta na construção de componentes. Ao que se junta a capacidade de manter a consistência entre os diversos níveis de abstracção que apenas tem par no SUIF (apesar de subsistirem algumas dúvidas acerca da eficácia dos mecanismos de consistência deste modelo).

É também um modelo que inova em alguns aspectos, por exemplo, em conjunto com a arquitectura proposta no Capítulo 5, é o único modelo que tem a capacidade de permitir estender a RIC, garantindo as condições necessárias para assegurar a consistência entre a informação anexa e a RIC. É também o único modelo que propõe a utilização de múltiplas camadas de elementos: uma formada pelo núcleo de classes da DIR; e as restantes formadas pelas diversas *vistas*. O que tem reflexos extremamente positivos na facilidade de utilização, quer na vertente de construção da RIC, quer na vertente de implementação de componentes.

Todos estes atributos contribuem para tornar mais simples a construção dos compiladores, indo assim de encontro aos objectivos deste doutoramento.

CAPÍTULO 5

Framework Dolphin: Problemas e soluções

Índice

O trabalho até aqui apresentado, descreve uma solução para desenvolvimento de compiladores que tem por base uma *framework*, formada por vários componentes implementados como classes de C++, que funcionam sobre um único modelo de representação de código. Foi também explicado que conceptualmente há seis tipos diferentes de componentes: os *front-ends*, que processam o código submetido ao compilador para construir a RIC; os componentes de análise, que servem para determinar informação sobre a RIC; os componentes de suporte ao *back-end*, que á semelhança dos componentes de análise também apuram informação sobre a RIC, mas visaM suportar as tarefas de *back-end*; os componentes de optimização, que produzem transformações sobre a RIC de forma a optimizar o código final; os *back-ends*, que convertem a RIC para outros formatos, como por exemplo: assembly, código binário, XML, etc; e finalmente os componentes de inspecção, que servem para apurar determinados

parâmetros que podem ser utilizados, por exemplo, para aferir da eficiência do processo de compilação.

Foi também explicado que a construção de compiladores é feita através da instanciação dos componentes que fazem parte da *framework Dolphin*. Significa isto que a complexidade de implementação de um compilador é proporcional ao número de componentes utilizados, ou seja, de ordem linear. O que é bastante aceitável e até mesmo um bom resultado. Se em termos conceptuais isto é verdade, na prática quanto maior o número de componentes utilizados, bastante mais complicado fica a construção do compilador e muito menos eficiente fica o processo de compilação. A relação será efectivamente de ordem linear, se forem contabilizados todos os componentes utilizados, nomeadamente os componentes que mantêm e actualizam a informação anexa à RIC e os componentes de suporte (que são utilizados pelos componentes principais incluídos na especificação do compilador). O que normalmente não acontece, dado que para isso é necessária uma especificação bastante detalhada do compilador. Especificação que requer um conhecimento bastante aprofundado da *framework*, em especial sobre a forma como os componentes estão implementados. É claro que isto não é desejável, nem sequer aceitável, principalmente tendo em consideração o objectivo deste doutoramento.

A próxima secção descreve detalhadamente os problemas que advêm da utilização da *framework* no seu estado original (ver Capítulo 3) e que são comuns a todos os sistemas analisados. Problemas esses que influenciam principalmente a qualidade do processo de compilação, a qual tende a deteriorar com o aumento do número de componentes utilizados na construção do compilador. A Secção 5.2 descreve as soluções desenvolvidas para solucionar esses problemas. A última secção faz o resumo deste capítulo.

5.1 Afinal quais são os problemas?

Conforme foi explicado no Capítulo 3, os compiladores construídos com base na *framework Dolphin* contêm pelo menos um *front-end* e um ou mais componentes dos restantes tipos, podendo mesmo conter vários *back-ends* e até mesmo múltiplas instâncias do mesmo componente. No estado actual, a *framework Dolphin* possui essencialmente os componentes que foram necessários para testar as ideias e soluções que surgiram durante a preparação dos trabalhos para esta dissertação. O que de forma alguma significa que apenas existam esses componentes. Só no livro "*Advanced Compiler Design and Implementation*" de Steven Muchnick [Muc97], são descritas mais de uma centena de diferentes formas da análise, optimizações e rotinas de suporte ao processo de compilação. Fora deste livro ficam ainda muitas outras soluções, nomeadamente para arquitecturas de computação mais específicas, que envolvem microprocessadores super-escalares, VLIW ou arquitecturas paralelas.

Como é natural, muitas das soluções utilizadas na construção de compiladores têm procedimentos comuns, tipicamente utilizados para assegurar determinados pré-requisitos, mas que também podem servir para normalizar resultados, ou mesmo para outros fins. Em termos de implementação, significa que os componentes vão ter procedimentos que são iguais ou que pelo menos visam executar as mesmas funções. É assim de todo conveniente implementar tais procedimentos em rotinas à parte, para que assim possam ser reutilizados. No caso do *Dolphin*, eles são implementados como componentes da própria *framework*, designados por *componentes de suporte*. Em oposição ao termo *componentes principais*, que identifica os componentes que fazem uso dos componentes de suporte. Os dois conceitos podem sobrepor-se, isto é, um componente principal pode também funcionar como componente de suporte.

O facto de se decompor um componente, em componentes menores, tem assim por

objectivo minimizar a quantidade de código a implementar e como tal, as potenciais fontes de erro e os custos de manutenção. Em determinadas circunstâncias contribui também para simplificar a implementação dos componentes, dado que é possível reutilizar procedimentos que já estão implementados. Um componente complexo, poderá assim ser implementado através da reutilização de componentes mais simples. Isto por si só é uma excelente opção, que facilita o desenvolvimento dos componentes e como tal dos compiladores, contribuindo directamente para alcançar os objectivos deste doutoramento. No entanto, e como mais adiante se verá, nem tudo são vantagens. Por exemplo, a decomposição das soluções implica a existência de mais componentes, o que dificulta a gestão e utilização da *framework*.

No entanto, esta solução não só evidencia boas práticas de programação, como potencia melhores resultados, mas o que parece ser uma questão simples e praticamente inconsequente coloca alguns problemas quando ao objectivo inicial, de se simplificar o processo de desenvolvimento de compiladores, se tenta potenciar a construção de compiladores mais eficientes. As próximas secções explicam o tipo de vantagens que se pretendeu obter, os problemas que daí surgiram e como é que a solução alcançada resolveu estes e outros problemas.

5.1.1 Reutilização de componentes: inclusão implícita vs. explícita

Implementar integralmente uma solução sob a forma de um único componente, faz com que este contenha tudo o que é necessário à sua execução, é por assim dizer autónomo dos restantes componentes e a sua execução apenas obedece a uma determinada sequência imposta pela estrutura do próprio compilador. Por exemplo, um componente que implemente integralmente um *front-end* pode ser executado de forma independente dos demais. A única restrição à qual deve obedecer, prende-se com a própria estrutura do compilador, que força a execução deste componente antes de todos os outros. Diz-se assim existir uma *dependência estrutural*, entre o *front-end* e os demais componentes.

A decomposição de uma solução em vários componentes (um principal e um ou mais de suporte), cria *dependências funcionais* entre esses componentes, isto é, a execução de um componente principal passa a requerer a execução de todos os componentes que o suportam. O que poderá ser feito antes, durante ou após a execução das tarefas inerentes ao componente principal.

Este tipo de dependência, que advêm do facto da solução se encontrar decomposta em vários componentes, por si só não cria grandes problemas. Basta que o componente principal inclua os componentes de suporte e tudo funcionará como antes. O componente de suporte é assim utilizado implicitamente, dado que é inserido no compilador por outro componente e não por parte de quem especifica a estrutura do compilador.

A principal vantagem da inclusão implícita advém do facto de ocultar do utilizador os componentes de suporte, fazendo com que a especificação de um compilador seja mais simples e como tal mais acessível de descrever. O Exemplo 5.1 ilustra como é que a inclusão implícita dos componentes de suporte simplifica a especificação dos compiladores.

Exemplo 5.1

A Static Single Assignment (SSA) é uma forma de RIC que facilita a implementação de muitas rotinas, nomeadamente de análise e optimização de código. O *Dolphin* fornece um componente, o *cnv2SSA*, que permite converter a RIC da forma normal para a forma SSA. Componente esse que reutiliza outros componentes, como se encontra ilustrado na Figura 5.1. Se considerarmos que esses componentes de suporte são incluídos implicitamente pelo *cnv2SSA*, o utilizador não necessita sequer de saber da sua existência e muito menos tem que lidar com eles, basta instanciar e executar o *cnv2SSA* como está exemplificado na Figura 5.2.

§

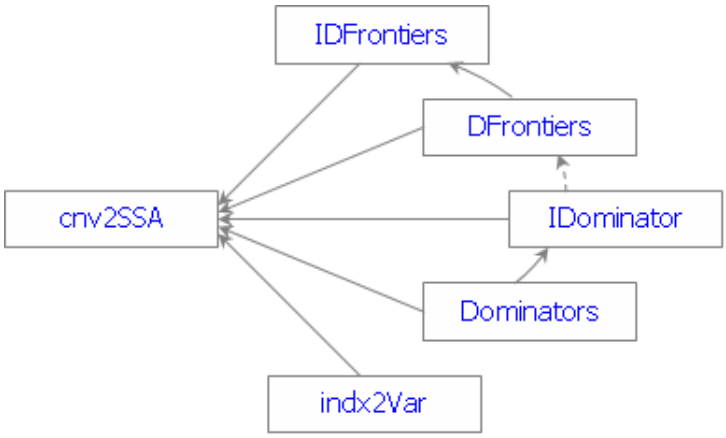


Figura 5.1: *cnv2SSA* e respectivos componentes de suporte.

```
(1)     DIR d(argc,argv);   // Instanciação do objecto que contém a RIC
(2)     littleC fe(&d);     // Instanciação do front-end
(3)     fe.execute();       // Execução do front-end
(4)                         // para construção da RIC
(5)     ...
(6)     cnv2SSA cnv(&d);    // Instanciação de cnv2SSA
(7)     ...
(8)     cnv.execute();      // Execução de cnv2SSA para converter a RIC
(9)                         // da forma normal para a forma SSA
(10)    ...
```

Figura 5.2: Especificação contendo as operações necessárias à conversão da forma normal para a forma SSA.

Apesar de todas as vantagens da inclusão implícita de componentes, a verdade é que a reutilização do processo em si continua a não existir, isto é, reutiliza-se o código mas não o processo inerente a esse código. Isto leva a que muitas operações sejam executadas inúmeras vezes sem que daí advenha qualquer vantagem, antes pelo contrário, contribui de forma muito significativa para deteriorar o tempo de compilação, mas não só.

O exemplo da Figura 5.1 serve também para ilustrar esta situação. É possível constatar que determinados componentes servem de suporte a mais do que um componente. Por exemplo, *DFrontiers* suporta directamente *cnv2SSA*, mas também *IDFrontiers*. Incluir implicitamente *DFrontiers*, faz com que existam pelo menos duas instâncias deste componente no compilador: uma que suporta *cnvSSA* e outra que suporta *IDFrontiers*. O mesmo acontece com os componentes *IDominator* e *Dominators*. A Figura 5.3 representa todas as instâncias que são necessárias para a utilização de *cnv2SSA*, caso os componentes sejam incluídos de forma implícita.

As consequências são claramente gravosas para o processo de compilação. Isto porque cada instância é independente, e como tal tem que ser executada individualmente, con-

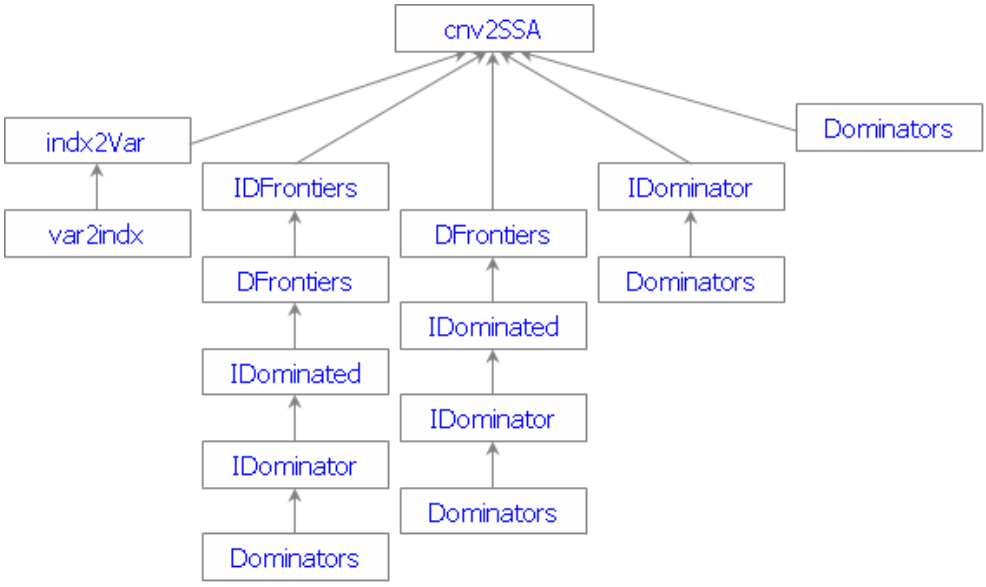


Figura 5.3: Instâncias requeridas por *cnv2SSA*, se os componentes forem incluídos implicitamente.

sumindo espaço de memória e tempo de processamento. O que se traduz em tempos de compilação mais longos (compiladores mais lentos do que eventualmente seria necessário).

O processo de compilação deteriora-se ainda mais quando os componentes de suporte, visam disponibilizar informação sobre a RIC. Isto porque, a informação é normalmente disponibilizada através dos próprios componentes, ou seja, é o próprio componente que mantém a informação que previamente determinou. Em termos do processo de compilação, a inclusão implícita dos componentes vai fazer com que uma quantidade substancial de informação seja desnecessariamente replicada. O Exemplo 5.1 serve para ilustrar esta situação. Como foi já dito, *cnv2SSA* converte a RIC da forma normal para a forma SSA, mas não há qualquer tipo de informação que fique retida no componente, ou seja, na prática o componente recebe a RIC (na forma normal), processa-a e coloca-a novamente na saída (agora na forma SSA). O mesmo não acontece com os componentes de suporte. Cada um destes componentes, com base na RIC, apura um determinado tipo de informação. É essa informação que na realidade é utilizada pelo componente principal (*cnv2SSA*). Por exemplo, *Dominators* constrói internamente um dicionário, que contém para cada nodo do Grafo de Fluxo de Controlo (GFC), os nodos que o dominam [LT79, CFR+91].

À dependência criada pelo facto de a execução de um componente requerer informação previamente determinada por outro componente (componente de suporte), designou-se por *dependência de dados*, a qual é um caso especial da dependência funcional entre componentes, mas que tem outros requisitos.

Uma possibilidade para tornar a reutilização dos componentes mais eficiente seria utilizá-los de forma explícita, isto é, aquando da construção do compilador, todos os componentes seriam explicitamente instanciados e executados pelo programador. Caberia a este gerir as instâncias dos vários componentes de forma a evitar a sua replicação, tornando assim o processo de compilação mais eficiente.

Para quem especifica compiladores, a inclusão explícita é o procedimento natural de utilização dos componentes. No entanto apenas tem sido aplicada a componentes:

- Cuja utilização é imposta pela estrutura do compilador (dependências estruturais);

- Cuja execução é recomendada mas não fundamental[1];
- Cuja execução é fundamental, resultando numa dependência funcional, mas em que cabe a terceiros requerer a execução desses componentes (normalmente a quem especifica o compilador)[2].

Associação de componentes

Os mecanismos existentes na *framework* original não permitem lidar com a maioria dos casos em que há dependências funcionais, mas em especial quando há dependências de dados. Isto porque faz falta um mecanismo que vincule o componente de suporte ao componente principal. No entanto não é difícil obter uma solução, por exemplo, basta registar o componente de suporte no componente principal, à semelhança do que é feito para o registo da RIC nos componentes. A Figura 5.4 ilustra a aplicação desta solução, apresentando as operações necessárias à conversão da forma normal para a forma SSA, com recurso à inclusão explícita dos componentes.

É fácil detectar pelos exemplos das Figura 5.2 e Figura 5.4, que para a mesma operação (conversão da forma normal para a forma SSA), a especificação de um compilador com base na inclusão explícita de componentes é substancialmente mais longa e complexa do que a especificação com base na inclusão implícita. Mais grave ainda, é que a inclusão explícita de componentes requer que o utilizador conheça minimamente a forma como os componentes estão implementados, nomeadamente: quais são os componentes de suporte, qual a ordem pela qual devem ser aplicados, como devem ser utilizados, quais os efeitos da sua execução, etc. Com a agravante de que esta desvantagem é recursiva, isto é, aplica-se aos próprios componentes de suporte (agora no papel de componentes principais). O pior é que há outros factores que agravam ainda mais esta situação, como se explica adiante.

Conhecer a estrutura da RIC

Como já foi referido, a Dolphin Internal Representation (DIR) é um modelo de representação de código composto por objectos com diferentes níveis de abstracção. Nos níveis de maior abstracção temos objectos do tipo *Program*, *Function* ou mesmo de *CFG*. Enquanto nos níveis de menor abstracção temos objectos do tipo *DT* ou de *Expression*. A existência de diversos níveis de abstracção, faculta a possibilidade de se escolher o nível (conjunto

[1] Por exemplo, a utilização de componentes cuja execução contribui, directa ou indirectamente, para criar melhores condições e oportunidades para os componentes que são executados posteriormente.

[2] Esta situação acontece, por exemplo, com o componente que faz a conversão da forma normal para a forma SSA (*cnv2SSA*). A forma SSA facilita a implementação de muitas rotinas de análise e optimização de código, permitindo em muitos casos obter soluções mais eficientes. Não significa no entanto que tais rotinas não possam ser implementadas sobre a forma normal. Existem mesmo alguns componentes que seleccionam a solução a utilizar, mediante a forma em que se encontra a RIC. Mas outros componentes há, que apenas estão aptos a executar sob uma das formas. No entanto como o processo de conversão para SSA, e posterior reconversão para a forma normal, é bastante pesado, a conversão só deverá ser feita nas seguintes condições: quando se pretende incluir no compilador um número considerável de componentes a funcionarem sobre a forma SSA (e que sejam executados sequencialmente); ou utilizar componentes que sejam fundamentais e que funcionem exclusivamente sobre a forma SSA. Estes motivos são no entanto de ordem conceptual, isto é, dependem da concepção do compilador, nomeadamente do tipo de componentes que se pretende utilizar. Por isso, e apesar da conversão para a forma SSA ser fundamental para alguns componentes, normalmente só é executada se for explicitamente requerida por quem especifica o compilador. Daí que muitos componentes da *framework Dolphin*, que funcionam exclusivamente sobre a forma SSA, estejam implementados de maneira a não forçarem a conversão, funcionando apenas se esta tiver sido previamente executada. Caso contrário, o pedido de execução do componente falha.

```
(1)    DIR d(argc,argv);    // Instanciação do objecto que contém a RIC
(1)    littleC fe(&d);      // Instanciação do front-end
(2)    fe.execute();        // Execução do front-end
(3)                         // para construção da RIC
(4)    ...
(5)    var2Indx vi(&d);     // Instanciação de var2Indx
(6)    indx2Var iv(&d);     // Instanciação de indx2Var
(7)    Dominators dom(&d);  // Instanciação de Dominators
(8)    IDominator idom(&d); // Instanciação de IDominator
(9)    IDominated ided(&d); // Instanciação de IDominated
(10)   DFrontiers df(&d);   // Instanciação de DFrontiers
(11)   IDFrontiers idf(&d); // Instanciação de IDFrontiers
(12)   cnv2SSA cnv(&d);     // Finalmente, instanciação de cnv2SSA
(13)   ...
(14)   vi.execute();        // Execução da instância de var2Indx
(15)   iv.execute();        // Execução da instância de indx2Var
(16)   dom.execute();       // Execução da instância de Dominators
(17)   idom.execute();      // Execução da instância de IDominator
(18)   ided.execute();      // Execução da instância de IDominated
(19)   df.execute();        // Execução da instância de DFrontiers
(20)   idf.execute();       // Execução da instância de IDFrontiers
(21)   ...
(22)   cnv.execute();       // Execução da instância de cnv2SSA
(23)   ...
```

Figura 5.4: Especificação parcial de um compilador com inclusão explícita de componentes.

de elementos da RIC) que melhor se adequa à implementação e execução de cada um dos componentes.

Nos exemplos até aqui apresentados, como é o caso da especificação da Figura 5.4, a utilização dos componentes faz-se exclusivamente sobre um objecto do tipo DIR (que caracteriza toda a RIC). Tal tem sido feito para simplificar os exemplos e a explicação dos mesmos. Na realidade, a utilização de elementos do tipo DIR, ou mesmo do tipo *Program*, visa: a execução de tarefas de alto nível (análises ou optimizações de código inter-procedimentais); ou aplicar o componente de forma generalizada sobre todos os elementos de um nível de abstracção menor. Mas a maior parte dos componentes requer a utilização de elementos mais específicos, isto é, de menor abstracção.

Com a inclusão explícita, o utilizador é obrigado a "navegar" através dos diversos níveis de abstracção para alcançar os elementos necessários à execução dos componentes. O mesmo acontece com inclusão implícita, mas a responsabilidade de "navegar" na RIC cabe neste caso a quem desenvolve o componente. Em ambos os casos é necessário conhecer o tipo de elementos que compõem a DIR e a própria estrutura da RIC. No entanto, subentende-se que tais conhecimentos serão mais acessíveis a quem desenvolve componentes, do que a quem vai apenas utilizá-los.

Dimensão do problema

O problema torna-se particularmente complicado, quando se tem em conta que um programa submetido ao compilador normalmente contém: um único objecto do tipo DIR; algumas dezenas de objectos de alto nível, do tipo *Function* e *CFG*; e várias centenas, senão milhares, de objectos de baixo nível do tipo *DT* e *Expression*. Significa isto, que um componente que utilize elementos de baixo nível de abstracção poderá ter várias centenas senão milhares de instâncias. Gerir essas instâncias pode ser bastante complicado, nomeadamente quando os componentes são incluídos explicitamente. É que se com a inclusão implícita, o problema fica restrito ao contexto de implementação de cada componente (apenas há que lidar com as respectivas instâncias dos componentes de suporte); já com a inclusão explícita, esta situação pode ocorrer para vários componentes, fazendo com que o número de instâncias a gerir seja consideravelmente superior.

No caso da inclusão explícita, há ainda que ter em conta que cabe ao utilizador relacionar os componentes de suporte com os componentes principais, isto é, fazer o registo dos componentes de suporte nos componentes principais. Com tantas instâncias envolvidas que derivam de diferentes componentes pode ocorrer que esta operação, que é aparentemente simples, se torne bastante complexa.

Relacionamento das instâncias

De notar que as instâncias dos componentes, mas também dos elementos da RIC, são identificados por endereços, na melhor das hipóteses por variáveis. Supondo, por exemplo, que a RIC contém vários elementos do tipo A e B ($\{A_0, \ldots, A_n, B_0, \ldots, B_m\}$), que o componente C_A é aplicado a cada um dos elementos do tipo A ($\{C_{A0}, \ldots, C_{An}\}$), que o componente C_B é aplicado a cada um dos elementos do tipo B ($\{C_{B0}, \ldots, C_{Bm}\}$), e que os componentes do tipo C_A suportam a execução dos componentes do tipo C_B. A questão que se levanta é saber como relacionar as instâncias de C_A com as instâncias de C_B? Isto é, como fazer o registo das instâncias de C_A nas instâncias de C_B?

Para responder a esta questão, há que saber como é que os elementos da RIC estão relacionados; e como aceder aos elementos do tipo A a partir dos elementos do tipo B (ou vice-versa). O que requer novamente conhecimentos sobre a DIR e sobre a estrutura da RIC, que como já se disse, não deveria ser um requisito necessário a quem só pretende utilizar os componentes.

Além disso não basta conhecer a DIR e a estrutura da RIC, para se conseguir associar as instâncias de C_A às instâncias de C_B. Por exemplo, para se executar C_{Bi}, é necessário garantir a execução prévia da correspondente instância ou instâncias de C_A. Aceder a essa instância, que vamos supor que é representada por C_{Aj}, requer aceder ao elemento B_i, e a partir deste aceder ao elemento A_j. Ambas operações são possíveis e relativamente fáceis de executar, desde que se conheça a DIR e a estrutura da RIC. O que não é possível, ou pelo menos fácil, é aceder a C_{Aj} a partir de A_j. Esta situação encontra-se ilustrada na Figura 5.5.

Neste caso em concreto, a solução passa pelo utilizador assumir a responsabilidade de implementar mecanismos que associem os elementos de A com as instâncias de C_A. Generalizando a solução, significa implementar mecanismos que permitam determinar para cada elemento da RIC, quais os componentes que lhe estão associados. É de notar que a relação inversa é estabelecida quando se faz o registo do elemento da RIC no componente. Convém no entanto realçar que:

- A cada instância de um componente está associado um único elemento da RIC; mas a cada elemento da RIC podem estar associadas instâncias de vários componentes;

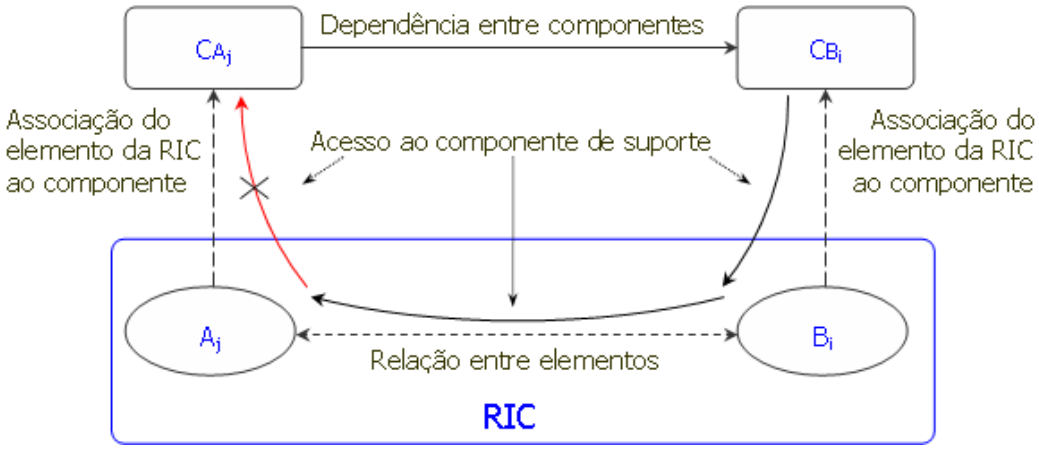


Figura 5.5: Relação entre componentes e elementos da RIC.

- Componentes que são dependentes, funcionam tipicamente sobre elementos do mesmo nível de abstracção, o que contribui para aumentar o número de instâncias e como tal, o número de dependências.

Em suma, foram apresentadas duas soluções: uma suportada pela inclusão implícita dos componentes, que apesar de facilitar a especificação dos compiladores, faz com que estes sejam muito pouco eficientes; e outra suportada pela inclusão explícita dos componentes, que permite produzir compiladores mais eficientes, mas exigindo a quem especifica o compilador, muito mais trabalho, conhecimento e experiência.

5.1.2 Consistência de dados

Quando existem dependências de dados entre os componentes, significa que a execução do componente principal requer os dados determinados pelo componente de suporte. Independentemente da solução utilizada para associar os componentes (incluídos explicitamente), a questão que se coloca é: tem o componente principal a garantia de que a informação contida no componente de suporte está correcta? De notar que não se está a colocar em causa se o componente de suporte está ou não bem implementado. Parte-se do princípio que sim. Apenas se coloca em causa, se a informação disponibilizada pelo componente de suporte está consistente com a sua fonte, isto é, com a RIC. E há razões para se colocar esta questão. Imagine-se que na linha 21 da especificação representada na Figura 5.4, é executado um ou mais componentes que visam optimizar o GFC, como por exemplo *elimJumpChains*. Este componente serve para eliminar cadeias de saltos incondicionais através da associação de nodos. Implica que a sua execução vai alterar o GFC, fazendo com que eventualmente determinados nodos deixem de existir. Assim e sem aparentemente haver qualquer ligação, acabou de se fazer com que toda a informação apurada pelas instâncias de *Dominators*, *IDominator*, *DFrontiers* e *IDFrontiers* (que trabalhem sobre os mesmos elementos da RIC utilizados por *elimJumpChains*), tenha deixado de estar consistente com a RIC. Nas circunstâncias actuais, a forma que o utilizador tem para evitar, ou pelo menos minimizar, a ocorrência deste tipo de situações, é:

- Executar os componentes de suporte imediatamente antes do componente principal, reduzindo a probabilidade da informação ficar inconsistente;

- Ou ter conhecimento do modo de funcionamento e dos efeitos directos e indirectos, de todos os componentes que pretenda vir a utilizar, evitando assim este tipo de situação.

Repare-se no entanto que não basta saber o modo de funcionamento e os efeitos dos componentes de suporte, há também que saber como funcionam os componentes principais, nomeadamente para identificar a que tipo de elementos acedem e como é que o fazem. E depois há a prática ...

Enquanto autor desta dissertação e responsável pela implementação da DIR, da *framework Dolphin*, e do *Dolphin* System, tenho alguma legitimidade para afirmar que sou quem melhor conhece a implementação, o funcionamento, os efeitos directos e indirectos, e tudo mais, de cada um dos componentes da *framework Dolphin*. No entanto, a verdade é que foram inúmeras as vezes que surgiram problemas que tiveram por base componentes cuja informação não estava consistente com a RIC e que só deixaram de surgir com a adaptação da arquitectura proposta nesta dissertação à versão original da *framework*. Torna-se muito difícil ter presente todo o conhecimento necessário sobre a *framework*, quando se está a lidar com um sistema:

- Cuja dimensão já é considerável, quer em número de componentes, quer em quantidade de código (aproximadamente 10 Mbytes);
- Que contém componentes desenvolvidos há já vários anos;
- Que visa construir aplicações que por si só são bastante complexas (compiladores);
- Que pela própria filosofia de desenvolvimento, que potencia a decomposição de soluções em vários componentes, leva a existência de muitas dependências;
- Que, por último mas de forma alguma menos importante, visa evoluir para uma solução *open-source*, para assim crescer através da comparticipação e colaboração de elementos externos ao projecto.

Há ainda que considerar que a detecção deste tipo de erro, cuja origem é semântica, é extremamente difícil, pois passa por depurar o funcionamento do compilador, isto é, analisar como se desenrola o processo de compilação para um conjunto de programas fonte (programas submetidos ao compilador). Bastaria atender ao facto de que a compilação envolve um número considerável de tarefas, algumas das quais bastante complexas e muitas delas construídas com base em soluções recursivas, para se perceber que a depuração seria logo à partida um processo complicado.

Mesmo quando os testes são realizados com programas fonte muito pequenos, a quantidade de dados produzida é bastante grande[3]. Com a agravante de que em termos de depuração, há uma nova "instância" da RIC entre o fim de uma tarefa e o início da seguinte. Isto contribui para aumentar de forma exponencial a quantidade de dados a analisar, dado que a origem do erro é, numa primeira fase, detectada comparando a RIC entre tarefas consecutivas.

Acresce ainda que a informação é em grande parte representada por objectos cuja identificação é feita através dos respectivos endereços de memória. Os quais são humanamente

[3]De notar que existe uma relação, que é inversamente proporcional, entre o capacidade semântica de uma linguagem e o tamanho dos programas que são descritos com essa linguagem. Em termos de capacidade semântica, a RIC pode ser comparada a uma linguagem de baixo nível, isto é, semanticamente pobre, pelo que o tamanho da RIC é relativamente grande, nomeadamente quando comparado com o tamanho do programa submetido ao compilador (programa fonte), que normalmente é descrito com uma linguagem semanticamente mais rica.

intratáveis, nomeadamente quando se tem em conta a quantidade de objectos envolvidos e o facto de que o espaço de execução atribuído pelo sistema operativo para a depuração do compilador se alterar, de forma completamente aleatória, de execução para execução. Isto faz com que a identificação dos objectos, que é fundamental para o processo de depuração, se torne numa tarefa dantesca.

Há ainda que ter a sorte de utilizar um programa de teste que "active" o erro. O que poderá acontecer muito tempo depois do compilador ter sido construído.

Se até ao início deste capítulo, se podia pensar que as soluções e ideias apresentadas, consubstanciadas na versão original da *framework Dolphin*, contribuíam por si só de forma significativa para atingir os objectivos deste doutoramento, resultando numa aplicação de fácil uso que faz com que a construção de compiladores seja mais simples, é de crer que após a exposição de todos estes problemas, tal ideia se tenha desvanecido. No entanto, e como se demonstra na próxima secção, todos estes problemas foram resolvidos, na sua grande maioria com soluções relativamente simples. É com a integração dessas soluções, que se concebeu a arquitectura proposta nesta dissertação, cuja aplicação permitiu fazer da *framework Dolphin* um sistema mais sólido, agradável de utilizar e que claramente contribui para alcançar os objectivos desta dissertação.

5.2 Desenho da arquitectura

Esta secção visa apresentar individualmente cada uma das soluções que foram desenvolvidas para resolver os problemas identificados no formato original da *framework Dolphin* (ver secção anterior) e que são comuns a todos os sistemas deste tipo que foram analisados. Mostrando também a forma como essas soluções foram sendo integradas, no sentido de construir uma única solução final, que é a grosso modo a arquitectura proposta nesta dissertação. O objectivo é fazer com que o leitor compreenda claramente o papel e a razão de ser de cada entidade que compõe a arquitectura, mas também fazer com que o leitor perceba que partes da arquitectura visam resolver cada um dos problemas anteriormente expostos. O Capítulo 7 aborda a arquitectura numa perspectiva de mais alto nível, não tão focada nas soluções apresentadas nesta secção, mas mais nas entidades que daí resultam e na sua integração no modelo de compilação que está inerente à *framework Dolphin*. Em síntese, esta secção visa apresentar as soluções desenvolvidas para cada um dos seguintes problemas:

- Reutilização de componentes - É necessária uma solução que permita reutilizar de forma simples e eficiente os componentes que tenham sido previamente instanciados, minimizando assim o número de instâncias utilizadas e, como tal, o tempo total de execução;
- Associação de componentes - Para este problema, que surge da necessidade de reutilizar os componentes, é necessária uma solução prática e eficiente para associar instâncias que sejam funcionalmente dependentes;
- Consistência dos dados - É necessária uma solução que garanta a consistência entre os dados de cada instância e a RIC, mas que minimize o número de vezes que as instâncias são executadas.

É sempre bom recordar que, para além de resolver estes problemas, a solução final deverá contribuir para:

- Simplificar a especificação dos compiladores;

- Potenciar a construção de compiladores eficientes.

Convém ainda realçar que os problemas expostos acima não são independentes. Aliás, quer a reutilização, quer a associação de componentes, têm por base uma única solução. O que até é natural, dado que a associação de componentes é um problema que advém da própria reutilização dos componentes. Mesmo a solução desenvolvida para garantir a consistência dos dados, parte da solução utilizada para a resolução dos dois outros problemas.

5.2.1 Reutilização de componentes

Como foi anteriormente explicado (ver Secção 5.1.1), há duas abordagens para utilizar os componentes: a implícita e a explícita. A inclusão implícita não permite a reutilização dos componentes, mas simplifica substancialmente a construção dos compiladores. Já a inclusão explícita, potencia a reutilização dos componentes, permitindo construir compiladores mais eficientes, mas por outro lado, faz com que todo o processo de construção seja bastante mais complexo e difícil de realizar. O ideal seria que, independentemente da forma como os componentes são incluídos, o processo de construção dos compiladores fosse acessível e simples, como potencia a inclusão implícita; e que permitisse construir compiladores eficientes, como potencia a inclusão explícita dos componentes. Será isto possível? Ou é pedir demasiado?

Com os recursos existentes na versão original *framework*, os componentes apenas podem ser reutilizados se forem incluídos explicitamente. No entanto, se é compreensível que o utilizador tenha que gerir as instâncias dos componentes que vai utilizar, já não é tão compreensível ter que gerir as instâncias dos componentes de suporte, sobre as quais eventualmente nada sabe e que muito provavelmente prefere evitar ter que saber.

Há a possibilidade de passar a responsabilidade de gerir as instâncias de suporte para quem desenvolve os componentes. Acontece no entanto, que quem o faz não tem como saber de antemão se as instâncias de suporte vão ou não estar disponíveis aquando da execução do componente que está a implementar, ou seja, à falta de um mecanismo que permita verificar dinamicamente (aquando da execução do componente), se as instâncias de suporte estão ou não disponíveis, resta apenas incluí-las sempre de forma implícita.

É claro que se pretende ocultar do utilizador as instâncias de suporte e daí obter todas as vantagens da utilização implícita dos componentes, designadamente a simplicidade. Mas para tal, há que fornecer uma solução a quem desenvolve os componentes, que permita lidar de forma eficiente com a reutilização. Neste sentido foram analisadas duas alternativas:

- A primeira consiste em desenvolver os componentes sem incluir as instâncias de suporte. Em vez disso, associa-se a cada componente meta-informação que assinale a necessidade das instâncias de suporte. A especificação construída com esses componentes, antes de ser compilada, é submetida a um pré-processador, que vai incluir as instâncias em falta e efectuar as associações necessárias. Cabendo assim ao pré-processador minimizar o número de instâncias a utilizar e assim optimizar o processo de compilação;

- A segunda alternativa consiste em criar uma solução dinâmica, que permita controlar e gerir as instâncias utilizadas aquando da execução dos componentes, isto é, aquando da compilação, minimizando assim o número de instâncias envolvidas.

Após a análise da viabilidade de ambas alternativas, a segunda foi a que em termos de abordagem prometia ser mais aliciante, dado que: a implementação é mais acessível; é mais intuitiva; e não só é compatível, como permite desenvolver boas soluções, para os restantes problemas detectados.

No essencial, apenas é necessário um mecanismo que controle e mantenha o registo das instâncias utilizadas no processo de compilação. Assim, sempre que é necessária uma determinada instância, há que consultar o mecanismo de controlo para averiguar se ela já existe. Se assim acontecer, então faz-se uso da instância que já se encontra registada no mecanismo de controlo, caso contrário cria-se uma nova instância, que obviamente também terá que ser registada no mecanismo de controlo.

Para que esta ideia seja funcional e por motivos mais ou menos óbvios, mas que serão explicados com maior detalhe mais adiante, o mecanismo de controlo deve indexar as instâncias pelo tipo do componente e pelo elemento da RIC sobre o qual as instâncias funcionam.

Optou-se no entanto por descentralizar a solução, de tal forma que cada elemento da RIC possui o seu próprio mecanismo de controlo. Esta opção, como mais adiante é demonstrado, permitiu ocultar os detalhes de implementação da solução, "escondendo" o mecanismo de controlo do utilizador. É no entanto de realçar que esta opção é puramente "estética" e não é de forma alguma fundamental. Até porque, se na solução centralizada era necessário um único dicionário com chave dupla, ao se associar o mecanismo de controlo aos elementos da RIC, são necessários muitos mais dicionários (um por cada elemento da RIC), se bem que basta utilizar dicionários simples, isto é, com uma única chave (o tipo do componente).

Continua no entanto a ser necessário efectuar o registo das instâncias, mas ao contrário do que acontecia até aqui, em que apenas se efectuava o registo do elemento da RIC no componente (instância), agora, é também necessário efectuar o registo inverso, isto é, registar o componente (instância) no elemento da RIC, ou mais especificamente no mecanismo de controlo associado ao elemento da RIC.

Para que esta solução fosse implementada de forma independente dos elementos concretos da RIC e dos componentes, foi necessário definir uma nova interface (classe virtual em C++), a *compManager*, que deve ser utilizada na implementação dos elementos da RIC. Houve também a necessidade de reformular a interface *Component*. A Figura 5.6 mostra a representação parcial em UML das duas interfaces. Alguns dos métodos dessas interfaces são virtuais, pelo que devem ser reescritos/implementados pelas classes derivadas.

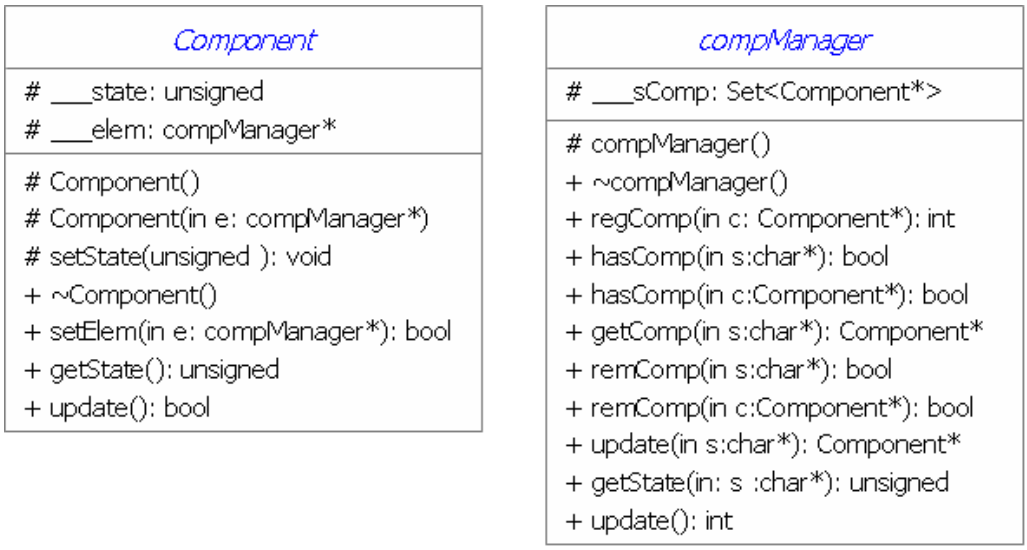


Figura 5.6: Interfaces *compManager* e *Component*.

A interface *Component*, cujo papel na *framework* original é essencialmente servir de classe abstracta para representar um qualquer componente, assume agora um papel funda-

mental. A variável *___elem* que era do tipo *DObject* passa agora a ser do tipo *compManager*, resultando em alterações num dos construtores e no método *bool setElem(...)*, que recebiam como parâmetro um apontador do tipo *DObject*, e que agora passam a receber um apontador do tipo *compManager*. O *bool setElem(compManager*)*, que já efectuava o registo do elemento da RIC no componente, passa também a invocar o método da interface *compManager* que faz o registo inverso (*bool regComp(Component*)*), conforme ilustra o exemplo da Figura 5.7.

```
(1)   class Component:virtual public Protocol{
(2)     protected:
(3)       Component(compManager *e):___elem(NULL){
(4)         setElem(e);
(5)         ...
(6)       }
(7)       ...
(8)     public:
(9)       virtual bool setElem(compManager *e){
(10)        bool st;
(11)        if(___elem)
(12)          ___elem->remComp(this);
(13)        ___elem=e;                    // Registo do elemento
(14)                                      // da RIC no componente
(15)        ___state = OUTDATED;
(16)        if(___elem){
(17)            ___elem->regComp(this);   // Registo do componente
(18)                                      // no elemento da RIC
(19)            st=true;
(20)        }else st=false;
(21)        return st;
(22)      }
(23)      ...
(24)  };
```

Figura 5.7: Procedimentos para efectuar o registo dos componentes.

Component introduz ainda uma segunda variável, o *___state*, e três novos métodos: *unsigned getState()*, *void setState(unsigned)* e *bool update()*. Os quais fazem parte da solução desenvolvida para garantir a consistência de dados e são explicados com o devido detalhe na Secção 5.2.3. Por agora basta dizer que a variável *___state* assinala o estado da instância, isto é, indica se a instância está ou não consistente com a RIC.

compManager é a interface que implementa o mecanismo de controlo, o qual contém: uma estrutura de dados (*___sComp*) onde ficam registados as instâncias; e os métodos necessários à gestão dessa estrutura, a saber:

- *int regComp(Component*)*: Método utilizado para efectuar o registo da instância de um componente em *compManager*. Devolve um valor positivo, caso o registo seja efectuado com sucesso. Caso contrário, devolve zero.

- *bool hasComp(char*)*: Método que permite determinar se existe alguma instância, de um dado tipo de componente, registada em *compManager*. Se existir, o método retorna *true*, caso contrário retorna *false*.
- *bool hasComp(Component*)*: Método que permite determinar se uma dada instância está ou não registada em *compManager*. Se estiver, o método retorna *true*, caso contrário retorna *false*.
- *Component *getComp(char*)*: Método que permite aceder à instância, de um dado tipo de componente, que se encontre registada em *compManager*. Retorna o endereço da instância caso esta exista, caso contrário retorna um valor nulo (*NULL*).
- *bool remComp(char*)*: Método que permite remover a instância, de um dado tipo de componente, que se encontre registada em *compManager*. Se a instância existir e se for devidamente removida, o método devolve *true*. Caso contrário devolve *false*.
- *bool remComp(Component*)*: Método que permite remover uma dada instância de *compManager*. Se a instância existir e se for devidamente removida, o método devolve *true*. Caso contrário devolve *false*.
- *Component *execute(char*)*: Método que permite requerer a execução da instância, de um dado tipo componente, que se encontre registada em *compManager*. Se a instância existir, é executada e o método retorna o seu endereço. Caso contrário, o método retorna um valor nulo (*NULL*).
- *Component *update(char*)*: Método que permite requerer a actualização da instância, de um dado tipo de componente, que se encontre registada em *compManager*. Este método faz parte da solução desenvolvida para garantir a consistência dos dados (ver Secção 5.2.3).
- *unsigned getState(char*)*: Método que permite determinar o estado da instância, de um dado tipo de componente, que se encontre registado em *compManager*. Faz também parte da solução desenvolvida para garantir a consistência dos dados.
- *int update()*: Método que visa actualizar todas as instâncias registadas em *compManager*.

De notar que nos métodos anteriormente apresentados, os componentes são identificados através de duas soluções distintas: uma que tem por base o endereço da instância (*Component**); e outra que recorre a uma *string* (*char**), que corresponde ao identificador do componente (designação do componente). Esta última solução, que não tem que ser necessariamente feita com base numa *string*, fornece a abstracção necessária para que a identificação das instâncias se faça pelo tipo de componente e não por referências directas às instâncias. Este tipo de identificação é fundamental para a reutilização e associação de componentes. Isto porque, quando um componente principal vai à procura de uma determinada instância (de suporte), apenas conhece o componente do qual a instância derivou, não possui qualquer outra informação que lhe permita identificar directamente a instância. Aliás, esta é a informação que se pretende apurar (o endereço da instância) para se poder reutilizar os componentes de suporte.

A Figura 5.8 ilustra os procedimentos a realizar para se aceder a uma determinada instância, em que $Comp_s$ representa o componente de suporte e $Elem_s$ o elemento da RIC utilizado pela instância de $Comp_s$. O acesso à instância de suporte faz-se através do método *Component* getComp(char*)*, indicando para tal o tipo de componente ao qual a instância

pertence ($Comp_s$). Pretende-se através deste método, apurar se a instância existe e, se tal acontecer, qual é a sua referência. Se não existir nenhuma instância de $Comp_s$ aplicada a $Elem_s$, então é criada uma nova instância e registada em $Elem_s$.

```
(1)   ...
(2)   Comp_s *c1=Elem_s->getComp("Comp_s"); // Acesso à instância de Comp_s
(3)   if(!c1){                            // Se a instância não existir então:
(4)        c1=new Comp_s(Elem_s);         // cria uma nova
(5)        c1->execute();                 // e executa-a
(6)   }
(7)   ...
```

Figura 5.8: Procedimentos a realizar para a reutilização de componentes.

A interface *compManager* possui outros métodos, alguns dos quais serão abordados posteriormente dado que fazem parte das soluções desenvolvidas para os restantes problemas. Há no entanto outros métodos que visam resolver situações específicas, por exemplo, apesar de não ser possível que uma instância de um determinado componente possua vários registos associados a um único elemento da RIC (não faz sentido ter a mesma instância registada várias vezes sobre o mesmo elemento da RIC), é no entanto possível ter várias instâncias do mesmo componente registadas no mesmo elemento. Solução que é por vezes utilizada para analisar a evolução do processo de compilação[4]. Como os métodos anteriormente apresentados não permitem lidar com este tipo de situação, a de ter várias instâncias do mesmo componente registadas sobre o mesmo elemento, foi necessário definir algumas variantes desses métodos.

Existem ainda variantes para os casos em que o componente (instância) deve ser registado em vários elementos da RIC. Estas variantes só devem ser utilizadas em casos muito específicos, nomeadamente: quando o componente requer elementos distintos e não pode ser decomposto em sub-componentes cujos registos sejam efectuados num único elemento; ou quando os elementos requeridos pelo componente, não possuem um ascendente comum que os permita representar a todos.

Apesar da solução até aqui apresentada, já ser perfeitamente funcional, falta no entanto explicar como identificar e aceder ao elemento da RIC, onde eventualmente está registada a instância a reutilizar. O que mais não é que fazer a associação de componentes.

5.2.2 Associação de componentes

Como foi anteriormente explicado, a associação de um componente de suporte a um componente principal, é um problema que advém da reutilização dos componentes e que cria algumas dificuldades relacionadas com a gestão e identificação das instâncias. Utilizando apenas os mecanismos base da *framework*, é impraticável efectuar a associação dos componentes, designadamente quando o número de instâncias envolvidas e a variedade de componentes é muito grande (ver explicação na Secção 5.1.1).

[4]Como acontece com alguns componentes de inspecção, que requerem a aplicação de várias instâncias sobre o mesmo elemento da RIC. Cada uma das instâncias, que é executada num ponto distinto do processo de compilação, computa um determinado conjunto de parâmetros. É comparando os parâmetros computados pelas várias instâncias, que se consegue analisar a evolução do processo de compilação.

Como também já foi dito, cada componente está associado a um elemento da RIC. O facto de dois componentes serem dependentes, nomeadamente quando há dependências de dados, deriva da existência de uma relação entre os respectivos elementos da RIC, como mostra a Figura 5.5. Se tal relação não existir, então não faz sentido que existam dependências entre os componentes aplicados a esses elementos. Na realidade o problema de associação entre componentes deve ser visto pela seguinte perspectiva (definida com base no exemplo da Figura 5.5): a dependência entre componentes surge da necessidade de uma instância C_{Bi}, aplicada ao elemento B_i, necessitar de determinada informação sobre um ou mais elementos que estejam relacionados com B_i (por exemplo: A_j). Informação essa que é determinada pelo componente C_A, isto é, para o elemento A_j é determinada pela instância C_{Aj}. É neste contexto que surge a dependência entre as instâncias C_{Bi} e C_{Aj}, que se reflecte nos respectivos componentes, fazendo de C_A um componente de suporte a C_B (componente principal). Olhando para o problema por esta perspectiva e atendendo que:

- Cada componente está associado a um elemento da RIC;
- Todos os componentes fazem uso da RIC, dado que é o mecanismo base utilizado para fazer passar a informação entre componentes;
- E que a DIR potencia a construção de representações intermédias de código, em que é normalmente possível e fácil aceder, a partir de um determinado elemento, a outros que estejam com ele relacionados (directa ou indirectamente), independentemente do nível de abstracção dos elementos envolvidos;

A solução torna-se então evidente: a associação dos componentes pode e deve ser feita através da própria RIC. Na *framework* original, o único problema a ultrapassar residia no facto de as instâncias dos componentes não estarem associadas aos elementos da RIC[5]. No entanto esta relação, entre instância e elemento da RIC, está agora efectivada com o mecanismo de controlo criado para a reutilização dos componentes. É assim possível aceder a partir de um elemento da RIC às instâncias a este associadas. E como tal aceder a partir de um componente principal aos respectivos componentes de suporte, através da RIC. A Figura 5.9 ilustra como é que este acesso se efectua.

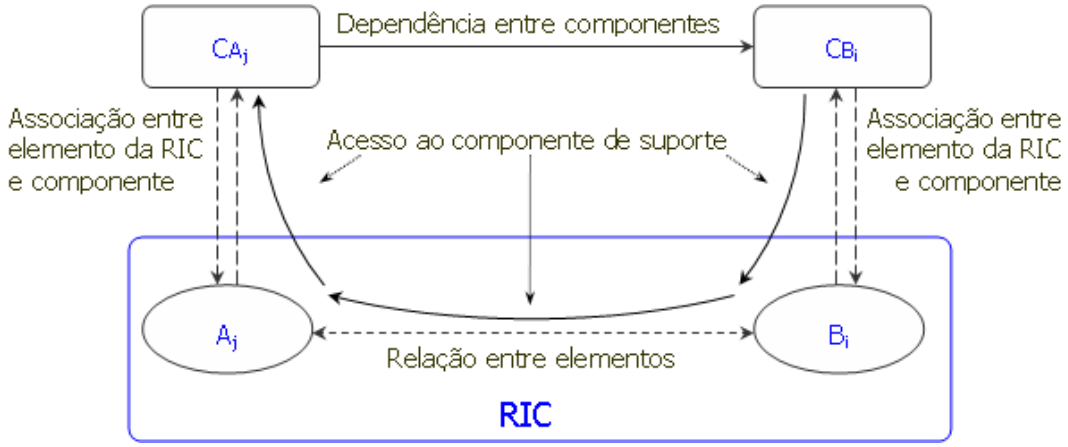


Figura 5.9: Relação entre componentes e elementos da RIC.

[5]Apenas se associava elementos da RIC às instâncias, permitindo aceder a partir de uma instância ao respectivo elemento da RIC, mas não o inverso.

Em termos práticos, não é necessário implementar nada de novo, dado que a solução utilizada para a reutilização de componentes já contém tudo o que é necessário. Partindo do princípio que as instâncias já se encontram vinculadas aos respectivos elementos da RIC, ou seja, que os registos estão devidamente efectuados (conforme ilustra a Figura 5.9), basta então realizar os procedimentos equivalentes aos da Figura 5.8. Para melhor se entender o processo de associação de componentes, a Figura 5.10 mostra a solução utilizada para associar *IDFrontiers* a *cnv2SSA* (recorrendo aos elementos da RIC que são efectivamente utilizados por estes componentes).

```
(1)   class cnv2SSA : public Component{
(2)     ...
(3)     protected:
(4)       ...
(5)       bool execute1(){
(6)         bool st=false;
(7)         if(___elem){
(8)           Function *f = dynamic_cast<Function*>(___elem);
(9)           if(f){
(10)            // Acesso ao elemento do componente de suporte
(11)            CFG *cfg=f->getCFG();
(12)            if(cfg){
(13)              // Acesso ao componente de suporte
(14)              IDFrontiers *idf =
(15)                (IDFrontiers*)(cfg->getComp("IDFrontiers"));
(16)              if(!idf) idf=new IDFrontiers(cfg);
(17)              if(idf && idf->execute()){
(18)                // Execução de cnv2SSA
(19)                ...
(20)                setState(UPDATED);
(21)                st=true;
(22)              }
(23)            }
(24)          }
(25)        }
(26)        return st;
(27)      }
(28)  };
```

Figura 5.10: Exemplo parcial da associação do componente *IDFrontiers* a *cnv2SSA*.

Esta solução, que permite reutilizar e associar os componentes, apresenta várias vantagens, a saber:

- Os componentes são associados dinamicamente e não através de "hard-code", o que permite associar, desassociar ou mesmo substituir os componentes, em plena execução do processo de compilação. Isto faz com que seja possível:
  - Instanciar os componentes de suporte apenas quando são efectivamente necessá-

rios;

– Libertar os componentes de suporte assim que deixem de ser necessários, sem que isso signifique destruir o componente principal.

- Em termos de implementação dos componentes, apenas é necessário acrescentar os procedimentos da linha 2 e 3 da Figura 5.8. Tudo o resto é igual;

- Em termos de utilização dos componentes, os procedimentos a realizar são exactamente os mesmos da versão original da *framework Dolphin* (que faz uso da inclusão implícita de componentes). Por exemplo, a utilização de *cnv2SSA* faz-se exactamente da mesma forma como está ilustrado na Figura 5.2;

- Liberta assim o utilizador da responsabilidade de controlar e gerir as várias instâncias e dependências entre componentes;

- Obtendo a simplicidade da inclusão implícita e a eficiência da inclusão explícita de componentes.

Mas o mais interessante desta solução, é que não só resolve o problema da associação de componentes, como elimina o problema, isto é, para o utilizador dos componentes deixa de fazer sentido falar em associação de componentes!!! Basta só e simplesmente registar os componentes (o que já acontecia na *framework* original), sejam eles inseridos implicitamente ou explicitamente. De notar que este registo é feito entre componente e elemento da RIC e não directamente entre componentes.

É de realçar ainda que é indiferente se estamos a incluir implicitamente ou explicitamente os componentes. Se os componentes forem desenvolvidos tendo em consideração os procedimentos expostos, então a sua utilização não requer qualquer conhecimento sobre os respectivos componentes de suporte. Quem desenvolve componentes, tem apenas de ter o cuidado de implementar os mecanismos que verificam se, aquando da execução, já existem instâncias dos componentes de suporte. Se assim acontecer, o componente principal deverá fazer uso dessas instâncias, caso contrário, deve criar novas instâncias efectuando os respectivos registos.

A solução desenvolvida até aqui, que faz parte da arquitectura proposta nesta dissertação, minimiza assim o número de instâncias utilizadas e maximiza a reutilização dessas mesmas instâncias, optimizando o processo de compilação. É ainda de salientar que qualquer componente inserido pelo utilizador, ou seja, inserido explicitamente, ficará também ele disponível para ser reutilizado. Para além de tudo isto, liberta o utilizador de muitos detalhes e preocupações, fazendo com que a construção dos compiladores seja mais simples, indo assim de encontro aos objectivos estabelecidos para este doutoramento.

Optimização do registo dos componentes

Conhecer detalhadamente a DIR e a estrutura da RIC é um requisito fundamental para se implementar novos componentes, mas é também um requisito importante para a utilização explícita dos componentes. Conforme foi anteriormente explicado, os componentes da *framework Dolphin* utilizam os elementos da RIC que, na perspectiva de quem os desenvolve, melhor se adequam à implementação e/ou execução desses componentes. Significa isto que, para utilizar os componentes na especificação, isto é, de forma explícita, o utilizador poderá ter de "navegar" na RIC, à procura dos elementos onde deve efectuar o registo dos componentes.

Para evitar ou minimizar a necessidade do utilizador conhecer a DIR e a estrutura da RIC, definiram-se algumas recomendações, que não sendo uma parte essencial da arquitectura proposta nesta dissertação, podem facilitar substancialmente a utilização dos componentes. Essas recomendações destinam-se a quem desenvolve os componentes e incentivam a implementação de soluções que: por um lado facilitem o registo dos componentes; e por outro generalizem a aplicação dos componentes.

Para facilitar o registo dos componentes, recomenda-se a implementação de métodos que permitam utilizar elementos da RIC de maior abstracção (em princípio mais acessíveis ao utilizador), através dos quais se possa efectuar o registo. Isto é particularmente adequado quando um componente faz uso de um elemento da RIC pouco acessível, ou que seja pouco conhecido dos utilizadores. Por exemplo, o *elimJumpChain* (ver Secção 5.1.2) é um componente que funciona sobre elementos do tipo *CFG*, que normalmente fazem parte de elementos do tipo *Function*. Para o utilizador é mais fácil e intuitivo fazer o registo sobre *Function*, do que sobre *CFG*, dado que é um elemento de um nível de abstracção superior, cuja utilização é mais generalizada e que é mais fácil de aceder a partir dos objectos *DIR* ou *Program* (elementos produzidos pelos *front-ends*).

Disponibilizar métodos que permitam efectuar o registo utilizando elementos de maior abstracção, não só é possível como é fácil, basta que a partir desses elementos seja possível identificar de forma inequívoca o elemento sobre o qual o componente efectivamente funciona. A Figura 5.11 ilustra como implementar um novo método que permite utilizar *Function*, em vez de *CFG*, para registar o componente *elimJumpChain*

```
(1)    bool elimJumpChain::setElem(CFG *cfg){...}   // Método nativo para
(2)                                                 // efectuar o registo
(3)    bool elimJumpChain::setElem(Function *f){    // Método auxiliar
(4)         if(f) return setElem(f->getCFG());
(5)         return false;
(6)    }
```

Figura 5.11: Método auxiliar para facilitar o registo dos componentes.

A segunda recomendação vai no sentido de disponibilizar soluções que facilitem a utilização de componentes cuja aplicação se faz de forma generalizada, isto é, que são utilizados simultaneamente sobre vários elementos da RIC. Por exemplo, há vários componentes que funcionam sobre elementos de baixo nível de abstracção, mas cuja aplicação visa normalmente ser feita para todos elementos desse tipo. É o caso do *NodeDefVar*, que serve para determinar as definições de variáveis existentes num nodo do GFC[6]. No entanto, este componente raramente é utilizado apenas para um nodo, o mais comum é ser aplicado a todos os nodos do GFC. Se for o utilizador a efectuar esta operação, tem que aceder a *CFG*, para depois percorrer todos os nodos (*FlowNodes*) e aplicar *NodeDefVar*. O que pode ser feito utilizando uma instância para cada nodo ou então utilizando uma única instância, que terá que ser registada e executada sucessivamente para cada um dos nodos.

Para evitar que seja o utilizador a efectuar este tipo de operação, recomenda-se a implementação de soluções que generalizem a aplicação dos componentes. Isto poderá ser feito acrescentando novos métodos, à semelhança do foi realizado para a recomendação anterior,

[6]Este componente é utilizado, por exemplo, pelo *DefReach*, uma forma de análise que permite determinar o alcance das definições das variáveis.

ou implementando componentes que visem efectuar exclusivamente esta operação. Tudo depende dos elementos da RIC e dos componentes envolvidos, e de quem implementa esses componentes. No entanto, a opção por implementar um novo componente faz-se normalmente quando há necessidade de manter, para lá da execução do componente, instâncias individuais para cada um dos elementos, como acontece com *NodeDefVar*. A Figura 5.12 ilustra a implementação desta opção e a Figura 5.13 o respectivo código.

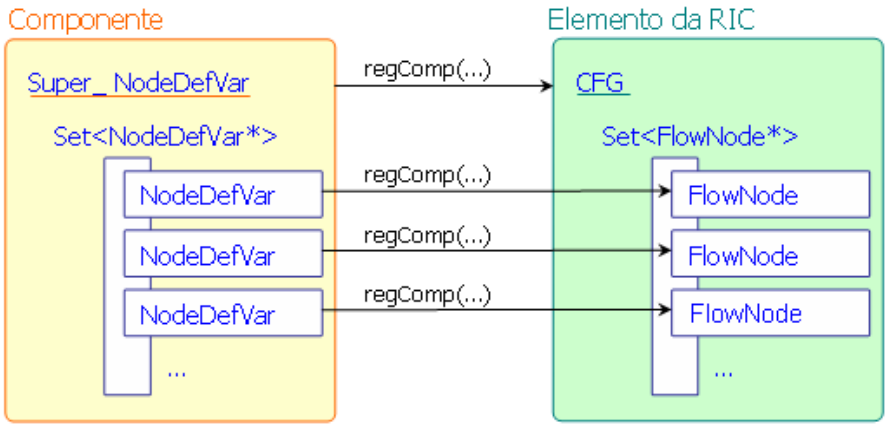


Figura 5.12: Componente para generalizar a aplicação de *NodeDefVar*.

5.2.3 Consistência de dados

Uma vez resolvidos os problemas inerentes à reutilização e associação de componentes, continua no entanto a faltar uma solução que garanta a consistência entre os componentes e a RIC, ou mais especificamente, entre a informação contida nas instâncias dos componentes e a RIC.

Este problema é facilmente ilustrado através da Figura 5.10. Na linha 15 é utilizado o método *getComp(...)* para aceder à instância de *IDFrontiers*, que se encontra registada no elemento *cfg*. Se a instância não existir, então uma nova é criada e executada. Naturalmente que os dados dessa instância, após a execução, vão estar consistentes com a RIC, mas se a instância já existir (o que pressupõe que foi previamente executada), não há como saber se a informação que contém continua consistente com o estado da RIC, isto é, *cnv2SSA* não tem garantias que *idf* (instância de *IDFrontiers*) esteja consistente com a RIC. Não sabe como tal, se a informação de *idf* é válida e se pode ser reutilizada sem ser recomputada. Pelo que, para se utilizar *idf* de forma segura, resta forçar sempre a sua recomputação.

Para evitar que tal ocorra, nomeadamente quando a instância está devidamente actualizada, faz falta uma solução que permita controlar o estado dos componentes. Neste sentido foram analisadas as seguintes soluções:

- Delegar nas instâncias dos componentes a responsabilidade de controlar o seu próprio estado, isto é, periodicamente ou quando interrogada, a instância deverá comparar os seus dados com o estado actual da RIC, para assim determinar se está ou não consistente;

- Delegar nos componentes que produzem alterações na RIC, a responsabilidade de notificarem os outros componentes, fornecendo indicações sobre os elementos da RIC que alteraram ou, em alternativa, informação sobre as operações realizadas. Permitindo

```
(1)    class Super_NodeDefVar :public Component{
(2)       public:
(3)          Super_NodeDefVar(CFG *g):___sComp(0){setElem(g);}
(4)          bool setElem(CFG *g){
(5)             FlowNode *n;
(6)             bool st=Component::setElem(g);
(7)             if(st && ___elem)
(8)                for(n=___elem->init();n;n=___elem->getNext())
(9)                   st &=___sComp.ins(new NodeDefVar(n));
(10)            return st;
(11)         }
(12)         ...
(13)      private:
(14)         Set<NodeDefVar*> ___sComp;
(15)         bool execute1(){
(16)            bool st=true;
(17)            NodeDefVar *c;
(18)            if(___elem)
(19)               for(c=___sComp.init();c;c=___sComp.getNext())
(20)                  st &= c->execute();
(21)            return st;
(22)         }
(23)         ...
(24)   };
```

Figura 5.13: Exemplo da implementação do componente *Super_NodeDefVar*.

assim aos componentes notificados determinar se continuam ou não consistentes com a RIC;

- Delegar na RIC a responsabilidade de notificar os componentes, sempre que sofrer alterações, indicando os elementos alterados e/ou as alterações sofridas.

A primeira alternativa pode ser tão complexa e elaborada como ter que voltar a recomputar toda a informação da instância. Pior ainda, é que para verificar a consistência da instância há que analisar detalhadamente a RIC, o que pode requerer mais conhecimento sobre a DIR e sobre a estrutura da RIC, do que aquele que é necessário à própria implementação dos componentes.

A segunda alternativa tem um inconveniente muito grande, comum à primeira alternativa, que é confiar a implementação dos mecanismos de controlo da consistência de dados a quem desenvolve os componentes. As consequências podem ser graves, se os mecanismos não forem implementados correctamente, ou pior ainda, se quem desenvolve os componentes desconhecer em absoluto que tem de implementar esses mecanismos. Para além disto não é fácil estabelecer de antemão o relacionamento entre os componentes que efectuam transformações na RIC e os componentes cujo estado interno (dados) está dependente da RIC.

Com estas desvantagens, e atendendo aos objectivos deste doutoramento e a própria filosofia de desenvolvimento da *framework Dolphin*, não é possível fazer uso destas duas

alternativas. Resta assim a terceira que, de uma forma bastante acessível, permite resolver ambos problemas, isto é: o mecanismo que faz o controlo do estado dos componentes fica implementado na própria RIC, não depende assim de quem implementa os componentes; e o relacionamento entre os componentes que provocam alterações à RIC e os componentes que possuem informação sensível à RIC, é feito através da própria RIC.

A solução foi desenvolvida utilizando uma abordagem standard, definida pelo padrão de desenho *Observer*, que se aplica à resolução de problemas caracterizados por: "*There is a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically*", página 293 de [GHJV95]. O que encaixa na perfeição com o caso em estudo, isto é, há elementos da RIC que são utilizados pelos componentes para computar informação. Sempre que o estado desses elementos se altera, os componentes devem ser notificados e actualizados. A Figura 5.14 mostra os elementos intervenientes neste padrão de desenho, o *Observer* e o *Observed*, e a sua adaptação ao caso actual. Cada um desses intervenientes vai corresponder a uma interface com a mesma designação. Assim, *Observer* é a interface a implementar pelos observadores, isto é, pelos componentes; e *Observed*, a interface a implementar pelos observados, isto é, pelos elementos da RIC. Ambas interfaces estão representadas na Figura 5.15.

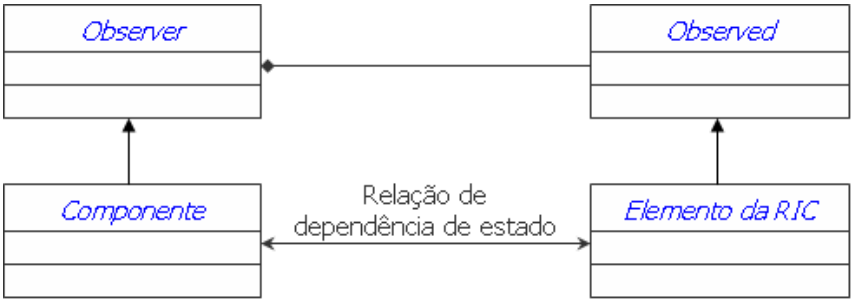


Figura 5.14: Adaptação do padrão de desenho *Observer* à *framework*.

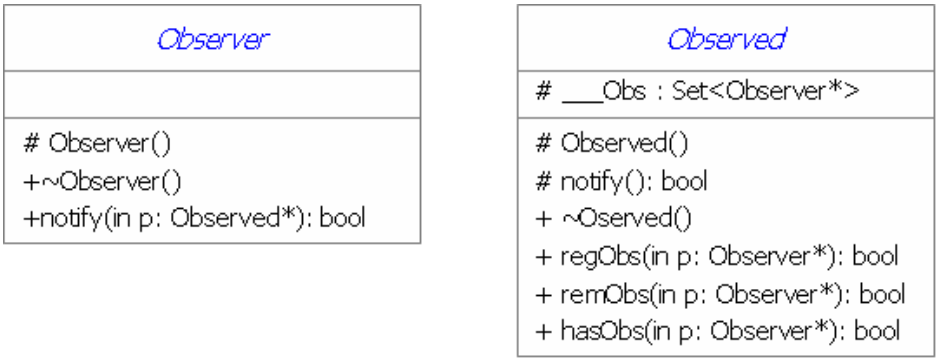


Figura 5.15: Interface *Observer* e *Observed*.

A Figura 5.16 mostra os intervenientes deste processo e a forma como estão relacionados. À esquerda está um componente, normalmente do tipo *Optimização*, que vai alterar o estado do elemento da RIC, que se encontra representado ao centro (*Observed*). A alteração do estado é representada através de *setState()*. Sempre que o estado do elemento da RIC se altera, este notifica os seus observadores, neste caso o componente que está representado do lado direito (*Observer*). A notificação é representada através de *notify()*.

Para ser notificado, um componente tem que se registar no elemento, ou elementos,

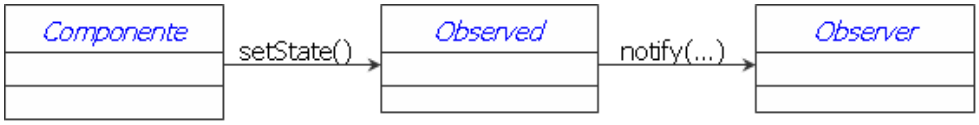


Figura 5.16: Esquema de funcionamento do padrão de desenho *Observer*.

da RIC que quer observar, utilizando o método *bool regObs(...)* da interface *Observed*. Assim, se o elemento da RIC sofrer alguma alteração, consequência da execução de outros componentes, envia uma notificação a todos os seus observadores, utilizando o método *bool notify(Observed*)* redefinido a partir da interface *Observed*. Com a notificação, o observador recebe a indicação de qual foi o elemento alterado (que em princípio será também o elemento que emite a notificação).

É aqui que entra a variável *__state* da interface *Component* (ver Figura 5.6). Antes de mais convém explicar que dentro do contexto actual, esta variável poderia pertencer à interface *Observed*. Está no entanto implementada na interface *Component*, porque integra outras soluções que também fazem uso desta interface e porque em termos conceptuais faz a ligação entre *Observer* e *Component*[7].

A variável *__state* é, no contexto actual, utilizada para identificar o estado do componente. Pode, como tal, assumir um dos dois seguintes valores: *UPDATED*, para quando os dados da instância estão consistentes com a RIC; e *OUTDATED*, para o caso oposto. Para ler e escrever na variável *__state* existem dois métodos: o *bool getState(...)* e o *void setState(bool)*. Este último é um método protegido, como aliás acontece com a variável *__state*, o que significa que só podem ser acedidos pelos métodos da própria classe (ou das classes derivadas), evitando assim que elementos externos, que não pertençam à solução desenvolvida para garantir a consistência dos dados, alterem o valor de *__state*.

Quando uma instância recebe a notificação (através do método *bool notify(Observed*)*), tem duas alternativas: colocar o seu estado em *OUTDATED*; ou actualizar de imediato os seus dados. Para ambas alternativas é necessário redefinir *bool notify(Observed*)*, isto porque, no primeiro caso não é possível aceder a partir da interface *Observer* à variável *__state*; e no segundo caso, porque a actualização da instância depende do componente em causa. É também importante que após a execução (efectuada através do método *bool execute()* da interface *Component*) se coloque o estado das instâncias em *UPDATED*. Ambas situações encontram-se ilustradas na Figura 5.17.

A interface *Observed* contém a estrutura de dados, onde ficam registados os observadores, e os métodos necessários à sua gestão, a saber:

- *bool regObs(Observer*)*: Serve para efectuar o registo de um dado observador. O método devolve *true* se o registo for efectuado com sucesso, caso contrário devolve *false*;
- *bool remObs(Observer*)*: Serve para remover o registo de um dado observador. Devolve *true* se o observador for removido com sucesso, caso contrário devolve *false*;
- *bool hasObs(Observer*)*: Serve para averiguar se um dado observador está ou não registado;

[7]Para a grande maioria dos componentes faz sentido que *Observer* derive de *Component*. Optou-se no entanto por não estabelecer esta relação de herança, permitindo assim utilizar a interface *Observer* sem a interface *Component*. O dá uma maior versatilidade à arquitectura, permitindo por exemplo utilizar a interface *Observer* nos próprios elementos da RIC.

```
(1)     class X :public Component, public Observer{
(2)       protected:
(3)         bool execute1(){
(4)           ...
(5)           setState(UPDATED);
(6)           return ...;
(7)         }
(8)         ...
(9)       public:
(10)        bool notify(Observed *p){
(11)          ...
(12)          setState(OUTDATED);
(13)          return ...;
(14)        }
(15)        ...
(16)    };
```

Figura 5.17: Procedimentos para a manutenção da variável *__state*.

- *bool notify()*: Método protegido, utilizado para despoletar o envio das notificações para os observadores, cuja implementação encontra-se na Figura 5.18.

```
(1)     bool Observed::notify(){
(2)       bool st=true;
(3)       Observer *obs;
(4)       for(obs=___Obs.init();obs;obs=___Obs.getNext())
(5)         st &= obs->notify(this);
(6)       return st;
(7)     }
```

Figura 5.18: Implementação do método *bool notify()* da interface *Observed*.

Falta também acrescentar que a implementação desta solução passa em parte pela própria RIC, à qual cabe: a implementação da interface *Observed* para os vários elementos da RIC que possam vir a ser observados; e o controlo absoluto do acesso às variáveis que definem o estado dos elementos da RIC. Esta última condição significa que não deve ser possível aceder directamente às variáveis dos elementos da RIC e que os métodos que alteram o valor dessas variáveis devem ser supervisionados, de forma a despoletarem o processo de notificação. A explicação de como isto é realizado encontra-se na Secção 7.3.

Por parte de quem desenvolve os componentes, deverá ter o cuidado de efectuar o registo nos elementos da RIC que pretende observar, isto é, nos elementos cujo estado influencia os dados do componente. Estado esse que é formado pelo valor das várias variáveis do elemento observado. É também necessário implementar a interface *Observer*, o que no pior dos casos

significa redefinir o método *bool notify(Observed*)*. Para os restantes componentes e em termos de utilização, nada se altera.

Em termos de implementação, existe agora uma solução que permite verificar o estado das instâncias de suporte, evitando por um lado usá-las sem a garantia de que os seus dados estão consistentes com a RIC; e por outro lado, não ter que as recomputar sempre que são reutilizadas. A Figura 5.19 mostra duas alternativas dos procedimentos a executar, em que $Comp_s$ é o componente de suporte e $Elem_s$, o elemento da RIC onde está a instância do componente de suporte. A primeira alternativa, apenas acresce à solução apresentada na Figura 5.8, o teste que verifica se a instância devolvida pelo método *getComp(...)* está desactualizada, e se tal acontecer, requer a sua execução.

```
(1)   // Primeira solução
(2)   ...
(3)   Elem_s *e=...
(4)   Comp_s *c1=e->getComp("Comp_s");
(5)   if(!c1){
(6)      c1=new Comp_s(e);
(7)      c1->execute();
(8)   }else if(c1->getState()==OUTDATED)
(9)      c1->execute();
(10)  ...
(11)  // Segunda solução
(12)  ...
(13)  Elem_s *e=...
(14)  Comp_s *c1=e->update("Comp_s");
(15)  if(!c1){
(16)     c1=new Comp_s(e);
(17)     c1->execute();
(18)  }
(19)  ...
```

Figura 5.19: Procedimentos a executar para a utilização segura dos componentes de suporte.

A segunda alternativa faz uso do método *Component* update(char*)* da interface *compManager*. Este método visa automatizar o processo, efectua para tal a pesquisa da instância e caso esta exista, requer a sua actualização invocando o método *bool update()* da interface *Component*. A implementação destes métodos encontra-se representada na Figura 5.20.

Para o utilizador dos componentes nada se altera, apenas passa a ser muito mais seguro ou eficiente, consoante a solução anteriormente utilizada, fazer uso destes.

Optimização do registo dos observadores

Pelo facto de agora existirem dois tipos de registo: o do componente e o dos observadores, convém estabelecer uma convenção sob a forma de designar estes registos, para que não surjam interpretações erradas do texto. Assim, designar-se-á por *registo do componente*, quando se pretende referir ao registo que permite reutilizar e associar os componentes (*bool regComp(Component*)*); e por *registo dos observadores*, quando se pretende referir ao registo

```
(1)     Component *compManager::update(const char *s){
(2)        Component *cx=getComp(s);
(3)        if(cx) cx->update();
(4)        return cx;
(5)     }
(6)     bool Component::update(){
(7)        if(getState()==OUTDATED) return execute();
(8)        else return false;
(9)     }
```

Figura 5.20: Métodos *update* das interfaces *Component* e *compManager*.

dos componentes enquanto observadores (*bool regObs(Observer*)*).

Também no caso do registo dos observadores, há o problema de se ter que "navegar" na RIC para aceder aos elementos onde o registo deve ser efectuado (*bool regObs(Observer*)*), ou mesmo para outras operações como remover um observador (*bool remObs(Observer*)*), ou confirmar se um observador já está registado (*bool hasObs(Observer*)*). Se por um lado, este problema é atenuado pelo facto dos registos dos observadores ser efectuado por quem desenvolve os componentes, por outro, tem dificuldades acrescidas. O elemento da RIC utilizado para o registo de um componente é, como já se disse, aquele que na perspectiva de quem devolve o componente, fornece as melhores condições para a sua implementação e execução. O qual não tem que ser necessariamente o mesmo elemento onde se vai efectuar o registo do componente como observador. Aliás, o componente poderá necessitar de observar vários elementos da RIC.

Normalmente, os elementos a observar integram o elemento onde se efectua o registo do componente. Significa isto, que são elementos de menor abstracção e como tal menos acessíveis. Acresce ainda que um único componente pode ter que observar vários elementos da RIC e, eventualmente, elementos de tipos distintos, o que requer vários registos. É claro que isto, dificulta a implementação dos componentes, colocando mesmo em questão a utilização da solução que garante a consistência dos dados. Houve assim necessidade de conceber mecanismos que facilitassem o registo dos observadores.

É no entanto de referir, que a solução utilizada no registo dos componentes não serve para este caso, dado que o problema é substancialmente diferente, isto porque:

- Disponibilizar métodos nos elementos de maior abstracção para efectuar o registo dos observadores, à semelhança do que foi feito para o registo dos componentes, de nada vale. Dado que a identificação dos elementos de menor abstracção, onde se vai efectuar o registo dos observadores, não é na maioria dos casos inequívoca;
- Com a agravante de que o registo dos observadores é normalmente feito para vários elementos e por vezes em elementos de tipos distintos.
- Acresce ainda que o registo do componente no elemento da RIC é efectuado por quem utiliza os componentes; já o registo enquanto observador deverá ser efectuado por quem implementa os componentes.

Foi assim necessário recorrer a outro tipo de solução. Utilizou-se para tal uma solução padrão, definida pelo padrão de desenho *Responsability Chain*, que visa a resolução

de problemas definidos da seguinte forma: "*Chain the receiving objects and pass the request along the chain until an object handles it*" página 223 de [GHJV95]. Esta definição encaixa claramente no problema do registo dos observadores, se considerarmos que:

- Há uma cadeia de objectos formada pelos elementos da RIC;
- O primeiro objecto, corresponderá ao elemento onde se faz o registo e o último objecto ao elemento a observar, isto é, o elemento onde efectivamente o observador é registado;
- E que o objectivo é fazer passar a mensagem de registo (ou outras) através dos vários elementos até alcançar o destinatário.

Esta solução envolve elementos com dois tipos distintos de funções: os destinatários, isto é, os elementos a observar; e os elementos que reencaminham as mensagens. O registo dos observadores é assim efectuado directamente nos destinatários, ou então num elemento que não sendo um destinatário, tem a capacidade de reencaminhar a mensagem de registo até este. Em termos de implementação, o destinatário corresponde à interface *Observed*. Já para os elementos que encaminham as mensagens, foi necessário definir uma nova interface que se designou por *regObserver*.

Pretendeu-se, no entanto, permitir que através de uma única mensagem fosse possível efectuar vários registos, minimizando assim o esforço de quem desenvolve os componentes. Em termos conceptuais, isto significa que na cadeia de objectos poderá haver mais do que um destinatário, com a particularidade de que os destinatários não têm que ser necessariamente os últimos elementos da cadeia. A cadeia de objectos poderá assim ser formada por: elementos que apenas reencaminham as mensagens, ocupando posições intermédias na cadeia de objectos, e que devem implementar a interface *regObserver*; elementos que são apenas destinatários, ocupando posições terminais da cadeia[8], e que devem implementar a interface *Observed*; e elementos que possuem ambas funcionalidades, ocupando como tal posições intermédias da cadeia, pelo que implementam ambas interfaces.

É no entanto importante explicar que um elemento da RIC pode ser destinatário para um determinado componente, mas não ser para os demais. Mas dado que é destinatário de pelo menos um componente, deve então implementar a interface *Observed*. Se para esse mesmo componente ou para os demais (dos quais não é destinatário), o elemento tiver que reencaminhar mensagens, então deve também implementar a interface *regObserver*. No entanto, isto não significa que uma mensagem que alcance um elemento que implemente a interface *Observed*, tenha necessariamente que destinar-se a esse elemento. Poderá apenas ser para reencaminhar. Como fazer a distinção entre as duas situações? Faz falta um mecanismo de identificação dos elementos. Isto é, cada tipo de elemento deve possuir um identificador e a mensagem deve levar a identificação dos elementos visados (destinatários). Assim, quando um elemento que implementa a interface *Observed* recebe a mensagem, deve processá-la para averiguar se é um dos destinatários dessa mensagem. Se assim for, deverá então efectuar o registo do observador. Se houver outros destinatários na mensagem e o elemento implementar também a interface *regObserver*, então deverá reencaminhar a mensagem.

A identificação dos elementos poderia ser feita, à semelhança da solução utilizada para os componentes, com base na designação da classe associada ao elemento. Atendendo no entanto, que uma mensagem pode ter vários destinatários e que esta solução (de reencaminhar as mensagens) requer muitas comparações, faz com que a utilização de *strings* não seja a mais

[8] De notar que no caso da RIC a cadeia é na realidade um grafo acíclico directo, que em termos de propagação das mensagens e com algum cuidado comportar-se como uma árvore, possuindo como tal vários elementos terminais.

adequada. Optou-se por utilizar valores inteiros, resultantes de 2^n ($\{1, 2, 4, 8, 16, 32, ...\}$), para identificar os diversos tipos de elementos da RIC (mais concretamente classes da DIR). A particularidade destes valores é que a sua representação binária inclui um único um, tudo o resto são zeros. Por exemplo, o valor decimal 8 corresponde em binário a 0...1000.

Esta solução permite identificar em simultâneo vários elementos, bastando para tal adicionar o valor dos respectivos identificadores. Por exemplo, elementos do tipo *Function* são identificados pelo valor 8 (0...1000), e os elementos do tipo *IdentTable* pelo valor 2 (0...0010). O valor a utilizar numa mensagem que vise ambos elementos, mais não é que 8+2=10 (0...1000 + 0...0010 = 0...1010). De forma a identificar as principais classes da DIR, foi necessário utilizar um identificador do tipo *long long*[9]. Os identificadores estão definidos no ficheiro *~/Framework/Include/Messages.h*.

A implementação deste mecanismo de identificação de elementos, passou por acrescentar: uma nova variável à interface *Observed* (*__msg*), a qual serve para salvaguardar o identificador do elemento; e os métodos de acesso a esta variável, nomeadamente *long long getMsg()* e *void setMsg(long long)*. Para implementar a solução que permite enviar mensagens com vários destinatários, foi necessário acrescentar novos métodos à interface *Observed*, a saber:

- *long long regObs(Observer*,long long)*: Serve para efectuar o registo de um observador em vários elementos.
  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Identifica o tipo de elementos onde se pretende efectuar os registos;
  - Valor de retorno: Devolve o tipo de elementos onde o registo foi efectuado com sucesso.
- *long long remObs(Observer*,long long)*: Serve para remover o registo de um dado observador, de todos os elementos de um determinado tipo.
  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Identifica o tipo de elementos de onde se pretende remover os registos;
  - Valor de retorno: Devolve o tipo de elementos de onde o registo foi removido com sucesso.
- *long long hasObs(Observer*)*: Substitui o método *bool hasObs(Observer*)* e serve para determinar o tipo de elementos onde se encontra registado um dado observador.
  - Primeiro parâmetro: Identifica o observador;
  - Valor de retorno: Devolve o tipo de elementos onde o observador está registado.
- *long long getObs(Observer*,List<Observed*>*)*: Serve para obter a lista de elementos onde um determinado observador está registado.
  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Funciona como parâmetro de saída, devolvendo uma lista com os elementos onde o observador está registado;

[9]Numa arquitectura convencional, um *long long* corresponde a 8 bytes, ou seja, 64 bits.

  - Valor de retorno: Devolve o tipo de elementos onde o observador está registado.

- *bool certify(Observer*,long long)*: Serve para certificar se um observador está registado em determinado tipo de elementos.

  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Identifica o tipo de elementos onde o observador deve estar registado;
  - Valor de retorno: Devolve *true* se o observador estiver efectivamente registado no tipo de elementos assinalados pelo segundo parâmetro, caso contrário devolve *false*.

A Figura 5.21 mostra a representação UML da interface *Observed* (com os novos métodos e variável).

Apesar das interfaces *regObserver* e *Observed*, terem finalidades distintas, o facto é que fazem parte de uma só solução. Quando um componente faz o registo ou quando um elemento reencaminha uma mensagem, não sabe de antemão qual o tipo de interface implementada pelo próximo elemento da cadeia. No mínimo, é importante que os métodos disponibilizados possuam o mesmo tipo de protótipo, mesmo que executem operações diferentes. Mas não chega, por restrições da linguagem utilizada (C++), o elemento seguinte da cadeia tem que ser identificado por um único tipo (classe), independentemente das funções que desempenha. Isto significa, que *regObserver* e *Observed* devem ter um ascendente comum, ou uma das interfaces derivar da outra. A que servir de interface base, deve definir os métodos que serão disponibilizados ao utilizador, independentemente do tipo de elemento (destinatário ou simples reencaminhador de mensagens). Dado que *Observed* requer uma estrutura de dados onde são guardados os registos (*__Obs*) e um identificador do tipo de elemento (*__msg*), que não são necessários à *regObserver*, optou-se por fazer desta última a classe pai e da primeira a classe derivada. Assim, qualquer elemento que faça parte da cadeia, implementa pelo menos a interface *regObserver*. Significa isto, que todos os elementos vão ter a capacidade de reencaminhar mensagens, mas uns fazem-no, outros não. Os elementos destinatários apenas devem implementar a interface *Observed*, que já inclui a *regObserver*.

Tudo o que foi dito até aqui sob a interface *Observed* continua válido, no entanto convém agora perceber que quem define os métodos públicos é a interface *regObserver*, e esses métodos apenas existem em *Observed* porque necessitam de ser redefinidos (dado que visam executar outras operações). A Figura 5.22 mostra a representação em UML da interface *regObserver*. De notar que, para além do construtor, esta interface inclui cinco métodos protegidos, cada um deles é responsável por efectuar o reencaminhamento das mensagens de cada um dos métodos públicos. Por exemplo, *long long regChain(Observer*,long long)* serve para reencaminhar os pedidos de registo, efectuados através de *long long regObs(Observer*,long long)*. Estes métodos protegidos, possuem uma implementação por omissão que não faz qualquer tipo de reencaminhamento. Assim, se um elemento da RIC, mais concretamente uma classe da DIR, necessitar de reencaminhar as mensagens, então deve reescrever estes métodos. Isto porque, o reencaminhamento é específico de cada tipo de elemento, isto é, cada classe da DIR é que sabe por onde deve reencaminhar as mensagens.

A Figura 5.23 mostra a implementação, por omissão, dos métodos: *regChain(...)* e *regObs(...)* da interface *regObserver*; e a Figura 5.24, a implementação de *regObs(...)* da interface *Observed*.

Um exemplo do que deve ser feito por parte de um elemento (neste caso *Function*) que tenha de reencaminhar mensagens, encontra-se na Figura 5.25. Consiste essencialmente em redefinir, se assim for necessário, o método *regChain(...)*. Se o elemento também for um

Figura 5.21: Redefinição da interface *Observed*, por forma a implementar os mecanismos de reencaminhamento de mensagens.

Figura 5.22: Interface *regObserver*.

potencial destinatário, isto é, um elemento a observar, então deverá registar o respectivo identificador, conforme está ilustrado na linha 2 da Figura 5.25.

A Figura 5.26 mostra a implementação de alguns métodos de um componente (*DFrontiers*) que implementa as interfaces *Component* e *Observer*. As instruções da linha 4 e 5 efectuam o registo do componente como observador, o que é feito através de *__elem*. Mas enquanto no primeiro caso, *__elem* é o próprio elemento a observar; no segundo caso, *__elem* é apenas o elemento através do qual se efectua o registo do componente como observador. Os elementos a observar, neste último caso, são do tipo *FlowNode*, isto é, objectos derivados

```
(1)     class regObserver:virtual public FObject{
(2)       protected:
(3)         virtual long long regChain(Observer*,long long){
(4)           return 0;
(5)         }
(6)         ...
(7)       public:
(8)         virtual long long regObs(Observer *c,long long m){
(9)           return regChain(c,m);
(10)        }
(11)        ...
(12)    };
```

Figura 5.23: Implementação dos métodos *regObs(...)* e *regChain(...)* da interface *regObserver*.

```
(1)     class Observed :public regObserver{
(2)       public:
(3)         long long regObs(Observer *c,long long m){
(4)           long long mm=0;
(5)           if(m & ___msg)
(6)             if(___Obs.insKey(c)) mm=___msg;
(7)           return mm|=regChain(c,m);
(8)         }
(9)         ...
(10)    };
```

Figura 5.24: Implementação do método *regObs(...)* da interface *Observed*.

de *JNode*, *CSNode*, *RNode* e *CJNode*.

É também de destacar o controlo efectuado sobre a variável *__state* de *Component*, que é colocada em *UPDATED* no fim da execução do componente (linha 20 da Figura 5.26), e em *OUTDATED* quando notificado pelo elemento observado (linha 27).

A aplicação da solução concebida para garantir a consistência dos dados, é o terceiro e último pilar da arquitectura que se desenvolveu neste trabalho de doutoramento. Conseguiu-se, através desta solução, garantir uma utilização segura das instâncias, minimizando o número de vezes que estas são recomputadas. Tudo isto, sem qualquer custo para quem utiliza os componentes e com um custo mínimo para quem os desenvolve, basta incluir a interface *Observer*, efectuar os registos nos elementos a observar e redefinir o método *bool notify(Observed*)*, como ilustra o exemplo da Figura 5.26.

Como se pode confirmar através do Capítulo 6, foram também desenvolvidas algumas soluções para optimizar o funcionamento da arquitectura, nomeadamente no que diz respeito à recomputação das instâncias dos componentes.

```
(1)     Function::Function(){
(2)        setMsg(WFUNCT);
(3)        ...
(4)     }
(5)     long long Function::regChain(Observer *c,long long m){
(6)        long long mm=0;
(7)        Function *f;
(8)        regObserver *ro=dynamic_cast<regObserver*>(___table);
(9)        if(ro) mm |=ro->regObs(c,m);
(10)       ro=dynamic_cast<regObserver*>(___cfg);
(11)       if(ro) mm |=ro->regObs(c,m);
(12)       for(f=___lFunct->init(0);f;f=___lfunct->getNext()){
(13)          ro=dynamic_cast<regObserver*>(f);
(14)          if(ro) mm |=ro->regObs(c,m);
(15)       }
(16)       return mm;
(17)    }
```

Figura 5.25: Procedimentos a efectuar por um elemento que implementa a interface *Observed*.

5.3 Resumo do capítulo

Findo este capítulo, é já possível identificar as principais entidades que vão formar a arquitectura proposta neste doutoramento. No entanto, considerou-se importante apresentar as soluções de forma individualizada para cada um dos problemas identificados, permitindo ao leitor compreender a razão de ser dessas soluções, bem como identificar que entidades da solução final visam resolver cada um dos problemas expostos. Para uma melhor compreensão das soluções desenvolvidas recomenda-se a leitura do Capítulo 8, que ilustra a utilização das várias interfaces na perspectiva de quem desenvolve componentes. A arquitectura em si, é apresentada numa perspectiva mais global, não orientada aos problemas, mas às entidades que compõem a arquitectura, no Capítulo 7.

```
(1)     bool DFrontiers::setElem(CFG *g){
(2)       bool st=Component::setElem(g);
(3)       if(st && ___elem){
(4)         st &=___elem->regObs(this);
(5)         st &=___elem->regObs(this,WFLOWNODE);
(6)       }else st=false;
(7)       return st;
(8)     }
(9)     bool DFrontiers::execute1(){
(10)      bool st=false ;
(11)      if(___elem){
(12)        Component *dom=___elem->update("IDominator");
(13)        if(!dom){
(14)          dom=new IDominator(___elem);
(15)          if(dom) st=dom->execute();
(16)        }else st=true;
(17)        if(st){
(18)          // Computação das fronteiras de dominância
(19)          ...
(20)          setState(UPDATED);
(21)          st=true;
(22)        }
(23)      }
(24)      return st;
(25)    }
(26)    bool DFrontiers::notify(Observed*){
(27)      setState(OUTDATED);
(28)      return true;
(29)    }
```

Figura 5.26: Exemplo da implementação das interfaces *Component* e *Observer*.

CAPÍTULO 6

Optimização do processo de compilação

Índice

A solução apresentada no Capítulo 5, da qual fazem parte as interfaces *Component*, *compManager*, *Observer*, *regObserver* e *Observed*, já resolve todos os problemas relatados na Secção 5.1 e satisfaz em grande parte os objectivos estabelecidos para este doutoramento. No entanto a sua implementação permitiu detectar novas oportunidades de melhorar a arquitectura.

A solução apresentada neste capítulo visa disponibilizar, através da arquitectura proposta, os meios necessários para que a recomputação das instâncias se faça de forma mais eficiente. Destina-se essencialmente às instâncias de componentes do tipo *Análise* (utilizadas para computar informação sobre a RIC), que implementem a interface *Observer*. O objectivo é fornecer aos observadores informação acerca das operações efectuadas sobre os elementos da RIC. Essa informação poderá permitir, em alguns casos, actualizar os dados das instâncias sem ter que as recomputar integralmente.

6.1 Conceito de contra-método

Atendendo que as alterações que ocorrem na RIC, resultam da invocação de determinados métodos dos elementos que compõem a RIC, então a ideia base da solução aqui apresentada, consiste em criar contra-métodos que compensem, ao nível do componente, as alterações ocorridas na RIC. A Figura 6.1 ilustra a essência desta ideia. A função f_m representa as alterações provocadas pelo método m na RIC, em que o método m pertence a um qualquer elemento da RIC e é utilizado pelos componentes do tipo Optimização. RIC_1 e RIC_2 representam o estado da RIC, respectivamente, antes e depois da aplicação de f_m. A função S_{CA} corresponde às operações realizadas sobre a RIC para apurar o estado do componente C_A. Em termos práticos, equivale às operações efectuadas aquando da execução do componente C_A.

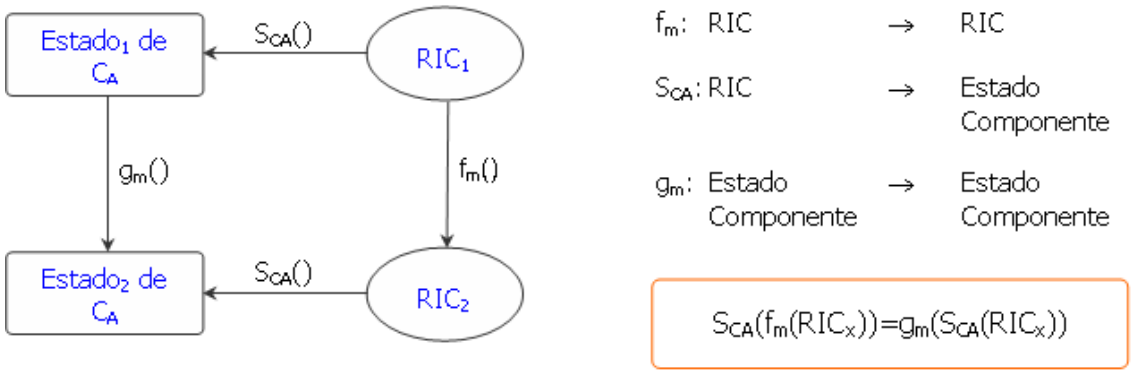


Figura 6.1: Princípio de funcionamento da solução desenvolvida para optimizar a recomputação das instâncias.

O procedimento normal para manter actualizado o estado de um componente é recomputá-lo sempre que ocorre uma transformação na RIC, que de alguma forma afecte a informação contida no componente. Por exemplo, após se aplicar f_m, o estado da RIC passa de RIC_1 para RIC_2; para actualizar o estado do componente C_A é assim necessário voltar a aplicar a função S_{CA} sobre RIC_2. Isto significa no entanto, recomputar integralmente o estado de S_{CA}, independentemente das transformações operadas por f_m sobre a RIC. Em muitos casos, essas transformações são suficientemente restritas e pontuais para que se possa facilmente actualizar o estado do componente C_A sem ter que o recomputar integralmente.

O objectivo é assim obter uma função g_m, a que designaremos por contra-método[1] de f_m, que permita determinar o estado de C_A correspondente a RIC_2 ($Estado_2$), a partir do seu estado actual ($Estado_1$), de tal forma que: $S_{CA}(f_m(x)) = g_m(S_{CA}(x))$, em que x representa um qualquer estado da RIC.

No entanto, a implementação dos contra-métodos requer informação detalhada acerca das operações efectuadas sobre os elementos da RIC. Para que tal informação esteja disponível, desenvolveu-se uma solução formada por dois mecanismos distintos. O primeiro e mais importante, consiste em gerar um relatório que é enviado aquando da notificação aos componentes observadores (componentes que implementam a interface *Observer*). Esse relatório contém informação sobre os métodos invocados sobre os elementos observados (elementos da DIR que implementam a interface *Observed*). O segundo mecanismo, que integra a solução apresentada neste capítulo, permite capturar o estado de um elemento da RIC antes de se efectivarem as alterações.

[1] Utilizando conceitos matemáticos, diz-se que g_m é o conjugado de f_m.

Com estes dois mecanismos, que são disponibilizados através da arquitectura, é possível dotar os componentes de soluções que lhes permitam actualizar o seu estado interno de forma localizada, isto é, executando apenas as operações necessárias para compensar as alterações ocorridas na RIC. Evita-se assim ter que recomputar integralmente toda a informação do componente. Conforme se mostra na Secção 8.3, em determinadas circunstâncias, a utilização desta solução resulta em melhorias significativas do processo de compilação.

De notar, que sem este tipo de solução, sempre que ocorre uma alteração num elemento da RIC, do qual um componente é dependente, o estado desse componente fica irremediavelmente desactualizado e como tal inutilizado. O componente só poderá voltar ser reutilizado após ter sido recomputado integralmente.

A Secção 6.2 descreve a concepção e o desenvolvimento do mecanismo para a geração dos relatórios, e a Secção 6.3 os mecanismos para a captura do estado dos elementos da RIC. A aplicação destes mecanismos é ilustrada no Capítulo 8.

6.2 Geração de relatórios

A geração de relatórios tem por objectivo fornecer informação acerca dos métodos invocados sobre os elementos da RIC que estejam sob a observação dos componentes (que implementam a interface *Observer*). O relatório é enviado aquando da notificação das instâncias. Para tal utiliza-se um objecto da classe *Report*. Este funciona como uma fila de alto nível, onde é guardada a informação dos métodos invocados sobre os elementos observados. Para cada método invocado é salvaguardado: a identificação da classe da DIR à qual o elemento observado pertence; a identificação do método utilizado; os valores dos parâmetros; e também, se existir, o valor de retorno.

No entanto, e atendendo que esta solução pode representar uma sobrecarga significativa para o sistema, o relatório apenas é disponibilizado a pedido dos componentes, os quais podem seleccionar o tipo de elementos (de entre aqueles que se encontram sob a sua observação) dos quais desejam receber os relatórios.

6.2.1 Reformulação das interfaces

Para implementar esta solução, houve a necessidade de modificar as interfaces *Observer*, *regObserver* e *Observed*. No caso da interface *Observer* apenas foi necessário acrescentar o método *bool notify(Observed*,Report*)*, conforme está representado na Figura 6.2. Caso tenha sido requerido o envio de relatório, será este o método utilizado pelos elementos observados para notificar os observadores.

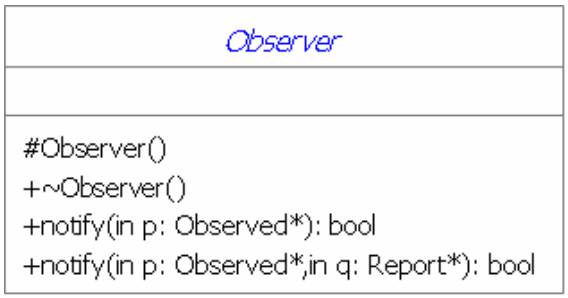


Figura 6.2: Redefinição da interface *Observer*, por forma a implementar os mecanismos necessários à emissão de relatórios.

No caso das interfaces *regObserver* e *Observed* foi necessário redefinir e acrescentar

novos métodos. Foi também necessário substituir o conjunto *__Obs* da interface *Observed*, por um dicionário do tipo *Dict<Observer*,short>*, que serve para manter o registo dos observadores e as funcionalidades requeridas por cada um, como por exemplo: se foi ou não requerida a geração e o envio de relatórios. Os métodos que foram acrescentados e alterados à interface *Observed*, são os seguintes:

- *bool regObs(Observer*,short)*: Este método é uma redefinição do método *bool regObs(Observer*)*, que para além de efectuar o registo directo de um observador num elemento observado, permite requerer a activação de determinados mecanismos, como por exemplo a emissão dos relatórios;

  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Funciona como um array de bits, em que o primeiro bit serve para requerer a geração e o envio de relatórios;
  - Valor de retorno: Devolve *true* se a operação de registo for efectuada correctamente, caso contrário devolve *false*.

- *long long regObs(Observer*,long long,short)*: Este método é uma redefinição do método *long long regObs(Observer*, long long)*, que para além de efectuar o registo do observador num conjunto de elementos, permite requerer a activação de determinados mecanismos, como por exemplo, a emissão dos relatórios;

  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Identifica o tipo de elementos onde deverá ser feito o registo do observador;
  - Terceiro parâmetro: Funciona como um array de bits, em que o primeiro bit serve para requerer a geração e o envio de relatórios;
  - Valor de retorno: Devolve o tipo de elementos nos quais foram efectivamente efectuados os registos.

- *Report *getReport()*: Permite obter o último relatório gerado por um determinado elemento da RIC.

  - Valor de retorno: Devolve o último relatório gerado pelo elemento sobre o qual é invocado o método.

- *long long getReport(Observer*,Dict<Observed*,Report*>*)*: Permite obter o último relatório gerado pelos elementos observados (de um dado observador).

  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Funciona como parâmetro de saída, devolvendo para cada elemento observado, o último relatório por este gerado.
  - Valor de retorno: Devolve o tipo de elementos dos quais foram obtidos relatórios.

- *short getRStatus(Observer*)*: Permite determinar o tipo de funcionalidades requeridas por um determinado observador a um elemento observado.

  - Primeiro parâmetro: Identifica o observador;

  - Valor de retorno: Devolve um *short* que funciona como um array de bits, que assinala as funcionalidades requeridas pelo observador ao elemento observado.

- *long long getRStatus(Observer*,Dict<Observed*,short>*)*: Permite determinar o tipo de funcionalidades requeridas por um determinado observador a cada um dos elementos observados.

  - Primeiro parâmetro: Identifica o observador;
  - Segundo parâmetro: Funciona como parâmetro de saída, devolvendo um dicionário que indica para cada elemento observado, o tipo de funcionalidades requeridas pelo elemento observador;
  - Valor de retorno: Devolve o tipo de elementos para os quais o observador está registado.

A representação gráfica da interface *Observed* encontra-se na Figura 6.3.

Figura 6.3: Redefinição da interface *Observed*, por forma a implementar os mecanismos necessários à emissão de relatórios.

No caso da interface *regObserver*, para além dos métodos *long long getReport(Observer*, Dict<Observed*, Report>*)* e *long long getRStatus(Observer*, Dict<Observed*, short>*)*, que foram anteriormente definidos, foi também necessário acrescentar e modificar os respectivos métodos de reencaminhamento:

- *long long regChain(Observer*,long long,short)*: encaminha as mensagens submetidas através do método *long long regObs(Observer*, long long, short)*;

- *long long getRChain(Observer*,Dict<Observed*,Report*>*)*: encaminha as mensagens submetidas através do método *long long getReport(Observer*, Dict<Observed*, Report*>*)*;
- *long long getRSChain(Observer*,Dict<Observed*,short>*)*: encaminha as mensagens submetidas através do método *long long getRStatus(Observed*, Dict<Observed*, short>*)*.

A representação gráfica da interface *regObserver* encontra-se na Figura 6.4.

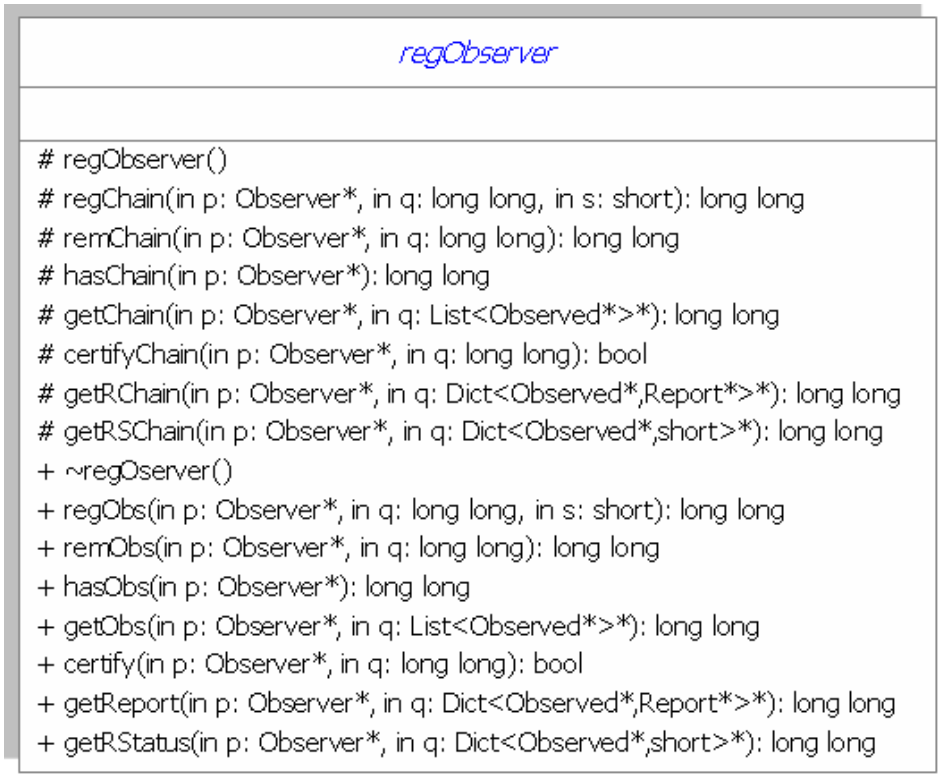


Figura 6.4: Redefinição da interface *regObserver*, por forma a implementar os mecanismos necessários à emissão de relatórios.

O pedido do relatório é feito aquando do registo do observador, utilizando um dos métodos de registo (*bool regObs(Observer*, short)* ou *long long regObs(Observer*, long long, short)*). O relatório é enviado, como já se disse, aquando da notificação ou pode ser requerido explicitamente utilizando um dos seguintes métodos: *Report* getReport()* ou *long long getReport(Observer*, Dict<Observed*, Report*>*)*.

6.2.2 Implementação do *Report*

A classe *Report* consiste numa estrutura de dados do tipo fila e respectivos métodos de acesso. A fila é formada por objectos (apontadores) do tipo *AbstractMethod*, cada um contendo informação respeitante à invocação de um método (identificação da classe e do método, valor dos parâmetros e, se existir, o valor de retorno).

Convém aqui fazer uma breve pausa para assinalar que na maioria das situações, a fila apenas conterá um único objecto do tipo *AbstractMethod*, isto porque, o processo de notificação dos observadores é despoletado sempre que se utiliza um qualquer método do elemento observado (que altere o estado desse elemento). Assim, se forem utilizados n métodos, serão

despoletadas n notificações e como tal gerados n *Reports*, cada um com um único objecto do tipo *AbstractMethod*. Pelo que a notificação poderia funcionar perfeitamente transmitindo ao observador apenas o *AbstractMethod*, em vez do *Report*. Optou-se no entanto por implementar o *Report* pelo desafio que foi desenvolver uma solução deste tipo, mas principalmente por ser uma solução mais abrangente e com bastante mais potencial.

Convém também esclarecer que, por restrições de tempo e mão-de-obra, ainda são poucos os componentes da *framework Dolphin* que tiram proveito da utilização do *Report*.

O *AbstractMethod* visa representar a informação de cada método invocado, designadamente: o identificador da classe e do método, o valor dos parâmetros e, se existir, o valor de retorno. É claro que a ideia é representar a informação de um qualquer método, independentemente da classe à qual pertence, do tipo e número de parâmetros, e do tipo do valor de retorno. No entanto a implementação de uma classe capaz de salvaguardar esta informação, levanta alguns problemas: primeiro, porque o número de parâmetros é variável; segundo, porque quer os parâmetros, quer o valor de retorno, podem ser de um qualquer tipo, nomeadamente de uma classe definida pelo utilizador.

Se o primeiro problema é, eventualmente, ultrapassado utilizando uma simples estrutura dinâmica do tipo lista ligada, já o segundo não é tão facilmente resolvido. Como em C++ (linguagem utilizada no desenvolvimento da *framework*) não existe um objecto universal, tipo o *Object* do Java, resta utilizar uma das duas seguintes alternativas: *templates* (tipos paramétricos), ou apontadores para nulo (*void**). As *templates*, por si só, não permitem construir estruturas dinâmicas com tipos distintos. Por exemplo, não é possível ter uma lista ligada implementada com base numa *template*, em que os elementos sejam de tipos distintos, sem que pelo menos derivem de uma mesma classe base.

A utilização de apontadores do tipo *void*, para além de não tornar credível o trabalho desenvolvido, apresenta fortes desvantagens. É que o problema não reside só na construção do *Report*, há o outro lado da questão, isto é, de quem vai fazer uso do *Report*. É fundamental que a informação que chega ao observador, permita que este identifique facilmente os métodos utilizados no objecto observado. Isto pode ser particularmente difícil, se atendermos que uma determinada classe pode conter vários métodos com a mesma designação e o mesmo tipo de valor de retorno. Nestes casos, a identificação dos métodos faz-se pelo número e tipo dos parâmetros. O que pode ser complicado e pouco seguro de efectuar, quando os parâmetros são representados através de apontadores do tipo *void*. Pois para tal, é necessário efectuar operações explícitas de *cast* sobre os argumentos, o que obviamente é muito indesejável.

Por forma a tornar mais acessível o reconhecimento dos métodos e evitar as operações de *cast* explícito, desenvolveu-se uma solução em que o reconhecimento faz-se com base no identificador da classe, no identificador do método e, para métodos da mesma classe com o mesmo identificador, através do número de parâmetros e/ou através do tipo dos parâmetros. Para determinar o tipo dos parâmetros utiliza-se as próprias capacidades polimórficas da linguagem C++.

Para que isto seja possível, é necessário implementar um contra-método no observador por cada um dos métodos existentes no elemento observado que seja capaz de alterar o estado desse elemento[2]. Cada contra-método deve conter (entre outros) os mesmos parâmetros do método base e, se necessário, mais um para o valor de retorno. O observador pode assim aplicar directamente os valores recebidos através do *Report*, que o polimorfismo da própria linguagem tratará de seleccionar o método a utilizar. A Figura 6.5 mostra um exemplo contendo algumas soluções, que apesar de não serem funcionais, seriam um bom contributo para se efectuar o reconhecimento dos métodos. O exemplo cobre duas situações distintas, a

[2]Para simplificar a leitura do texto, designar-se-á por *contra-método*, ao método implementado no observador; e por *método base* ao método do elemento observado.

saber:

- Reconhecimento de métodos com a mesma designação e com o mesmo número de parâmetros, mas em que o tipo dos parâmetros difere de método para método. Neste caso o reconhecimento é feito através do polimorfismo da linguagem (ver linha 18 da Figura 6.5);
- Reconhecimento de métodos com a mesma designação, mas com um número de parâmetros distinto. Neste caso o reconhecimento pode ser feito pelo número de parâmetros (ver linha 21 à 26 da Figura 6.5).

Os métodos *getMID()*, *getR()*, *getP0()* e *getP1()*, da Figura 6.5, servem para obter, respectivamente, o identificador do método, o valor de retorno, e o valor do primeiro e segundo parâmetros, do método representado por *AbstractMethod* (ver Figuras 6.11 e 6.8).

A implementação do *Report* no entanto só foi possível de concretizar graças à utilização de *typelists* (listas de tipos) [Ale01]. Trata-se de uma tecnologia relativamente nova que integra a linguagem C++, que só muito recentemente se tornou utilizável, isto porque até aqui eram poucos os compiladores capazes de processar este tipo de código e havia alguns problemas de standard[3]. As *typelists* têm por base as *templates* do C++ (uma forma de implementação de tipos paramétricos), e como tal são processadas ("executadas") no início do processo de compilação. Como o próprio nome indica, uma *typelist* mais não é que uma lista de tipos (e não de valores), o que por si só deixa antever algum do potencial desta tecnologia, que se traduz numa mais valia efectiva para os programadores, como mostra a implementação do *Report*.

Há muita coisa a dizer sobre *typelists*, e como este texto apenas inclui uma descrição com o que é estritamente necessário à compreensão da solução desenvolvida para o *Report*, recomenda-se vivamente a consulta do livro "*Modern C++ Design - Generic programming and design patterns applied*" de Alexandrescu [Ale01].

Uma *typelist* é implementada utilizando a definição recursiva de lista, isto é, uma lista ou é vazia/nula, ou é composta por um elemento à cabeça (*head*), seguido de uma lista (*tail*). Para definir a *typelist*, utiliza-se uma *template* de um estrutura (*struct*), a que se designou por *TypeList*. Esta *template* contém dois parâmetros, que são utilizados para definir os dois campos da estrutura: um que representa a cabeça da lista (*Head*), e outro que representa a restante lista (*Tail*). A lista vazia é representada pela classe *NullType*, conforme mostra a Figura 6.6. Por forma a facilitar a implementação do código, são normalmente definidas algumas macros que funcionam como "*templates*" de listas, isto é, como tipos de listas. Cada tipo corresponde a uma lista com um determinado número de elementos.

Com base na definição e implementação feita de *typelists*, é agora possível definir um conjunto de *templates* para representar a informação dos métodos invocados. É necessária uma *template* por cada tipo de lista, incluindo uma *template* para a lista vazia. Essas *templates* têm a particularidade de partilharem a mesma designação, existindo assim uma múltipla definição de *templates*, que ao contrário do que se poderia pensar não resulta necessariamente num erro de compilação. A explicação para este facto está na especialização parcial das *templates*. O C++ permite que se definam *templates* que sejam especializações parciais de uma *template* base. Essas *templates* especializadas mantêm a mesma designação da *template* base mas, por restrições impostas na sua própria implementação, têm um domínio de aplicação mais restrito, ou seja, são mais especializadas. Por exemplo, uma especialização parcial pode consistir em estabelecer, aquando da implementação, o valor de alguns dos parâmetros utilizados na *template* base. Convém no entanto realçar o seguinte:

[3] Actualmente a maior parte das aplicações que fazem uso de *typelists* já podem ser compiladas, nomeadamente com o Visual C++ ou com o g++.

```
(1)  class Comp :public Component,                class Elem:
(2)   public Observer{                              public compManager,
(3)   private:                                      public Observed{
(4)    bool csetX(Observed*p,int r,char *s);        ...
(5)    bool csetX(Observed*p,int r,int i);         public:
(6)    bool csetY(Observed*p,int r,char *s);        int setX(char *s);
(7)    bool csetY(Observed*p,int r,char *s,int i); int setX(int i);
(8)    ...                                          int setY(char *s);
(9)   public:                                       int setY(char *s,int i);
(10)   ...                                          ...
(11)   bool notify(Observed *p,Report *r){         };
(12)     AbstractMethod *m;
(13)     bool b=false;
(14)     int i, nn=r->howMany();
(15)     for(i=0;i<nn;i++){
(16)       m=r->dequeue();
(17)       if(!strcmp(m->getMID(),"setX"))
(18)         b=csetX(p,m->getR(),m->getP0()); // Contra-métodos seleccionados
(19)                                          // através de polimorfismo
(20)       else if(!strcmp(m->getMID(),"setY"))
(21)         switch(m->howManyParam()){       // Contra-métodos seleccionados
(22)           case 1:                         // pelo número de parâmetros
(23)             b=csetY(p,m->getR());break;
(24)           case 2:
(25)             b=csetY(p,m->getR(),m->getP0(),m->getP1());
(26)         }
(27)       r->enqueue(m);
(28)     }
(29)     return b;
(30)   }
(31) };
```

Figura 6.5: Exemplo do processamento do *Report*.

- Independentemente do número de *templates* especializadas ou da forma como estas estão implementadas, o número de parâmetros a utilizar para se criar uma "instância" dessas *templates* é sempre o definido pela *template* base;

- Como uma *template* especializada apenas cobre um sub-conjunto de situações abrangidas pela *template* base, então onde é utilizada a *template* especializada também pode ser utilizada a *template* base. Vendo isto numa outra perspectiva, significa que para uma dada instanciação de uma *template*, podem existir várias definições aplicáveis (*template* base + *templates* especializadas). Ao contrário do que acontece, por exemplo, com a múltipla definição de uma classe, está situação não dá origem a qualquer erro ou aviso, ou seja, é perfeitamente aceite pelos compiladores de C++. A escolha da *template* a utilizar é feita pela ordem de implementação.

```
(1)     class NullType;
(2)
(3)     struct EmptyType{};
(4)
(5)     template <class T,class U>
(6)     struct TypeList{
(7)       typedef T Head;
(8)       typedef U Tail;
(9)     };
(10)
(11)    #define TL0() NullType
(12)    #define TL1(T0) TypeList<T0,TL0() >
(13)    #define TL2(T0,T1) TypeList<T0,TL1(T1) >
(14)    #define TL3(T0,T1,T2) TypeList<T0,TL2(T1,T2) >
(15)    ...
```

Figura 6.6: Estruturas base para a definição de *typelists*.

A definição das *templates* utilizadas para representar a informação sobre a invocação de métodos (*MethodRep*), encontra-se representada na Figura 6.7. Neste caso, a *template* base não está implementada (não existe), apenas existe uma definição que indica que esta *template* requer um único parâmetro (linha 1 da Figura 6.7). As especializações são obtidas restringindo o formato do parâmetro. Por exemplo, no primeiro caso tem que ser uma lista do tipo *TL0()*, no segundo, uma lista do tipo *TL1(...)*, e no terceiro, uma lista do tipo *TL2(...)*. É importante realçar que o parâmetro das *templates* é a lista e não os parâmetros da lista, pelo que existe apenas um único parâmetro. É ainda importante assinalar que, pela forma como estão definidas, estas *templates* são mutuamente exclusivas, significa isto que, para uma dada utilização, não poderá haver mais do que uma definição da *template* que seja aplicável.

É o compilador que efectua a selecção da *template* especializada, que deverá ser utilizada na representação de um dado método. Com base no número de parâmetros do método a representar, o compilador determina o tipo de lista requerida e com base nesta, a *template* a instanciar. Desta forma, as instâncias de *MethodRep* vão diferir entre si no tipo de lista que utilizam, que indirectamente tem a ver com o número e tipo dos parâmetros a representar.

A construção de uma solução do tipo *Report*, em que é necessário lidar com diferentes "instâncias" numa única estrutura de dados (fila), requer uma classe comum/base a todas essas instâncias. Neste sentido, desenvolveu-se a *template Method*, representada na Figura 6.8, que em função do tipo de lista, isto é, do número de parâmetros, selecciona o *MethodRep* a utilizar (linha 28 da Figura 6.8). A dificuldade de implementar esta *template* reside no facto de cada *MethodRep* possuir um número distinto de parâmetros e cada "instância" de um *MethodRep* poder conter diferentes tipos de valores. Assim, para que *Method* possa disponibilizar métodos para aceder aos valores associados a cada parâmetro da *template*, é preciso que seja possível:

- Definir métodos que acedam a parâmetros cujo tipo difere de "instância" para "instância";

```
(1)   template <class TList> class MethodRep;
(2)
(3)   template < > class MethodRep<TL0()>{
(4)     public:
(5)       MethodRep(){}
(6)       ~MethodRep(){}
(7)   };
(8)
(9)   template <typename P0> class MethodRep<TL1(P0)>{
(10)    private:  P0 p0;
(11)    public:
(12)      MethodRep(){}
(13)      MethodRep(P0 q0){setParam(q0);}
(14)      ~MethodRep(){}
(15)      void setParam(P0 q0){p0=q0;}
(16)      void setP0)P0 q0){p0=q0;}
(17)      P0 getP0(){return p0;}
(18)  };
(19)
(20)  template <typename P0,typename P1> class MethodRep<TL2(P0,P1)>{
(21)    private:  P0 p0;P1 p1;
(22)    public:
(23)      MethodRep(){}
(24)      MethodRep(P0 q0,P1 q1){setParam(q0,q1);}
(25)      ~MethodRep(){}
(26)      void setParam(P0 q0,P1 q1){p0=q0;p1=q1;}
(27)      void setP0(P0 q0){p0=q0;}
(28)      void setP1(P1 q1){p1=q1;}
(29)      P0 getP0(){return p0;}
(30)      P1 getP1(){return p1;}
(31)  };
```

Figura 6.7: *Templates* para representar a informação dos métodos.

- Definir métodos que acedam a parâmetros que podem não existir.

É aqui que entram em força as *typelists*, é através destas que é possível apurar aquando da compilação o tipo de cada parâmetro. Isto é efectuado através da *template* *TypeAtNonStrict*[4] que recebe a *typelist*, o índice do elemento a que se pretende aceder e o valor a devolver caso o índice esteja fora dos limites (*EmptyType*). *TypeAtNonStrict* é definido pelas três *templates* representadas na Figura 6.9: a primeira é utilizada quando o índice está para lá do limite da lista, devolvendo neste caso o valor do terceiro parâmetro (neste caso *EmptyType*); a segunda *template* funciona como condição de paragem da terceira *template*, que é recursiva, e são estas duas que em condições normais (índice dentro dos limites) permitem determinar

[4] A implementação destas *templates* foi obtida da biblioteca *Loki*, cuja autoria é de Andrei Alexandrescu [Ale01].

```
(1)  template <typename R,class TList>
(2)  class Method :public AbstractMethod{
(3)   public:
(4)     typedef typename GetValidType<R,void,EmptyType>::Result RET;
(5)     typedef typename TypeAtNonStrict<TList,0,EmptyType>::Result Parm0;
(6)     typedef typename TypeAtNonStrict<TList,1,EmptyType>::Result Parm1;
(7)     typedef typename TypeAtNonStrict<TList,2,EmptyType>::Result Parm2;
(8)     ...
(9)     Method(){}
(10)    Method(Parm0 p0){___lPar.setParam(p0);}
(11)    Method(Parm0 p0,Parm1 p1){___lPar.setParam(p0,p1);}
(12)    Method(Parm0 p0,Parm1 p1,Parm2 p2){___lPar.setParam(p0,p1,p2);}
(13)    ...
(14)    void setR(RET r){___ret=r;}
(15)    void setP0(Parm0 p0){___lPar.setP0(p0);}
(16)    void setP1(Parm1 p1){___lPar.setP1(p1);}
(17)    void setP2(Parm2 p2){___lPar.setP0(p2);}
(18)    void setParam(Parm0 p0){___lPar.setParam(p0);}
(19)    void setParam(Parm0 p0,Parm1 p1){___lPar.setParam(p0,p1);}
(20)    void setParam(Parm0 p0,Parm1 p1,Parm2 p2){
(21)      ___lPar.setParam(p0,p1,p2);}
(22)    ...
(23)    RET getR(){return ___ret;}
(24)    Parm0 getP0(){return ___lPar.getP0();}
(25)    Parm1 getP1(){return ___lPar.getP1();}
(26)    Parm2 getP2(){return ___lPar.getP2();}
(27)  private:
(28)    MethodRep<TList > ___lPar;
(29)    REP ___ret;
(30) };
```

Figura 6.8: Implementação da *template Method*.

o tipo que ocupa uma determinada posição da *typelist*.

É com base no *TypeAtNonStrict* que é possível definir o tipo dos parâmetros *Parm0*, *Parm1* e *Parm2*, da *template Method*, isto independentemente do *MethodRep* em uso. Permitindo assim descrever os construtores e métodos de acesso necessários à *template Method*.

Em relação à variação do número de parâmetros, *Method* considera sempre o número máximo, que no exemplo da Figura 6.8 são três, isto significa que define (e são utilizados na compilação) métodos para aceder até três parâmetros. Se a lista utilizada na instanciação de *Method* contiver um número inferior de parâmetros, por exemplo, um único parâmetro, então os restantes (segundo e terceiro) são do tipo *EmptyType*. No entanto não será possível aceder a estes parâmetros, não por limitações da *template Method*, mas por limitações do *MethodRep* que vai ser utilizado (que neste caso apenas conterá métodos para aceder a um único parâmetro).

A *template Method* possui um segundo parâmetro, representado por *R*, que serve para

```
(1)   template <class TList,unsigned index,typename DefType=NullType>
(2)   struct TypeAtNonStrict{
(3)     typedef DefType Result;
(4)   };
(5)
(6)   template <class Head,class Tail,typename DefType>
(7)   struct TypeAtNonStrict<TypeList<Head,Tail>,0,DefType>{
(8)     typedef Head Result;
(9)   };
(10)
(11)  template <class Head,class Tail,unsigned i,typename DefType>
(12)  struct TypeAtNonStrict<TypeList<Head,Tail>,i,DefType>{
(13)    typedef typename TypeAtNonStrict<Tail,i-1,DefType>::Result Result;
(14)  };
```

Figura 6.9: *Templates* auxiliares para processar as *typelists* (*TypeAtNonStrict*).

definir a variável que salvaguarda o valor de retorno do método a representar (*__ret*). Foi no entanto necessário precaver contra possíveis casos em que os métodos não têm valor de retorno, isto é, quando são do tipo *void*. É que em C++ não é possível definir uma variável, neste caso *__ret*, como sendo do tipo *void*. Para tal desenvolveu-se *GetValidType*, cuja implementação se encontra na Figura 6.10. É uma *template*, implementada com base numa estrutura, que requer três parâmetros: o tipo do valor de retorno; o tipo não válido (neste caso *void*); e o tipo de substituição. O funcionamento é simples, *GetValidType<...>::Result*, que é um tipo de dado, será igual ao primeiro parâmetro se este não for inválido, isto é, se não for igual ao segundo parâmetro; caso contrário será igual ao terceiro parâmetro. Significa isto que se o valor de retorno for do tipo *void*, então *__ret* é definido como sendo do tipo *EmptyType* (ver Figura 6.6); caso contrário é definido como sendo do tipo *R*.

```
(1)     template <typename RetType,typename VoidType,typename DefType >
(2)     struct GetValidType;
(3)
(4)     template <typename RetType,typename DefType >
(5)     struct GetValidType<RetType,RetType,DefType>{
(6)       typedef DefType Result;
(7)     };
(8)
(9)     template <typename RetType,typename VoidType,typename DefType >
(10)    struct GetValidType{
(11)      typedef RetType Result;
(12)    };
```

Figura 6.10: Implementação de *GetValidType*.

Apesar de *Method* permitir utilizar de forma transparente as *templates* *MethodRep*, continua a faltar uma classe comum que permita definir a estrutura de dados necessária a *Report*, isto porque, *Method* não é uma classe mas sim uma *template*. A solução utilizada consistiu em definir uma classe abstracta, o *AbstractMethod*, da qual *Method* deriva. Isto pode parecer uma situação insólita dado que temos uma *template* a derivar de uma classe, mas se nos lembrarmos que as *templates* são processadas previamente, na realidade o que vai derivar de *AbstractMethod* é uma "instância" de *Method*, o que é perfeitamente viável e funcional. *AbstractMethod*, que está representado na Figura 6.11, contém os recursos comuns necessários à representação de um qualquer método, que neste caso se resume ao identificador da classe e do método (a representar), e respectivos métodos de acesso. Na Figura 6.12 encontra-se representada a classe *Report*.

```
(1)     class AbstractMethod{
(2)       protected:
(3)         char *___mid;
(4)         char *___cid;
(5)         AbstractMethod(){___mid="";___cid="";}
(6)       public:
(7)         virtual ~AbstractMethod(){}
(8)         virtual char *getCID(){return ___cid;}
(9)         virtual void setCID(char *s){___cid=s;}
(10)        virtual char *getMID(){return ___mid;}
(11)        virtual void setMID(char *s){___mid=s;}
(12)    };
```

Figura 6.11: Implementação de *AbstractMethod*.

```
(1)     class Report{
(2)       private:
(3)         Queue<AbstractMethod*> ___lstM;
(4)       public:
(5)         Report():___lstM(NULL){...}
(6)         virtual ~Report(){...}
(7)         void enqueue(AbstractMethod *a){___lstM.enqueue(a);}
(8)         AbstractMethod *dequeue(){return ___lstM.dequeue();}
(9)         int howMany(){return ___lstM.howMany();}
(10)        ...
(11)    };
```

Figura 6.12: Implementação de *Report*.

6.2.3 Construção e processamento dos relatórios

Já foi explicada a concepção e implementação do *Report* e já se estabeleceu a forma como o componente observador se relaciona com o elemento observado, para que o primeiro possa ter acesso aos relatórios gerados pelo último. Falta agora explicar como é que se constrói um objecto do tipo *Report*, isto é, quando, onde e como inserir a informação dos métodos invocados em *Report*; e explicar como é que o componente observador pode/deve processar a informação do *Report*.

A construção dos objectos *Report* está relacionada com a implementação da interface *Observed*, cuja adaptação à DIR e mais genericamente à versão original da *framework Dolphin* é explicada na Secção 7.3. Fica no entanto aqui um pequeno exemplo, representado na Figura 6.13, que ilustra como é que o processo se desenrola para o método *bool insNode(FlowNode*)* de *CFG*. Como é explicado na Secção 7.3, os métodos que alteram o estado dos elementos têm que ser reescritos. Por exemplo, neste caso *bool _O_CFG::insNode(FlowNode*)* reescreve o método base (*bool _R_CFG::insNode(FlowNode*)*). Isto não significa que o método base deixa de existir, apenas que o utilizador passa a ver o método reescrito, em vez do método base.

As operações efectuadas pelo método reescrito visam essencialmente assegurar a manutenção do *Report*. Não são efectuadas operações que estejam directamente relacionadas com o elemento em causa (elemento observado). Assim, o método reescrito começa por invocar o método base (*bool st=_R_CFG::insNode(n);*); depois testa a variável *___rep* para determinar se *Report* existe, se assim for, é criado um objecto do tipo *Method*, no qual se inserem os valores dos parâmetros, o valor do resultado e o identificador do método e da classe. Por fim, faz-se a notificação dos observadores através de *Observed::notify()*, enviando o relatório, se este tiver sido solicitado.

```
(1)     bool _O_CFG::insNode(FlowNode *n){
(2)       bool st=_R_CFG::insNode(n);
(3)       if(___rep){
(4)         Method<bool,TL1(FlowNode*)> *m
(5)           =new Method<bool,TL1(FlowNode*) >(n);
(6)         if(m){
(7)           m->setR(st);
(8)           m->setMID("insNode");
(9)           m->setCID("CFG");
(10)          ___rep->enqueue(m);
(11)        }
(12)      }
(13)      Observed::notify();
(14)      return st;
(15)    }
```

Figura 6.13: Exemplo da construção de um objecto do tipo *Report*.

Falta apenas explicar o que acontece do lado do componente observador. Este tem que processar cada um dos *AbstractMethods* contidos na fila de *Report*. Acontece no entanto, que através de *AbstractMethod* não é possível utilizar os métodos definidos em *Method*, funda-

mentais para se aceder aos parâmetros e ao valor de retorno. Infelizmente e tanto quanto foi possível apurar, em C++ não há muitas mais alternativas do que efectuar um *dynamic_cast* para determinar o tipo concreto de *Method* representado por *AbstractMethod*. Isto é feito, indicando ao *dynamic_cast* o tipo de *Method* que se espera obter, isto é, assinalando o tipo de retorno e os tipos dos parâmetros. Se a operação de *casting* for possível de efectuar, o que significa que o *AbstractMethod* realmente corresponde ao *Method* que se pretende obter, então será devolvido um apontador para este último, caso contrário será devolvido *NULL*.

É evidente que o componente não tem como saber qual o *Method* concreto representado por *AbstractMethod*, pelo que o processo de identificação é efectuado por tentativas. Isto é particularmente grave se o componente estiver a observar muitos elementos da RIC e se esses elementos possuírem muitos métodos que permitam alterar o seu estado. O que implica por um lado ter que efectuar muitos testes para identificar o método, e por outro lado, ter que implementar muitos contra-métodos, colocando em questão a utilização desta solução.

No entanto, a experiência adquirida com a aplicação do *Report* à *framework Dolphin* (versão definitiva), permitiu constatar que se a escolha dos elementos a observar for bem feita, raramente é necessário observar mais do que um ou dois tipos diferentes de elementos. Constatou-se também, que normalmente o número de métodos para os quais o componente tem que identificar e implementar contra-métodos, é bastante restrito. Isto porque:

- Em muitos casos, apenas parte da informação de um elemento, interfere com o estado do componente. Como tal, apenas é necessário controlar os métodos que alteram a informação da qual o componente é dependente, e não todos os métodos do elemento observado;

- Constata-se também que muitos métodos são implementados exclusivamente com base noutros métodos mais primitivos. Nestes casos, é suficiente implementar contra-métodos para os métodos primitivos. Os restantes apenas necessitam que se mantenha o estado do componente em *UPDATED*;

- E também, porque em determinados componentes nem sempre é fácil ou vantajosa a implementação de contra-métodos.

De notar que, por omissão e no sentido de precaver o correcto funcionamento de todo o sistema, quando um método não é identificado ou quando não existem contra-métodos, o estado do componente deve ser colocado em *OUTDATED*. Isto permite que a identificação dos métodos seja feita, considerando apenas os métodos para os quais se implementou contra-métodos, o que reduz substancialmente o número de operações de *dynamic_cast*. Convém no entanto explicar que há alguns casos particulares. Por exemplo, como anteriormente se disse, um método que invoque exclusivamente métodos mais primitivos, não requer a implementação de contra-métodos no sentido estrito do termo. Na prática, é necessário implementar um contra-método, não para contrapor as alterações ocorridas na RIC, mas para evitar que o estado do componente passe a *OUTDATED*. É como tal necessária a identificação prévia deste tipo de métodos.

A Figura 6.14 ilustra como é que, segundo esta solução, é possível implementar os contra-métodos e o método *bool notify(Observed*,Report*)*, para o exemplo da Figura 6.5. O Capítulo 8 contém um exemplo completo feito com um componente da *framework Dolphin*.

Esta forma de processar o *Report* não é certamente a solução ideal, mas foi a solução encontrada que melhor se adequava às necessidades do problema em causa. Não estão no entanto exploradas todas as alternativas, pelo que poderá haver soluções melhores para este problema em concreto.

```
(1)   bool Comp::procM(Observed *obs,Method<int,TL1(char*)> *m){...}
(2)   bool Comp::procM(Observed *obs,Method<int,TL1(int)> *m){...}
(3)   bool Comp::procM(Observed *obs,Method<int,TL2(char*,int)> *m){...}
(4)
(5)   bool Comp::notify(Observed *obs,Report *r){
(6)     bool st;
(7)     AbstractMethod *m;
(8)     if(r && getState()==UPDATED){
(9)       int i, nn=r->howMany();
(10)      for(i=0;i<nn;i++){
(11)        m=r->dequeue();
(12)        st=false;
(13)        Method<int,TL1(char*)> *m1
(14)          =dynamic_cast<Method<int,TL1(char*)>*>(m);
(15)        if(m1) st=procM(obs,m1);
(16)        else{
(17)          Method<int,TL1(int)> *m2
(18)            =dynamic_cast<Method<int,TL1(int)>*>(m);
(19)          if(m2) st=procM(obs,m2);
(20)          else{
(21)            Method<int,TL2(char*,int)> *m3
(22)              =dynamic_cast<Method<int,TL2(char*,int)>*>(m);
(23)            if(m3) st=procM(obs,m3);
(24)          }
(25)        }
(26)        if(!st) setState(OUTDATED);
(27)        r->enqueue(m);
(28)      }
(29)    }
(30)    return st;
(31)  }
```

Figura 6.14: Exemplo do processamento do *Report*.

6.2.4 Controlo das prioridades do processo de notificação

A solução até aqui apresentada é perfeitamente funcional para a grande maioria das situações, nomeadamente quando é aplicada a componentes isolados, isto é, que não possuem dependências com outros componentes. No entanto, quando existem vários componentes dependentes entre si, cujo estado é susceptível ao mesmo tipo de elementos da RIC, pode acontecer que a ordem pela qual os elementos da RIC notificam os componentes não seja a mais adequada. Principalmente, quando os contra-métodos são implementados utilizando informação dos componentes de suporte. O diagrama sequencial da Figura 6.15 ajuda a perceber o problema que advém do facto de os componentes não serem notificados pela ordem correcta. $Comp_A$ e $Comp_B$ são dois componentes dependentes ($Comp_B$ suporta $Comp_A$), cujo estado depende de $Elem_{obs}$, um elemento da RIC. $Comp_{opt}$ é o componente que, através do método

fictício *changeState()*, vai provocar alterações em $Elem_{obs}$. Quando tal acontece, $Elem_{obs}$ invoca o método *notify()* para despoletar o envio das mensagens de notificação aos componentes observadores. Neste caso, o primeiro componente a receber a mensagem é $Comp_A$, que possui um contra-método para *changeState()*, representado por *procM(...)*. Este método consegue repor o estado de $Comp_A$ utilizando informação de $Comp_B$, o que é representado através do método *getInfo(...)*. É aqui que reside o problema, a partir do momento que foi executado *changeState(...)*, quer o estado de $Comp_A$, quer o estado de $Comp_B$, deixaram de estar consistentes com a RIC. $Comp_A$ está assim a fazer uso de informação disponibilizada por $Comp_B$ que possivelmente não está correcta.

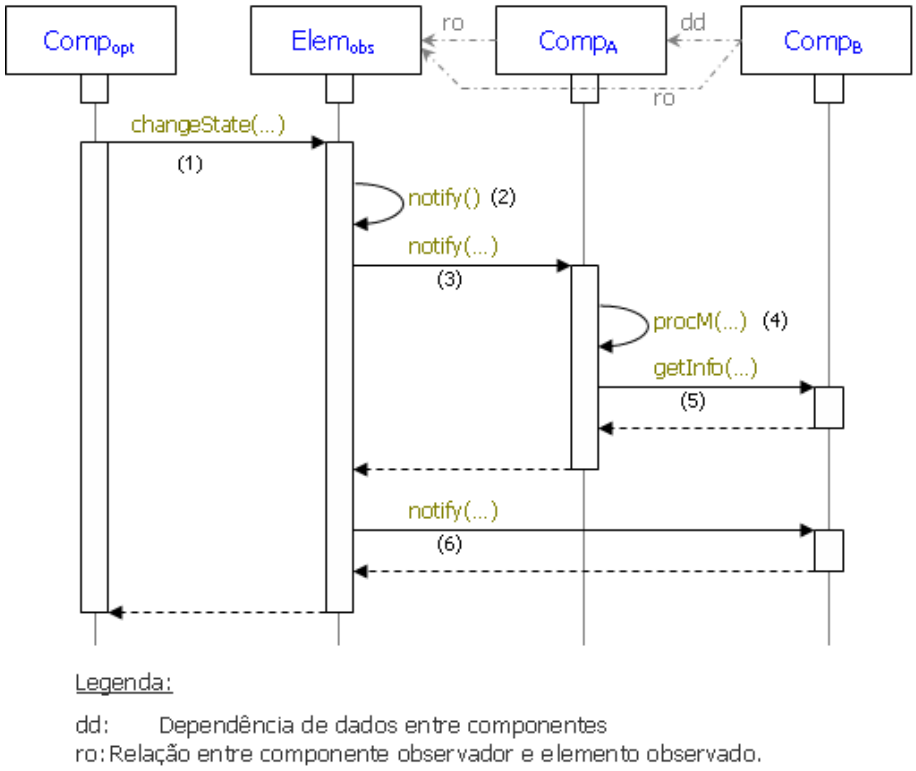


Figura 6.15: Diagrama sequencial ilustrando o problema que se deve à notificação dos componentes não ser efectuada pela ordem correcta.

Como é fácil de perceber, o problema não existiria se primeiro fosse notificado $Comp_B$ e só depois $Comp_A$. No entanto, não existe uma ordem definida pela qual os componentes devem ser notificados[5]. Desenvolver uma solução que de alguma forma permita estabelecer uma ordem global não é simples, dado que requer um modelo global capaz de gerir as dependências entre os vários componentes. Foi assim necessário desenvolver uma solução que permitisse uma ordenação relativa, nomeadamente entre componente principal e componentes de suporte. A solução desenvolvida estabelece esta relação aquando da própria notificação do componente principal. Para tal, antes de executar os contra-métodos, o componente principal deve verificar se os componentes de suporte já foram notificados. Se tal aconteceu então prossegue a sua execução, caso contrário cede a vez, isto é volta a ser inserido na fila de espera utilizada pelo elemento da RIC para notificar os componentes. A implementação desta solução requereu acrescentar os seguintes métodos à interface *Observed*:

[5] A notificação é feita pela ordem crescente dos endereços dos componentes.

- bool wasNotified(char*): Permite testar se um dado componente ainda está na fila de espera para ser notificado.
  - Primeiro parâmetro: Identifica o componente;
  - Valor de retorno: Devolve um booleano que indica se o componente está ou não na fila de espera.
- bool passNotification(Observer*): Permite inserir um determinado componente na fila de espera, para voltar a ser notificado.
  - Primeiro parâmetro: Identifica o componente;
  - Valor de retorno: Devolve um booleano que indica se a inserção foi ou não bem sucedida.
- bool passNotification(Observer*,char*): Permite testar se um dado componente de suporte ainda se encontra na fila de espera, e se tal acontecer insere novamente o componente principal na fila. Na prática este método mais não faz do que executar os dois métodos anteriores.
  - Primeiro parâmetro: Identifica o componente principal (a inserir na fila de espera);
  - Segundo parâmetro: Identifica o componente de suporte;
  - Valor de retorno: Devolve um booleano que indica se a operação foi ou não bem sucedida.

A Figura 6.16 ilustra os procedimentos que o componente $Comp_A$, da Figura 6.15, deve executar aquando da notificação, para se assegurar que todos os seus componentes de suporte ($Comp_B$) foram previamente notificados.

6.3 Captura do estado dos elementos da RIC

O segundo mecanismo incluído na solução desenvolvida para optimizar a recomputação dos componentes, passa por capturar o estado dos elementos observados antes de ocorrerem as alterações. Este mecanismo é fundamental para a implementação de determinados contra-métodos. A Figura 6.17 serve para ilustrar esta situação. $Elem_A$ é um elemento da RIC que contém duas variáveis ($\_\_elem$ e $\_\_sElem$), e pelo menos dois métodos que servem para alterar o valor dessas variáveis (*int remAll()* e *void setB(*$Elem_B$ **)*). $Comp_A$ é um componente cujo estado depende de $Elem_A$, pelo que possui dois contra-métodos que correspondem aos dois métodos base de $Elem_A$. Para que esses contra-métodos consigam actualizar o estado do componente, necessitam de informação acerca do estado actual, mas também do estado anterior à execução dos métodos base, isto é, requerem saber: qual o valor de $\_\_elem$ antes da execução de *void setB(*$Elem_B$*)*; e que elementos continha $\_\_sElem$ antes da execução de *int remAll()*. Um caso concreto, ilustrando este tipo de situação, encontra-se no exemplo apresentado no Capítulo 8.

Para estes casos, em que a actualização do estado do componente requer informação anterior à execução do método base, foi desenvolvido um mecanismo de captura de estado, que integra a arquitectura proposta nesta dissertação. Este mecanismo integra a interface *Observed* e faz uso do mesmo tipo de tecnologia utilizada para o desenvolvimento do *Report* (*typelists*).

```
(1)   bool Comp_A::notify(Observed *obs,Report *r){
(2)     bool st;
(3)     AbstractMethod *m;
(4)     if(r && getState()==UPDATED){
(5)       if(obs->passNotification(this,"Comp_B"))
(6)         st=true;
(7)       else{
(8)         int i, nn=r->howMany();
(9)         for(i=0;i<nn;i++){
(10)          m=r->dequeue();
(11)          ...
(12)          r->enqueue(m);
(13)        }
(14)      }
(15)    }
(16)    return st;
(17)  }
```

Figura 6.16: Procedimento a executar para garantir que os componentes de suporte são notificados antes do componente principal.

Aquando do registo de um observador, este pode requerer a captura do estado aos elementos observados, assinalando para tal o segundo bit do parâmetro do tipo *short* dos métodos *regObs(...)* (ver Figura 6.20). À interface *Observed* foi acrescentada uma nova variável, do tipo *bool*, que serve para assinalar se algum dos observadores requereu a captura do estado do elemento da RIC; e três novos métodos: o *bool getCapState()*, que devolve o valor de *__capstat*; *bool createState(short)*, que é um método virtual protegido, que serve para criar a variável onde vai ficar salvaguardado o estado; e o *AbstractState* getState()*, que devolve um apontador para um objecto do tipo *AbstractState*. Esta é uma classe abstracta que serve para representar objectos do tipo *State* e que desempenha o mesmo papel da classe *AbstractMethod* em relação à *template Method* (ver Figuras 6.11 e 6.8). *State* é a *template* que serve para representar o estado dos elementos observados e a sua implementação é em tudo semelhante à da *template Method*, mas apenas requer um único parâmetro do tipo *typelist*. Enquanto *Method* serve para representar os parâmetros e o valor de retorno de um método, *State* serve para representar o valor das variáveis de um elemento.

De notar que a variável que vai guardar o estado de um elemento *(__state)*, não pode ser definida na interface *Observed*. Isto porque o tipo dessa variável é dependente da *typelist* e dos parâmetros utilizados em *State*. Os quais vão depender de cada elemento em concreto, ou mais correctamente, de cada classe da DIR. Significa isto que a variável deve ser definida nas classes que implementam a interface *Observed*, que também devem redefinir os métodos virtuais disponibilizado por *Observed*, para criar e aceder a esta variável (*bool createState(short)* e *AbstractState* getState()*). A implementação deste mecanismo, para o exemplo da Figura 6.17, encontra-se ilustrada na Figura 6.18.

A Figura 6.19 mostra como efectivar o processo de captura, em que é necessário reescrever os métodos que alteram o estado dos elementos observados, à semelhança do que se fez no exemplo da Figura 6.13. Mais detalhes sobre a reescrita dos métodos são dados na

```
(1)   class Elem_A :public compManager, public Observed{
(2)     private:
(3)       Elem_B *___elem;
(4)       Set<Elem_C*> ___sElem;
(5)       ...
(6)     public:
(7)       void setB(Elem_B *e); // Actualiza o valor de ___elem
(8)       int remAll();         // Remove todos elementos de ___sElem
(9)       ...
(10)  };
(11)
(12)  class Comp_A :public Component, public Observer{
(13)    private:
(14)      Elem_A *___a;         // Elemento observado por Comp_A
(15)      bool procM(Observed *o,Method<int,TL0()> *m){
(16)        ...                 // Actualizações relacionadas com
(17)                            // o estado anterior de Elem_A
(18)        ...                 // Actualizações relacionadas com
(19)                            // o estado actual de Elem_A
(20)      }
(21)      bool procM(Observed *o,Method<void,TL1(Elem_B*)> *m){
(22)        ...                 // Actualizações relacionadas com
(23)                            // o estado anterior de Elem_A
(24)        ...                 // Actualizações relacionadas com
(25)                            // o estado actual de Elem_A
(26)      }
(27)      ...
(28)  };
```

Figura 6.17: Exemplo ilustrando a necessidade de um mecanismo de captura do estado dos elementos da RIC.

```
(1)   class Elem_A :public compManager, public Observed{
(2)     private:
(3)       Elem_B *___elem;
(4)       Set<Elem_C*> ___sElem;
(5)       State<TL2(Elem_B*,Set<Elem_C*>)> *___state;
(6)       bool createState(short sr){
(7)         if(!___state && sr & 2){
(8)           ___capstat=true;
(9)           ___state= new
(10)          State<TL2(Elem_B*,Set<Elem_C*>*)>(NULL,new Set<Elem_C*>());
(11)          return true;
(12)        }else return false;
(13)      }
(14)      ...
(15)    public:
(16)      int remAll(){...}
(17)      void setB(Elem_B *e){...}
(18)      State<TL2(Elem_B*,Set<Elem_C*>)> *getState(){return ___state;}
(19)      ...
(20)  };
```

Figura 6.18: Exemplo da aplicação do mecanismo de captura do estado.

```
(1)   void _O_Elem_A::setB(Elem_B *e){
(2)     if(getCapState()){       // Salvaguardando o estado do elemento
(3)       ___state->setV0(___elem);
(4)       *(___state->getV1())=*___sElem;
(5)     }
(6)     _R_Elem_A::setB(e);      // Invocação do método primitivo
(7)     if(___rep){              // Criação do Report
(8)       Method<void,TL1(Elem_B*)> *m=new Method<void,TL1(Elem_B*)>(e);
(9)       if(m){
(10)        m->setMID("setB");
(11)        m->setCID("Elem_A");
(12)        ___rep->enqueue(m);
(13)      }
(14)    }
(15)    Observed::notify();      // Notificação dos observadores
(16)  }
```

Figura 6.19: Reescrita dos métodos por forma a capturar o estado do elemento.

secção sobre a adaptação das interfaces à *framework Dolphin* (ver Secção 7.3).

A versão definitiva da interface *Observed*, que inclui os mecanismos de reencaminhamento de mensagens, a geração do *Report* e a captura do estado, encontra-se representada na Figura 6.20.

6.4 Resumo do capítulo

Conclui-se assim a descrição de todos os mecanismos que compõem a arquitectura proposta nesta dissertação. É no entanto importante salvaguardar que apesar das soluções apresentadas neste capítulo acarretarem alguma complexidade, a verdade é que são transparentes para os utilizadores da *framework*. Mesmo para quem implementa componentes, pode sempre evitar a utilização destas soluções, optando por implementações mais simples. Recomenda-se mesmo ponderar bem a sua utilização, isto porque:

- A implementação dos contra-métodos depende muito da natureza do componente, pode ser uma tarefa relativamente simples, mas também pode ser tão ou mais complicada do que desenvolver a própria solução base dos componentes;
- Em determinados casos, a implementação dos contra-métodos pode reduzir de forma muito significativa o tempo de compilação (ver Secção 8.1.5), mas também há casos em que ocorre o oposto. Isto porque:
  - A execução dos contra-métodos pode consumir mais recursos do que a simples recomputação do componente;
  - A eficiência dos contra-métodos depende muito do contexto de execução. Por exemplo, poderá não ser eficaz utilizar contra-métodos, se a RIC sofrer muitas alterações concentradas num determinado ponto do processo de compilação, ou se

Figura 6.20: Redefinição da interface *Observed* por forma a implementar os mecanismos para captura do estado (versão definitiva da interface *Observed*).

as alterações abrangem um número significativo de elementos de um determinado nível de abstracção. Nestes casos, provavelmente será melhor esperar que passe o período de turbulência e posteriormente requerer a recomputação integral do componente;

- As vantagens da implementação dos contra-métodos no processo de compilação está muito dependente do tipo de optimizações de código utilizadas na construção do compilador.

Recomenda-se a leitura da Secção 8.1.5, na qual se descreve a aplicação dos mecanismos descritos neste capítulo para optimizar a execução de um componente.

CAPÍTULO 7

Arquitectura

Índice

Uma vez apresentada a *framework Dolphin* (versão original), a Dolphin Internal Representation (DIR) e as soluções desenvolvidas para solucionar os problemas entretanto detectados, é chegada a altura de explicar em que é que consiste a arquitectura proposta nesta dissertação, qual a sua relação com as demais entidades e como é que funciona. Estes são os assuntos abordados nas próximas duas secções. Existe ainda uma terceira secção, onde se explica como é que a arquitectura foi adaptada à versão original da *framework Dolphin*.

7.1 Modelo, entidades e interfaces da arquitectura

A arquitectura que é proposta nesta dissertação consiste num conjunto de interfaces, concebidas em função de um modelo de compilação, que definem o comportamento e o relacionamento entre as várias entidades subjacentes a esse modelo.

O modelo de compilação e as entidades que lhe estão subjacentes foram um contributo da versão original da *framework Dolphin*. Conforme se explicou no Capítulo 3, este modelo assenta na utilização de componentes que funcionam sobre uma forma comum de representação de código, a qual é a espinha dorsal do processo de compilação (ver Figura 7.1). É essencialmente formado por dois tipos de entidades: os componentes (definidos através da interface *Component*) e os elementos da Representação Intermédia do Código-RIC (definidos através da interface *DObject*).

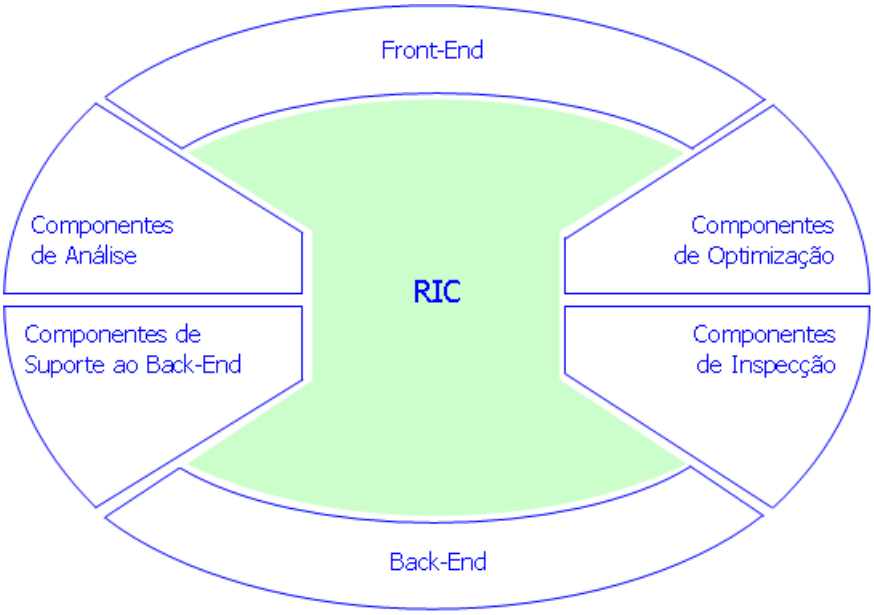


Figura 7.1: Modelo de compilação.

Na realidade, a essência da versão original da *framework Dolphin* é a implementação deste modelo. De tal forma que, o correcto será mesmo dizer, que é o modelo e as entidades por este definidas (interfaces *Component* e *DObject*) que são a *framework*. O que se designa por "*framework Dolphin*" é, na realidade, uma concretização que integra já vários componentes construídos segundo esse modelo e que implementam a interface *Component*.

Daí que o grande contributo da versão original da *framework Dolphin* foi o modelo de compilação, dado que aí foi aperfeiçoado e validado. No entanto subsistiram vários problemas, que não advinham do modelo de compilação, mas da abordagem utilizada pela versão original da *framework Dolphin* (e por outros sistemas semelhantes) para consubstanciar esse modelo. Essa abordagem tinha a vantagem da simplicidade (conforme se mostrou no Capítulo 3), mas não contemplava a qualidade do processo de compilação (conforme explicado no Capítulo 5). Foi no sentido de conjugar a simplicidade de utilização da *framework*, com a capacidade de produzir compiladores eficientes (em termos do processo de compilação), que se desenvolveram as várias soluções descritas nos Capítulos 5 e 6.

Houve no entanto o cuidado de que essas soluções fossem desenvolvidas apenas com base no modelo de compilação, de preferência sem o alterar e sem se comprometerem em relação à *framework Dolphin* (na sua versão original). Desta forma, as soluções desenvolvidas

não alteraram em nada o modelo de compilação, não acrescentaram novas entidades (continuam apenas a existir as entidades *componente* e *elemento da RIC*), apenas definiram novas funcionalidades e comportamentos para essas entidades, nomeadamente no que diz respeito à forma como elas se relacionam. O que é feito, como se mostrou no Capítulo 5, através de cinco interfaces: *Component*, que é uma redefinição da versão utilizada na *framework Dolphin-VO* (ver Figura 5.6); *compManager*, que complementa o *DObject* (ver Figura 5.6); *Observer* (ver Figura 6.2); *regObserver* (ver Figura 6.4); e *Observed* (ver Figura 6.20). As várias interfaces e sua relação com as entidades inerentes ao modelo de compilação, encontram-se representadas na Figura 7.2.

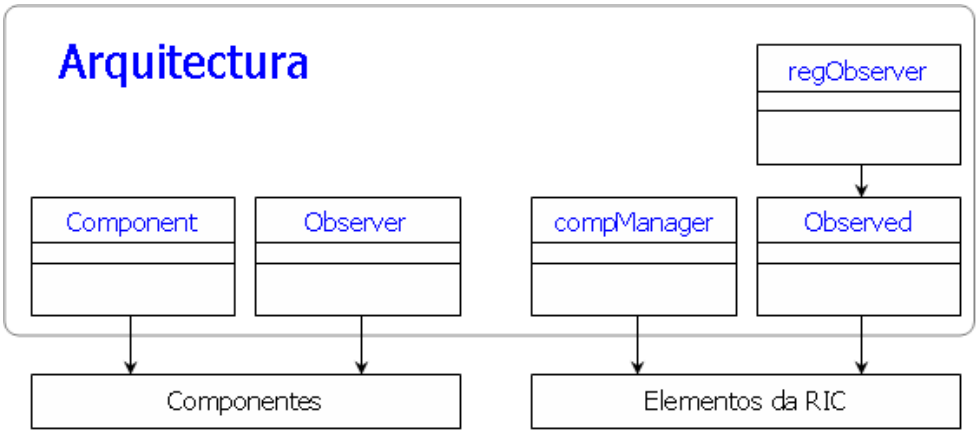


Figura 7.2: Representação gráfica da arquitectura.

A arquitectura proposta resulta assim numa solução capaz de concretizar o modelo de compilação, mantendo todas as suas virtudes (genérico, simples, modular e escalável), sem deteriorar a qualidade do código gerado, nem o processo de compilação, mesmo quando comparado em termos absolutos, isto é, com compiladores construídos directamente por especialistas sem recurso a qualquer tipo de ferramenta. Comparativamente com a solução utilizada na versão original da *framework Dolphin*, conseguiu-se manter todas as vantagens existentes, mas também assegurar que o processo de compilação é tão eficiente quanto possível, sem penalizar a qualidade do código gerado.

É no entanto de notar que a aplicação desta arquitectura resulta, no sentido mais puro da palavra, numa *framework*. Em que a sua concretização, isto é, a implementação de componentes segundo o modelo de compilação e segundo as interfaces definidas pela arquitectura, mais não é que desenvolver uma ferramenta para construção de compiladores, como é o caso da versão definitiva da *framework Dolphin*.

Atendendo ainda que a arquitectura é suficientemente genérica e independente, ao ponto de poder ser aplicada a outras *frameworks* que partilhem de um modelo semelhante ao que aqui é utilizado para o processo de compilação, então não será certamente errado classificá-la como uma meta-framework, que define o comportamento das entidades inerentes a este tipo de modelo. Basta para tal, que as aplicações a desenvolver possam ser construídas com base em componentes, que operem de forma encadeada sob um único conjunto de informação e que, quer os componentes, quer a informação, se encontrem representados sob a forma de objectos (condição que está subjacente à utilização de *frameworks*).

Não seria sensato acabar esta secção sem enumerar as vantagens da utilização desta arquitectura:

- É simples, inteligível e fácil de utilizar;

- A sua aplicação não requer alterações no modelo utilizado;
- As alterações a efectuar nas entidades do modelo são quase sempre mínimas (a maioria das entidades apenas tem que herdar as interfaces);
- Simplifica a construção dos compiladores, dado que permite que os componentes sejam incluídos implicitamente;
- Garantindo simultaneamente a eficiência do processo de compilação;
- Não altera em nada a qualidade do código produzido pelos compiladores;
- Permite efectuar a recomputação optimizada dos componentes;
- Potencia o desenvolvimento de componentes mais normalizados e a compatibilidade entre componentes.

7.2 Funcionamento da arquitectura

Uma vez explicado em que é que consiste a arquitectura, nomeadamente o papel de cada uma das suas interfaces, e sabendo de antemão quais são os seus objectivos, isto é, quais são os problemas que visa resolver, é então chegada a altura de mostrar como é que o faz. Pretende-se assim explicar que operações são executadas nos componentes e elementos da RIC, aquando dos três principais eventos: registo de um componente (registo no elemento da RIC + registo como observador); execução de um componente; e notificação de um componente.

Para além das operações concretas, pretende-se também mostrar que, apesar de ser transparente para os utilizadores, há todo um conjunto de operações que são executadas pela própria arquitectura, no sentido de garantir que tudo corre conforme o esperado, facilitando assim a utilização dos componentes e, como tal, a construção de compiladores.

7.2.1 Registo dos componentes

A Figura 7.3 representa o diagrama sequencial com as principais operações que são executadas aquando do registo de um componente. Na figura estão representados três tipos de intervenientes: o utilizador, que tanto pode ser um componente ou quem efectua a especificação do compilador; o componente que vai ser registado a pedido do utilizador ($Comp_1$) e que implementa a interface *Component*; e um conjunto de elementos da RIC. $Elem_v$ é o elemento mais acessível para o utilizador e sobre o qual este requer o registo de $Comp_1$. $Elem_{rc}$ é o elemento utilizado para se efectuar o registo do componente que, como tal, implementa a interface *compManager*. $Elem_{ro}$ é o elemento do qual o estado de $Comp_1$ está dependente. É, como tal, o elemento onde $Comp_1$ deverá efectuar o registo como observador. Significa isto que, $Comp_1$ implementa a interface *Observer* e que $Elem_{ro}$ implementa a interface *Observed*. $Elem_{rc}$ é também responsável por reencaminhar as mensagens provenientes do observador ($Comp_1$) até ao elemento observado ($Elem_{ro}$), como tal, implementa a interface *regObserver*.

O processo de registo de $Comp_1$ tem início na instanciação deste componente (*new* $Comp_1(Elem_v)$), operação que é despoletada pelo utilizador, indicando o elemento através do qual o registo se deverá efectuar ($Elem_v$). É de referir, no entanto, que o registo não tem que ser feito directamente no elemento no qual o componente vai ficar efectivamente registado, como aliás acontece neste exemplo. Esta facilidade de efectuar o registo num elemento diferente daquele onde o componente deve ficar registado, cabe a quem desenvolve os componentes, não é no entanto de implementação obrigatória mas recomendável, como

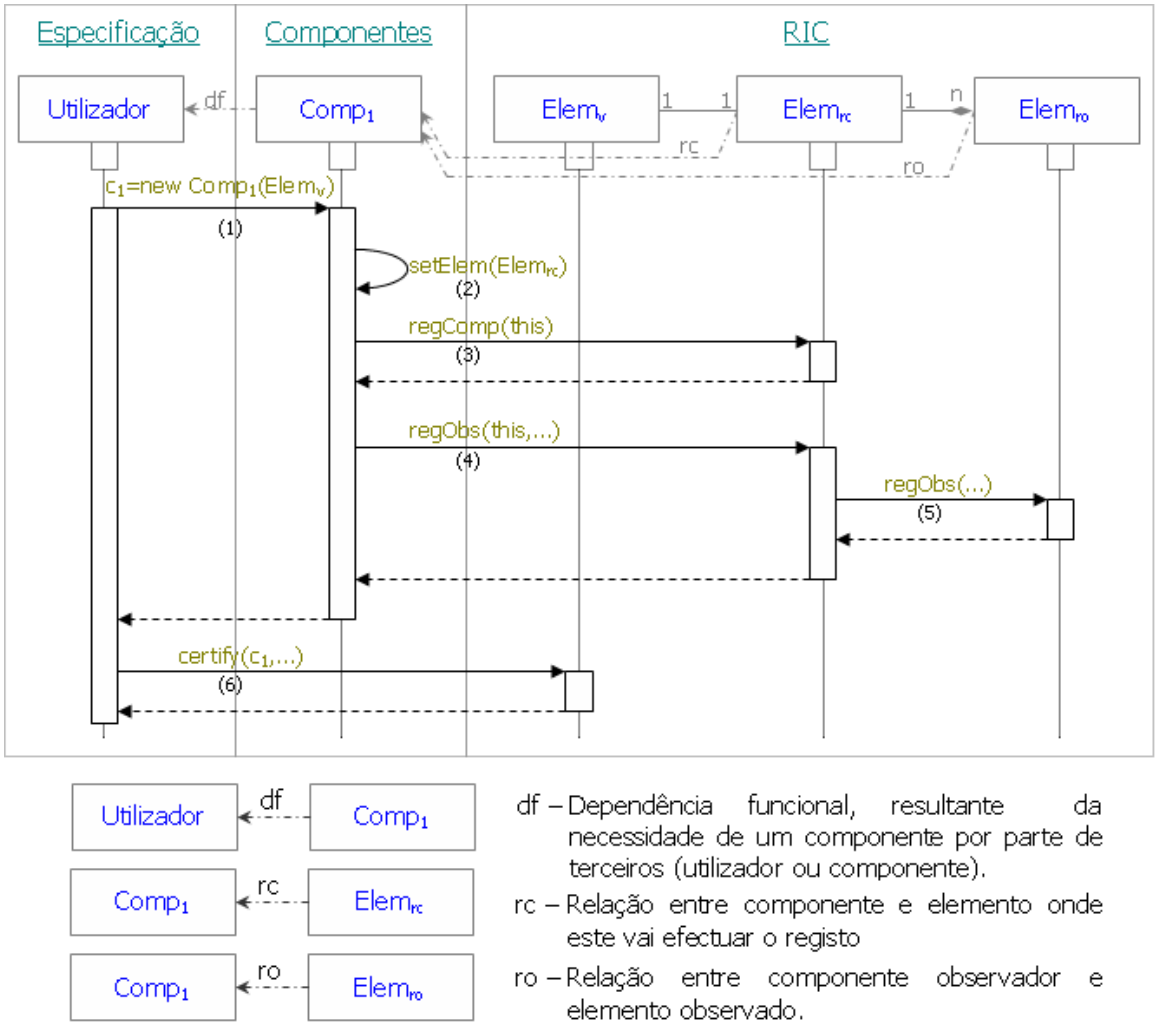


Figura 7.3: Diagrama sequencial do registo dos componentes.

é explicado na Secção 5.2.2. O construtor de $Comp_1$ acede a $Elem_{rc}$, a partir de $Elem_v$, para tal, é importante que a relação entre estes dois elementos seja de um para um (ver explicação na Secção 5.2.2). De seguida, o construtor invoca $setElem(Elem_{rc})$, que é o método de $Comp_1$ que efectivamente faz o registo do componente. Para tal utiliza o método $regComp(this)$, do elemento sobre o qual se efectua o registo ($Elem_{rc}$), ficando o processo concluído com a confirmação desta operação.

No entanto, se o estado de $Comp_1$ depender da RIC, o componente pode e deve requerer o registo como observador nos elementos que influenciam o seu estado. Este registo como observador é efectuado através do método $regObs(this, ...)$, que pode ser invocado directamente do elemento a observar ($Elem_{ro}$), ou a partir de um outro elemento que tenha a capacidade de reencaminhar a mensagem de registo (e outras) até $Elem_{ro}$. Como aliás acontece no exemplo da Figura 7.3, em que o registo é efectuado sobre $Elem_{rc}$, que apesar de não ser o destinatário da mensagem, tem a função de a reencaminhar até ao elemento a observar. Para que isto seja possível, $Elem_{rc}$ deve implementar a interface *regObserver*, os elementos da RIC devem estar devidamente identificados, e as mensagens devem levar a identificação dos destinatários (segundo parâmetro do método $regObs(this, ...)$). Convém ainda relembrar que, com uma única mensagem, é possível efectuar o registo em vários elementos da RIC e

até mesmo em elementos de tipos distintos. A implementação destes mecanismos de reencaminhamento, que são descritos na Secção 5.2.3, não cabe aos utilizadores da *framework*, mas cabe a estes requererem a sua inclusão nos elementos da RIC, como se explica na Secção 7.3.

Em relação ao registo dos componentes, convém acrescentar que existe um mecanismo de teste que permite averiguar se um determinado componente está, ou não, registado como observador num dado elemento. Isto porque a implementação dos componentes pode ser efectuada por terceiros, que por desconhecimento ou por outros motivos, podem não realizar os registos dos componentes enquanto observadores. Conforme explicado na Secção 5.2.3, a falta de tais registos pode ser grave dado que são fundamentais para controlar a consistência dos componentes em relação à RIC. A situação será mesmo grave se os componentes contiverem informação relacionada com a RIC; e se funcionarem como componentes de suporte, isto é, componentes que visam disponibilizar informação a outros componentes (principais).

Assim, e de forma a evitar uma utilização pouco segura dos componentes de suporte, existem dois mecanismos de segurança. O primeiro, passa pela utilização do método *getState(...)*, o qual por omissão, isto é, se quem desenvolve os componentes nada fizer para controlar o estado dos componentes, devolve *OUTDATED*. Significa isto que a informação do componente não está consistente com a RIC, mesmo que tal seja verdade. Isto protege contra situações de negligência ou de desconhecimento, mas não contra situações mal intencionadas, dado que quem implementa o componente pode sempre forçar a variável *__state* a assumir o valor *UPDATED*. A segunda solução, que não é 100% eficaz, permite pelo menos identificar se o componente está registado como observador nos elementos que eventualmente deveria supervisionar. Solução essa disponibilizada através do método *bool certify(Component*,long long)* da interface *regObserver* e *Observed*. Convém no entanto realçar que esta não é uma solução 100% eficaz, o componente pode perfeitamente efectuar os registos e no entanto não se comportar devidamente.

O esquema da Figura 7.3 é ainda importante para se perceber o quanto a arquitectura é fundamental para facilitar a utilização dos componentes. De notar que para o utilizador apenas é visível a parte correspondente à Especificação, isto é, à zona mais à esquerda da figura, que inclui apenas a instanciação do componente e quando muito o pedido de certificação ($certify(c_1, ...)$). Tudo o resto é invisível. É a arquitectura que garante uma utilização simples, eficiente e segura.

Mesmo para quem desenvolve componentes, a solução continua a ser bastante simples. No que diz respeito ao registo do componente basta implementar o *setElem(...)*, que essencialmente inclui a invocação do *regComp(...)* e dos *regObs(...)* (ver exemplo da Figura 5.26). Toda a parte direita da Figura 7.3, que diz respeito à RIC, faz parte da DIR. É, como tal, transparente a quem utiliza ou implementa os componentes.

7.2.2 Execução dos componentes

A Figura 7.4 representa o diagrama sequencial com as operações efectuadas aquando da execução de um componente. A figura contém quatro intervenientes: o utilizador do componente; o componente que vai ser executado a pedido do utilizador ($Comp_{princ}$); um componente de suporte ($Comp_{sup}$); e o elemento da RIC sobre o qual está registado o componente de suporte ($Elem_{sup}$).

A execução do componente é requerida pelo utilizador através de *bool execute()* ou de *bool update()*. O utilizador tanto pode ser quem especifica o compilador, um elemento da RIC ou outro componente. Se o componente que vai ser executado, neste caso $Comp_{princ}$, tiver dependências funcionais, como acontece neste exemplo, então deve requerer a execução dos componentes de suporte ($Comp_{sup}$), utilizando o método *bool update(char*)* (2). O pedido é efectuado ao elemento da RIC sobre o qual é suposto o componente de suporte funcionar

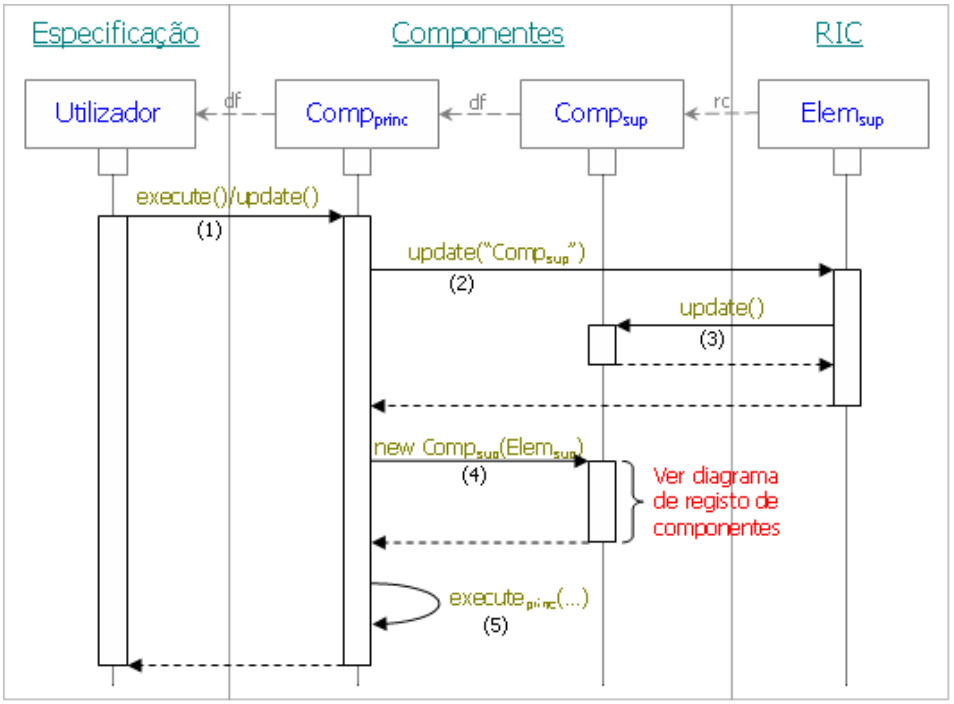


Figura 7.4: Diagrama sequencial da execução dos componentes.

($Elem_{sup}$), isto é, o elemento onde é efectuado o registo do componente de suporte. A execução deste pode ser efectuada antes, durante ou depois da execução do componente principal (tudo depende da forma como este último está implementado). Quando o elemento da RIC recebe a mensagem de *update(char*)* (2), verifica se já possui alguma instância registada do componente requerido. Se assim acontecer efectua um pedido de actualização directamente a essa instância, utilizando o método *bool update()* (3), devolvendo o valor que resulta da invocação deste último método. No entanto, se o elemento da RIC não possuir nenhuma instância do componente requisitado, devolve *false*. Neste caso, $Comp_{princ}$ pode criar directamente uma instância de $Comp_{sup}$ e registá-la em $Elem_{sup}$, em que os procedimentos a realizar são iguais aos do exemplo da Figura 7.3.

A execução do componente principal ($execute_{princ}(...)$) só poderá ser integralmente efectuada, se for possível obter todas as instâncias dos componentes de suporte devidamente actualizadas. Se assim acontecer, o processo termina devolvendo ao utilizador *true*. Se por algum motivo algo falhar na execução do componente, então o resultado final será *false*.

Também aqui, a Figura 7.4 permite perceber o quanto a arquitectura contribui para tornar a execução dos componentes simples e eficiente. Para o utilizador apenas é relevante a parte esquerda da figura (Especificação). Basta-lhe invocar o *execute()* ou o *update()* do componente, que tudo resto ficará a cargo dos componentes e da RIC, desde que tenham sido implementados segundo a arquitectura. Isto garante, que todos os componentes de suporte estarão presentes no momento em que são necessários, que a sua informação estará devidamente actualizada e que será minimizado o número de instâncias utilizadas e número de vezes que estas são recomputadas.

Mesmo para quem implementa componentes, apenas terá que requerer a actualização dos componentes de suporte ($update(Comp_{sup})$) e quando muito, ou seja, à falta destes, criar novas instâncias e registá-las ($new\ Comp_{sup}(Elem_{sup})$).

7.2.3 Notificação dos componentes

A Figura 7.5 representa o diagrama sequencial com as operações efectuadas aquando da notificação de um componente. Estão envolvidos três intervenientes: uma instância de um componente ($Comp_{observador}$), cujo estado depende de um ou mais elementos da RIC, estando assim registado nestes como observador; um dos elemento da RIC ($Elem_{observado}$) observado por $Comp_{observador}$; e um segundo componente ($Comp_{opt}$), cuja execução vai alterar o estado de $Elem_{observado}$.

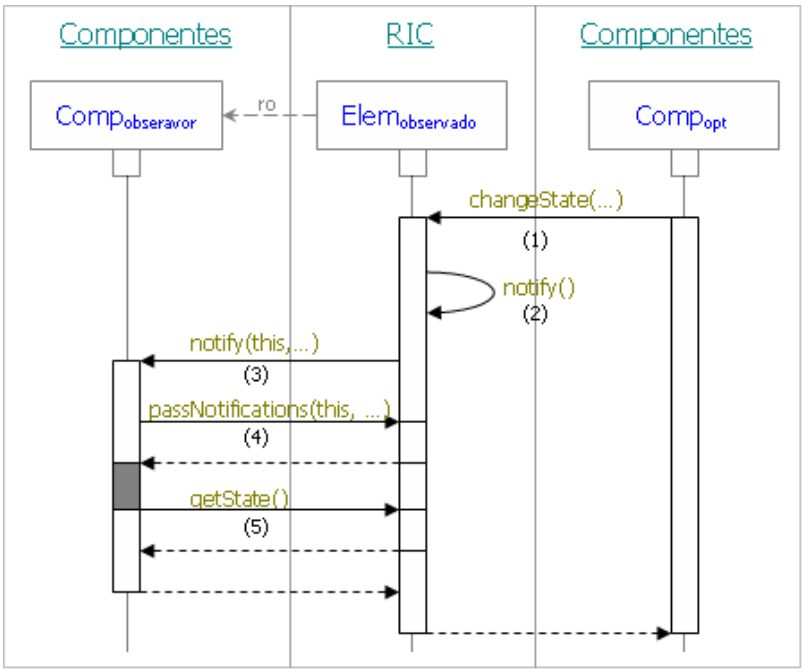


Figura 7.5: Diagrama sequencial da notificação dos componentes.

É na altura desta alteração do estado de $Elem_{observado}$, que se dá início ao processo de notificação. A alteração está representada através de um método fictício (*changeState(...)*), que representa um qualquer método de $Elem_{observado}$, capaz de alterar o estado desse elemento. Quando $Elem_{observado}$ recebe uma mensagem deste tipo, pode, se assim tiver sido requerido por pelo menos um componente, capturar o estado do elemento observado ($Elem_{observado}$), conforme explicado na Secção 6.3. Só posteriormente é que executa as operações que normalmente teria que efectuar. Depois, e conforme explicado na Secção 6.2.3, $Elem_{observado}$ pode construir um relatório com informação do método invocado (representado no exemplo por *changeState(...)*), desde que tal tenha sido solicitado por pelo menos um dos componentes observadores (ver Secção 6.2). Nesse relatório, construído a partir da classe *Report*, vai a identificação da classe e do método, os valores dos parâmetros e o valor de retorno (se estes existirem). Independentemente de ter sido ou não requerido o relatório, no final $Elem_{observado}$ invoca sempre *bool notify()* (2), que vai tratar de notificar todos os observadores registados em $Elem_{observado}$.

No entanto, o tipo de mensagem utilizada para notificar os componentes, depende se foi ou não requerido relatório. Se foi, é utilizado o método *bool notify(Observed*,Report*)* senão, é utilizado o método *bool notify(Observed*)*. Em ambos casos vai a identificação de $Elem_{observado}$ (*Observed**). Em alternativa, o componente observador pode requerer o relatório através dos métodos *Report* getReport()* ou *long long getReport(Observer*, Dict<Observed*, Report*>*)*.

A Figura 6.19 ajuda a compreender o tipo de procedimentos que são executados nos métodos que alteram o estado dos elementos observados (que no presente exemplo corresponde ao método *changeState()*).

Os métodos *bool notify(...)* devem ser sempre redefinidos pelo componente observador, quanto mais não seja para colocar o seu estado em *OUTDATED*. É no entanto possível reescrever estes métodos no sentido de efectuar uma actualização imediata do estado, o que pode ser feito recomputando integralmente o componente, ou efectuando apenas as operações necessárias para compensar as alterações ocorridas no elemento observado ($Elem_{Observado}$).

Caso o componente opte pela actualização imediata do seu estado e se estiver dependente de componentes de suporte, então deverá verificar se estes já foram notificados. Isto pode ser feito através do método *passNotify(this,*$Comp_{sup}$) que, conforme se explicou na Secção 6.2.4, permite ao componente principal (neste caso $Comp_{observador}$) ceder a vez na ordem de notificação. Se tal acontecer, o componente passa a execução para o elemento observado e fica a aguardar nova notificação. Nessa altura, ou caso não tenha sido necessário ceder a vez, o componente poderá então actualizar o seu estado. Para tal, poderá recorrer à informação disponibilizada por *Report* e/ou pela captura do estado do elemento observado ($Elem_{Observado}$), para efectuar apenas as operações necessárias a compensar as alterações ocorridas em $Elem_{Observado}$, minimizando assim o número de operações a efectuar e como tal o tempo de compilação.

Convém acrescentar que a Figura 7.5 apenas representa as principais operações efectuadas pelas entidades que compõem a arquitectura. Muitas outras operações há, que por não serem tão relevantes, não estão assinaladas.

Para concluir, vale a pena realçar que todo este processo de notificação e actualização de componentes é integralmente transparente para os utilizadores. Mesmo para quem implementa os componentes, pode optar simplesmente por integrar a interface *Observer* e redefinir os métodos *bool notify(...)* (ver Figura 8.14), o que bastará para que o componente esteja em conformidade com a arquitectura. No entanto recomenda-se vivamente tirar proveito das funcionalidades disponibilizadas para a recomputação optimizada dos componentes, que permitem reduzir significativamente o tempo de compilação. O Capítulo 8 inclui um exemplo que ilustra como é que a recomputação optimizada dos componentes é efectuada.

7.3 Adaptação da arquitectura à *framework Dolphin*

Como já foi referido por diversas vezes e como é possível confirmar ao longo desta secção, a arquitectura proposta nesta dissertação é para todos os efeitos independente da *framework Dolphin* (versão original). Isto foi intencionalmente feito para separar claramente a arquitectura, do que é uma concretização desta, como é o caso da versão definitiva da *framework Dolphin*. Serve ainda para mostrar que se trata de um modelo genérico, que pode ser aplicado a outros sistemas, como o SUIF Compiler System ou o RTL System, mas também em sistemas similares que nada tenham a ver com a construção de compiladores. No entanto, isto levanta uma questão perfeitamente legítima, que é saber como é que na prática a arquitectura foi aplicada à *framework Dolphin*.

Na perspectiva dos componentes a questão é simples, existem duas interfaces, *Component* e *Observer*, nenhuma é de implementação obrigatória. Os componentes podem funcionar perfeitamente sem implementar qualquer destas interfaces, apenas não será possível usufruir dos benefícios propostos pela arquitectura, nomeadamente tornar a utilização dos componentes simples e simultaneamente eficiente.

No que diz respeito à *framework Dolphin* (versão definitiva), e considerando que esta serviu de caso de estudo para analisar e testar a viabilidade da arquitectura, é lógico que

todos os componentes disponibilizados implementam ambas interfaces. Convém no entanto realçar que, por falta de tempo, nem todos componentes foram ainda adaptados para tirar o máximo proveito das funcionalidades disponibilizadas pela arquitectura, nomeadamente dos mecanismos de recomputação optimizada apresentados no Capítulo 6.

Já a adaptação da arquitectura aos elementos da RIC, ou melhor dizendo às classes da DIR, é um problema interno da *framework Dolphin*, que diz essencialmente respeito a quem implementa a DIR e que não afecta directamente os utilizadores. No entanto, não é um problema simples. É que a DIR deve disponibilizar a quem utiliza ou desenvolve componentes, todas as funcionalidades que a arquitectura propõe para a RIC. Isto porque, um dos grandes objectivos da DIR é facilitar o desenvolvimento dos componentes; mas também, porque não faz sentido adaptar a arquitectura à *framework Dolphin*, se depois não for possível tirar o máximo proveito da arquitectura por restrições da RIC.

A primeira questão que se coloca é saber até que ponto é que as classes da DIR devem disponibilizar todas as funcionalidades existentes na arquitectura, especificamente concebidas para os elementos da RIC. Apenas para relembrar, que dotar a RIC das funcionalidades disponibilizadas pela arquitectura passa pela adaptação das interfaces: *compManager*, *regObserver* e *Observed*. Para tal, as classes da DIR devem herdar estas interfaces e sempre que necessário completá-las, isto é, implementar e redefinir os métodos virtuais.

Acontece no entanto que, as funcionalidades disponibilizadas pela arquitectura requerem recursos. Por vezes, de uma ordem de grandeza superior à que seria necessária para implementar as classes sem essas funcionalidades. Por exemplo, a interface *compManager* contém, não só uma estrutura de dados, como também todos os métodos necessários para efectuar a sua gestão (ver Secção 5.2.1). Fará sentido que para determinadas classes muito simples, como é o caso do *Expression* e classes derivadas, se tenha de implementar sempre a interface *compManager*?

Isto levanta outras questões igualmente legítimas: Será possível implementar as funcionalidades da arquitectura apenas em algumas classes? E se tal for possível, então que funcionalidades devem ser implementadas em cada classe? E certamente a mais relevante das questões: Como fazê-lo sem envolver os utilizadores ou pelo menos sem sobrecarregá-los, de forma a não colocar em risco todo o trabalho até aqui desenvolvido?

A resposta a estas e outras questões, são dadas nas próximas secções onde se mostra em que é que consiste a implementação das várias interfaces às classes da DIR. Mais adiante são focados os entraves ultrapassados para se obter uma solução generalizada. Após o que se apresenta a solução desenvolvida.

7.3.1 Implementação individual das interfaces

A Figura 7.6 ilustra a adaptação da interface *compManager* à classe *CFG* da DIR. Para tal, a classe resultante herda a classe base de *CFG (_B_CFG)* e a interface *compManager*. Há apenas que acrescentar o construtor, o destrutor e os operadores de cópia e de atribuição.

A Figura 7.7 ilustra o caso da adaptação da interface *regObserver* à classe *CFG*. À semelhança do caso anterior, a classe resultante herda a classe base de *CFG* (*_B_CFG*) e a interface *regObserver*. Depois, para além de definir o construtor, o destrutor e os operadores de cópia e de atribuição, poderá ainda ser necessário redefinir os métodos de *regObserver*, que fazem o reencaminhamento das mensagens (*regChain(...)*, ..., *getRSChain(...)*). Isto porque, por omissão, *regObserver* não reencaminha as mensagens, até porque não tem como saber para onde é que deveria enviar as mensagens, dado que essa é uma informação que depende de cada classe que implementa a interface (as mensagens são normalmente reencaminhadas pelos objectos que integram a classe). Assim, sempre que haja necessidade de reencaminhar as mensagens, é fundamental redefinir estes métodos em conformidade com a classe em causa.

```
(1)   // ////////////////////////////////////////////////////
(2)   // _C_CFG é a classe derivada de _B_CFG
(3)   // que implementa a interface compManager
(4)   // ////////////////////////////////////////////////////
(5)   #include "compManager.h"
(6)   class _C_CFG : public _B_CFG, public compManager{
(7)     public:
(8)       // Construtor + Destrutor
(9)       _C_CFG();
(10)      _C_CFG(_C_CFG*);
(11)      virtual ~_C_CFG();
(12)      // Operadores do objecto
(13)      _C_CFG &operator=(_C_CFG&);
(14)      ...
(15)  };
```

Figura 7.6: Adaptação da interface *compManager* à classe *CFG* da DIR.

A adaptação da interface *Observed*, talvez a mais complexa de todas, encontra-se ilustrada na Figura 7.8, onde é possível constatar que foi necessário: isolar todos os métodos que davam acesso directo às estruturas de dados contidas em *CFG*, como acontece na linha 9; e redefinir todos os métodos que alteram o estado interno de *CFG*, como acontece na linha 19. Ambos procedimentos são necessários para controlar as alterações que *CGF* possa sofrer e assim despoletar os mecanismos de notificação.

Um exemplo típico contendo a redefinição de um método, encontra-se representado na Figura 6.19. É ainda de chamar atenção para a necessidade de registar o identificador da classe, como acontece na linha 13 e 14 da Figura 7.8.

7.3.2 Problemas de adaptação

Após os três exemplos anteriores, mais do que nunca é legítimo perguntar: que interfaces deve *CFG* implementar? *compManager*? *regObserver*? *Observed*? Todas? Nenhuma? Ou combinações de algumas das interfaces? Há pelo menos duas soluções possíveis. A primeira passa por implementar as três interfaces simultaneamente (o que na realidade consiste em implementar *compManager* e *Observed*, dado que esta última deriva de *regObserver*), ficando assim a classe *CFG* capaz de disponibilizar todas as funcionalidades da arquitectura que se destinam aos elementos da RIC. A desvantagem desta abordagem está no facto de sobrecarregar a RIC, com métodos e estruturas de dados que podem não ter qualquer utilidade. A segunda solução passa por utilizar apenas as interfaces requeridas pelos componentes. É claro que isto nos leva novamente à questão inicial: que interfaces deve *CFG* implementar? E mais genericamente, que interfaces deve cada classe da DIR implementar?

Para complicar um pouco mais a questão, há ainda que ter em conta os seguintes problemas:

1. Em muitos casos, cada uma das classes da DIR vai necessitar de implementar mais do que uma interface;

```
(1)   // ////////////////////////////////////////////////////
(2)   // _R_CFG é a classe derivada de _B_CFG
(3)   // que implementa a interface regObserver
(4)   // ////////////////////////////////////////////////////
(5)   #include "regObserver.h"
(6)   class _R_CFG :public _B_CFG, virtual public regObserver{
(7)     protected:
(8)       // Implementação das interfaces
(9)       long long regChain(Observer*,long long,short);
(10)      long long remChain(Observer*,long long);
(11)      long long hasChain(Observer*);
(12)      long long getChain(Observer*,List<Observed*>*);
(13)      long long getRChain(Observer*,DictT<Observed*,Report*>*);
(14)      bool certifyChain(Observer*,long long);
(15)      long long getRSChain(Observer*,Dict<Observed*,short>*);
(16)    public:
(17)      // Construtor+Destrutor
(18)      _R_CFG();
(19)      _R_CFG(_R_CFG);
(20)      virtual ~_R_CFG();
(21)      // Operadores do objecto
(22)      _R_CFG &operator=(_R_CFG&);
(23)      ...
(24)  };
```

Figura 7.7: Adaptação da interface *regObserver* à classe *CFG* da DIR.

2. Um determinado componente pode requerer um determinado conjunto de funcionalidades da arquitectura, que só e simplesmente não são necessários ou suficientes para os outros componentes em uso;

3. Implementar uma dada interface numa classe da DIR, poderá requerer que as classes relacionadas tenham também elas que implementar algumas das interfaces. Por exemplo, a implementação da interface *regObserver* numa classe A da DIR, só tem razão de ser se houver uma classe B, de alguma forma relacionada com A, que implemente a interface *regObserver* ou *Observed*;

4. No sentido de conjugar determinadas características que são comuns e reaproveitar determinadas funcionalidades, muitas das classes da DIR são implementadas por derivação de outras classes da própria DIR. É legítimo pensar que poderão ser requeridas a uma classe A, determinadas funcionalidades da arquitectura, que não o são à classe da qual A deriva ou às classes que derivam de A.

Os problemas expostos servem para demonstrar o quanto esta questão da adaptação das interfaces às classes da DIR é complexa, mas também para legitimar de uma vez por todas que a arquitectura é completamente independente da *framework Dolphin*.

O primeiro problema, dos que foram atrás enumerados, é talvez o mais simples de

```
(1)   // ////////////////////////////////////////////////////
(2)   // _O_CFG é a classe derivada de _R_CFG
(3)   // que implementa a interface Observed
(4)   // ////////////////////////////////////////////////////
(5)   #include "Observed.h"
(6)   class _O_CFG :public _R_CFG, public Observed{
(7)     protected:
(8)       // Métodos que devem ser isolados
(9)       Set<FlowNode*>& getINodes();
(10)      ...
(11)    public:
(12)      // Construtor + Destrutor
(13)      _O_CFG(){setMSG(WCFG);...}
(14)      _O_CFG(_O_CFG*){setMSG(WCFG);...}
(15)      virtual ~_O_CFG();
(16)      // Operadores do objecto
(17)      _O_CFG &operator=(_O_CFG&);
(18)      // Métodos controlados
(19)      bool insINode(FlowNode*);
(20)      ...
(21)  };
```

Figura 7.8: Adaptação da interface *Observed* à classe *CFG* da DIR.

resolver, dado que a implementação de mais do que uma interface numa mesma classe, não levanta qualquer tipo de obstáculo.

O segundo problema, que levanta a questão dos componentes terem exigências distintas, é fácil de resolver se atendermos que não deverá ser por limitações da RIC (que neste caso resulta na DIR), que os componentes não deverão ser implementados como seria desejado. Se considerarmos ainda que, segundo o modelo de compilação utilizado, a RIC é a coluna vertebral do processo de compilação, o que significa que os elementos da RIC utilizados numa determinada tarefa são a grosso modo os mesmos das restantes tarefas, então para se satisfazer as exigências de todos os componentes, a escolha das interfaces terá que ser feita por excesso. Isto é, se um determinado componente requerer a uma determinada classe todas as funcionalidades da arquitectura, então tal exigência deverá ser atendida independentemente dos restantes componentes terem ou não requerido essas funcionalidades.

O terceiro problema enumerado anteriormente, que coloca a possibilidade de que a adaptação de uma interface a uma classe, possa exigir que classes relacionadas tenham também elas que implementar determinadas interfaces, é resolvido da mesma forma que o problema anterior. Por exemplo, se a classe A implementa a interface *regObserver* e normalmente reencaminha as mensagens através da classe B, então esta última deverá no mínimo implementar a interface *regObserver*, mesmo que nenhum componente o tenha requerido.

O quarto e último problema, que levanta a possibilidade de haver requisitos distintos entre classes com relações de herança, é o mais complexo de resolver e o que nos leva ao cerne da questão, que é: como implementar as diversas variantes das classes da DIR? Como mais adiante é explicado, isto leva-nos ainda a um quinto problema que advém da compatibilização

entre classes distintas (que não estejam relacionadas por qualquer tipo de herança).

7.3.3 Modelo de heranças

A concepção da solução que permite adaptar a arquitectura à DIR, foi feita considerando os problemas anteriormente colocados, mas também que este é um problema que deve ficar tanto quanto possível restrito à DIR, evitando assim complicações a quem vai utilizar a *framework Dolphin*. Isto para não colocar em causa todo o trabalho até aqui desenvolvido. É como tal fundamental, que a implementação das interfaces seja parte integrante da *framework Dolphin*, isto é, não deverá caber ao utilizador alterar cada uma das classes da DIR, para poder usufruir das mais valias da arquitectura.

Não restou assim outra alternativa que não fosse fornecer variantes de cada uma das classes da DIR, com as várias combinações possíveis para a utilização das interfaces (como mais adiante é explicado não há necessidade de fornecer variantes de todas as classes).

Isto significa que para cada classe base da DIR vão surgir agora seis novas classes (designadas por *variantes*). Para as distinguir optou-se por associar a cada uma dessas classes um prefixo, que obedece à seguinte convenção:

_B_ Classe base;

_R_ Classe base com a implementação da interface *regObserver*;

_O_ Classe base com a implementação da interface *Observed* (e implicitamente da interface *regObserver*;

_C_ Classe base com a implementação da interface *compManager*;

_CR_ Classe base com a implementação das interfaces *compManager* e *regObserver*;

_CO_ Classe base com a implementação das interfaces *compManager* e *Observed*.

Houve ainda a necessidade de estabelecer um modelo que permitisse relacionar as variantes entre si. Isto porque se entendeu que poderia ser útil facultar aos utilizadores uma solução que lhes permitisse, em situações extremas, passar de uma variante para outra ou compatibilizar variantes. Neste sentido estabeleceu-se o modelo de heranças que se encontra representado na Figura 7.9.

Quando isolado, o modelo de heranças proposto é perfeitamente funcional e relativamente simples de implementar. Basta realizar os procedimentos anteriormente descritos para a implementação individual das várias interfaces. Que são acumulativos, isto é, se uma variante implementa mais do que uma interface, então tem que realizar os procedimentos inerentes a cada uma.

No entanto a adaptação do modelo de heranças à DIR exige alguns cuidados e está longe de ser um problema simples. Isto pelo facto de as classes da DIR não serem entidades isoladas, isto é, muitas das classes da DIR têm relações de herança entre si.

Conceptualmente, a solução ideal é adaptar o modelo de heranças a todas classes da DIR, sejam elas abstractas ou classes concretas. Relacionando as variantes das classes abstractas com as variantes das classes concretas, conforme ilustra a Figura 7.10. O que é fundamental para manter a herança nativa existente entre as classes, pois é através desta que é possível utilizar uma classe abstracta para representar objectos das suas classes derivadas. Como se pode confirmar pelo exemplo da Figura 7.10, para manter a herança nativa não basta associar as classes base (*_B_DT* com *_B_RAssign*). Se esta for a única relação de herança existente entre as variantes das duas classes, não será possível utilizar qualquer variante de

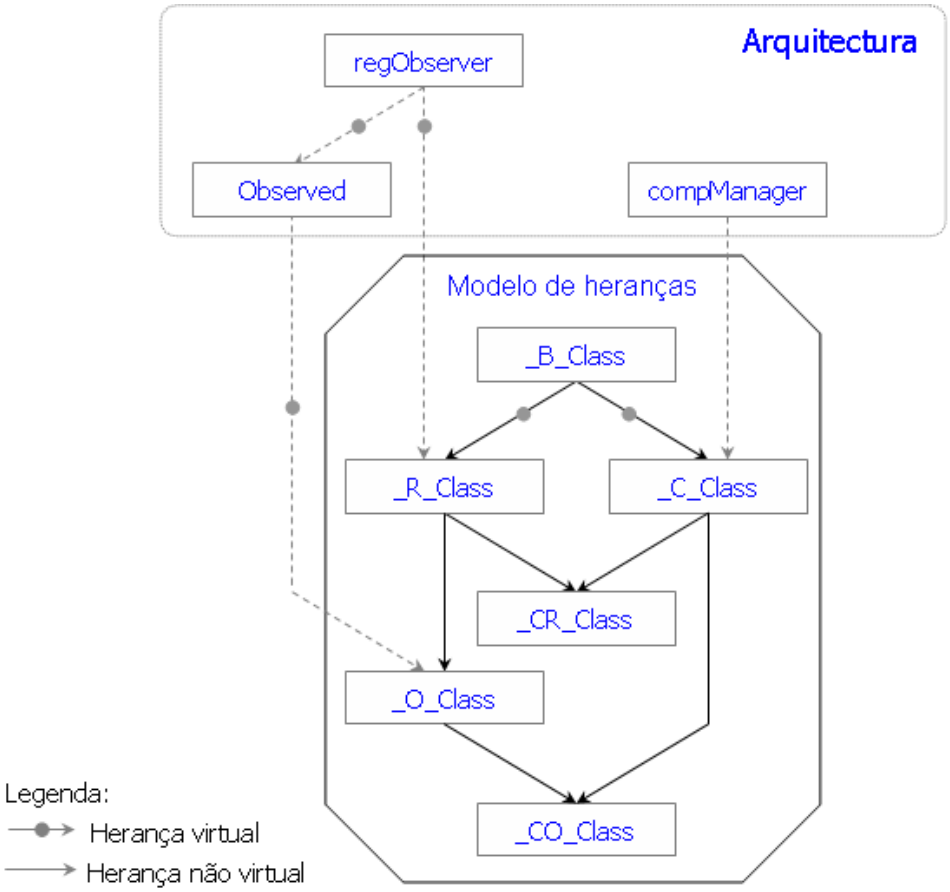


Figura 7.9: Modelo de heranças para as variantes de uma classe.

DT diferente de *_B_DT*, para representar objectos do tipo *RAssign*, independentemente do tipo de variante utilizada para esta última classe.

Para além do mais, estabelecer a relação de herança entre as variantes da classe abstracta e da classe concreta, permite que as variantes desta última aproveitem o código já implementado pelas variantes da classe abstracta. Por exemplo, *_O_RAssign* apenas necessita de reescrever os métodos de *_B_RAssign*, mas não os métodos de *_B_DT*, dado que estes já teriam sido reescritos por *_O_DT*. No entanto, isto cria problemas de ambiguidade, por exemplo, *_O_RAssign* vai ver os métodos de *DT* através de *_O_DT* e de *_R_RAssign*. Em alguns casos, a ambiguidade é resolvida pelos próprios compiladores, mas casos há em que o problema tem que ser resolvido reescrevendo os métodos nas variantes.

Para além disso, e como se pode ver pela Figura 7.11, que ilustra a aplicação desta solução às classes *DT*, *RAssign* e *CAssign*, o resultado é no mínimo assustador. A quantidade de heranças múltiplas é aberrante e mais grave do que isso, força a inserção de mais heranças virtuais[1].

Não obstante a estes problemas, foram realizados vários testes com esta solução. Se isoladamente era exequível e permitia compatibilizar grande maioria das variantes, quando adaptada à DIR, mostrou-se bastante instável, ou porque perdia a referência interna das

[1]Em C++ a utilização de herança virtual, que surge associada à utilização de herança múltipla, aplica-se quando uma classe *A* herda por diferentes vias uma classe *B* e se pretende evitar que a estrutura dos objectos do tipo *A* contenha várias instâncias da classe *B*.

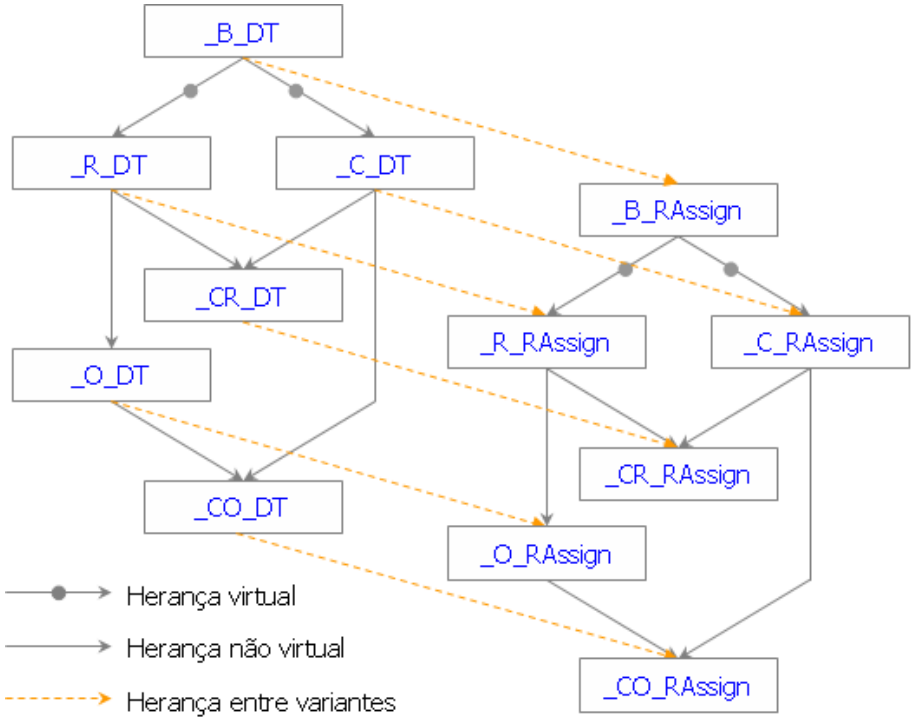


Figura 7.10: Exemplo de aplicação generalizada do modelo de heranças.

instâncias que provinham das heranças virtuais; ou porque os compiladores não conseguiam processar o código.

Optou-se assim por desenvolver uma nova solução, que mantivesse as vantagens da actual, mas que fosse mais simples, nomeadamente que fizesse uso de menos heranças múltiplas, mas principalmente de menos heranças virtuais. O resultado foi o que se designou por *modelo de heranças flutuante*, ou simplesmente por *modelo flutuante*, que foi desenvolvido considerando que o utilizador da *framework Dolphin*, fará uso de uma única variante por cada classe da DIR. O conjunto das variantes utilizadas será escolhido conforme os requisitos dos componentes e segundo as condicionantes descritas anteriormente (ver Secção 7.3.2). Mesmo que as variantes a utilizar para cada tipo de classe da DIR não sejam bem escolhidos, o modelo flutuante é bastante flexível para permitir que, na maioria dos casos, seja possível passar de uma variante para outra.

O modelo de heranças flutuante assenta no princípio de que não há necessidade de estabelecer uma relação de herança entre todas as variantes da classe abstracta e da classe concreta. Basta estabelecer a relação de herança entre as variantes em uso. Por exemplo, no caso da Figura 7.10 haverá uma única ligação entre as variantes de *DT* e de *RAssign*, ligação essa que vai depender das variantes escolhidas. No entanto, o segredo do modelo de heranças flutuante está na forma como se estabelece esta ligação. A solução, que é muito simples, consiste em dar uma designação genérica (normalmente o designação da própria classe) à variante em uso de cada uma das classes da DIR, como ilustra o exemplo da Figura 7.12.

Esta definição, que deve ser feita pelos utilizadores da *framework Dolphin*[2], permite que o vínculo entre a classe abstracta e as classes concretas (ou entre duas classes concretas que

[2]A *framework Dolphin* contém vários ficheiros com definições deste tipo, que fazem uso de diferentes combinações de variantes. Esses ficheiros podem ser utilizadas directamente ou modificados conforme as necessidades de cada utilizador.

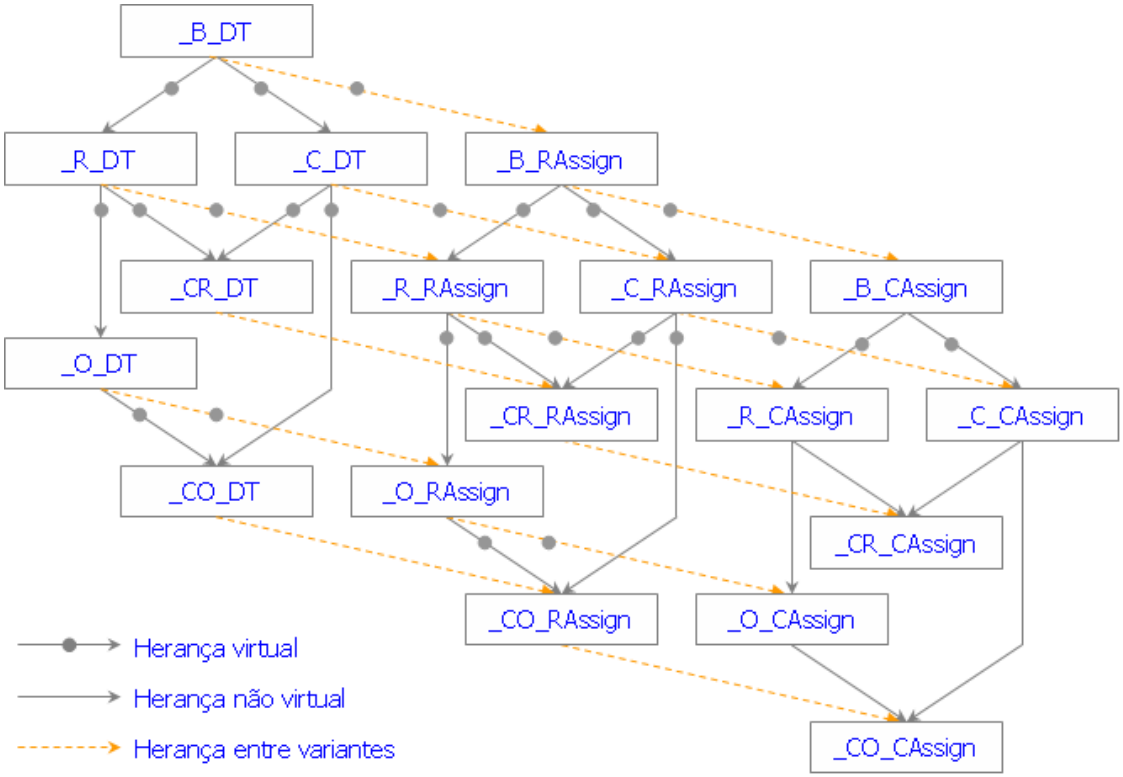


Figura 7.11: Adaptação generalizada do modelo de heranças às classes *DT*, *RAssign* e *CAssign*.

```
(1)     class _C_DT;
(2)     typedef _C_DT DT;
```

Figura 7.12: Definição de tipos genéricos para as classes da DIR.

possuam uma relação de herança entre si), seja efectuado aquando da compilação. Para tal, apenas houve que fazer com que a variante base das classes concretas herdasse a designação genérica da classe abstracta da qual deriva. Como ilustra o exemplo da Figura 7.13.

A Figura 7.14 representa graficamente a relação que se estabelece entre as variantes da classe *DT* e *RAssign*, considerando que é utilizada a variante *_C_DT*. De notar que esta variante pode ser utilizada para representar um qualquer objecto do tipo *RAssign* ou qualquer outro objecto de classes derivadas de *DT*. Mantém-se assim uma das principais vantagens do modelo inicial.

A implementação das variantes obedece a uma regra muito simples, que se resume ao seguinte: *Cada variante deve apenas cuidar dos aspectos inerentes à implementação das respectivas interfaces e restringir o seu espaço de intervenção aos métodos e variáveis da respectiva classe base.*

Convém também acrescentar que, em termos de compilação, é aceite qualquer tipo de combinação entre as variantes de uma classe abstracta e uma classe concreta. Por exemplo, é possível utilizar a variante *_CO_* para a classe abstracta e utilizar a variante *_B_* para a

```
(1)     class _B_RAssign:  public DT{
(2)        ...
(3)     };
```

Figura 7.13: Implementação do modelo de heranças flutuante, entre a classe *DT* e *RAssign*.

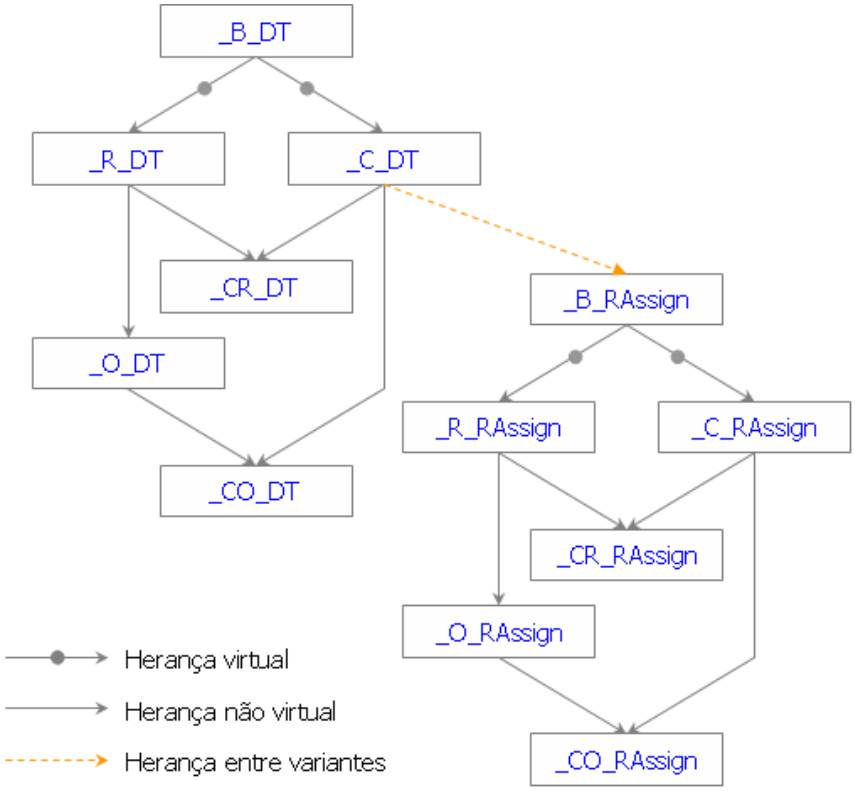


Figura 7.14: Relacionamento das variantes utilizando o modelo flutuante.

classe concreta, apesar disto não fazer muito sentido. Como é evidente, cabe ao utilizador ter o bom senso de escolher as variantes mais apropriadas a cada classe ou, como já se disse, aproveitar um dos ficheiros disponibilizados pela *framework Dolphin*.

O modelo flutuante permite também reaproveitar o código já implementado nas variantes das classes superiores (abstractas e/ou concretas), desde que salvaguardadas algumas condições.

A primeira é que só se pode reaproveitar o que existe. Significa isto que se a variante utilizada numa classe superior não implementar uma determinada interface, então as classes derivadas não vão poder reaproveitar qualquer tipo de código que seja inerente a essa interface. Por exemplo, se a variante utilizada na classe abstracta for a *_R_*, e se para a classe derivada for utilizada a variante *_O_*, então apenas os métodos da classe derivada é que vão estar controlados. Qualquer invocação de um método da classe abstracta, mesmo quando feita através da classe derivada, não vai produzir qualquer tipo de notificação, dado que os métodos desta classe não estão controlados, porque a variante apenas implementa a interface *regObserver*. Para que os métodos estivessem controlados, a classe abstracta deveria

implementar a interface *Observed*, ou seja, deveria ser utilizada a variante *_O_*.

A segunda condição aplica-se exclusivamente à construção da variante *_R_* das classes derivadas (concretas). Esta variante faz o reencaminhamento das mensagens redefinindo alguns métodos de *regObserver*. Segundo o princípio estabelecido para o desenvolvimento das variantes, o reencaminhamento deverá apenas ser feito para os elementos pertencentes à classe em causa. Significa isto que não haverá reencaminhamento para os elementos que integram a classe ascendente.

A Figura 7.15 ilustra esta situação, em que *CSNode* é uma classe que deriva de *JNode*, que por sua vez deriva de *FlowNode*. Na figura estão representados os métodos *regChain(...)* para estas três classes[3]. Uma mensagem do tipo *regObs(...)* submetida a *CSNode* (variante *_R_*), vai fazer uso de *regChain(...)*. Acontece no entanto que o *regChain(...)* que é utilizado, é o do *CSNode*, que apenas reencaminha as mensagens para os objectos incluídos directamente nesta classe, ficando de fora os objectos herdados, como por exemplo, os que são definidos na classe *FlowNode*.

```
(1)     long long _R_FlowNode::regChain(Observer *c,long long m,int sr){
(2)       long long mm=0;
(3)       regObserver *ro=dynamic_cast<regObserver*>(___lnode);
(4)       if(ro) mm |=ro->regObs(c,m,sr);
(5)       ro=dynamic_cast<regObserver*>(___block);
(6)       if(ro) mm |=ro->regObs(c,m,sr);
(7)       return mm;
(8)     }
(9)
(10)    long long _R_JNode::regChain(Observer *c,long long m,int sr){
(11)      return 0;
(12)    }
(13)
(14)    long long _R_CSNode::regChain(Observer *c,long long m,int sr){
(15)      long long mm=0;
(16)      regObserver *ro=dynamic_cast<regObserver*>(___funct);
(17)      if(ro) mm |=ro->regObs(c,m,sr);
(18)      return mm;
(19)    }
(20)
(21)    CSNode n;
(22)    n.regObs(...);
```

Figura 7.15: Problemas derivados do reencaminhamento de mensagens.

Para evitar que tal aconteça, a variante *_R_* de cada classe que contenha ascendentes (classes abstractas ou outras classes concretas), deve reencaminhar as mensagens para os elementos pertencentes a essa classe, mas deve também requerer o reencaminhamento das mensagens às classes ascendentes. Um exemplo da utilização desta solução, encontra-se representado na Figura 7.16. De notar que antes de se efectuar o pedido de reencaminhamento

[3]De realçar que *JNode* na realidade não redefine este método (é herdado directamente da interface regObserver).

à classe ascendente, é necessário certificar que essa classe implementa a interface *regObserver*, como se encontra ilustrado nas linhas 12 e 13, e 21 e 22 da Figura 7.16.

```
(1)   long long _R_FlowNode::regChain(Observer *c,long long m,int sr){
(2)     long long mm=0;
(3)     regObserver *ro=dynamic_cast<regObserver*>(___lnode);
(4)     if(ro) mm |=ro->regObs(c,m,sr);
(5)     ro=dynamic_cast<regObserver*>(___block);
(6)     if(ro) mm |=ro->regObs(c,m,sr);
(7)     return mm;
(8)   }
(9)
(10)  long long _R_JNode::regChain(Observer *c,long long m,int sr){
(11)    long long mm=0;
(12)    _R_FlowNode *rf=dynamic_cast<_R_FlowNode*>(this);
(13)    if(rf) mm|=rf->regObs(c,m,sr);
(14)    return mm;
(15)  }
(16)
(17)  long long _R_CSNode::regChain(Observer *c,long long m,int sr){
(18)    long long mm=0;
(19)    regObserver *ro=dynamic_cast<regObserver*>(___funct);
(20)    if(ro) mm |=ro->regObs(c,m,sr);
(21)    _R_JNode *rj=dynamic_cast<_R_JNode*>(this);
(22)    if(rj) mm|=rj->regObs(c,m,sr);
(23)    return mm;
(24)  }
(25)
(26)  CSNode n;
(27)  n.regObs(...);
```

Figura 7.16: Resolução do problema ilustrado pela Figura 7.15.

Utilização do modelo de heranças

Uma vez estabelecida a forma de utilizar as interfaces da arquitectura, que resultou na definição de um conjunto de variantes e num modelo de heranças para essas variantes, falta agora explicar como é que esta solução é utilizada, quer por quem desenvolve componentes, quer pelas próprias classes da DIR.

De notar que a utilização desta solução, pode criar algumas dúvidas sobre como é que as variáveis das classes da DIR devem ser declaradas. A Figura 7.17 corresponde à representação parcial da classe CFG antes da adaptação da arquitectura. *__iNodes* e *__funct* são duas variáveis, uma do tipo *Set<FlowNode*>* e outra do tipo *Function*. A questão que se coloca é saber com que variantes estas variáveis devem ser definidas na adaptação da arquitectura à DIR.

Uma solução possível consiste em substituir *FlowNode* e *Function*, que no exemplo da

```
(1)     class CFG : public DObject, ...{
(2)       protected:
(3)         // Variáveis do objecto
(4)         Set<FlowNode*> ___iNodes;
(5)         Function *___funct;
(6)         ...
(7)     };
```

Figura 7.17: Representação parcial da classe *CFG*.

Figura 7.17 representam as classes originais da DIR (antes da adaptação da arquitectura), pelas variantes base, isto é, por *_B_FlowNode* e *_B_Function*, que são em tudo semelhantes às versões originais. Esta solução apresenta no entanto algumas desvantagens. Por exemplo, se *___funct* for do tipo *_R_Function* poderá ocorrer que *CFG* queira reencaminhar as mensagens através deste objecto (apontador). No entanto, como *___funct* é declarado como sendo do tipo base (*_B_Function*), então para que seja possível aceder aos métodos de reencaminhamento, será necessário fazer um *cast* de *___funct* (com todas as desvantagens que daí resultam).

A solução mais simples, funcional e que não requer qualquer tipo de *cast*, consiste em não alterar absolutamente nada. Desde que *FlowNode* e *Function* sejam designações genéricas das correspondentes classes, isto é, o tipo associado à variante em uso para cada um dos casos. Basta para tal, definir *FlowNode* e *Function*, como se fez para *DT* na Figura 7.13.

Desta forma, quer os utilizadores, quer as próprias classes da DIR, podem utilizar *Function* para definir objectos deste tipo, o que vai corresponder à variante em uso. Assim, a definição da classe base de *CFG* (e restantes classes da DIR) em nada se altera, isto é, *___funct* continua a ser definido com *Function*. O mesmo acontece caso o utilizador pretenda declarar uma variável desta classe.

As definições dos identificadores genéricos das várias classes da DIR, encontram-se no ficheiro *~/Framework/DIR/Include/Definitions.h*, o qual deverá ser modificado, em conformidade com as exigências colocadas pelos componentes em uso. Para além deste ficheiro, existem outros com modelos preestabelecidos para a utilização das variantes.

Definição dos identificadores para os elementos observados

Convém ainda acrescentar que a definição dos identificadores dos objectos observados (elementos que implementam a interface *Observed*), faz parte do processo de adaptação da arquitectura à *framework Dolphin*, e mais concretamente à DIR. Os identificadores definidos para as principais classes da DIR encontram-se representados na Figura 7.18. Para a maioria das situações, basta definir identificadores para as classes abstractas e classes que sejam únicas (não derivem, nem tenham derivados). Os identificadores, como já se disse, são utilizados pelas classes da DIR que implementam a interface *Observed*, para identificar os elementos da RIC. Servem também para identificar os destinatários das mensagens submetidas pelos componentes que implementam a interface *Observer*, conforme explicado na Secção 5.2.3. A definição destes identificadores encontra-se no ficheiro *~/Framework/Include/Messages.h*.

```
(1)   #define NIDENTIFIERS   33    // Número de identificadores definidos
(2)   #define WNULL          0     // Identificador nulo
(3)   #define WDIR           1     // Identificador da classe DIR
(4)   #define WIDTABLE       2     // Identificador da classe IdentTable
(5)   #define WCFG           4     // Identificador da classe CFG
(6)   #define WFUNCT         8     // Identificador da classe Function
(7)   #define WFUNCTS        16    // Identificador da classe Functions
(8)   #define WFLOWNODE      32    // Identificador das classes FlowNode
(9)   #define WPROG          64    // Identificador da classe Program
(10)  #define WLDT           128   // Identificador da classe LDT
(11)  #define WDT            256   // Identificador das classes DT
(12)  #define WEXPR          512   // Identificador das classes Expression
(13)  #define WCELLTABLE     1024  // Identificador da classe CellTable
(14)  // ...  Incluir aqui novos identificadores
```

Figura 7.18: Identificadores definidos para as classes da DIR.

7.4 Resumo do capítulo

Pretendeu-se com este capítulo dar uma perspectiva global da arquitectura, mostrando que esta foi desenvolvida tendo por base um modelo de compilação, que resultou da versão original da *framework Dolphin*, e num conjunto de interfaces que definem o comportamento das entidades inerentes a esse modelo. Serviu também para explicar em que consiste a linha divisória que existe entre a arquitectura e a *framework Dolphin*.

No sentido de se explicar o contributo da arquitectura, descreveram-se os procedimentos efectuados aquando dos três principais eventos: registo, execução e notificação de componentes. Mostrou-se então que em todos estes eventos, cabia à arquitectura a execução da maior parte dos procedimentos, ficando apenas uma parte muito residual a cargo dos utilizadores da *framework*.

Neste capítulo também foi explicado como se adaptou a arquitectura à versão original da *framework Dolphin*, mas em especial à DIR, dando assim origem aquela que é a versão actual deste sistema para desenvolvimento de compiladores. Para um melhor entendimento da adaptação e utilização da arquitectura, recomenda-se a leitura do capítulo 8, onde se exemplifica a implementação e utilização de componentes, utilizando a versão actual da *framework Dolphin* que já implementa a arquitectura.

CAPÍTULO 8

Desenvolvimento de um componente

Índice

Uma vez apresentada a Dolphin Internal Representation (DIR), a arquitectura e a adaptação desta à *framework Dolphin*, é certamente o momento oportuno para provar a utilidade prática destas soluções, nomeadamente quando integradas numa única entidade que é a versão definitiva da *framework Dolphin*. Considerou-se que a forma mais cabal de efectuar esta prova, é descrever este capítulo na perspectiva de um utilizador da *framework Dolphin*, que tenha por objectivo desenvolver integralmente um compilador.

Este capítulo resulta assim num exercício de cariz claramente prático, em que se ilustra o desenvolvimento integral de um componente que tira proveito das funcionalidades dispo-

nibilizadas pela DIR (enquanto modelo de RIC), e pela arquitectura. Numa segunda fase, fazendo uso dos recursos disponibilizados pela *framework Dolphin*, constrói-se integralmente um compilador.

Conclui-se o capítulo apresentando um pequeno estudo elaborado para determinar o impacto da arquitectura no modelo de compilação, nomeadamente com e sem a recomputação optimizada das instâncias dos componentes. Pretende-se assim demonstrar a mais valia que é a utilização da DIR e da arquitectura, mas também da abordagem utilizada para a concepção e desenvolvimento de ferramentas para construção de compiladores e componentes para compiladores.

8.1 Desenvolvimento de um componente

O exemplo escolhido para ilustrar o desenvolvimento de um componente, recaiu sobre *IDominator*. Trata-se de um componente que constrói um dicionário de dados, contendo para cada nodo do Grafo de Fluxo de Controlo (GFC), o correspondente dominador imediato. Este componente foi escolhido pelo facto de que:

- É um componente de análise e como tal, permite aproveitar melhor as funcionalidades disponibilizadas pela arquitectura;
- É um componente de implementação muito acessível e executa uma tarefa relativamente simples de compreender;
- Foi utilizado em vários exemplos do Capítulo 5;
- É implementado reutilizando outros componentes;
- É utilizado para suportar a execução de outros componentes, nomeadamente do *cnv2SSA* (componente responsável pela conversão da RIC da forma normal para a forma SSA);
- É especialmente adequado para fazer uso dos mecanismos disponibilizados pela arquitectura, para optimizar a recomputação do componente (utilização do *Report* e do mecanismo de captura de estado);
- E requer apenas a observação de três elementos da RIC (*CFG*, *Jump* e *CJump*) e o controlo de cinco métodos.

8.1.1 Cálculo dos dominadores imediatos

Antes de se avançar para a implementação, é fundamental perceber como se determinam os dominadores imediatos e, antes disto, é também conveniente explicar o conceito de dominância.

Sejam n e m dois nodos pertencentes ao GFC, diz-se que n domina m (e representa-se por $n\ dom\ m$) sse todos os caminhos de execução possíveis, que vão desde o nodo inicial até m, passarem obrigatoriamente por n. Ao conjunto de todos os nodos dominados por n representa-se por $dom(n)$.

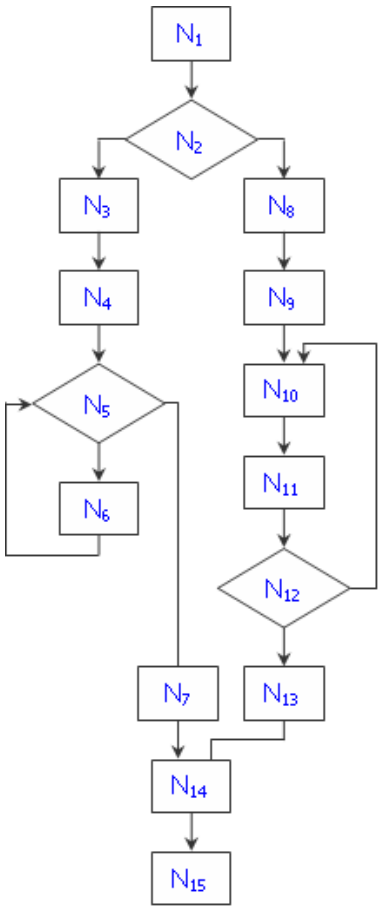
Se $n\ dom\ m$ e $n \neq m$ então diz-se que n domina estritamente m e representa-se por $n\ sdom\ m$ (Eq. 8.1). Ao conjunto de todos os nodos dominados estritamente por n, representa-se por $sdom(n)$.

$$n\ sdom\ m, sse\ n\ sdom\ m \wedge n \neq m \tag{8.1}$$

Para $n \neq m$, se n *sdom* m e se não existir nenhum nodo p diferente de n e de m, tal que n *dom* p e p *dom* m, então n é o dominador imediato de m, o que se representa por n *idom* m. Quando existe, o dominador imediato é único. A definição formal de dominador imediato encontra-se representada na Eq. 8.2, em que N corresponde ao conjunto dos nodos do GFC.

$$n \; idom \; m, sse \; n \neq m \wedge n \; sdom \; m \wedge \nexists \; p \in N \backslash \{n, m\} : n \; dom \; p \wedge p \; dom \; m \tag{8.2}$$

A Figura 8.1(a) representa o GFC obtido para o programa da Figura 8.1(b). Os dominadores imediatos encontram-se representados na Figura 8.1(c).



(a) GFC.

```
(1)   int fx(int a){
(2)       if(a>0)
(3)           while(a) a=a-1;
(4)       else
(5)           do{a=a+1;}while(a);
(6)       return a;
(7)   }
```

(b) Programa fonte.

$$\begin{cases} idom(N_1) = \emptyset \\ idom(N_2) = \{N_1\} \\ idom(N_3) = \{N_2\} \\ idom(N_4) = \{N_3\} \\ idom(N_5) = \{N_4\} \\ idom(N_6) = \{N_5\} \\ idom(N_7) = \{N_5\} \\ idom(N_8) = \{N_2\} \\ idom(N_9) = \{N_8\} \\ idom(N_{10}) = \{N_9\} \\ idom(N_{11}) = \{N_{10}\} \\ idom(N_{12}) = \{N_{11}\} \\ idom(N_{13}) = \{N_{12}\} \\ idom(N_{14}) = \{N_2\} \\ idom(N_{15}) = \{N_{14}\} \end{cases}$$

(c) Dominadores imediatos.

Figura 8.1: Cálculo dos dominadores imediatos de um GFC.

A solução utilizada para computar os dominadores imediatos de um GFC, foi retirada do livro "Advanced Compiler Desgin and Implementation" de Steven Muchnick [Muc97], e encontra-se representada na Figura 8.2. É uma solução relativamente simples, que determina os dominadores imediatos com base nos dominadores. De notar que a própria definição de

dominador imediato faz uso do conceito de dominância estrita, que por sua vez faz uso do conceito de dominância.

```
(1)   PROC IDominator(G:GFC, dom:Dict<FlowNode,Set<FlowNode> >)
(2)   :Dict(FlowNode,FlowNode))
(3)     VAR n, s, t, r :FlowNode
(4)     VAR Tmp :Dict<FlowNode,Set<FlowNode> >
(5)     VAR idom :Dict<FlowNode,FlowNode>
(6)     INÍCIO
(7)       r ← getRNode(G)
(8)       PARA ∀ n ∈ G FAZER
(9)         Tmp(n) ← dom(n) - {n}
(10)      FPARA
(11)      PARA ∀ n ∈ G - {r} FAZER
(12)        PARA ∀ s ∈ Tmp(n) FAZER
(13)          PARA ∀ t ∈ Tmp(n)-{s} FAZER
(14)            SE t ∈ Tmp(s) ENTÃO
(15)              Tmp(n)-={t}
(16)            FSE
(17)          FPARA
(18)        FPARA
(19)      FPARA
(20)      PARA ∀ n ∈ G - R FAZER
(21)        idom(n) ← Tmp(n).getFirst()
(22)      FPARA
(23)      RET idom
(24)  FIM
```

Figura 8.2: Algoritmo para determinar os dominadores imediatos.

A solução algorítmica utilizada, começa por considerar para cada nodo n, que todos os seus dominadores, à excepção do próprio n, são seus dominadores imediatos (o que corresponde ao conjunto $Tmp(n)$). Depois para cada nodo n pertencente ao GFC (à excepção do nodo raiz), testa se entre os dominadores imediatos existe algum nodo (t) que domine outro (s). Se tal acontecer, então o nodo dominador (t) é removido dos dominadores imediatos de n. Desta forma os nodos vão sendo eliminados do conjunto $Tmp(n)$, que no final conterá no máximo um único nodo (que corresponde ao dominador imediato de n) que, conforme já foi referido, quando existe é único.

Como o algoritmo apresentado requer informação sobre os dominadores de cada nodo, o utilizador é obrigado a implementar uma solução para apurar esta informação, de preferência sob a forma de um componente. Felizmente neste caso, o utilizador pode ainda optar por utilizar a solução já existente na *framework Dolphin* (componente *Dominators*). Caso opte pela primeira alternativa, o utilizador deverá realizar os mesmos procedimentos que vão ser efectuados ao longo deste capítulo para implementar o componente que vai determinar os dominadores imediatos.

Convém assinalar que o componente que se encontra implementado na *framework Dolphin* para determinar os dominadores (*Dominators*), constrói um dicionário que rela-

ciona cada nodo, com o conjunto de nodos que o dominam.

8.1.2 Estrutura do componente

É altura de decidir que tipo de interfaces devem ser utilizadas. Pelo simples facto de se tratar de um componente, é de todo recomendada a implementação da interface *Component*. Se a informação que apura é susceptível ao estado de determinados elementos da RIC, então deve também implementar a interface *Observer*. A implementação destas interfaces, passa por incluí-las na classe utilizada para construir o componente, conforme está representado na linha 1 da Figura 8.3.

```
(1)   class IDominator :public Component, public Observer{
(2)     private:// Declaração das variáveis
(3)       CFG *___cfg;
(4)       DictT<FlowNode*,FlowNode*> ___idom;
(5)       // Definição dos métodos
(6)       bool compute(Dominators*);
(7)       bool procReport(Observed*,Method<bool,TL1(Label*)>*);
(8)       bool procReport(Observed*,Method<bool,TL1(FlowNode*)>*);
(9)       // Interface Protocol
(10)      bool execute0();
(11)      bool execute1();
(12)    public:// Definição dos construtores e destrutor
(13)      IDominator();
(14)      IDominator(CFG*);
(15)      IDominator(Function*);
(16)      virtual ~IDominator();
(17)      // Definição dos métodos
(18)      void setCFG(CFG*);
(19)      CFG *getCFG();
(20)      void setFunction(Function*);
(21)      Function *getFunction();
(22)      FlowNode *getIDominator(FlowNode*);
(23)      // Definição dos operadores
(24)      FlowNode *operator[](FlowNode*);
(25)      int buildQueue(QueueL<FlowNode*>*,int st=0);
(26)      int buildQueue(QueueL<CellKey<FlowNode*,FlowNode*>*>*,int st=0);
(27)      // Interface FObject
(28)      static const char *id;
(29)      const char *getClass();
(30)      // Interface Observer
(31)      bool notify(Observed*,Report*);
(32)  };
```

Figura 8.3: Interface do componente IDominator.

A classe *IDominator* contém três variáveis: *id*, cuja definição é, como mais adiante

se explica, imposta pela interface *FObject*; $\_\_cfg$, que serve para salvaguardar a referência do GFC (objecto *CFG* da DIR), para o qual se vai computar os dominadores imediatos; e $\_\_idom$, o dicionário que relaciona cada nodo do GFC com o seu dominador imediato, ou seja, a estrutura de dados utilizada para salvaguardar a informação apurada pelo componente.

Estão também definidos vários métodos, alguns que visam a implementação das interfaces (ver Secção 8.1.4 e 8.1.5), e outros que fazem parte da estrutura base do próprio componente, como é o caso dos construtores, do destrutor e dos métodos de acesso às variáveis. A Figura 8.4 mostra a implementação dos construtores e do destrutor. Os construtores, para além de permitirem a instanciação dos objectos, servem para inicializar as variáveis. $\_\_cfg$ é inicializado a *NULL* ou, caso seja utilizado o construtor *IDominator(CFG*)*, com o valor do parâmetro. $\_\_idom$, que é um dicionário implementado através da *template DictT* (uma das várias *templates* disponibilizadas pela *framework Dolphin*), requer ser inicializado com os valores a utilizar para representar o valor nulo da chave e da informação. Como neste caso, quer a chave, quer a informação associada à chave, são apontadores, então a iniciação de $\_\_idom$ faz-se utilizando valores do tipo *NULL*.

```
(1)     IDominator::IDominator():___idom(NULL,NULL){setCFG(NULL);}
(2)
(3)     IDominator::IDominator(CFG *g):___idom(NULL,NULL){setCFG(g);}
(4)
(5)     IDominator::~IDominator(){}
```

Figura 8.4: Construtores e destrutor do componente.

Os construtores fazem uso do método *void setCFG(CFG*)*, que é público e que serve para: efectuar o registo do elemento da RIC no componente; o registo inverso, isto é, o registo do componente no elemento da RIC; e ainda o registo do componente como observador. É neste método que se executam algumas das principais tarefas necessárias à implementação das interfaces. A implementação deste método encontra-se na Figura 8.5.

Conforme explicado na Secção 3.4, e como se pode conformar pela Figura 8.5, a versão original da *framework Dolphin* coabita com a versão definitiva. A primeira através do protocolo *P0* e a segunda através do protocolo *P1*. Isto permite que através de uma simples operação (*setProtocol(...)*), se passe da versão original para a versão definitiva da *framework Dolphin*, e vice-versa.

Um detalhe que poderá passar despercebido ao leitor, é que o registo do elemento da RIC no componente, é efectuado em duplicado: a primeira vez ocorre na linha 3 da Figura 8.5, em que o elemento da RIC fica registado na variável $\_\_cfg$; e a segunda vez ocorre aquando da invocação do método *setElem(...)* (linha 11 da Figura 8.5), em que o elemento fica registado na variável $\_\_elem$ da interface *Component* (ver Figura 5.7). Este último registo faz parte do funcionamento da interface *Component*, pelo que não deverá ser alterado. De notar ainda que é através do método *setElem(...)* que se efectua o registo do componente no elemento da RIC e que se remove registos anteriores do componente.

O motivo que leva a utilizar uma variável local para manter o registo do elemento da RIC, deve-se ao facto de $\_\_elem$ ser definida, em *Component*, como sendo um apontador do tipo *compManager*. Significa isto, que não é possível aceder directamente aos métodos de CFG a partir de $\_\_elem$, sem antes efectuar uma operação de *cast*. Para evitar esta operação, recomenda-se a utilização de uma variável local que aponte para o mesmo objecto

```
(1)   void IDominator::setCFG(CFG *g){
(2)     Framework frm;
(3)     ___cfg=g;
(4)     if(___cfg){
(5)       switch(getProtocol()){
(6)         case P0:  // Protocolo da versão original da framework
(7)           break;
(8)         case P1:{// Protocolo que implementa a arquitectura
(9)           Observed *obs;
(10)          compManager *cmp=dynamic_cast<compManager*>(___cfg);
(11)          if(cmp) setElem(___cfg);
(12)          obs=dynamic_cast<Observed*>(___cfg);
(13)          if(obs){
(14)            obs->regObs(this,WCFG,3);
(15)            obs->regObs(this,WJMP|WCJMP,3);
(16)          }
(17)        }break;
(18)      default:
(19)        frm.error("IDominator","setCFG(Function*)",
(20)        "Protocolo não definido");
(21)      }
(22)    }
(23)  }
(24)
(25)  CFG *IDominator::getCFG(){return ___cfg;}
```

Figura 8.5: Implementação dos métodos *void setCFG(CFG*)* e *CFG *getCFG()*.

que *__elem*, mas que seja definida como um apontador para um objecto do tipo *CFG*, permitindo assim aceder sem qualquer operação de *cast* aos métodos desta classe.

Para que os registos sejam efectuados de forma segura, é conveniente testar se o elemento da RIC utilizado pelo componente implementa a interface *compManager* (linha 10 e 11 da Figura 8.5); e se os elementos através dos quais vão ser enviadas as mensagens para efectuar o registo do componente como observador, implementam pelo menos a interface *regObserver* (linha 12 e 13 da Figura 8.5). Caso o elemento utilizado para submeter as mensagens, seja um dos elementos a observar (como acontece neste exemplo), então é conveniente assegurar que implementa a interface *Observed*.

Convém também assinalar que o registo do componente como observador, que é efectuado em três tipos de elementos da DIR (*CFG*, *Jump* e *CJump*), requer a geração e envio dos relatório (*Report*) e a captura do estado (valor do terceiro parâmetro dos métodos *regObs(...)*). Os detalhes acerca da escolha dos elementos a observar e os motivos que levaram a requerer a activação dos dois mecanismos, encontram-se descritos na Secção 8.1.5.

A Figura 8.5 contém também a implementação do método que permite aos utilizadores acederem ao valor de *__cfg* (*CFG *getCFG()*).

Para além dos construtores definidos na Figura 8.4, existe mais um construtor e alguns métodos (linha 15, 20 e 21 da Figura 8.3), cuja implementação visa satisfazer as recomenda-

ções feitas na Secção 5.2.2. O objectivo é disponibilizar métodos que tornem mais simples a utilização do componente. O que neste caso passa por permitir que o registo dos componentes, se faça utilizando elementos da RIC que sejam mais acessíveis para o utilizador. Por exemplo, *Function* que está relacionado com *CFG* numa relação de um para um, é mas acessível a partir dos objectos principais (objectos do tipo *Program* ou *DIR*), do que é *CFG*. Pelo que, para o utilizador, a operação de instanciação de um objecto deste componente ou a operação de registo da instância (no elemento da RIC), tornar-se-á mais acessível se ambas operações poderem ser efectuadas utilizando elementos do tipo *Function*. Neste sentido foi implementado mais um construtor e dois métodos de acesso à variável $\_\_cfg$ via *Function*, que se encontram representados na Figura 8.6.

```
(1)     IDominator::IDominator(Function *f)
(2)     :___idom(NULL,NULL){setFunction(f);}
(3)
(4)     void IDominator::setFunction(Function *f){
(5)       if(f) setCFG(f->getCFG());
(6)       else setCFG(NULL);
(7)     }
(8)
(9)     Function *IDominator::getFunction(){
(10)      if(___cfg) return ___cfg->getFunction();
(11)      else return NULL;
(12)    }
```

Figura 8.6: Implementação dos métodos que visam facilitar a utilização do componente.

A classe *IDominator* define também um método e um operador (linha 22 e 24 da Figura 8.3) que permitem obter o dominador imediato de um dado nodo. Este método e este operador encontram-se representados na Figura 8.7.

```
(1)     FlowNode *IDominator::getIDominator(FlowNode *n){
(2)       return ___idom[n];
(3)     }
(4)
(5)     FlowNode *IDominator::operator[](FlowNode *n){
(6)       return ___idom[n];
(7)     }
```

Figura 8.7: Método e operador para obter o dominador imediato de um dado nodo.

Como $\_\_idom$ é uma variável privada, é conveniente disponibilizar métodos que permitam um acesso mais generalizado ao seu conteúdo, sem que no entanto permitam que seja modificada. Para este efeito recomenda-se a implementação dos métodos *buildQueue(...)*. A ideia base destes métodos é permitir construir uma fila com os elementos existentes em

__idom, que poderá ser utilizada, por exemplo, para iterar através de *__idom*. A Figura 8.8 mostra a implementação destes métodos, que é extremamente simples, dado que se limitam a utilizar os métodos que estão implementados na *template DictT* (utilizada para definir *__idom*). No primeiro caso devolve uma fila (através do primeiro parâmetro) com os nodos chave de *__idom*; no segundo caso devolve uma fila com apontadores para elementos do tipo *CellKey*. Cada elemento deste tipo contém um par de valores formado por um nodo e respectivo dominador imediato (endereços).

```
(1)   int IDominator::buildQueue(QueueL<FlowNode*> *q,int st){
(2)     return ___idom.buildQueue(q,st);
(3)   }
(4)
(5)   int IDominator::buildQueue
(6)   (QueueL<CellKey<FlowNode*,FlowNode*>*> *q,int st){
(7)     return ___idom.buildQueue(q,st);
(8)   }
```

Figura 8.8: Implementação dos métodos *buildQueue(...)*.

8.1.3 Implementação do algoritmo

Para se concluir a definição das variáveis e métodos base do componente, falta apenas efectuar a implementação do algoritmo que permite apurar os dominadores imediatos (Figura 8.2).

Calcular os dominadores imediatos é a operação que se espera que o componente efectue, isto é, a operação que deve ser realizada aquando do pedido de execução do componente. É, como tal, uma invariante do componente que deverá ser executada independentemente do protocolo em uso. Deve como tal ser implementada num método à parte, que no presente caso corresponde a *bool compute(Dominators*)*, cuja implementação encontra-se na Figura 8.9.

A solução implementada é praticamente uma tradução directa do algoritmo, se atendermos que cada ciclo do tipo PARA $\forall$ n $\in$ N FAZER , é convertido nas instruções que se encontram representadas na Figura 8.10. É como tal necessário introduzir para cada conjunto N, uma fila do tipo *QueueL*[1], aqui representada por qN. É o caso das filas G, Tn e Tn_s que correspondem, respectivamente, ao conjunto de nodos do GFC, a $Tmp(n)$ e a $Tmp(n) - \{s\}$.

Como as filas funcionam como cópias dos respectivos conjuntos, modificar a fila em nada afecta esse conjunto. Por um lado permite proteger os dados, mas por outro requer alguns cuidados extras. Por exemplo, a operação do algoritmo que remove t de $Tmp(n)$ (linha 15 da Figura 8.2), requer em termos de implementação que se remova t de $Tmp(n)$, mas também t da fila correspondente $Tmp(n)$, ou seja de Tn (linha 20 e 21 da Figura 8.9).

[1] A *framework Dolphin* disponibiliza dois tipos de estruturas abstractas de dados, implementadas sob a forma de *templates*, para a definição de objectos do tipo fila: *Queue*, que é uma *template* construída de raiz que contém apenas métodos típicos de uma fila; e *QueueL*, que tendo sido desenvolvida com base na *template List* (que define uma lista ligada simples), disponibiliza os métodos típicos de uma fila, mas também os métodos típicos de uma lista, como por exemplo o *remObj(...)* (linha 21 da Figura 8.9).

```
(1)   bool IDominator::compute(Dominators *dom){
(2)     FlowNode *n,*s,*t, *r=___cfg->getRNode();
(3)     SetDict<FlowNode*,FlowNode*> Tmp(NULL,NULL);
(4)     QueueL<FlowNode*> G(NULL),Tn(NULL),Tn_s(NULL);
(5)     Set<FlowNode*> *Saux;
(6)     ___cfg->_B_CFG::buildQueue(&G);
(7)     for(n=G.dequeue();n;n=G.dequeue())
(8)       Tmp.setSet(n,(*dom)[n]-n);
(9)     ___cfg->_B_CFG::buildQueue(&G,1);
(10)    G.remObj(r);
(11)    for(n=G.dequeue();n;n=G.dequeue()){
(12)      Saux=Tmp._getVal(n);
(13)      Saux->buildQueue(&Tn);
(14)      for(s=Tn.dequeue();s;s=Tn.dequeue()){
(15)        (*Saux)-=s;
(16)        Saux->buildQueue(&Tn_s);
(17)        (*Saux)(s);
(18)        for(t=Tn_s.dequeue();t;t=Tn_s.dequeue()){
(19)          if(Tmp[s]<=t){
(20)            (*Saux)-=t;
(21)            Tn.remObj(t);
(22)          }
(23)        }
(24)      }
(25)    }
(26)    ___cfg->_B_CFG::buildQueue(&G);
(27)    for(n=G.dequeue();n;n=G.dequeue())
(28)      ___idom(n,Tmp.getVal(n).getFirst());
(29)    return true;
(30)  }
```

Figura 8.9: Implementação do algoritmo para computar os dominadores imediatos.

De notar que as estruturas abstractas de dados, utilizadas na *framework Dolphin* e implementadas através de *templates*, fazem um uso intensivo da redefinição de operadores, dos quais se destacam os seguintes:

D[n] Devolve o valor associado a n do dicionário D;

S-n Remove o elemento n do conjunto S;

S-=n Remove o elemento n do conjunto S;

N(n) Insere o elemento n na estrutura de dados N;

S<=n Resulta em *true*, caso n pertença ao conjunto S, caso contrário resulta em *false*.

É ainda de realçar que, para o correcto funcionamento do algoritmo, o ciclo *for* que tem início na linha 11 (Figura 8.9), deve processar os nodos segundo uma travessia do tipo

```
(1)     N.buildQueue(&qN);
(2)     for(n=qN.dequeue();n;n=qN.dequeue(){
```

Figura 8.10: Tradução para linguagem C dos ciclos PARA ∀ n ∈ N FAZER .

breadth-first posorder. O que é fácil de conseguir, dado que o método *buildQueue(...)* de *CFG* é capaz de ordenar os nodos do GFC de várias formas. Neste caso, o valor 1 do segundo parâmetro da linha 9 da Figura 8.9, vem exactamente requerer este tipo de ordenação.

8.1.4 Implementação da interface *Component*

Uma vez definidos e implementados todos os métodos específicos do componente, falta agora fazer a implementação dos métodos requeridos pelas interfaces, que permitem que o componente aproveite e funcione segundo os princípios estabelecidos pela arquitectura.

Component não requer a definição directa de variáveis, nem de métodos específicos, o que já não acontece com as interfaces *FObject* e *Protocol*, herdadas através de *Component*. Conforme foi explicado no Capítulo 3, *FObject* define os procedimentos para se efectuar a identificação das classes. O que em termos de protótipo passa por definir a variável *id* e o respectivo método de acesso (linha 28 e 29 da Figura 8.3). Em termos de implementação, há que inicializar a variável *id* e implementar o método *getClass()*, conforme ilustra a Figura 8.11.

```
(1)     const char *IDominator::id="IDominator";
(2)
(3)     inline const char *IDominator::getClass(){return id;}
```

Figura 8.11: Implementação da interface *FObject*.

A interface *Protocol* requer a redefinição dos métodos associados aos protocolos em uso, que neste caso corresponde a *P0* e *P1*. Significa isto que é necessário redefinir os métodos *bool execute0()* e *bool execute1()* da interface *Protocol* (linhas 10 e 11 da Figura 8.3). Estes métodos são invocados aquando do pedido de execução do componente (via *bool execute()*) e em conformidade com o protocolo em uso.

De notar que a implementação de *bool execute0()*, visa permitir que o componente seja utilizado sem fazer uso da arquitectura. Para tal, inclui implicitamente *Dominators* e requer a sua execução, para que possa posteriormente invocar o método *bool compute(Dominators*)*, que é onde efectivamente se computa os dominadores imediatos (ver Figura 8.9). A implementação de *bool execute0()* encontra-se na Figura 8.12.

De notar que é possível utilizar *IDominator* fazendo a inclusão explícita de *Dominators*, bastando para tal tornar público o método *bool compute(Dominators*)*.

O método *bool execute1()* é parte fundamental da adaptação da arquitectura à *framework Dolphin*. Serve para testar, segundo as interfaces e o comportamento definido pela arquitectura, se estão reunidas as condições necessárias à execução do componente. Só posteriormente é que passa a execução propriamente dita, que é efectuada pelo método *bool compute(Dominators*)*.

```
(1)     bool IDominator::execute0(){
(2)       bool st
(3)       if(___cfg){
(4)         Dominators dom(___cfg);
(5)         dom.execute();
(6)         st=compute(&dom);
(7)       } else st=false;
(8)       return st;
(9)     }
```

Figura 8.12: Implementação do método *bool execute0()*.

As operações executadas por *bool execute1()*, cuja implementação se encontra na Figura 8.13, consiste em testar se o componente já está associado a um elemento da RIC e, se tal acontecer, requerer a actualização dos componentes de suporte. O que neste caso se resume ao componente *Dominators* (linha 4 da Figura 8.13). Se o componente de suporte não existir, *IDominator* cria uma nova instância deste componente, efectua o seu registo e pede a sua actualização (linha 5, 6 e 7 da Figura 8.13). Só após haver garantia de que os componentes de suporte existem e estão devidamente actualizados, é que a execução prossegue. O que, no presente exemplo, consiste em limpar a estrutura *__idom* e requerer a execução do método que vai efectivamente determinar os dominadores imediatos (linha 10 e 11 da Figura 8.13). Se esta operação for executada sem problemas, o estado de *IDominator* será assinalado como actualizado (*UPDATED*).

De notar que, se este componente já fizesse parte da *framework* (versão original), a adaptação da arquitectura consistia apenas em fazer com que herdasse as interfaces e em implementar o *case P1* do método *void setCFG(CFG*)* (linhas 8 à 17 da Figura 8.5) e o método *bool execute1()*. O que mais uma vez vem provar a independência e a simplicidade de utilização da arquitectura.

8.1.5 Implementação da interface *Observer*

Para concluir a implementação de *IDominator* falta apenas tratar da interface *Observer*. O componente ficará perfeitamente funcional e em conformidade com a arquitectura, redefinindo apenas os métodos *notify(...)* como ilustra a Figura 8.14.

É também possível optar por actualizar de imediato o estado do componente, recomputando integralmente toda a informação, conforme ilustra o exemplo da Figura 8.15. De notar que deve ser utilizado o método *execute()* e não o método *update()*, uma vez que este último apenas executa o componente se a variável *__state*, da interface *Component*, estiver em *OUTDATED*. O que não acontece, pois apesar do componente estar efectivamente desactualizado, a variável *__state* ainda indica que o componentes está UPDATED.

Na realidade, é mesmo possível evitar reescrever estes métodos, fazendo com que os componentes do tipo *Análise*, como é o caso do *IDominator*, derivem da classe *Analysis*, que é uma classe abstracta que implementa as interfaces *Component* e *Observer* e que redefine, por omissão, alguns dos métodos destas interfaces, como é o caso do *notify(...)*.

```
(1)     bool IDominator::execute1(){
(2)       bool st=false;
(3)       if(___cfg){
(4)         Dominators *dom=(Dominators*)(___cfg->update("Dominators"));
(5)         if(!dom){
(6)           dom=new Dominators(___cfg);
(7)           if(dom) st=dom->update();
(8)         }else st=true;
(9)         if(st){
(10)          ___idom.remAll();
(11)          st=compute(dom);
(12)          if(st){
(13)            setState(UPDATED);
(14)            return true;
(15)          }
(16)        }
(17)      }
(18)      return false;
(19)    }
```

Figura 8.13: Implementação do método *bool execute1()*.

```
(1)     bool IDominator::notify(Observed *){
(2)       setState(OUTDATED);
(3)       return false;
(4)     }
(5)     bool IDominator::notify(Observed*,Report*){
(6)       setState(OUTDATED);
(7)       return false;
(8)     }
```

Figura 8.14: Redefinição base dos métodos *notify(...)*.

Escolha dos elementos a observar

Para que a notificação do componente funcione, conforme esperado, é necessário proceder ao registo do componente como observador. O que deverá ser feito nos elementos da RIC que de alguma forma influenciam o estado do componente. Conforme se explica na secção anterior, *IDominator* requer a observação dos elementos do tipo *CFG*, *Jump* e *CJump*. Não significa isto que não pudessem ser escolhidos mais ou outros elementos. No entanto, e como se mostra mais adiante, basta controlar estes elementos para se controlar todas as alterações que possam ocorrer na RIC que influenciam o estado de *IDominator*.

As operações que podem alterar a informação sobre os dominadores imediatos, são:

```
(1)     bool IDominator::notify(Observed *){
(2)       return execute();
(3)     }
(4)     bool IDominator::notify(Observed*,Report*){
(5)       return execute();
(6)     }
```

Figura 8.15: Redefinição dos métodos *notify(...)*, forçando a recomputação integral do componente.

- Inserção de um novo nodo no GFC;
- Remoção de um nodo do GFC;
- Alteração dos ramos entre nodos, isto é, das saídas dos elementos do tipo *FlowNode*.

A inserção e remoção de nodos do GFC é efectuada via *CFG*, pelo que é imprescindível o controlo deste tipo de elementos. Já as ligações entre nodos (ramos do GFC) podem ser modificadas através dos elementos do tipo *FlowNode*. Pelo que, para manter a consistência do componente, poder-se-á optar por controlar os elementos deste tipo, isto é, *JNode*, *CSNode*, *CJNode* e o *RNode* (classes derivadas de *FlowNode*). Se à primeira vista é necessário controlar quatro classes, na realidade só duas são relevantes para o estado de *IDominator*. Isto porque a ligação entre nodos faz-se a partir do nodo origem, utilizando por exemplo o método *n.setOut(p)*, em que n é o nodo origem e p o nodo destino. Isto elimina a necessidade de controlar *RNode*, dado que nunca poderá ser utilizado como nodo de origem. Basta assim controlar *JNode* e *CJNode*, uma vez que *CSNode* deriva de *JNode* e não acresce operações que possam provocar alterações no estado de *IDominator*. De notar no entanto, que se forem controladas as quatro classes, o pior que pode acontecer é que o componente vai ser notificado mais vezes.

No entanto, quer *JNode*, quer *CJNode*, contêm vários métodos para alterar as ligações entre nodos. Se os métodos *bool notify(...)* se limitarem a efectuar as operações mínimas, isto é, colocar o estado do componente a *OUTDATED* (ver Figura 8.14), ou a requerer de imediato a recomputação integral do componente (ver Figura 8.15), então a quantidade de métodos que podem alterar o estado dos elementos *JNode* e *CJNode*, é irrelevante. O que já não é verdade, se o objectivo for desenvolver contra-métodos que permitam efectuar a recomputação optimizada do componente.

Todos os métodos das classes *JNode* e *CJNode*, que permitem alterar as ligações entre nodos, fazem-no através de dois métodos: um pertencente à classe *Jump* (*bool setJLabel(Label*)*); e outro à classe *CJump* (*bool setCJLabel(Label*)*). É como tal menos trabalhoso implementar os contra-métodos, se se optar por controlar *Jump* e *CJump*, em vez de *JNode* e *CJNode*. Pelo que foi esta a opção escolhida para a implementação de *IDominator*[2]. In-

[2]Conforme explicado no Capítulo 4, apesar da DIR disponibilizar múltiplos níveis de abstracção e de garantir a consistência entre eles, é o nível de menor abstracção, que é formado pelas *Expressions*, que suporta a RIC (à excepção das tabelas de identificadores) e sobre o qual se reflectem todas as operações efectuadas nos restantes níveis de abstracção. Significa isto que, sempre que possível, os elementos da RIC devem ser controlados neste nível, uma vez que é mais simples, requer a implementação menos contra-métodos e é mais seguro.

formação mais detalhada sobre as classes da DIR, que permite perceber melhor esta opção, encontra-se no Capítulo 4.

Implementação dos contra-métodos

Como já foi dito, os procedimentos efectuados até aqui, são suficientes para que *IDominator* possa ser utilizado em conformidade com a arquitectura proposta. É no entanto possível ir mais longe, optimizando a recomputação do componente. Para tal, há que redefinir os métodos *notify(...)* e implementar os contra-métodos (ver Capítulo 6.2.1).

Conforme se mostra mais adiante, a implementação dos contra-métodos resulta, sob determinadas circunstâncias, em melhoria muito significativas, nomeadamente quando se compara o tempo de execução do componente com e sem contra-métodos.

É no entanto importante realçar, que a implementação dos contra-métodos é um assunto que requer alguns conhecimentos acerca da DIR, da arquitectura, da própria *framework Dolphin* e também dos próprios processos implementados nos componentes. Acresce ainda, que a implementação de contra-métodos não é eficaz para todo o tipo de componente, nem em todas as situações.

Como foi anteriormente explicado, os elementos observados por *IDominator* são do tipo *CFG*, *Jump* e *CJump*. O próximo passo a executar é determinar que métodos destas classes requerem contra-métodos.

No caso de *CFG* são essencialmente três os métodos base que requerem contra-métodos, é o caso de *int remAll()*, *bool insNode(FlowNode*)* e *bool remNode(FlowNode*)*. Todos eles extremamente simples de implementar. O primeiro requer que se removam todas as entradas de *__idom*; o segundo, que se acrescente uma nova entrada em *__idom* para o nodo inserido; o terceiro, que se elimine de *__idom* a entrada do nodo removido e que se coloque todos os nodos, que tinham o nodo removido como dominador imediato, como não tendo dominador imediato. Como estes dois últimos métodos base partilham o mesmo número e tipo de operandos e de valor de retorno, optou-se por juntar os respectivos contra-métodos, conforme ilustra a Figura 8.16.

Para as classes *Jump* e *CJump* basta criar contra-métodos para *bool setJLabel(Label*)* e *bool setCJLabel(Label*)*. O primeiro método faz parte da classe *Jump*, a qual representa um expressão de salto incondicional, que está sempre associada a um nodo do tipo *JNode*; o segundo método faz parte da classe *CJump*, a qual representa uma expressão de salto condicional, que está sempre associada a um nodo do tipo *CJNode*. Convém acrescentar que *CJump* deriva de *Jump*, e que o método *bool setJLabel(Label*)* é utilizado em *Jump* para estabelecer a ligação de salto incondicional, mas também é utilizado em *CJump* para estabelecer a ligação para o caso em que a condição de teste é verdadeira. Para o caso em que a condição de teste é falsa, utiliza-se o método *bool setCJLabel(Label*)*. Para lá destas diferenças, o tratamento dado a cada um dos métodos é muito semelhante, até porque:

- Ambos métodos se destinam a estabelecer ligações entre dois nodos do GFC;
- Em termos do cálculo dos dominadores imediatos é indiferente o significado atribuído ao tipo de ligação. É, como tal, indiferente se a ligação é ou não condicional ou, no caso de ser condicional, se corresponde à ligação para quando a condição de teste é verdadeira, ou se corresponde à ligação para quando a condição de teste é falsa.

Como o número e tipo dos parâmetros e do valor de retorno, também é igual para os dois métodos, optou-se por fazer uma implementação conjunta dos respectivos contra-métodos, conforme se pode comprovar pela Figura 8.18.

```
(1)   bool procM(Observed *obs,Method<int,TL0()> *m){
(2)     bool st=false ;
(3)     if(m->getR()>0)
(4)       if(strcmp(m->getMID(),"remAll")==0){
(5)         ___idom.remAll();
(6)         st=true;
(7)       }
(8)     return st;
(9)   }
(10)  bool procM(Observed *obs,Method<bool,TL1(FlowNode*)> *m){
(11)    bool st=false ;
(12)    if(m->getR()){
(13)      char *mid=m->getMID();
(14)      FlowNode *n=m->getP0();
(15)      if(strcmp(mid,"insNode")==0){
(16)        ___idom(n);
(17)        st=true;
(18)      }else if(strcmp(mid,"remNode")==0){
(19)        CellKey<FlowNode*,FlowNode*> *cell;
(20)        QueueL<CellKey<FlowNode*,FlowNode*>*> qc(NULL);
(21)        ___idom.buildQueue(&qc,0);
(22)        for(cell=qc.dequeue();cell;cell=qc.dequeue())
(23)          if(cell->getVal()==n) cell->setVal(NULL);
(24)        st=true;
(25)      }
(26)    }
(27)    return st;
(28)  }
```

Figura 8.16: Contra-métodos para a classe CFG.

Aquando da notificação, o componente recebe o endereço do elemento observado, isto é, do elemento que despoletou o envio da notificação. Para os métodos em causa, este endereço corresponde a um objecto do tipo *Jump* ou *CJump*. Ambas, são classes derivadas de *Expression* que, como tal, estão associadas a um nodo do GFC, isto é, a um objecto do tipo *FlowNode* (mais precisamente a um objecto do tipo *LDT*). Este *FlowNode* corresponde ao nodo origem da ligação que se estabelece através dos métodos *bool setJLabel(Label*)* e *bool setCJLabel(Label*)*.

O parâmetro utilizado na invocação destes métodos serve para apurar o nodo destino. Isto porque representa um objecto do tipo *Label*, que deriva de *Expression* pelo que também associado a um nodo do GFC[3].

A Figura 8.17 ajuda a perceber a relação que existe entre *Jump* e *JNode*, entre *CJump* e *CJNode*, e entre todos estes elementos e *Label*.

A solução utilizada na implementação dos contra-métodos requer que se determine: o

[3]Enquanto *Jump* e *CJump* correspondem sempre à última expressão contida num nodo, *Label* corresponde sempre à primeira expressão.

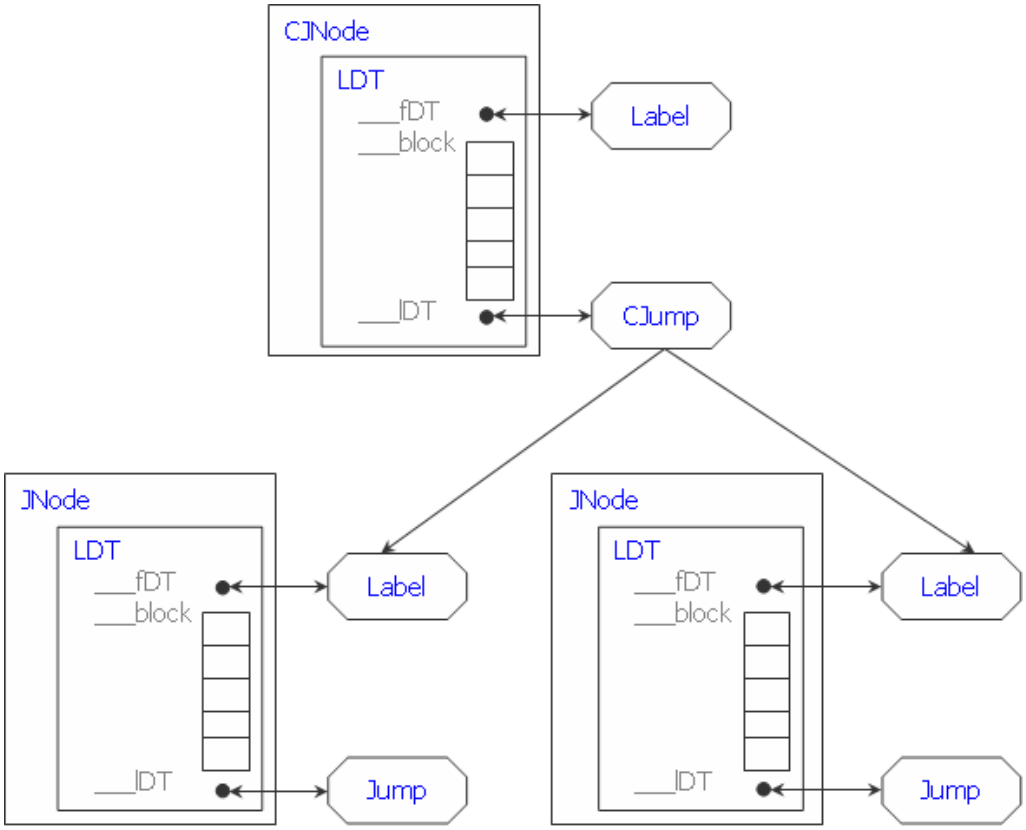


Figura 8.17: Relação entre *Jump*, *CJump*, *Label*, *JNode* e *CJNode*.

nodo origem (n), computado na linha 10 da Figura 8.18; o nodo destino anterior à execução do método (no_ant), computado na linha 14 da Figura 8.18; e o nodo destino posterior à execução do método (no_actual), computado na linha 22 da Figura 8.18. Isto porque se parte do princípio que inicialmente existe uma ligação entre n e no_ant e que, após a execução do método, esta ligação deixa de existir e passa a existir uma ligação entre n e no_actual.

Para se determinar no_ant recorre-se ao mecanismo de captura de estado (ver linha 9 da Figura 8.18). No caso do *Jump*, o objecto que representa o estado do componente (do tipo *State*), possui uma única variável: um apontador do tipo *Label*, que identifica o nodo destino antes da execução do método (ver linhas 12, 13 e 14 da Figura 8.18).

Para se determinar no_actual recorre-se ao *Report* enviado com a notificação, que neste caso contém um único parâmetro do tipo apontador para *Label*, através do qual é possível apurar o nodo destino após a execução do método base (ver linhas 20, 21 e 22 da Figura 8.18).

Desta forma, a solução utilizada na implementação dos contra-métodos, desenrola-se em duas fases distintas: uma que visa actualizar *IDominator*, pelo facto da ligação entre n e no_ant ter deixado de existir; e outra que visa actualizar *IDominator*, pelo facto de passar a existir uma nova ligação entre n e no_actual.

A Figura 8.19 ilustra as situações que podem ocorrer pelo facto de deixar de haver ligação entre n e no_ant. No caso de no_ant deixar de ter antecessores então $idom(no\_ant)$ é nulo (linha 16 da Figura 8.18); se possuir apenas um antecessor então $idom(no\_ant)$ é igual a esse antecessor (linha 17 da Figura 8.18); e possuir mais do que um antecessor então é necessário efectuar a recomputação de no_ant (linha 18 da Figura 8.18). Neste último caso, utiliza-se a mesma solução do algoritmo principal, mas apenas se computa o dominador imediato de no_ant. De notar que se considera que não há alterações em relação aos nodos

```
(1)   bool procM(Observed *obs,Method<bool,TL1(Label*)> *m){
(2)     bool st=false;
(3)     if(m->getR()==true){
(4)       char *mid=m->getMID();
(5)       FlowNode *no_actual, *no_ant, *n;
(6)       if(strcmp(mid,"setJLabel")==0){
(7)         Jump *jmp=dynamic_cast<Jump*>(obs);
(8)         if(jmp){
(9)           State<TL1(Label*)> *stat=jmp->getState();
(10)          n=jmp->getNode();
(11)          if(stat){
(12)            Label *lb=stat->getV0();
(13)            if(lb){
(14)              no_ant=lb->getNode();
(15)              int nn=no_ant->howManyIn();
(16)              if(nn==0)___idom(no_ant,0);
(17)              else if(nn=1) ___idom(no_ant,no_ant->getFirstIn());
(18)              else recompute(no_ant);
(19)            }
(20)            lb=m->getP0();
(21)            if(lb){
(22)              no_actual=lb->getNode();
(23)              if(no_actual->howManyIn()==1) ___idom(no_actual,n);
(24)              else recompute(no_actual);
(25)            }
(26)            st=true;
(27)          }
(28)        }
(29)      }else if(strcmp(mid,"setCJLabel")==0){
(30)        ...
(31)      }
(32)    }
(33)    return st;
(34)  }
```

Figura 8.18: Contra-método para *bool setJLabel(Label*)* e *bool setCJLabel(Label*)*.

alcançáveis a partir de no_ant. Significa isto, que mesmo não existindo caminho entre o nodo raiz e no_ant, este último poderá ser localmente dominador imediato.

Para o caso da inserção da ligação entre n e *no_actual*, pode ocorrer um das duas seguintes situações: *no_actual* tem apenas um antecessor, que terá que corresponder a n pelo que será este o dominador imediato de *no_actual* (linha 23 da Figura 8.18); caso contrário, é necessário recomputar *no_actual*, fazendo uso da solução principal para determinar os dominadores imediatos, mas neste caso apenas aplicada a *no_actual* (linha 24 da Figura 8.18).

Dado que se requereu a geração e envio dos relatórios (*Reports*), não há necessidade de redefinir o método *bool notify(Observed*)*. Pelo que para concluir a construção de *IDominator*

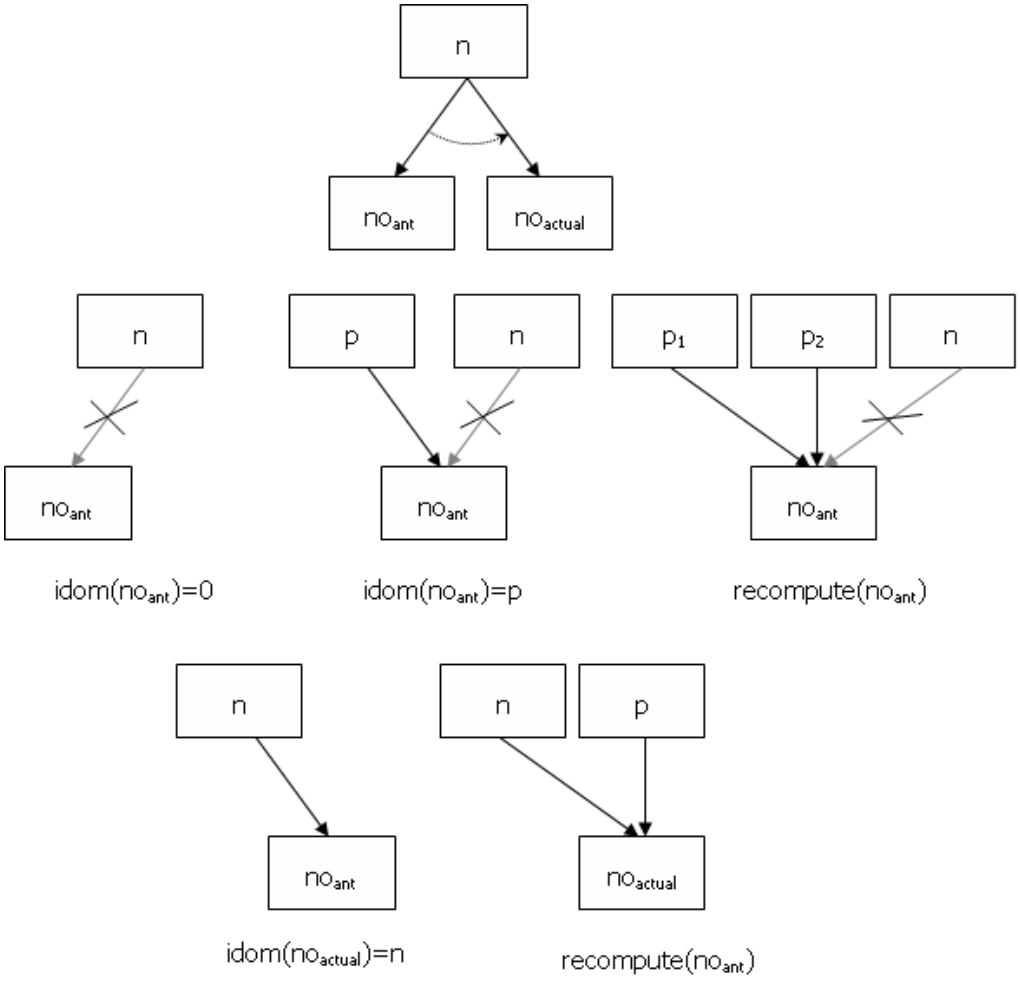


Figura 8.19: Recomputação do dominador imediato.

falta apenas mostrar como é que o método *bool notify(Observed*,Report*)* deve ser redefinido. Figura 8.20 ilustra a implementação deste método.

O primeiro teste a realizar, é verificar se até ao instante em que foi executado o método que despoletou a notificação, o componente estava actualizado (linha 4 da Figura 8.20). De notar que se o componente estava já desactualizado, muito provavelmente de nada adiantará efectuar o procedimento de recomputação optimizada.

O passo seguinte é verificar se os componentes de suporte de *IDominator* já foram notificados (linha 5 da Figura 8.20). O que apenas é preciso, caso os contra-métodos façam também eles uso dos componentes de suporte, como aliás acontece neste caso, em que a recomputação (linhas 18 e 24 da Figura 8.18) requer acesso aos dominadores (calculados pelo componente *Dominators*).

Finalmente, há apenas que verificar se a notificação visa algum dos contra-métodos implementados. O que é feito, neste caso, em duas etapas distintas. A primeira consiste em detectar o tipo de *Method*, através do número e do tipo dos parâmetros e do valor de retorno. Com esta informação é possível seleccionar o *bool procM(...)* no qual está implementado o contra-método. Numa segunda etapa, e apenas se for necessário, recorre-se ao identificador do método base (*m→getMID()*) para assim determinar as operações a efectuar (ver linhas 6 e 29 da Figura 8.18). Caso estas duas etapas não sejam suficientes para identificar claramente

o método base, invocado sobre o elemento observado, é ainda possível utilizar o identificador do elemento que despoletou a notificação (*m→getCID()*).

Apenas para concluir, falta dizer que se não existir contra-método, o estado do componente é colocado em OUTDATED (ver linha 24 e 25 da Figura 8.20).

```
(1)   bool IDominator::notify(Observed *obs,Report *r){
(2)     AbstractMethod *m;
(3)     bool st=false;
(4)     if(r && getState()==UPDATED){
(5)       if(obs->passNotification(this,"Dominators")) st=true;
(6)       else{
(7)         int i, nn=r->howMany();
(8)         for(i=0;i<nn;i++){
(9)           m=r->dequeue();
(10)          st=false;
(11)          Method<bool,TL1(FlowNode*)> *m1
(12)          =dynamic_cast<Method<bool,TL1(FlowNode*)>*>(m);
(13)          if(m1) st=procReport(obs,m1);
(14)          else{
(15)            Method<int,TL0()> *m2
(16)            =dynamic_cast<Method<int,TL0()>*>(m);
(17)            if(m2) st=procReport(obs,m2);
(18)            else {
(19)              Method<bool,TL1(Label*)> *m3
(20)              =dynamic_cast<Method<bool,TL1(Label*)>*>(m);
(21)              if(m3) st=procReport(obs,m3);
(22)            }
(23)          }
(24)          if(!st){
(25)            setState(OUTDATED);
(26)            break;
(27)          }
(28)          r->enqueue(m);
(29)        }
(30)      }
(31)    }
(32)    return st;
(33)  }
```

Figura 8.20: Redefinição do método *bool notify(Observed*,Report*)*.

Está assim pronto o componente *IDominator*, fazendo proveito de todas as facilidades concedidas pela DIR e pela arquitectura. Para utilizar este componente basta criar uma instância indicando, directa ou indirectamente, o *CFG* para o qual se pretende computar os dominadores imediatos. Depois, é apenas necessário requerer a execução do componente. A partir daí poder-se-á fazer uso do componente sabendo de antemão que os seus dados vão estar sempre consistentes com a RIC, que será minimizado o número de vezes que terá que

ser recomputado e que não haverá mais instâncias deste componente do que aquelas que são estritamente necessárias.

8.2 Construção de um compilador

A construção de compiladores com base na *framework Dolphin*, e fazendo prova do principal objectivo deste doutoramento, é um processo simples. Provavelmente, a operação mais complexa é identificar e escolher os componentes. No entanto para facilitar esta tarefa, encontra-se actualmente implementada uma pequena aplicação que permite navegar na estrutura de directorias da *framework Dolphin* e aceder a informação sobre os componentes. Esta aplicação, que surge no âmbito do projecto apresentado no Capítulo 9, requer que se associe a cada componente uma descrição em XML contendo a seguinte meta-informação:

- Objectivo do componente;
- Data de implementação;
- Versão;
- Identificação do(s) elemento(s) da RIC a utilizar para se efectuar o registo do componente;
- Identificação das interfaces requeridas para as várias classes da DIR que estão relacionados com o componente em causa;
- Descrição da solução (que pode incluir algoritmo);
- Descrição dos métodos;
- Exemplos de utilização;
- Identificação dos classes da DIR, cujas instâncias influenciam o estado do componente (elementos observados);
- Identificação dos elementos da RIC modificados pelo componente;
- Descrição das dependências com outros componentes.

A informação é associada ao componente, criando um ficheiro XML que é normalmente colocado junto ao ficheiro dos protótipos do componente. O *Schema* que define a estrutura destes ficheiros XML, encontra-se representado no Apêndice E.

Uma vez escolhido o componente, é extremamente simples ao utilizador aplicá-lo. Basta para tal obter uma instância, efectuar o registo (o que pode ser feito aquando da instanciação do componente) e finalmente executá-lo, conforme ilustram os vários exemplos da Figura 8.21. Nos quais se inclui a utilização de *XCOMPD(a,b,c)* e *XCOMPI(a,b,c)*, duas das várias macros definidas para auxiliar a construção dos compiladores.

Para ilustrar e explicar os diversos detalhes com que o utilizador pode ter que lidar, será utilizado o exemplo da Figura 8.22, que consiste na especificação de um pequeno compilador que inclui explicitamente oito componentes, a saber:

- O *littleC* que é o *front-end*;

```
(1)     // Exemplo um
(2)     elimLoads el1(f);
(3)     el1.execute();
(4)
(5)     // Exemplo dois
(6)     elimLoads el2 = new elimLoads(f);
(7)     el2->execute();
(8)
(9)     // Exemplo três
(10)    #define XCOMPD(A,B,C) A B(C);B.execute();
(11)    XCOMPD(elimLoads,el3,f);
(12)
(13)    // Exemplo quatro
(14)    #define XCOMPI(A,B,C) A B=new A(C);B->execute();
(15)    XCOMPI(elimLoads,el4,f);
```

Figura 8.21: Exemplos da instanciação e execução de um componente.

- O *elimJumpChains* e o *elimUnreachCode* que são dois componentes que visam efectuar optimizações do fluxo de controlo (o primeiro elimina cadeias consecutivas de saltos e o segundo código inalcançável);
- O *cnv2SSA* que converte a RIC da forma normal para a forma SSA;
- O *elimComSubExpr* e o *elimLoads*, que são dois componentes que efectuam optimizações do fluxo de dados (o primeiro elimina sub-expressões comuns e o segundo minimiza o número de operações de *load*);
- O *cnv2NF* que converte a RIC da forma SSA para a forma normal;
- E o *genPseudoCode*, um *back-end* capaz de gerar um assembly genérico, que considera que o número de registos é ilimitado e indiferenciado.

De notar que na realidade não são apenas estes os componentes utilizados, isto porque há componentes que são incluídos implicitamente. A Figura 8.23 representa todos os componentes concretos que são utilizados, num total de 22, e as dependências existentes entre eles. É de realçar que dois terços dos componentes são incluídos implicitamente e que o número de instâncias utilizadas, caso não se faça uso da arquitectura, é de 37.

Convém também relembrar que, normalmente, a especificação de um compilador não inclui explicitamente os componentes de suporte, como é o caso das *Análises*, dado que são incluídos implicitamente pelos componentes principais que deles fazem uso. As duas excepções a esta regra, correspondem aos componentes *cnv2SSA* e *cnv2NF*, cuja inclusão, e conforme foi explicado na Secção 5.1.1, deve ser feita por quem constrói o compilador.

O primeiro componente a executar tem que ser sempre do tipo *Front-End*, como aliás ilustra o exemplo da Figura 8.22 (linha 17). Este tipo de componente, ao contrário dos demais, não requer o registo sobre um elemento da RIC, em contrapartida necessita de ter acesso aos argumentos utilizados na invocação do compilador, o que é feito registando o *argc*

```
(1)     // Inclusão de ficheiros
(2)     ...
(3)     #define REGEXE(a,b) a.setFunction(b);a.execute();
(4)
(5)     int main(int argc,char *argv[]){
(6)       Program *p;
(7)       Function *f;
(8)       QueueL<Function*> q(0);
(9)       elimJumpChains ejc();
(10)      elimUnreachCode euc();
(11)      cnv2SSA cs();
(12)      elimComSubExpr ecse();
(13)      elimLoads el();
(14)      cnv2NF cn();
(15)      genPseudoCode ps();
(16)      littleC fe(argc,argv);
(17)      fe.execute();
(18)      if(p=fe.getProgram()){
(19)        p->buildQueue(&q);
(20)        for(f=q.dequeue();f;f=q.dequeue()){
(21)          REGEXE(ejc,f);
(22)          REGEXE(euc,f);
(23)          cs.setFunction(f);
(24)          if(cs.execute()){
(25)            REGEXE(ecse,f);
(26)            REGEXE(el,f);
(27)            REGEXE(cn,f);
(28)          }
(29)          REGEXE(ps,f);
(30)        }
(31)        delete p;
(32)      }
(33)      return 0;
(34)    }
```

Figura 8.22: Especificação integral do compilador.

e o *argv* da função *main* (linha 16). É através deste registo que é possível determinar as opções de compilação escolhidas pelo utilizador e o ficheiro a compilar.

A execução do *front-end*, se for bem sucedida, dá origem a um objecto do tipo *Program* ou do tipo *DIR*, o qual é utilizado posteriormente pelos restantes componentes para acederem aos elementos sobre os quais efectuam os registos. Componentes há que fazem o registo sobre o próprio objecto *Program*.

No exemplo da Figura 8.22, à excepção do *front-end*, os componentes são aplicados a todas as funções incluídas no objecto *Program*, isto é, as funções implementadas no ficheiro submetido ao compilador. De notar no entanto que apesar de poderem existir várias funções,

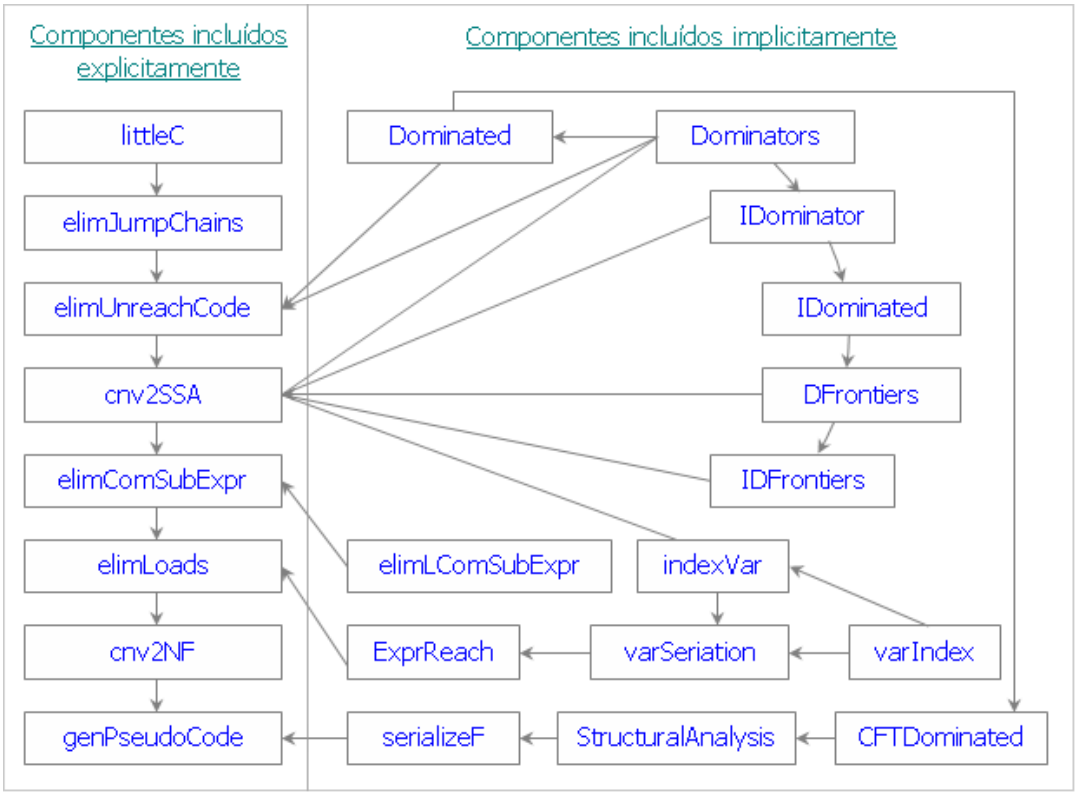


Figura 8.23: Dependências entre os componentes utilizados pelo compilador da Figura 8.22.

apenas existe uma única instância por componente.

Uma vez que a especificação do compilador é feita utilizando a própria linguagem C++, é então possível utilizar as próprias instruções desta linguagem para estruturar o compilador. Por exemplo, na Figura 8.22 é utilizado o método *buildQueue(...)* juntamente com um ciclo do tipo *for* (linhas 19 e 20) para aplicar os vários componentes às funções contidas em *Program*. Outro exemplo está na utilização da estrutura do tipo *if* (linha 24) que testa a viabilidade de efectuar a conversão para a forma *SSA*, antes de se fazer uso de determinados componentes que requerem este tipo de formato.

Apesar do compilador da Figura 8.22 apenas incluir um único *back-end*, é possível fazer uso de mais do que um por compilador e até mesmo fazer uso do mesmo *back-end* várias vezes. Um exemplo típico, ocorre quando é necessário monitorizar o processo de compilação para, por exemplo, detectar problemas de implementação. Nestes casos faz-se normalmente uso de uma outra ferramenta, o *Dolphin-MAS* [MH03g], desenvolvida no âmbito do projecto apresentado no Capítulo 9. Esta ferramenta requer informação em formato XML, da evolução da RIC ao longo das várias etapas do processo de compilação. Para tal, utiliza-se o componente *genXML*, um *back-end* capaz de colocar a informação contida na RIC em formato XML, que é aplicado após a execução de cada um dos restantes componentes. Ou seja, é utilizado várias vezes durante o mesmo processo de compilação.

Para especificar a estrutura do compilador, poderá também ser útil fazer uso dos operadores de cópia e de atribuição implementados pelas classes da DIR, como ilustra o exemplo fictício da Figura 8.24 (linha 11). Em que se cria uma cópia do objecto *Program* obtido pelo *front-end*. Como o processo de cópia dos objectos do tipo *Program* se propaga aos demais elementos contidos neste objecto, resulta numa cópia integral da RIC. Com esta duplicação da RIC, é possível dar destinos distintos à versão original e à cópia, criando linhas diferenci-

adas de compilação. Por exemplo, na Figura 8.24, uma vez criada a cópia da RIC, aplica-se sobre esta os componentes $Optimizacao_B$ e $BackEnd_B$, enquanto que sobre a versão original são aplicados os componentes $Optimizacao_A$ e $BackEnd_A$.

```
(1)   int main(int argc,char *argv[]){
(2)     Program *p,*q;
(3)     Optimizacao_A opt_A;
(4)     Optimizacao_B opt_B;
(5)     BackEnd_A be_A;
(6)     BackEnd_B be_B;
(7)     littleC fe(argc,argv);
(8)     fe.execute();
(9)     if(p=fe.getProgram()){
(10)      ...
(11)      q=new Program(p);
(12)      opt_A.setProgram(p);
(13)      opt_A.execute();
(14)      be_A.setProgram(p);
(15)      be_A.execute();
(16)      if(q){
(17)        opt_B.setProgram(q);
(18)        opt_B.execute();
(19)        be_B.setProgram(q);
(20)        be_B.execute();
(21)        ...
(22)        delete q;
(23)      }
(24)      ...
(25)      delete p;
(26)    }
(27)    return 0;
(28)  }
```

Figura 8.24: Utilização dos operadores de cópia na especificação de compiladores.

Convém no entanto salientar que os elementos da RIC que integram a cópia (q), não contêm qualquer componente associado. Isto é, este processo de cópia não inclui a duplicação dos componentes até aí registados na RIC. Significa isto que um componente que tenha sido aplicado à versão original da RIC (p) antes da operação de cópia, vai após esta manter-se disponível na versão original, mas não na cópia da RIC entretanto criada (q). Encontra-se actualmente em concepção uma solução alternativa que visa estender a operação de cópia aos componentes.

Apenas para concluir, falta dizer que o processamento das opções de compilação fica normalmente a cargo do *front-end*. No entanto, esta informação deverá estar de alguma forma disponível para que possa ser utilizada na própria especificação do compilador. Por exemplo, é comum que os compiladores disponibilizem opções para activar/desactivar determinadas optimizações de código. Este tipo de opção reflecte-se necessariamente na estrutura

do compilador, dado que o componente que implementa a optimização em causa será utilizado mediante tenha sido ou não seleccionada a opção. No caso do *littleC*, que foi construído utilizando o Eli [GHK+90], as opções de compilação são disponibilizadas através de variáveis globais, que podem ser utilizadas directamente na especificação do compilador para condicionar o seu funcionamento, conforme ilustra a Figura 8.25, em que *___ELIMLOADS* é a variável que assinala se foi requerido ou não a aplicação do componente *ElimLoads*.

```
(1)     int main(int argc,char *argv[]){
(2)        Program *p;
(3)        ElimLoads el;
(4)        ...
(5)        if(___ELIMLOADS){
(6)           el.setProgram(p);
(7)           el.execute();
(8)        }
(9)        ...
(10)       return 0;
(11)    }
```

Figura 8.25: Utilização dos argumentos de compilação para definir a estrutura do compilador.

Recomenda-se no entanto fazer uso da classe *DIR*, que contém um dicionário para armazenar as opções de compilação. Basta para tal registar *argc* e *argv* aquando da instanciação desta classe, para que automaticamente sejam determinadas as opções de compilação e preenchido o dicionário, conforme ilustra a Figura 8.26.

```
(1)     int main(int argc,char *argv[]){
(2)        DIR d(argc,argv);
(2)        FrontEnd fe(&d);
(2)        fe.execute();
(2)        ...
(2)        ElimLoads el(&d);
(7)        el.execute();
(9)        ...
(10)       return 0;
(11)    }
```

Figura 8.26: Utilização da classe DIR para gerir os argumentos do compilador.

A opção de utilizar o objecto DIR pode simplificar significativamente a estrutura do compilador. De notar que este é um dos poucos casos em que o *front-end* requer o registo num elemento da RIC (objecto DIR). A execução do *front-end* deverá complementar o preenchimento do objecto *DIR*, associando-lhe para tal o objecto *Program*. Depois tudo é muito mais simples, por exemplo, ao se registar o objecto DIR na instância de *ElimLoads*, esta

passa a conter toda a informação necessária para determinar se deve ou não ser executada. Ou seja, a decisão é tomada pelo próprio componente em conformidade com as opções de compilação utilizadas. De notar no entanto que para que isto seja funcional, o método *bool execute()* da interface *Protocol* deve ser reescrito.

8.3 Avaliação das soluções propostas

Chegado a este ponto em que quase tudo já foi dito, nomeadamente sobre a DIR, a arquitectura e as optimizações, e em que já se explicou como se procede para implementar um componente e para fazer uso da *framework Dolphin* para construir compiladores, é certamente a altura de avaliar o impacto real do trabalho até aqui desenvolvido. Na sequência do que foi feito neste capítulo, pretende-se avaliar até que ponto as soluções desenvolvidas e propostas no contexto deste doutoramento, são uma mais valia e satisfazem os objectivos estabelecidos.

Considerou-se importante analisar o trabalho desenvolvido segundo duas perspectivas distintas: uma que visa avaliar se as soluções apresentadas realmente simplificam a construção de compiladores e de componentes para compiladores; e outra que visa avaliar se as soluções desenvolvidas permitem que os componentes sejam simples de utilizar, sem com isso afectar a qualidade do processo de compilação, nem a qualidade do código produzido pelos compiladores.

8.3.1 Soluções que simplificam o desenvolvimento de compiladores

Avaliar se as soluções desenvolvidas permitem ou não tornar a construção de compiladores mais simples, não só é algo de relativo, como bastante subjectivo. Passa essencialmente por reunir um conjunto significativo de utilizadores, com experiência na utilização de soluções semelhantes, que se predisponha a fazer uma avaliação comparativa. Infelizmente, não foi possível reunir um número suficiente de utilizadores, com o perfil desejado, que permitisse fazer um estudo (por mais pequeno que fosse) para avaliar até que ponto é que as soluções desenvolvidas contribuem realmente para simplificar a construção de compiladores. Resta assim apelar para alguns exemplos, como os que foram expostos ao longo dos vários capítulos desta dissertação.

Numa perspectiva mais global, ao se apostar numa *framework* promove-se claramente a reutilização em detrimento de outras soluções, como é o caso da geração de componentes com base numa especificação. Conforme explicado no Capítulo 3, a reutilização, quando devidamente suportada, é das formas mais simples de construir compiladores. Com a vantagem adicional de que a *framework* utilizada, não inviabiliza a utilização de outras soluções, antes pelo contrário, é uma opção a ter em conta para a integração dessas soluções.

É de crer que os exemplos da Secção 5.1.1 (ver Figura 5.2) e da Secção 8.2 permitem por si só comprovar, que a reutilização dos componentes (de forma implícita) é realmente acessível. Com a garantia de que, utilizando as soluções propostas nesta dissertação, também é bastante eficiente.

É também de crer que o trabalho desenvolvido contribui de forma muito significativa para simplificar a construção dos componentes. Destaca-se o papel da DIR, que ao disponibilizar diversos níveis de abstracção e ao conter mecanismos intrínsecos que permitem que a RIC seja dinâmica e consistente, faz com que a implementação dos componentes seja substancialmente mais simples. Veja-se, por exemplo, o cálculo dos dominadores imediatos feito através do componente *IDominator*, que está documentado na Secção 8.1.3, o qual é realizado fazendo uso de um único elemento (*CFG*) que, sem excessos, contém de forma muito acessível toda a informação que o componente requer. Com a garantia de que quem

implementa o componente pode, sempre que tal seja necessário, passar a trabalhar num nível de abstracção superior ou inferior, sabendo de antemão que a informação entre os vários níveis de abstracção é actualizada automaticamente.

De notar que noutros modelos mais comuns de RIC, como é o caso dos tuplos ou das árvores de expressões, aceder à informação que define o GFC, como por exemplo aos antecessores ou aos sucessores de um nodo, não é uma tarefa trivial que se consiga efectuar com a simplicidade do exemplo da Figura 8.9 para computar os dominadores imediatos. Só mesmo em modelos mais evoluídos de RIC, como é o caso do Stanford University Intermediate Format-SUIF [ADH+00c] ou do modelo utilizado pelo RTL System [MRS90a], é que é possível encontrar soluções deste tipo.

Acresce a tudo isto a utilização de *vistas* (ver Secção 4.3.3), que permitem facultar perspectivas mais simples da RIC e aproximar a DIR de outros modelos de RIC, fazendo com que seja o modelo a ir de encontro ao utilizador e não o contrário.

Há ainda o contributo dado pela arquitectura, que ao fornecer todo um conjunto de soluções permite, por um lado, libertar o construtor dos compiladores de muitos detalhes e, por outro, implementar determinadas funcionalidades que de outra forma não seriam possíveis ou pelo menos simples de efectuar.

A utilização do projecto descrito no Capítulo 9, o *Sistema Dolphin*, que tem por base a *framework Dolphin*, será a prova final para determinar se as soluções propostas nesta dissertação contribuem ou não para simplificar a construção de compiladores e de componentes. Infelizmente, é um projecto que ainda está em fase de concepção.

8.3.2 Soluções que garantem a eficiência do processo de compilação

Atendendo que a principal característica enunciada para a arquitectura proposta nesta dissertação, é permitir reutilizar os componentes de forma simples, garantindo simultaneamente a eficiência do processo de compilação, é então importante avaliar até que ponto isto é verdade. Falta assim quantificar a influência da arquitectura na qualidade do processo de compilação. É particularmente pertinente avaliar:

1. Os mecanismos desenvolvidos para a recomputação optimizada de componentes, nomeadamente em termos de tempo de execução;

2. A parte central da arquitectura, isto é, a arquitectura sem as optimizações descritas no Capítulo 6, quer em termos de tempo de execução, quer em termos do número de instâncias utilizadas;

3. A arquitectura numa perspectiva global, quantificando também o tempo de execução e o número de instâncias utilizadas.

Qualquer uma destas avaliações é efectuada durante o processo de compilação. É como tal necessário construir, para cada uma, um compilador através do qual se possa apurar as medidas requeridas. O que, consoante o caso, envolve determinar o tempo de execução e/ou o número de instâncias utilizadas.

Apurar o tempo de execução como um valor absoluto de pouco serve, dado que depende muito de factores externos, como por exemplo da arquitectura de computação sobre a qual são efectuados os testes, da carga computacional do sistema, da forma como a *cache* é gerida, etc. É assim importante existir um valor de referência que permita saber se as soluções utilizadas contribuem ou não para garantir a eficiência do processo de compilação, o que passa por desenvolver, para cada uma das avaliações, um segundo compilador. Para o primeiro avaliação, que visa verificar o impacto dos contra-métodos, o segundo compilador

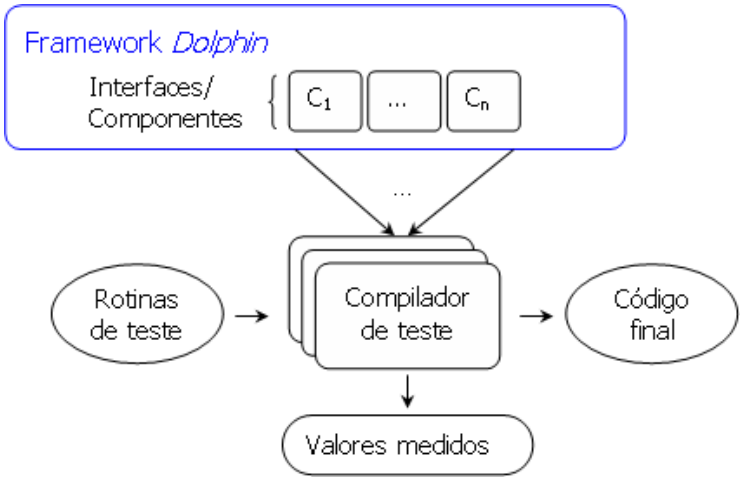


Figura 8.27: Modelo para aplicação dos testes de avaliação.

apesar de fazer uso da arquitectura, não aplica os contra-métodos. Para a segunda e terceira avaliação, o segundo compilador é implementado sem fazer uso de qualquer uma das soluções propostas.

É assim necessário construir, para cada uma das avaliações, dois compiladores: um para apurar o valor de referência e outro para efectuar os testes fazendo uso das soluções visadas em cada uma das avaliações. Para a segunda e terceira avaliação, isto significa que um dos compiladores é construído utilizando componentes da versão original da *framework Dolphin*, enquanto o outro é construído utilizando componentes, que consoante a avaliação, implementam parcialmente ou integralmente as soluções apresentadas.

É de realçar que, para lá da arquitectura, dos algoritmos utilizados e da forma como estão implementados, os componentes aplicados na construção dos compiladores de teste, são em tudo semelhantes aos encontrados em sistemas similares à *framework Dolphin*. Pelo que, os resultados obtidos com estes testes representam muito mais do que apenas o contributo dado pela arquitectura à versão original da *framework Dolphin*. Demonstram que a arquitectura pode realmente ser uma mais valia para os sistemas que dela façam uso, sendo como tal uma contribuição válida e com resultados palpáveis em termos científicos.

Claro que existem imensas variantes na realização destes testes, como por exemplo: o tipo de componentes utilizados na construção dos compiladores, a ordem pela qual os componentes são utilizados, as rotinas de teste, etc. Pelo que os resultados apresentados não devem ser interpretados como provas do que quer que seja, mas apenas como uma demonstração dos benefícios que se podem obter com a utilização das soluções propostas nesta dissertação.

A avaliação pressupõe submeter rotinas de código aos compiladores desenvolvidos para efectuar os testes, conforme ilustra a Figura 8.27. Os compiladores, para além de executarem os procedimentos inerentes aos componentes em uso, nomeadamente efectuar a geração do código final, estão adaptados para apurar os valores estipulados para cada uma das avaliações (tempo de execução e/ou número de instâncias).

Para se efectuarem os testes, escolheu-se o seguinte conjunto de rotinas, que é representativo de várias situações e cuja implementação se encontra no Apêndice D:

1. Multiplicação de matrizes bidimensionais;

2. Pesquisa numa lista simples (solução não recursiva);

3. Pesquisa numa árvore binária;

4. Implementação *hard-code* de um autómato (20 estados);

5. Uma versão, em linguagem C, da rotina que determina os dominadores imediatos.

É conveniente explicar que por limitações dos *front-ends* que estão actualmente construídos na *framework Dolphin*, não foi possível fazer uso de *benchmarks* existentes no mercado.

De notar também que o estado da execução das rotinas é irrelevante para a avaliação que se pretende efectuar. Por exemplo, a dimensão da matriz da primeira rotina ou o número de elementos contidos na lista ou na árvore (respectivamente, segunda e terceira rotina), são completamente irrelevantes para a avaliação do desempenho das soluções desenvolvidas na preparação desta dissertação, dado que estas visam a optimização do processo de compilação e não do código a produzir pelos compiladores.

Os tempos obtidos, que são medidos em número de ciclos, são dependentes da arquitectura de computação, isto é, do processador, do sistema operativo, da quantidade de memória, da dimensão e da forma como a cache é utilizada, da carga do sistema, etc. Significa isto que o tempo de execução de um programa sofre pequenas variações de execução para execução. O mesmo acontece com a execução de um compilador. Para minimizar a influência destes factores externos, os resultados apresentados correspondem ao valor médio de 1000 execuções/teste. Em que a média é feita após o teste ter sido executado 50 vezes (no total cada teste é executado 1050 vezes).

Para concluir e antes de se avançar para a descrição e apresentação dos resultados de cada uma das avaliações, falta apenas dizer que os testes foram efectuados num computador equipado com um Pentium III a 1GHz, com 512Mbytes de memória e com sistema operativo Linux (Red-Hat).

Recomputação optimizada dos componentes

Na sequência do trabalho já documentado neste capítulo, os testes preparados para efectuar a avaliação dos mecanismos de recomputação optimizada incidem sobre a execução do componente *IDominator*. É preciso no entanto um segundo componente que, ao produzir alterações na RIC, torne a informação computada por *IDominator* inconsistente. Para tal foi escolhido o *ElimJumpChains*. A Figura 8.28 mostra o programa de teste (compilador) preparado para esta avaliação. O tempo de execução é apurado para a execução de *IDominator*, de *ElimJumpChains* e para o pedido posterior de actualização de *IDominator* (linhas 21, 22 e 23).

O teste referência, em que *IDominator* faz uso da arquitectura, mas sem utilizar os contra-métodos, designa-se por $Teste_A$. A avaliação dos contra-métodos, e como tal da utilização do *Report*, e dos mecanismos de captura do estado dos elementos da RIC (ver Secção 8.1.5), é efectuada através do $Teste_B$.

Os resultados obtidos encontram-se representados na Tabela 8.1, através da qual se pode confirmar que há casos em que o tempo aumenta e outros em que reduz. No pior dos casos o tempo aumenta em 15% e no melhor dos casos diminui em 42%. Em termos médios há uma redução de aproximadamente 6%.

Os resultados vêm assim de encontro ao que já foi anteriormente dito. Os contra-métodos podem reduzir substancialmente o tempo de execução, mas tudo depende das rotinas a compilar, dos componentes utilizados na construção dos compiladores, inclusive na ordem pela qual estes são aplicados.

```
(1)     #include "Program.h"
(2)     #include "littleC.h"
(3)     #include "IDominator.h"
(4)     #include "elimJumpChains.h"
(5)     #include <time.h>
(6)
(7)     int main(int argc,char *argv[]){
(8)       int i;
(9)       Program *p;
(10)      littleC fe(argc,argv);
(11)      IDominator idom;
(12)      elimJumpChains el;
(13)      double duration[1050], soma=0.0;
(14)      clock_t start, finish;
(15)      for(i=0;i<1050;i++){
(16)        fe.execute();
(17)        if(p=fe.getProgram()){
(18)          idom.setFunction(p->getMFunction());
(19)          el.setFunction(p->getMFunction());
(20)          start=clock();
(21)          idom.execute();
(22)          el.execute();
(23)          idom.update();
(24)          finish=clock();
(25)          duration[i]=(double)(finish-start);
(26)          delete p;
(27)        }
(28)      }
(29)      for(i=50;i<1050;i++) soma+=duration[i];
(30)      cout << soma/1000 << endl;
(31)      return 0;
(32)    }
```

Figura 8.28: Teste para avaliar o impacto da recomputação optimizada dos componentes.

Rotinas	**Teste**$_A$	**Teste**$_B$	T_B/T_A
Multiplicação de matrizes	9.33333	5.43704	58.25%
Pesquisa lista simples	15.7778	17.1753	108.86%
Pesquisa árvore binária	17.1753	19.7469	114.97%
Autómato	475.778	351.398	73.86%
Dominadores imediatos	56.8889	64.9654	114.20%
Média			94.03%

Tabela 8.1: Resultados dos testes efectuados para avaliar o impacto da recomputação optimizada de componentes para *IDomonator*.

É o contexto no qual o componente é utilizado que vai determinar as vantagens de se utilizar contra-métodos. Supondo que um contra-método leva a executar, em termos médios, um décimo do tempo que é necessário para efectuar a recomputação integral do componente, então a utilização de contra-métodos vai ser prejudicial, se este for executado dez ou mais vezes entre duas utilizações consecutivas do componente.

Por outro lado, a não implementação de contra-métodos leva a que qualquer notificação coloque o estado do componente em OUTDATED, mesmo que a execução do método base, que provoca a notificação, em nada afecte o estado do componente. O que pode ser evitado utilizando um contra-método que simplesmente evite que o componente passe a OUTDATE. Evita-se assim, que posteriormente o componente tenha que ser integralmente recomputado. Mesmo que o método que provocou a notificação do componente, seja executado muitas vezes, o tempo requerido para a execução do contra-método é tão pequeno que compensará a utilização dos contra-métodos.

É ainda de realçar que o exemplo que se utilizou para verificar a eficiência dos contra-métodos (Figura 8.28), é claramente agressivo e pouco favorável. Isto porque inclui essencialmente o componente de análise e um componente de optimização que despoleta um grande número de notificações, as quais requerem por vezes contra-métodos relativamente pesados (ver linhas 18 e 24 da Figura 8.18). Mesmo assim, e como se pode confirmar pela Tabela 8.1, é possível obter resultados interessantes. De notar que em termos médios, houve um ganho de aproximadamente 6%.

É ainda importante realçar que por vezes, a implementação dos contra-métodos num dado componente, requer informação proveniente dos seus componentes de suporte, como aliás acontece entre *IDominator* e *Dominator*. Nestes casos, é fundamental que os componentes de suporte implementem contra-métodos, mesmo que por vezes não sejam muito eficientes. Se tal não acontecer, o componente de suporte será executado integralmente, tantas quantas as vezes forem executados os contra-métodos do componente principal que requerem informação disponibilizada pelo componente de suporte.

Visto isto de uma outra perspectiva, a implementação de contra-métodos em *IDominator* permite que os seus componentes principais (que o utilizem como componente de suporte), implementem contra-métodos. Os quais podem permitir obter resultados bem mais interessantes, como aliás acontece entre *IDominated* e *IDominators*. O que é comprovado pela Tabela 8.2, onde os resultados são sempre positivos, podendo mesmo chegar a optimizar o tempo de execução em mais de 92%.

Convém no entanto relembrar que a utilização dos contra-métodos é muito dependente do tipo de componente, da estrutura do compilador e do contexto de execução. Servem assim estes resultados para demonstrar que é possível obter ganhos significativos, mas de forma alguma para afirmar que a utilização de contra-métodos traduz-se sempre em ganhos

Rotinas	**Teste**$_A$	**Teste**$_B$	T_B/T_A
Multiplicação de matrizes	9.66667	6.77407	70.08%
Pesquisa lista simples	48.8889	19.6543	40.18%
Pesquisa árvore binária	47.77800	23.5309	49.25%
Autómato	5391.86	419.798	7.79%
Dominadores imediatos	247.111	73.7457	29.84%
Média			39.43%

Tabela 8.2: Resultados dos testes efectuados para avaliar o impacto da recomputação optimizada para o componente *IDominated*.

desta ordem, ou mesmo que se traduz sempre em ganhos.

Parte central da arquitectura

Para se efectuar a segunda avaliação, isto é, avaliar o impacto da arquitectura no processo de compilação, sem no entanto fazer uso das optimizações propostas no Capítulo 6 (que visam essencialmente a implementação de contra-métodos), construíram-se dois compiladores: que contêm exactamente o mesmo número de componentes, que são do mesmo tipo e que são executadas pela mesma ordem. Mas conforme já foi dito, o compilador preparado para apurar o valor de referência ($Teste_A$) utiliza exclusivamente componentes da versão original da *framework Dolphin*, isto é, componentes que não fazem uso da arquitectura proposta e que contêm apenas as rotinas essenciais à sua execução. Já no $Teste_B$, todos os componentes implementam a arquitectura proposta sem no entanto fazerem uso dos contra-métodos.

Os testes efectuados consistiram em submeter as rotinas de teste anteriormente apresentadas a ambos compiladores, de forma a obter o tempo de execução e o número de instâncias utilizadas por cada um. Para apurar o número de instâncias bastou efectuar um teste por cada rotina. Já para apurar o tempo médio de execução, foram feitas as 1050 medições por rotina e por compilador, conforme foi anteriormente explicado.

É conveniente salientar que para garantir a fidedignidade dos testes e que efectivamente se estava a avaliar o impacto da arquitectura, houve o cuidado de certificar que as rotinas base, isto é, as rotinas que desempenham as tarefas que se espera que o componente efectue independentemente de estar ou não a utilizar a arquitectura proposta, fossem para componentes do mesmo tipo, exactamente iguais.

A Figura 8.29 representa o teste (compilador) preparado para esta avaliação. A sua estrutura é essencialmente a mesma do compilador construído na Secção 8.2, mas aqui adaptado para se poder executar o processo de compilação várias vezes e se determinar o tempo de execução. Contém um *front-end*, um *back-end* e várias rotinas de optimização, incluindo algumas que funcionam sobre a forma SSA. Implicitamente são inseridos outros componentes conforme se encontra ilustrado na Figura 8.23.

Os resultados apurados encontram-se representados na Tabela 8.3, através da qual é possível constatar que a redução do tempo de compilação pode chegar aos 62% e que mesmo no pior dos casos o ganho é superior a 24%.

Outra vantagem reside no facto de o compilador, que faz uso da arquitectura proposta, apenas requerer 22 instâncias, enquanto que o compilador implementado para determinar os valores de referência acusa a utilização de 37 instâncias.

Em termos médios e de forma geral, o impacto da arquitectura será tanto mais relevante quanto maior o número de componentes utilizados.

```
(1)     ...// Inclusão de ficheiros
(2)     #define REGEXE(a,b) a.setFunction(b);a.execute();
(3)     int main(int argc,char *argv[]){
(4)       int i;
(5)       double duration[1050], soma=0.0;
(6)       clock_t start, finish;
(7)       Program *p;
(8)       Function *f;
(9)       QueueL<Function*> q(0);
(10)      elimJumpChains ejc();
(11)      elimUnreachCode euc();
(12)      cnv2SSA cs();
(13)      elimComSubExpr ecse();
(14)      elimLoads el();
(15)      cnv2NF cn();
(16)      genPseudoCode ps();
(17)      littleC fe(argc,argv);
(18)      for(i=0;i<1050;i++){
(19)        start=clock();
(20)        fe.execute();
(21)        if(p=fe.getProgram()){
(22)          p->buildQueue(&q);
(23)          for(f=q.dequeue();f;f=q.dequeue()){
(24)            REGEXE(ejc,f);
(25)            REGEXE(euc,f);
(26)            cs.setFunction(f);
(27)            if(cs.execute()){
(28)              REGEXE(ecse,f);
(29)              REGEXE(el,f);
(30)              REGEXE(cn,f);
(31)            }
(32)            REGEXE(ps,f);
(33)          }
(34)          delete p;
(35)        }
(36)        finish=clock();
(37)        duration[i]=(double)(finish-start);
(38)      }
(39)      for(i=50;i<1050;i++) soma+=duration[i];
(40)      cout << soma/1000 << endl;
(41)      return 0;
(42)    }
```

Figura 8.29: Compilador construído para avaliar o impacto da arquitectura.

Rotinas	**Teste**$_A$	**Teste**$_B$	T_B/T_A
Multiplicação de matrizes	64.1429	48.2857	75.28%
Pesquisa lista simples	98.1429	60.2857	61.47%
Pesquisa árvore binária	117	69.8571	59.71%
Autómato	2647.06	993.857	37.55%
Dominadores imediatos	499.857	291.571	58.33%
Média			58.47%

Tabela 8.3: Resultados dos testes efectuados para avaliar o impacto da arquitectura sem os mecanismo de recomputação optimizada de componentes.

Arquitectura: utilização integral

Os testes realizados para avaliar o impacto da utilização global da arquitectura são muito semelhantes aos da avaliação anterior, com a pequena diferença de que alguns componentes do $Teste_B$ implementam contra-métodos. No entanto os componentes que o fazem são do tipo *Análise* e como tal incluídos implicitamente. São eles os seguintes: *Dominators*, *IDominator*, *IDominated* e *Dominated*.

Os resultados encontram-se representados na Tabela 8.4, que novamente confirmam as vantagens da utilização da arquitectura. Não diferem no entanto de forma muito significativa dos valores apurados na avaliação anterior. O que é em parte justificável, dado que são poucos os componentes utilizados que implementam contra-métodos. Na realidade é até um pouco injusto dizer que se está a utilizar integralmente a arquitectura, mas infelizmente, por falta de tempo, não foi possível desenvolver contra-métodos para todos os componentes que podem tirar partido desta solução.

Rotinas	**Teste**$_A$	**Teste**$_B$	T_B/T_A
Multiplicação de matrizes	64.1429	44.5714	69.49%
Pesquisa lista simples	98.1429	56.7143	57.79%
Pesquisa árvore binária	117	69.8571	59.71%
Autómato	2647.06	982.143	37.10%
Dominadores imediatos	499.857	291	58.22%
Média			56.46%

Tabela 8.4: Resultados dos testes efectuados para avaliar o impacto da utilização integral da arquitectura.

Acresce ainda que na realidade, devido à estrutura dos compiladores de teste, são poucos os componentes reutilizados. Por exemplo, o componente que faz a conversão para a forma SSA é dos que mais partido tira da arquitectura, no entanto finda a sua execução e apesar de todos os componentes de suporte estarem actualizados, poucos são os que vão ser posteriormente reutilizados. Aliás, a única excepção é *Dominators*.

8.4 Resumo do capítulo

Conclui-se assim este capítulo, esperando que o seu conteúdo tenha contribuído para clarificar a utilização das soluções apresentadas, nomeadamente no desenvolvimento de componentes e de compiladores. Espera-se também que tenha servido para demonstrar algumas

das vantagens de se fazer uso dessas soluções.

CAPÍTULO 9

Sistema Dolphin

Índice

O *Sistema Dolphin* é um projecto que surge no seguimento do trabalho até aqui apresentado e que visa disponibilizar via Web um conjunto diversificado de serviços relacionados com o desenvolvimento de compiladores e, de forma geral, de software. O desenvolvimento deste sistema, que se encontra ainda numa fase muito inicial, é o que se pode designar por trabalho futuro, que apesar de extravasar muito os limites deste doutoramento é fundamental para melhor se compreender a motivação que levou à concepção e implementação da arquitectura e para mostrar o potencial da *framework Dolphin*.

Como mais adiante se mostra, a *framework Dolphin* é parte fundamental do *Sistema Dolphin*. Entre ambos existe uma forte relação de simbiose, o primeiro sustenta o segundo,

e o segundo promove o desenvolvimento do primeiro. Aliás, foi neste sentido, de promover o desenvolvimento da *framework Dolphin*, que surgiu a ideia de desenvolver o *Sistema Dolphin*.

Convém aqui relembrar, que a *framework Dolphin* surgiu como um pequeno projecto desenvolvido para colocar em prática e testar as soluções idealizadas para satisfazer os objectivos deste doutoramento. Serviu assim (e continua a servir) como uma *test-bed* caseira para fins de investigação, mas também para fins pedagógicos. No entanto, a sua afirmação, quer como ferramenta pedagógica, quer como ferramenta de investigação, mas principalmente como opção credível para desenvolvimento de compiladores, requer a implementação de um vasto conjunto de componentes e também de algumas ferramentas de suporte. Para tal, são necessários conhecimentos e mão-de-obra que estão muito para lá dos recursos afectos a este projecto.

A solução encontrada para ultrapassar estes obstáculos passou por atrair recursos externos, que ao tirarem proveito da *framework Dolphin* como ferramenta pedagógica, de investigação e de desenvolvimento de compiladores, contribuíssem também para o seu crescimento. Para tal, era fundamental exteriorizar o projecto, colocando-o visível para os potenciais utilizadores. Claro está, que a via natural para o fazer é através da Web, mas não bastava desenvolver um simples site Web. Era necessário disponibilizar todo um conjunto de facilidades, que realmente tornassem apetecível a utilização de *framework Dolphin*, mas que também levassem os utilizadores a contribuir para o seu desenvolvimento. É dentro deste contexto que surge o *Sistema Dolphin* que, à excepção de algumas ferramentas e da própria *framework Dolphin*, está em fase de desenvolvimento.

No entanto as expectativas criadas à volta deste projecto são bastante grandes, ao ponto de se ter tornado o principal foco de atenção, em detrimento da própria *framework Dolphin*. É que ao fazer uso das novas tecnologias Web, o *Sistema Dolphin* vem estimular o desenvolvimento desta área científica que estava algo esmorecida. Uma vez que se aproxima dos eventuais interessados e injecta alguma da dinâmica, atracção e motivação que está inerente ao fenómeno Web.

O *Sistema Dolphin* está, em termos de investigação, a abrir novas portas que apesar de não estarem directamente relacionados com o desenvolvimento de compiladores, são bastante promissoras. É o caso do desenvolvimento de ambientes cooperativos, mais especificamente de laboratórios virtuais, em que o *Sistema Dolphin* é, pelo tipo de recursos que envolve, um excelente *case-study*.

9.1 A arquitectura do *Sistema Dolphin*

O *Sistema Dolphin*, cuja representação gráfica se encontra na Figura 9.1, é constituído por vários sub-projectos, nos quais se inclui a própria *framework Dolphin*. Esta ocupa uma posição nuclear no sistema e é sobre ela que vão funcionar a grande maioria das aplicações.

O *Sistema Dolphin* foi concebido tendo em conta as diversas etapas envolvidas no desenvolvimento de compiladores, que vão desde o desenvolvimento de componentes; passando pela sua aplicação, isto é, pela construção de compiladores; e terminando na utilização destes, que também poderá ser entendida como fase de testes. Entendeu-se que a cada uma destas etapas está associado um utilizador tipo, isto é, o construtor de componentes, o construtor de compiladores e o utilizador de compiladores. De forma a disponibilizar todas as funcionalidades necessárias a cada tipo de utilizador, estabeleceram-se três sub-projectos:

- O *Dolphin*-Compiler Components Development System;
- O *Dolphin*-Compilers Development System;

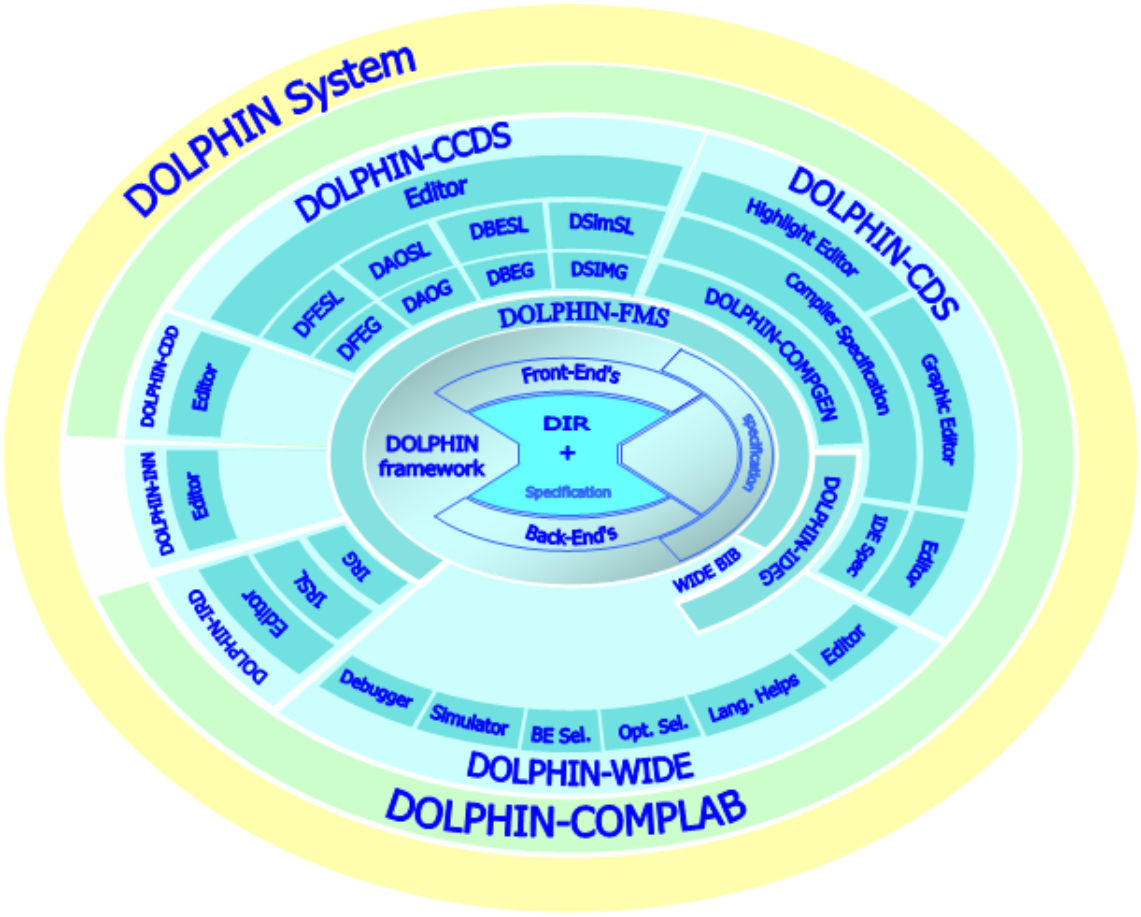


Figura 9.1: Arquitectura do *Sistema Dolphin*.

- E o *Dolphin*-Web Integrated Development Environment.

9.1.1 *Dolphin*-Compiler Components Development System

O *Dolphin-Compiler Components Development System* (*Dolphin-CCDS*) visa a construção de um ambiente integrado de desenvolvimento (IDE) que permita aos utilizadores construir novos componentes. Para além do editor, do sistema de ajuda, do sistema de interacção com a *framework Dolphin*, o IDE deverá dar acesso às ferramentas específicas para a construção de componentes. É como tal, o projecto que também visa promover o desenvolvimento/integração de ferramentas de geração na *framework Dolphin*.

O *Dolphin-CCDS* contempla assim o desenvolvimento ou integração/adaptação de uma ferramenta de geração de *front-ends*, designada por *Dolphin-Front-End Generator* (*DFEG*), para a construção das rotinas de análise léxica, sintáctica, semântica e de geração de código intermédio. Algo semelhante ao sistema Eli [GHK+90], mas aqui devidamente adaptado de forma a fazer uso directo da Dolphin Internal Representation (DIR). De preferência, a especificação das tarefas de *front-end* deverá ser feita utilizando uma única linguagem, a que se designou por *Dolphin-Front-End Specification Language* (*DFESL*).

Pretende-se também fazer uso da tecnologia utilizada na construção do Program Analyzer Generator-PAG [AM95, NNH99], do OPTIMIX [Aßm00] e de outros sistemas de reescrita, como é o caso do Bottom-Up Rewrite Generator-BURG [FHP91], para disponibilizar uma

solução integrada para o desenvolvimento dos componentes. Nomeadamente componentes de análise, optimização e de suporte às tarefas de *back-end*. De preferência permitindo que os utilizadores possam especificar os vários componentes através de uma única linguagem, o *Dolphin-Analysis and Optimizations Specification Language* (*DAOSL*). A ferramenta de geração, designada por *Dolphin-Analysis and Optimizations Generator* (*DAOG*), processará a especificação e gerará os componentes em conformidade com a arquitectura que é proposta nesta dissertação.

Por último, o *Dolphin-CCDS* deverá ainda incluir uma ferramenta para construção de *back-ends*, que permita a geração das rotinas de atribuição de registos, selecção de instruções, geração de código *assembly*/binário e rotinas de optimização de baixo nível. Esta ferramenta, designada por *Dolphin-Back-End Generator* (*DBEG*), deverá ser desenvolvida tendo por base o trabalho já realizado no Back-End Development System [Mat02], que faz uso da tecnologia aplicada no Improved Bottom-Up Rewrite Generator-IBURG [FHP92] e no *NJMCT* [RF95]. Também neste caso se pretende que a especificação das várias tarefas de *back-end*, seja feita com base numa única linguagem, o *Dolphin-Back-End Specification Language* (*DBESL*).

Claro está que para além destas ferramentas, o *Dolphin-CCDS* deverá também permitir desenvolver componentes através de implementação directa, o que é designado por *Dolphin-Components Direct Development*.

9.1.2 *Dolphin*-Compilers Development System

O *Dolphin-Compilers Development System* (*Dolphin-CDS*) é o projecto que visa o desenvolvimento de um IDE específico para a construção de compiladores, que contém essencialmente dois elementos: o editor e o gerador de compiladores, designado por *Dolphin-COMPilers GENerator* (*Dolphin-COMPGEN*) .

No caso do editor, pretende-se disponibilizar duas alternativas: um editor textual, do tipo *highlight*, eventualmente integrado com um sistema de navegação nos componentes; e um editor gráfico, em que o utilizador pode "desenhar" o compilador e ajustar os parâmetros dos componentes. Na realidade o editor gráfico deverá funcionar como um *front-end* para o editor textual, que como tal deverá integrar um conversor da representação gráfica para representação textual (e vice-versa).

A especificação dos compiladores, é actualmente feita recorrendo à própria linguagem na qual se encontra implementada a *framework Dolphin*, isto é, C++. Pelo que a geração dos compiladores mais não é do que efectuar a compilação dos componentes utilizados. Ou seja, o editor textual mais não é que um editor de C++.

Encontra-se no entanto em estudo a possibilidade de desenvolver uma linguagem própria para a especificação dos compiladores, que permita lidar com os componentes, mas que também permita tornar a estrutura dos compiladores mais flexível, por exemplo, activar/desactivar optimizações de código, seleccionar o tipo de *back-end* ou mesmo de *front-end*, etc. Se se confirmar a necessidade de desenvolver esta linguagem, será também necessário construir o gerador de compiladores (*Dolphin-COMPGEN*), que faça a conversão da especificação para C++ e que posteriormente invoque o compilador+*linker*.

Convém apenas realçar que foi feito um esforço, quer no sentido de redefinir os operadores nativos da linguagem C++, quer no sentido de simplificar as interfaces (parte que é visível aos utilizadores), para que a especificação dos compiladores utilizando o C++ fosse simples e eficiente.

9.1.3 *Dolphin-Web Integrated Development Environment*

O *Dolphin*-Web Integrated Development Environment (*Dolphin*-WIDE) é o projecto que visa disponibilizar IDEs para a utilização dos compiladores. Os quais poderão ser empregues como típicos ambientes de desenvolvimento de software, mas em que o principal objectivo é servirem para testar o funcionamento dos compiladores desenvolvidos através da *framework Dolphin*.

Este é um projecto bastante ambicioso. Em primeiro lugar porque a ideia é que os IDEs sejam gerados automaticamente a partir da especificação dos compiladores. Segundo porque cada IDE deverá disponibilizar (dentro dos possíveis) um vasto conjunto de funcionalidades, nomeadamente: acesso ao compilador construído através do *Dolphin-CDS*; um editor específico para o tipo de *front-end* utilizado no compilador; ajudas sobre o IDE, sobre a linguagem do *front-end* e sobre o próprio compilador (estrutura, opções de compilação, etc); e soluções gráficas para seleccionar as opções de compilação.

Pretende-se ir ainda mais longe, os IDEs devem incluir um simulador da arquitectura de computação, gerado a partir da informação associada ao *back-end*; e um *debugger*, que de princípio será genérico (único para todos os IDEs).

Para que tudo isto seja possível, entendeu-se associar a cada componente e classe da DIR, uma descrição feita em XML com meta-informação. Os próprios compiladores deverão conter uma descrição, que integra as descrições dos componentes que utiliza.

O *Schema* que valida a informação associada às classes da DIR já está especificado e também já existe um *Schema*, que deverá funcionar para a grande maioria dos componentes, à excepção dos *front-ends* e *back-ends*, que deverão incluir informação sobre a linguagem a compilar e sobre a arquitectura de computação.

A própria DIR, enquanto estrutura que integra várias classes, pode ser representada em XML. A *framework Dolphin* inclui mesmo um *back-end*, designado por *genXML*, capaz de gerar XML a partir da RIC. Encontra-se também em desenvolvimento um componente para converter XML para a RIC utilizada pela *framework Dolphin*, possibilitando assim o desenvolvimento de componentes sobre XML.

Através do *Dolphin-CDS* o utilizador terá oportunidade de efectuar alguns ajustes à descrição utilizada para gerar o IDE, nomeadamente relacionados com a própria estrutura do compilador, com aspectos gráficos do IDE, com a selecção das opções que devem estar visíveis aos utilizadores do IDE, etc.

A descrição será depois submetida, ao que se designou por *Dolphin-IDE Generator* (*Dolphin-IDEG*), que dentro dos possíveis deverá gerar todos os elementos necessários ao IDE.

Existe actualmente um protótipo construído manualmente, do tipo de IDE que se pretende gerar (sem simulador, nem *debugger*). O qual funciona e deverá funcionar cada vez mais como uma espécie de *show-case* da tecnologia existente na *framework Dolphin*.

9.2 *Dolphin-COMPilers LABoratory*

O três projectos apresentados até aqui, cobrem as principais fases do processo de desenvolvimento de componentes: a fase de implementação propriamente dita, que é feita através do *Dolphin-CCDS*; e a fase de teste e avaliação dos componentes. Esta última é efectuada construindo compiladores com os componentes entretanto desenvolvidos, utilizando para tal o *Dolphin-CDS*. Posteriormente, submetem-se os compiladores a vários testes, o que significa colocá-los a compilar várias rotinas de código. É nesta última fase do processo que entra o *Dolphin-WIDE*.

Com o objectivo de integrar sob uma única interface todos estes serviços e outros, e simultaneamente disponibilizar os meios necessários ao desenvolvimento cooperativo dos componentes/compiladores, surgiu o projecto *Dolphin-COMPilers LABoratory*, cuja arquitectura se encontra representada na Figura 9.2.

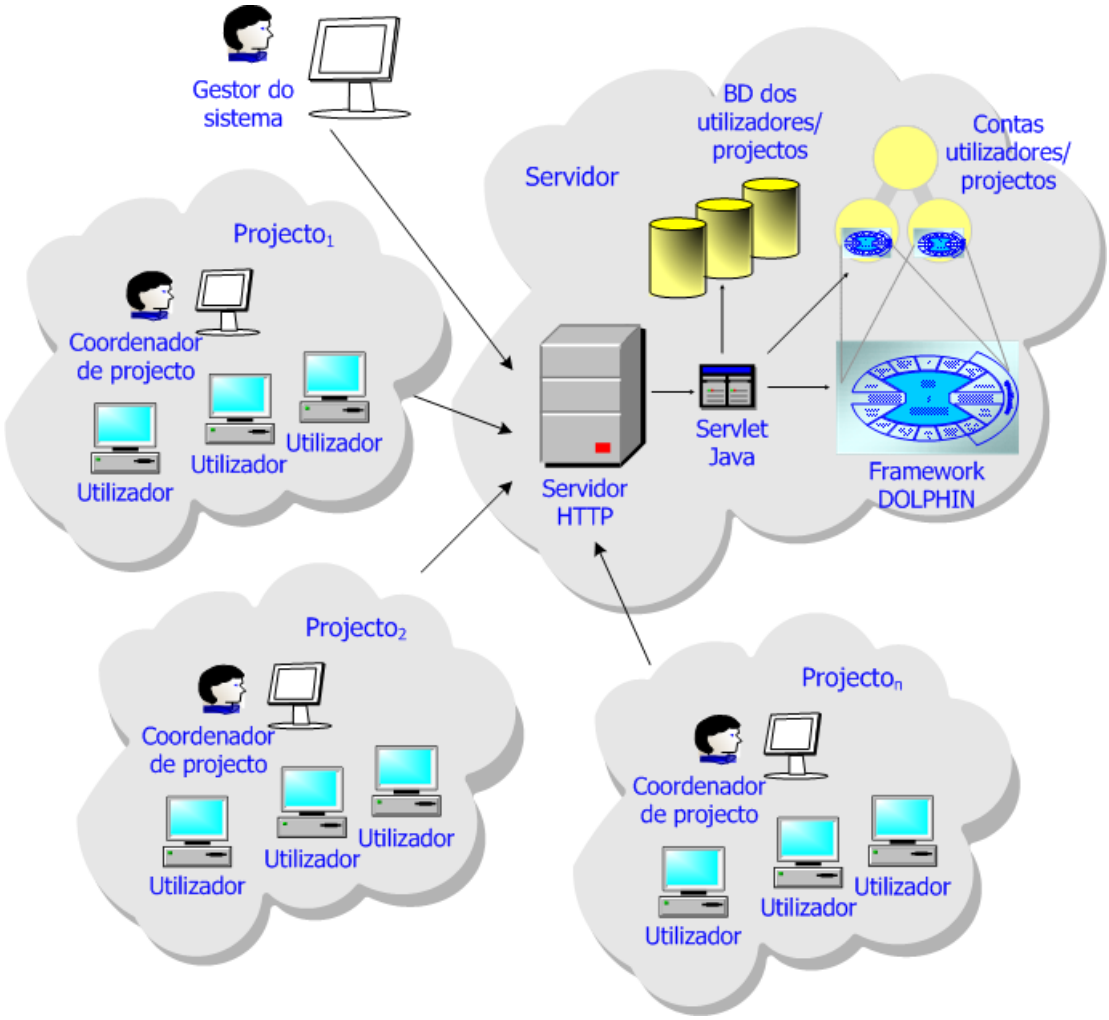


Figura 9.2: Arquitectura do *Dolphin-COMPLAB*.

A ideia base é permitir aos utilizadores, sejam eles estudantes, professores, investigadores ou profissionais, poderem desenvolver os seus projectos com todos os recursos necessários, nomeadamente fazendo uso de soluções especificamente desenvolvidas para trabalho cooperativo.

Para tal, cada utilizador deverá estar registado no sistema, o que lhe dará acesso a uma conta pessoal e à possibilidade de requerer a abertura de projectos ou a integrar projectos já existentes.

A requisição de novos projectos, será feita automaticamente pelo utilizador que entender ser o coordenador do projecto. Cabe este escolher o tipo de recursos disponíveis no projecto, nomeadamente ferramentas, componentes e outras funcionalidades específicas do *Dolphin-COMPLAB*.

A criação de um novo projecto, dará origem à abertura de uma conta própria para o projecto, que apesar de ser gerida pelo coordenador, será partilhada por todos os elementos

que integram o projecto. Nessa conta será depositado automaticamente uma cópia da *framework Dolphin*. É sobre essa cópia que serão realizados todos os testes e sobre a qual se desenvolverá o projecto.

A adesão de novos elementos a um projecto, é feita pelo coordenador de projecto, o qual será também responsável por definir o tipo de privilégios de cada elemento. Por exemplo, que ferramentas pode utilizar, que componentes estarão visíveis, se pode ou não requerer a integração de novos componentes na *framework* local, etc.

De notar que a possibilidade do coordenador controlar os privilégios dos demais elementos do projecto, foi pensada essencialmente para utilização com fins pedagógicos do *Dolphin-COMPLAB*. Em que o papel de coordenador do projecto caberá ao docente, sendo os demais utilizadores os discentes. Aliás, a utilização do *Dolphin-COMPLAB* para fins pedagógicos, levou a idealizar várias outras funcionalidades, como por exemplo a integração de um diário de actividades, a integração de um sistema de planeamento por projecto e por utilizador, a calendarização e posterior execução de eventos (por exemplo, para efectuar testes à *framework* local), etc.

Como já foi dito, cada utilizador terá uma conta pessoal, mas terá também acesso às contas dos projectos do qual faz parte. Um utilizador pode assim manter os seus próprios trabalhos ou partilhá-los com os restantes elementos do projecto. Aliás, no sentido de partilhar informação, estão perspectivadas várias funcionalidades para o *Dolphin-COMPLAB* que permitirão melhorar a comunicação e a troca de experiências entre elementos. Por exemplo, a integração de um sistema de conversação (*chat*), a utilização de listas de e-mail, o já referido diário de actividades (com o registo das operações efectuadas sobre as partes comuns do projecto); um *dash-board* para afixar mensagens; etc.

Caberá ao coordenador do projecto manter o diálogo com o sistema central, por exemplo, para requerer novos componentes, ou pedir a actualização dos componentes que já possua. Será também o coordenador de projecto, o único elemento que terá privilégios para submeter novos componentes ao sistema central. Está previsto que esta operação seja efectuada de forma completamente automática. Para tal, o componente submetido à *framework* central, deverá passar por uma série de testes de compatibilidade e de avaliação de desempenho. Se ficar aprovado nesses testes, entra na fase *beta*, ficando disponível para os demais utilizadores (desse e dos outros projectos) durante um período pré-determinado, que servirá para o componente ser testado por todos os utilizadores do sistema. Se após este período, não for detectado qualquer problema no componente, será então integrado de forma definitiva na *framework* central.

Para além das ferramentas e soluções dos projectos *Dolphin-CCDS*, *Dolphin-CDS* e *Dolphin-WIDE*, o *Dolphin-COMPLAB* deverá ainda disponibilizar outras ferramentas/soluções que apenas têm razão de ser quando se olha para o desenvolvimento de componentes de forma integrada. É o caso do *Dolphin-Monitor and Analyzer System* [MH03b, MH03g, MH03f], que permitirá observar graficamente a evolução do processo de compilação (uma espécie de *debugger* gráfico); ou dos vários testes de *benchmark* a desenvolver, para se efectuar a avaliação dos compiladores e, indirectamente, dos componentes.

Espera-se que a implementação de todas as funcionalidades até aqui descritas, permitam fazer do *Dolphin-COMPLAB* uma excelente solução para quem eventualmente tenha que lidar com o desenvolvimento de componentes/compiladores, seja aluno, docente, investigador ou profissional.

9.3 Outros componentes e projectos

9.3.1 *Dolphin-Framework Management System*

Dado que nos vários projectos apresentados até aqui, existem diversos intervenientes que necessitam de interactuar com a *framework Dolphin*, de formas e com objectivos muito distintos, considerou-se importante estabelecer um protocolo que defina quem pode e como deve fazer para aceder à *framework*. A implementação desse protocolo, que ainda não está definido, passará em parte por um sistema de gestão da *framework* (versão central), designado por *Dolphin-Framework Management System*. Será através deste sistema que deverão ocorrer todos os acessos à *framework*.

9.3.2 *Dolphin-Intermediate code Representation Definition*

A DIR é um modelo de representação de código bastante flexível, poderoso e com muito potencial. No entanto apenas foi utilizado para representar código obtido da compilação de linguagens imperativas. Não que isto seja particularmente grave, dado que ao longo do processo de compilação, a representação do código deverá tender necessariamente para este tipo de paradigma, uma vez que é o paradigma utilizado no código final do processo de compilação, nomeadamente se for assembly ou código máquina.

Pode no entanto acontecer que alguém queira utilizar a *framework Dolphin*, não para construir especificamente compiladores, mas conversores/tradutores. Pode mesmo acontecer que a DIR, no seu estado actual, não reúna as condições necessárias para representar código proveniente de linguagens que sigam outros paradigmas ou mesmo formas mais complexas do paradigma imperativo, como é o caso das linguagens orientadas por objectos.

Foi neste sentido, mas também para dar um carácter ainda mais inovador ao *Sistema Dolphin*, que se entendeu incluir um projecto, designado por *Dolphin-Intermediate code Representation Definition* (DIRD), para analisar até que ponto a DIR se adequa aos demais paradigmas e, caso se constate que não serve, que tipo de soluções se poderão disponibilizar aos utilizadores para colmatar esta lacuna. Se possível, tentar criar uma solução que permita modelar a DIR em conformidade com as necessidade de cada utilizador, sem no entanto inviabilizar a utilização dos componentes que já integram a *framework Dolphin*.

Dentro deste projecto, pretende-se ainda analisar a viabilidade de utilizar a DIR, ou a solução entretanto desenvolvida, para representar problemas de outras áreas. O que permitirá utilizar as soluções desenvolvidas para o *Sistema Dolphin* e os componentes construídos para a *framework Dolphin*, na resolução desses problemas. Por exemplo, existem boas perspectivas para a aplicação das técnicas de análise de fluxo de dados em problemas de suporte à decisão; ou a aplicação de técnicas de *scheduling* e de paralelização de código na resolução de problemas de planeamento. O que iria permitir abrir novos campos de intervenção e dar outra dimensão ao trabalho descrito nesta dissertação.

9.3.3 *Dolphin-INNovation*

Por último, e acreditando que o *Sistema Dolphin* poderá albergar e apadrinhar vários projectos de investigação, decidiu-se incluir um projecto que visa analisar a viabilidade de utilizar tecnologias de outras áreas, nomeadamente de inteligência artificial, na implementação de algumas tarefas do processo de compilação.

As soluções utilizadas actualmente na resolução dos problemas de compilação são essencialmente determinísticas e estáticas. Pretende-se neste projecto averiguar a viabilidade de utilizar tecnologias mais versáteis e com capacidades adaptativas, como é o caso dos agen-

tes e das redes neuronais, para resolver problemas como por exemplo, o da atribuição de registos, a selecção de instruções, e mesmo algumas formas de optimização de código.

9.4 Resumo do capítulo

Conclui-se assim este capítulo, que deve ser visto antes de mais como uma descrição relativamente detalhada do que poderá ser o trabalho futuro a realizar no seguimento deste doutoramento. Em que o futuro já começou. Por exemplo:

- Já existe um protótipo do IDE que se pretende utilizar no projecto *Dolphin-WIDE* [MH03e];
- A ferramenta de monitorização (*Dolphin-MAS*) também já se encontra implementada [MH03g, MH03b];
- Já foram efectuados alguns estudos sobre a utilização da DIR na representação de problemas de outras áreas [MH03c, MH03a];
- O BEDS [Mat99, Mat02] apenas requer alguns pequenos ajustes (adaptações necessárias à nova arquitectura) para poder funcionar como gerador de *back-ends* (projecto *Dolphin*-CCDS).

É claro que o trabalho que falta fazer é muito mais do que aquele que está feito e que os objectivos são muito ambiciosos.

CAPÍTULO 10

Conclusão

Índice

Este doutoramento iniciou-se com um objectivo muito claro e concreto: contribuir para tornar o processo de construção de compiladores mais simples. É claro que o caminho a seguir, como acontece na grande maioria dos trabalhos de investigação, é quase sempre uma incógnita. Conhece-se o ponto de partida, por vezes o ponto de chegada, e pouco mais. Desenvolver um trabalho conducente à preparação de uma dissertação de doutoramento é em grande parte construir, num processo gradual e efectuado com base em tentativas, o caminho que liga esses dois pontos.

Assim aconteceu com o trabalho realizado neste doutoramento. O objectivo era claro mas logo à partida se sabia que havia várias alternativas por onde avançar. Podia passar pela investigação de tarefas específicas do processo de compilação, analisando assim a viabilidade de desenvolver ferramentas para a sua construção; simplificar a utilização de ferramentas já existentes; ou integrar várias ferramentas/sistemas e assim obter soluções mais abrangentes e homogéneas.

Estas alternativas investiam em lacunas que de alguma forma estavam por preencher, ou em trabalho já realizado, melhorando-o ou combinando-o, no sentido de obter mais valias que fossem de encontro aos objectivos propostos.

Neste sentido, deu-se início ao desenvolvimento da *framework Dolphin*, mas foi com alguma surpresa, que se detectou que a melhor contribuição que podia resultar deste douto-

ramento, não residia em atacar a construção de tarefas pontuais do processo de compilação, ou tentar obter uma solução que suportasse o desenvolvimento do maior número de tarefas, mas tão só e simplesmente idealizar, segundo os objectivos estabelecidos para este doutoramento, a solução mais adequada à construção de compiladores. Assim, com recurso a técnicas de engenharia relativamente simples, concebeu-se e implementou-se uma arquitectura para esta solução, dando uma especial ênfase a dois aspectos muito concretos: à construção dos componentes para compiladores; e à reutilização desses componentes.

Pensou-se inicialmente, que este era um trabalho relacionado com tecnologias da compilação. No entanto, a solução obtida é essencialmente um trabalho de engenharia, mas por se tratar de uma solução de base, que estabelece um modelo de compilação, que define claramente o tipo de entidades envolvidas e a forma como estas se relacionam, com todas as vantagens que daí advêm, resulta numa contribuição bastante significativa para a área científica de tecnologias da compilação. De notar que, dificilmente as alternativas inicialmente propostas resultariam numa solução tão abrangente.

10.1 Contribuições

A principal contribuição que resulta deste doutoramento, consiste numa arquitectura, que define: o tipo de entidades que estão subjacentes a um determinado modelo de compilação; e a forma como essas entidades se relacionam entre si. Arquitectura esta que é genérica o suficiente para poder ser aplicada a outros sistemas, mesmo que em áreas não afins à compilação, desde que visem o desenvolvimento de aplicações que façam uso de um modelo semelhante. Significa isto que as vantagens que advêm da utilização desta arquitectura, não se restringem ao sistema sobre o qual foi testada (*framework Dolphin*).

É pelo facto de ser uma solução genérica e de contribuir para simplificar a reutilização de componentes, garantindo simultaneamente a qualidade do processo (de compilação), que esta arquitectura é a maior contribuição deste doutoramento. Foram mesmo identificados, ao longo desta dissertação, vários sistemas sobejamente conhecidos que poderão obter benefícios significativos com a utilização desta arquitectura.

A sua utilização visa essencialmente simplificar a reutilização dos componentes, garantindo:

- A conectividade entre componentes;
- A simplicidade da inclusão implícita e a eficiência da inclusão explícita;
- A consistência entre componentes e o repositório de dados (RIC);
- A recomputação optimizada dos componentes.

Conforme se pretendeu provar nesta dissertação, a aplicação desta arquitectura, comparativamente com as demais soluções, simplifica a reutilização dos componentes e optimiza o tempo de execução (por vezes com reduções superiores a 50%).

É também importante realçar que esta arquitectura é simples, inteligível, fácil de utilizar e pode ser aplicada a posteriori, não requerendo alterações de fundo nos sistemas que a implementam. Acresce ainda que, pelo simples facto de fazer uso de um modelo (de compilação) e de definir de forma clara o tipo e o relacionamento das entidades que compõem esse modelo, promove:

- O desenvolvimento de componentes mais normalizados, mais reutilizáveis, que são mais facilmente integrados entre si, e que têm menores custos de manutenção e de actualização;

- A homogeneidade do sistema que implementa a arquitectura, do processo de implementação e de reutilização dos componentes, e da estrutura das aplicações;

Convém realçar que a aplicação desta arquitectura, ao caso concreto da construção de compiladores, não afecta a qualidade do código por estes gerados.

O próprio modelo que é empregue é simples, modular, escalável e suficientemente genérico para poder ser utilizado por um grande número de aplicações, em especial compiladores.

A aplicação deste modelo, e mais genericamente a aplicação da arquitectura, à construção de compiladores, é complementada com a utilização de um modelo de Representação Intermédia do Código (RIC), que por ter um papel fundamental na construção de novos componentes foi também alvo de investigação, do que resultou a segunda contribuição deste doutoramento, a Dolphin Internal Representation (DIR).

À semelhança de outros modelos, a DIR é genérica e flexível, mas não só. É comparável aos melhores modelos, disponibilizando diversos níveis de abstracção e mecanismos de consistência. Vai mesmo um pouco mais longe no que diz respeito à facilidade de utilização, quer na perspectiva da construção da RIC, quer na perspectiva da implementação de novos componentes. A DIR ao introduzir o conceito de *vistas*, como camadas complementares que reformulam a perspectiva que o utilizador tem da RIC, permite:

- Aproximar a DIR aos demais modelos. O que por si só, contribui para:
  - Aproximar a DIR dos utilizadores, facultando vistas próximas dos modelos utilizadas por estes;
  - Abrir caminho para reutilizar componentes provenientes de outros sistemas;
- Filtrar a RIC, disponibilizando aos utilizadores apenas as partes de que estes necessitam (simplifica a utilização da RIC);
- Acrescentar níveis de abstracção mais adequados às necessidades dos utilizadores.

É também importante fazer referência à *framework Dolphin*, que apesar de não ser uma contribuição científica, é a parte "visível" do trabalho realizado ao longo deste doutoramento e que tem a particularidade de incluir a arquitectura proposta como sendo apenas mais um dos protocolos que são disponibilizados por este sistema. É igualmente importante fazer referência ao *Sistema Dolphin*, que apesar de estar numa fase embrionária, deve no entanto a sua concepção ao trabalho efectuado neste doutoramento.

10.2 Análise crítica

Os aspectos menos conseguidos deste doutoramento prendem-se quase todos com a argumentação, principalmente no que diz respeito ao modelo de RIC que é proposto. O que em parte levou a fazer deste, uma contribuição menor deste doutoramento. A avaliação de um modelo de RIC é algo de muito subjectivo e que só pode ser feita com alguma seriedade, se os utilizadores tiverem experiência em construção de compiladores e se tiverem trabalhado com outros modelos de RIC. Reunir um conjunto de pessoas com este perfil e com a disponibilidade necessária, é algo que não é fácil. Apesar desta falha ter sido em parte preenchida, validando o modelo através de publicações [MH04, MH03h], fica no entanto a sensação de que a descrição feita da DIR e a sua adequação à implementação de novos componentes, não faz jus ao trabalho desenvolvido.

Sobre a escolha do modelo de compilação e da solução utilizada para a construção dos compiladores, existe a convicção de que são realmente as melhores opções. Apesar de nunca ter sido objectivo deste doutoramento, seria interessante ter-se demonstrado a adequação desta solução na integração de ferramentas de geração.

Em relação à arquitectura, os resultados excederam as expectativas iniciais. No que diz respeito à facilidade de reutilização dos componentes, os resultados já eram previsíveis, até porque o objectivo principal era garantir que a reutilização fosse tão simples como quando os componentes são utilizados implicitamente. Foi no entanto com alguma surpresa que se constatou que a optimização do processo de compilação, que resulta do simples facto de se minimizar o número de componentes utilizados, apresentava resultados muito relevantes. A maior surpresa adveio no entanto dos mecanismos de recomputação optimizada, quando se constatou que também permitiam obter melhorias muito significativas. Tudo isto com uma simplicidade notável.

Apesar de se ter conseguido quantificar a contribuição da arquitectura para os objectivos propostos, seria no entanto interessante analisar de uma forma pragmática e segundo a perspectiva dos utilizadores, até que ponto as soluções propostas afectam o processo de construção de compiladores.

10.3 Trabalho futuro

O trabalho descrito nesta dissertação encontra-se inacabado em muitos aspectos. No que diz respeito ao modelo de RIC proposto, seria conveniente testar a sua adequação na representação de código proveniente de outras linguagens, de preferência que fizessem uso de outros paradigmas que não o imperativo. Seria igualmente interessante analisar até que ponto a DIR se adequa à representação de problemas de outras áreas científicas, abrindo assim caminho para se aplicar as soluções utilizadas nos compiladores, na resolução desses problemas. O que foi aliás umas das vertentes que se chegou a explorar durante a realização deste doutoramento [MH03c, MH03a]. No seguimento desta ideia, seria ainda interessante averiguar até que ponto a DIR pode ser alterada para suportar sincronismo, concorrência, eventos, etc.

Em relação à arquitectura ficaram em aberto algumas oportunidades para melhoramentos, nomeadamente no que diz respeito à geração de relatórios. Seria desejável, obter uma solução mais eficiente para efectuar a "ligação" entre os elementos que geram os relatórios e os componentes que deles fazem uso.

No que diz respeito à *framework Dolphin*, esta só será considerada uma solução credível quando disponibilizar um número significativo e diversificado de componentes, que permita efectivamente construir compiladores para as linguagens e arquitecturas mais comuns. Pelo que neste sentido há imenso trabalho a desenvolver.

A grande maioria dos tópicos apresentados para trabalho futuro, estão no entanto incluídos num único projecto, o *Sistema Dolphin*, que visa disponibilizar via Web um conjunto de recursos que abrangem todo o processo de construção de compiladores: desde a implementação dos componentes; passando pela construção propriamente dita dos compiladores; até ao desenvolvimento dos ambientes de teste e de utilização (IDEs). Será através da concretização deste projecto, que se poderá provar a mais valia das soluções apresentadas nesta dissertação e passar de um trabalho de investigação a uma solução de carácter prático e verdadeiramente utilizável. O Capítulo 9 descreve detalhadamente os objectivos deste sistema, identificando as partes que já estão desenvolvidas e aquelas que fazem efectivamente parte do trabalho futuro.

Bibliografia

[ADH+00a] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C.Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Computer System Laboratory, University of Stanford, Portland, August 2000.

[ADH+00b] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. The basic SUIF programming guide. Technical report, Computer System Laboratory, University of Stanford, Portland, August 2000.

[ADH+00c] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. The SUIF program representation. Technical report, Computer System Laboratory, University of Stanford, Portland, August 2000.

[ADR98] A. Appel, J. Davidson, and N. Ramsey. The Zephyr compiler infrastructure. Technical report, University of Virginia, University of Princeton, 1998.

[AGT89] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 4:491–516, October 1989.

[AJ76] A. Aho and S. Johnson. Optimal code generation for expression trees. *Journal ACM*, 3(23):488–501, 1976.

[AK02] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures.* Morgan Kaufmann Publishers, 2002.

[Ale01] Andreu Alexandrescu. *Modern C++ Design - Generic programming and design patterns applied.* Addison-Wesley, 2001.

[AM91] A. Appel and D. MacQueen. Standard ML of New Jersey. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming*, 528, pages 1–13. Springer Verlag, August 1991.

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 33–50. Springer, September 1995.

[AMW95] M. Alt, F. Martin, and R. Wilhelm. Generating dataflow analyzers with PAG. Technical Report A 10/95, Universität des Saarlandes, 1995.

[Aßm95] U. Aßmann. Optimix language report. Technical Report 31/95, Universität Karlsruhe, 1995.

[Aßm96] U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proceedings of the International Conference on Compiler Construction*, pages 121–135, 1996.

[Aßm97] U. Aßmann. *Optimix Language Manual*. Universität Karlsruhe, August 1997.

[Aßm00] U. Aßmann. Graph rewrite systems for program optimization. *ACM Transactions on programming Languages and Systems (TOPLAS)*, 22(4), 2000.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[BCS96] P. Briggs, K. Cooper, and T. Simpson. Value numbering. *Software-pratical and experience*, September 1996.

[BD88] M. Benitez and J. Davidson. A portable global optimizer and linker. In *Proceedings of the Conference on Programming Language Design and Implementation*, number 7 in 23, pages 329–338. ACM SIGPLAN, 1988.

[BD93] M. Benitez and J. Davidson. A retargetable integrated code improver. Technical Report CS-93-64, Departmente of Computer Science, University of Virginia, November 1993.

[BD94] M. Benitez and J. Davidson. Target-specific global code improvement: principles and applications. Technical Report CS-94-42, Departmente of Computer Science, University of Virginia, 1994.

[BD95] M. Bailey and J. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the Conference on Principles of Programming Languages*, pages 298–310, San Francisco, USA, January 1995. ACM.

[BDB00] V. Bala, E. Duesterwald, and S. Banerijia. DYNAMO: A transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation*, Vancover, Canada, 2000.

[BHS95] P. Briggs, T. Harvey, and T. Simpson. Static single assignment construction. Technical report, Rice University, May 1995.

[Bis92] K. Bischoff. Design, implementation, use, and evaluation of Ox: An attribute-grammar compiling system based on Yacc, Lex, and C. Technical Report TR92-31, Department of Computer Science, Iowa State University, December 1992.

[Bis93a] Kurt M. Bischoff. *Ox: An attribute grammar compiling system based on Yacc, Lex and C - Tutorial Introduction*, 1993.

[Bis93b] Kurt M. Bischoff. *Ox: An attribute grammar compiling system based on Yacc, Lex and C - User Reference Manual*, 1993.

[BM94] M. Brandis and H. Mossenbock. Single-pass generation of static single assignment form for structured languages. *Transactions on Programming Language and Systems*, pages 1684–1698, 1994.

[Bur02] Kevin Burton. *.NET Common Language Runtime Unleashed*. SAMS, 2002.

[CAC+81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via colouring. *Computer Languages*, 6(1):47–57, January 1981.

[CCK+97] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algoritm for partial redundancy elimination based on SSA form. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, Nevada, June 1997.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, April 1991.

[Cha82] G. Chaitin. Register allocation and spilling via graph colouring. In *Proceedings of the Symposium on Compiler Construction*, number 17 in 6, pages 98–105, Boston, USA, June 1982. ACM SIGPLAN.

[CS95] K. Cooper and T. Simpson. Value-driven code motion. Technical report, Rice University, October 1995.

[DF84a] J. Davidson and C. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[DF84b] J. Davidson and C. Fraser. Register allocation and exhaustive peephole optimization. *Software-Practice and Experience*, 14(9):857–865, September 1984.

[DGS97] E. Duesterwald, R. Gupta, and M. Soffa. A pratical framework for demand-driven interprocedural data flow analysis. *Transactions on Programming Languages and Systems*, 18(6):992–1030, November 1997.

[DW89] J. Davidson and D. Whalley. Quick compilers using peephole optimization. *Software - Practice and Experience*, 19(1):79–97, 1989.

[EG90] H. Emmelmann and J. Grosch. A tool-box for compiler construction. Technical report, GMD Karlsruhe, 1990.

[ESL89] H. Emmelmann, F. Schroer, and R. Landwehr. BEG - A generator for efficient back-end's. In *Proceedings of the Conference on Programming Language Design and Implementation*, number 7 in 24, Portland, Oregon, July 1989.

[Exp03a] ACE Associated Compiler Experts. *BEG-CoSy Manual*, cosy-8005-beg edition, 2003.

[Exp03b] ACE Associated Compiler Experts. *CCMIR Definition, Specification in SDL Description and Rationale*, cosy-8002-ccmir edition, 2003.

[Exp03c] ACE Associated Compiler Experts. *The CoSy framework, a compiler construction system*, cosy-8006-fw edition, 2003.

[Exp03d] ACE Associated Compiler Experts. *DSP-C Extensions to DWARF*, cosy-8117p-dwarf edition, 2003.

[Fay00] M. Fayad. Introduction to the computing survey's electronic symposium on object-oriented application frameworks. *ACM Computing Surveys*, 32(1), March 2000.

[FD80] C. Fraser and J. Davidson. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[Fer96] Mary Fernandez. *A retargetable optimizing linker*. PhD thesis, Department of Computer Science, Princeton University, 1996.

[FH90] C. Fraser and D. Hanson. A code generation interface for ANSI C. Research report, CS-TR-270-90, 1990.

[FH91] C. Fraser and D. Hanson. A retargetable compiler for ANSI C. Research report, CS-TR-303-91, 1991.

[FH95] Christopher Fraser and David Hanson. *A retargetable C compiler: design and implementation*. Addison Wesley Publishing Company, 1995.

[FHP91] C. Fraser, R. Henry, and T. Proebsting. BURG - Fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1991.

[FHP92] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FK89] R. French and J. Kohl. *The Zephyr programmer's manual*. MIT Project Athena, April 1989.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fra77] Christopher Fraser. *Automatic generation of code generators*. Phd dissertation, Yale University, New Haven, 1977.

[FSJ99] Mohamed E. Fayad, D. Schmidt, and Ralph Johnson. *Building application frameworks: Object-oriented foundations of framework design*. John Wiley and Sons, Inc, 1999.

[FW88] C. Fraser and A. Wendt. Automatic generation of fast optimizing code generators. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 79–84, Atlanta, Georgia, 1988. ACM SIGPLAN.

[G5096] Open Group Guide G508. *Architecture Neutral Distribution Format Guide*. Open Group, January 1996.

[GE90a] J. Grosch and H. Emmelmann. A toolbox for compiler construction. Technical Report 20, University Karlsruhe, 1990.

[GE90b] Josef Grosch and Helmut Emmelmann. *Cocktail - A toolbox for compiler construction*. CoCoLab-Datenverarbeitung, January 1990.

[GF88] M. Ganapathi and C. Fischer. Integrating code generation and peephole optimization. *Acta Informática*, 25:85–109, 1988.

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Proceedings of the Conference on Principles of Programming Languages*, San Diego, California, January 1998. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of reusable object-orient software.* Addison-Wesley, 1995.

[GHK+90] R. Gray, V. Heuring, S. Kram, A. Sloam, and W. Waite. Eli: A complete, flexible compiler construction system. Research report, University of Colorado, 1990.

[Gla77] R. S. Glanville. *A machine independent algorithm for code generation and its use in retargetable compilers.* Phd dissertation, University of California, Berkeley, 1977.

[Gou01] John Gough. *Compiling for the .NET Common Language Runtime (CLR).* Prentice Hall, 2001.

[Gro90] J. Grosch. Lalr - A generator for efficient parsers. *Software - Practice & Experience*, 11(20):1115–1135, November 1990.

[Gro92a] Josef Grosch. *The parsers generators Lalr and Ell.* CoCoLab Germany, July 1992.

[Gro92b] Josef Grosch. *WAG - Efficient evaluation of Well-formed Attribute Grammars and beyond.* CoCoLab Germany, October 1992.

[Gro98a] Compiler Tools Group. *Lexical Analysis.* Department of Electrical and Computer Enginnering, University of Colorado, 1998.

[Gro98b] Compiler Tools Group. *Syntactic Analysis.* Department of Electrical and Computer Enginnering, University of Colorado, 1998.

[Gro00a] Josef Grosch. *Reuseable software - A collection of C-Modules.* CoCoLab Germany, December 2000.

[Gro00b] Josef Grosch. *Reuseable software - A collection of Modula-Modules.* CoCoLab Germany, December 2000.

[Gro00c] Josef Grosch. *Rex - A scanner generator.* CoCoLab Germany, December 2000.

[Gro02a] Josef Grosch. *Ag - An attribute evaluator generator.* CoCoLab Germany, September 2002.

[Gro02b] Josef Grosch. *Ast - A generator for abstract syntax trees.* CoCoLab Germany, September 2002.

[Gro04] Josef Grosch. *PUMA - A generator for the transformation of attributed trees.* CoCoLab Germany, July 2004.

[Han83] D. Hanson. Simple code optimizations. *Software-Practice and Experience*, 13:745–763, 1983.

[Hau97] J. Hautamaki. Survey of frameworks. Report A-1997-3, Department of Computer Science, University of Tampere, March 1997.

[Hei92] Nevin Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[Hei95] N. Heintze. Control-flow analysis and type systems. In *Proceedings of the International Conference on Static Analysis Symposium*, Glasgow, Scotland, September 1995. Springer-Verlag.

[Hen84] Robert R. Henry. *Graham-Glanville code generators*. Phd dissertation, Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, 1984.

[HH98] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the Conference on Programming Languages Design and Implementation*, Montreal, Canada, June 1998. ACM.

[HP00] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the Symposium on Software Testing and Analysis*, Portland, Oregon, August 2000.

[JGZ88] R. Johnson, J. Graver, and L. Zurawski. TS: An optimizing compiler for smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, number 11 in 23, pages 18–26, November 1988.

[JM91] R. Johnson and C. McConnell. The RTL System: A framework for code optimization. Technical Report UIUCDCS-R-9-1698, University of Illinois, 1991.

[JML91] R. Johnson, C. McConnell, and J. Lake. The RTL System: A framework for code optimization. In *Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, May 1991.

[Joh75] S. Johnson. Yacc - Yet Another Compiler-Compiler. Technical Report 32, Computer Science Technical Report, Bell Laboratories, July 1975.

[Joh79] Steven Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, USA, 1979.

[Joh93] R. Johnson. How to design frameworks - Tutorial Notes. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, 1993.

[Joh97] R. Johnson. Components, Frameworks, Patterns. In *Symposium on Software Reusability*, pages 10–17, Boston, Massachusetts, United States, February 1997.

[Kas94] U. Kastens. Construction of application generators using Eli. Technical report, Reihe Informatik, Universität Paderborn, March 1994.

[Kas97] Uwe Kastens. *LIDO-Reference Manual*. Compiler and Programming Language Group, University of Paderborn, 1997.

[KM89] E. Klein and M. Martin. The parser generating system PGS. *Software - Practice & Experience*, 11(19):1015–1028, November 1989.

[KPJ98] U. Kastens, P. Pfahler, and M. Jung. The Eli System. In *Proceedings of the International Conference on Compiler Construction*, volume 1383, pages 294–297. Springer Verlag, 1998.

[KR94] J. Knoop and O. Ruthing. Optimal code motion: Theory and Pratice. *Trans. on Progr. Languages and Systems*, 16(4):1117–1155, 1994.

[KU76] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 21(3):158–171, 1976.

[Les75] M. Lesk. A lexical analyzer generator. Technical Report 39, Computer Science Technical Report, Bell Laboratories, October 1975.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly and Associates, 1992.

[LT79] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1(1):121–141, July 1979.

[Mar98] F. Martin. PAG: An efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[Mar99] Florian Martin. *Generation of program analyzers*. PhD thesis, Universität des Saarlandes, 1999.

[Mat99] Paulo Matos. Estudo e desenvolvimento de sistemas de geração de back-end's do processo de compilação. Master's thesis, Universidade do Minho, Braga, Portugal, Julho 1999.

[Mat02] P. Matos. DOLPHIN framework. Technical report, University of Minho, October 2002.

[Mat03] P. Matos. DOLPHIN: A system for compilers development, teach and use. Technical report, Universidade do Minho, Braga, Portugal, October 2003.

[MD97] John Meyer and Troy Downing. *Java Virtual Machine*. OReilly & Associates, Inc., 1997.

[MH03a] P. Matos and P. Henriques. Applying compilers technology to solve generic workflow problems. In *Proceedings of the Third Congresso Luso-Moçambicano de Engenharias*, pages 1327–1338, Maputo, Moçambique, August 2003. INEGI-FEUP.

[MH03b] P. Matos and P. Henriques. Construção dinâmica de um sistema interactivo para visualização do código intermédio do processo de compilação. In *Actas 12 Encontro Português de Computação Gráfica*, pages 173–177, Porto, Portugal, October 2003.

[MH03c] P. Matos and P. Henriques. Data flow analysis applied to optimize generic workflow problems. In *Proceedings of the International Conference on Industrial Engineering and Production Management*, Porto, Portugal, May 2003. FUCaM.

[MH03d] P. Matos and P. Henriques. DOLPHIN-COMPLAB: A virtual compilers laboratory. In *Proceedings of the Second International Conference on Multimedia and ICTs in Education*, pages 1637–1641, Badajoz, Spain, December 2003.

[MH03e] P. Matos and P. Henriques. DOLPHIN-FEW: An architecture for compilers development, monitoring and use on the web. In *Proceedings of the Fifth International Conference on Information Integration and Web-based Applications and Service*, pages 154–158, Jakarta, Indonésia, September 2003. Data da conferência: 15-17/September 2003.

[MH03f] P. Matos and P. Henriques. DOLPHIN-FEW: An example of a web system to analyze and study compilers behavior. In *Proceedings of the International Conference e-Society*, volume 2, pages 966–970, Lisbon, Portugal, June 2003. IADIS.

[MH03g] P. Matos and P. Henriques. A solution to dynamically build an interactive visualization system to the DOLPHIN-FEW. In *Proceedings of the International Conference on Visualization, Imaging, and Image Processing*, pages 868–873, Benalmádena, Spain, September 2003. IASTED.

[MH03h] Paulo Matos and Pedro Henriques. DIR: Um modelo para representação de código no processo de compilação. In *Aceite para publicação no Simpósio Brasileiro de Informática*, Brasil, Outobro 2003.

[MH04] P. Matos and P. Henriques. DIR - A code representation approach for compilers. In *Proceedings of the International Conference on Applied Computing*, pages 518–526, Lisbon, Portugal, March 2004. IADIS.

[Mor97] John Morgenthaler. *Static analysis for a software transformation tool*. PhD thesis, University of California, 1997.

[MRS90a] C. McConnell, J. Roberts, and C. Schoening. The RTL System. Technical report, Departement of Computer Science, University Illinois at Urbana-Champaign, October 1990.

[MRS90b] C. McConnell, J. Roberts, and C. Schoening. Using SSA form in a code optimizer. Technical report, Departement of Computer Science, University Illinois at Urbana-Champaign, 1990.

[Muc97] Steven Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[Nil00] Hans-Peter Nilsson. Porting GCC for dunces. Master's thesis, Lund Institute of Technology, 2000.

[NNH99] F. Nielson, H. Nielson, and Hankin. *Principles of program analysis*. Springer Verlag, 1999.

[P5296] Open Group Specification P527. *Architecture Neutral Distribution Format Specification*. Open Group, January 1996.

[Pau96] Larry C. Paulson. *ML for the working programmer (Second Edition)*. Cambridge University Press, 1996.

[Pau98] Larry C. Paulson. *Elements of ML programming, ML97 Edition*. Prentice-Hall, 1998.

[Pax88] V. Paxson. *Flex - Manual Page*. Public Domain Software, 1988.

[PDC92] T. Parr, H. Dietz, and W. Cohen. PCCTS 1.00: The Purdue Compiler Construction Tool Set. *ACM SIGPLAN Notices*, 27(2):88–165, 1992.

[PHEK97] M. Poletto, W. Hsieh, D. Engler, and F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 109–121, 1997.

[PL87] Eduard Pelegri-Llopart. *Tree transformation in compiler systems*. Phd dissertation, University of Berkeley, California, 1987.

[Pol99] Massimiliano Poletto. *Language and compiler support for dynamic code generation*. PhD thesis, Massachusetts Institute of Technology, September 1999.

[Pro88] GNU Project. *Bison - Manual Page*. Public Domain Software, 1988.

[Pro92a] T. Proebsting. Simple and efficient BURS table generation. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 331–340. ACM, 1992.

[Pro92b] Todd A. Proebsting. *Code generation techniques*. Phd. dissertation, Wisconsin University, Madison, 1992.

[Pro95] T. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[RD98] N. Ramsey and J. Davidson. Machine description to build tools for embedded systems. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474, pages 172–188. ACM SIGPLAN, Springer-Verlag, June 1998.

[RF94] N. Ramsey and M. Fernandez. New Jersey Machine-Code Toolkit reference manual. Technical Report TR-471-94, Department of Computer Science, Princeton University, 1994.

[RF95] N. Ramsey and M. Fernandez. The New Jersey Machine-Code Toolkit. In *Proceedings of The USENIX Technical Conference*, pages 289–302, New Orleans, USA, 1995. ACM SIGPLAN.

[RF96] N. Ramsey and M. Fernandez. New Jersey Machine-Code Toolkit architecture specifications. Technical report, Department of Computer Science, Princeton University, 1996.

[RJ96] D. Roberts and R. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of the Conference on Pattern Languages of Programs*, 1996.

[RM97] N. Ramsey and M.Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[RP86] B. Ryder and M. Paull. Elimination algorithms for data analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.

[SA94] Z. Shao and A. Appel. A type-based compiler for Standard ML. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, California, USA, June 1994. ACM SIGPLAN.

[San95] G. Sander. VCG: Visualization of Compiler Graphs. Technical Report A01-95, Universität des Saarlandes, 1995.

[Sch97] Friedrich Schroer. *The Gentle compiler construction system manual*, 1997.

[Sha80] M. Sharir. Structural analysis: A new approach to flow analysis in the optimizing compilers. *Computer Languages*, 5(3–4):715–728, 1980.

[Sha94] Zhong Shao. *Compiling Standard ML for efficient execution on modern machines*. Phd. dissertation, Princeton University, New Jersey, November 1994.

[Sim96] Taylor Simpson. *Value-driven redundancy elimination*. PhD thesis, Rice University, May 1996.

[Sta94] Richard Stallman. *Using and porting GNU CC*. Free Software Foundation, 1994.

[Sta00] Richard Stallman. *Using and porting the GNU Compiler Collection (GCC)*. iUniverse.com, Inc, 2000.

[TH92] S. Tjiang and J. Hennessy. Sharlit - A tool for building optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 82–93. ACM SIGPLAN, July 1992.

[Tji93] Steven Tjiang. *Automatic generation of data-flow analyzers: A tool for building optimizers*. PhD thesis, Stanford University, Computer Systems, Laboratory, 1993.

[TMAL98] S. Thesing, F. Martin, M. Alt, and O. Lauer. *PAG user's manual*, 1998. Version 1.0.

[Tof95] J. Toft. Formal specification of ANDF semantics. Technical Report CS-93-64, ESPIRIT Project 6062 OMI/GLUE, DDC-I, 1995.

[Vie89] B. Vielsack. *Spezification und implementierung der transformation attributierter*. Forschungsstelle an der Universitat Karlsruhe, June 1989.

[WFW+94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. The SUIF compiler system: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, 1994.

[WS91] D. Whitfield and M. Soffa. Automatic generation of global optimizers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 120–129, Toronto, Ontario, Canada, 1991. ACM SIGPLAN.

Glossário

Abstract Syntax Description Language (ASDL) Linguagem utilizada no Zephyr (ver Secção 2.6) para definir modelos de representação de código, p. 21.

Application Programming Interface (API) Define a forma como dois ou mais programas (ou partes de um programa) comunicam entre si, p. 25.

Architectural Neutral Distribution Format (ANDF) Modelo de representação que é independente de qualquer tipo de arquitectura e que visa suportar o desenvolvimento de software "portável". É também utilizado como modelo de RIC na construção de compiladores, p. 21.

back-end Conjunto de tarefas terminais de um compilador, que são tipicamente dependentes da arquitectura de computação, como é o caso do selector de instruções, do sistema de *alocação* de registos e gerador de código final (*assembly*/código binário), p. 5.

Back-End Development System (BEDS) Sistema para geração integrada das tarefas de back-end, p. 5.

Back-End Generator (BEG) Ferramenta de geração de back-end's utilizada no âmbito dos sistemas Cocktail (ver Secção 2.4) e CoSy (ver Secção 2.3), p. 10.

Bottom-Up Rewrite Generator (BURG) Ferramenta de geração de selectores de instruções e de optimizações de código que tenham por base a reescrita de expressões, p. 10.

Call Conventions Language (CCL) Linguagem que permite definir a convenção utilizada para a passagem de parâmetros e para a devolução do valor de retorno, p. 22.

código final Código que resulta do processo de compilação, p. 2.

código fonte Código que é submetido a um compilador, p. 2.

Central Processing Unit (CPU) Designação dada à unidade central de processamento de um microprocessador, p. 212.

Common CoSy Medium-level Intermediate Representation (CCMIR) Modelo de representação de código utilizado pelo sistema CoSy (ver Secção 2.3), p. 15.

Complex Instruction Set Computer (CISC) Termo que surge em oposição a RISC, que visa designar CPU's cuja arquitectura contempla instruções que podem conter várias operações de baixo nível, por exemplo: *load* + operações aritméticas/lógicas + *store*., p. 16.

Computer Systems Description Languages (CSDL) Conjunto de linguagens utilizadas no Zephyr (ver Secção 2.6) para especificar a arquitectura de computação. A especificações são utilizadas para gerar os back-end's, p. 22.

cross-compiler Designação dada a um compilador que tenha a capacidade de gerar código para outras arquitecturas que não seja aquela sob a qual está instalado, p. 19.

Defense Advanced Research Projects Agency (DARPA) Agência do Departamento do Defesa dos USA responsável pelo desenvolvimento de novas tecnologias para uso militar, p. 9.

Digital Signal Processing (DSP) Classe de microprocessadores dedicados ao processamento digital de sinal, p. 15.

Dolphin Internal Representation (DIR) Modelo de representação de código utilizado na *framework Dolphin*. Designa-se também por "DIR" a classe mais abstracta deste modelo, p. 11.

Dynamic Link Library (DLL) Biblioteca de funções e/ou dados, em que o vínculo entre o programa que faz uso dessas funções/dados com a biblioteca é estabelecido dinamicamente, isto é, aquando da execução do programa., p. 25.

eXtended Meta Language (XML) Linguagem utilizada para descrever a estrutura e o conteúdo de documentos, p. 10.

Framework Dolphin *Framework* para construção de compiladores, composta por vários componentes que funcionam sobre um mesmo modelo de representação de código, a DIR, p. 11.

Free Software Foundation (FSF) Organização sem fins lucrativos que visa promover o desenvolvimento de software livre, p. 18.

front-end Conjunto de tarefas iniciais de um compilador, que são tipicamente dependentes da linguagem fonte, como é o caso da análise léxica, da análise sintáctica e da análise semântica, p. 17.

GNU Compilers Collection (GCC) Projecto da FSF que visa o desenvolvimento de compiladores para as principais linguagens e arquitecturas de computação. A sigla GCC é também utilizada para designar GNU C Compiler, p. 18.

GNU General Public License (GNU GPL) Licença que define os termos de utilização do software desenvolvido no âmbito do projecto GNU, p. 19.

Grafo de Dependências de Dados (GDD) Estrutura de dados que caracteriza as dependências entre as variáveis. É fundamental para se efectuar a reordenação das instruções, para se efectuar optimizações de *pipeline*, de gestão de cache e de uma forma geral de outros recursos existentes ao nível do microprocessador, p. 49.

Grafo de Fluxo de Controlo (GFC) Grafo que representa as estruturas de controlo de um programa, p. 23.

High Performance Fortran (HPF) Extensão à linguagem Fortran que disponibiliza os recursos necessários ao desenvolvimento de programas para arquitecturas de elevada performance, p. 17.

hot-spot Termo que identifica as partes de uma *framework*, que poderão ter que ser expandidas pelo utilizador no sentido de desenvolver as suas aplicações, p. 36.

Improved Bottom-Up Rewrite Generator (IBURG) Ferramenta de geração de selectores de instruções e de optimizações de código que tenham por base a reescrita de expressões, p. 22.

instruction set Designação dada ao conjunto de todas as instruções de um microprocessaor, p. 214.

Integrated Development Environment (IDE) Ambiente integrado de desenvolvimento de software, p. 189.

Light C Compiler (lcc) Compilador de C, pequeno, eficiente e capaz de gerar código para várias arquitecturas, p. 21.

linguagem fonte Linguagem utilizada para descrever os programas submetidos ao compilador (ver código fonte), p. 2.

medium-level Conjunto de tarefas do compilador que são independentes da linguagem fonte e da arquitectura de computação. Corresponde normalmente à etapa do processo de compilação que é executada após as tarefas de front-end e antes das tarefas de back-end, p. 19.

National Science Foundation (NSF) Agência independente dos USA que visa promover a investigação e a educação nas mais variadas áreas científicas, p. 9.

Network Processor Units (NPU) Processadores dedicados ao processamento de pacotes em redes de dados, p. 16.

New Jersey Machine Code Toolkit (NJMCT) Ferramenta que permite desenvolver, com base numa especificação, a biblioteca com as funções necessárias à geração de código final, p. 10.

peephole optimizations Optimizações de código localizadas, normalmente efectuadas nas últimas tarefas do processo de compilação, p. 5.

Peephole Optimizer (PO) Solução que visa a implementação de optimizações de código de baixo nível de forma independente da arquitectura de computação, p. 19.

PipeLine Unifying Notation: Graphs and Expressions (PLUNGE) Notação utilizada no sistema Zephyr (ver CSDL) para definir as estruturas de pipeline de um processador, p. 22.

Program Analyzer Generator (PAG) Ferramenta para construção de rotinas de análise de fluxo de dados, p. 10.

Reduced/Regular Instruction Set Computer (RISC) É uma filosofia de desenho de CPU's que favorece o desenvolvimento de um instruction set formado por instruções simples, reduzidas e com tempos de execução similares, p. 16.

Register Transfer List (RTL) Modelo de representação de código proposto com o sistema Peephole Optimizer, que se caracteriza por ser independente da arquitectura de computação e da linguagem fonte. Em alguma bibliografia, a sigla RTL é utilizada com o mesmo significado, mas designando Register Transfer Language, p. 19.

Representação Intermédia do Código (RIC) Representação do código utilizada no medium-level do processo de compilação, p. 15.

RTL System Framework concebida para o desenvolvimento de optimizações de código, que utiliza como modelo de representação de código o RTL, p. 9.

Sistema Dolphin Sistema que visa disponibilizar, via Web, um conjunto de recursos para desenvolvimento de compiladores e que tem por base a *framework Dolphin*, p. 11.

Specification for Encoding and Decoding (SLED) Linguagem do CSDL que permite construir as funções para a geração do código final, p. 22.

Stanford University Intermediate Format (SUIF) Modelo de representação de código utilizado no SUIF Compiler System, p. 21.

Static Single Assignment (SSA) Forma de representação de código que se caracteriza por apenas admitir uma única atribuição por variável (requer a instanciação das variáveis), p. 56.

SUIF Compiler System Sistema para desenvolvimento de compiladores e componentes de compiladores, nomeadamente para arquitecturas paralelas, que tem por base o modelo de representação de código SUIF, p. 8.

SUIF Macro Generator (Smgn) Ferramenta do SUIF Compiler System, que automatiza o processo de implementação de novos elementos para o modelo de RIC, p. 52.

test-bed Termo que designa uma solução utilizada para efectuar testes e experiências, p. 9.

typelist Tecnologia que tem por base as *templates* da linguagem C++ e que permite manusear listas de tipos aquando do processo de compilação, p. 112.

Unified Modeling Language (UML) Linguagem gráfica de modelação, p. 83.

Very Large Instruction Word (VLIW) Tipo de arquitectura utilizada na construção de microprocessadores, que visa permitir representar várias operações através de uma única instrução. O que requer que os microprocessadores sejam capazes de decompor as instruções e de darem andamento às várias operações a efectuar, p. 16.

Very Portable Optimizer (VPO) Ferramenta que visa o desenvolvimento de rotinas de optimização de código de baixo nível (peephole optimizations), de forma independente da arquitectura de computação, p. 20.

Web Diminutivo de World Wide Web, p. iv.

XML Schema Especificação utilizada para descrever a estrutura de documentos XML, p. 10.

Yet Another Compiler Compiler (Yacc) Ferramenta para geração de parsers, p. 3.

APÊNDICE A

Interfaces da arquitectura

A.1 Interface *Protocol*

```
(1)     #ifndef _PROTOCOL
(2)     #define _PROTOCOL
(3)     #include "FObject.h"
(4)     #define P0 0
(5)     #define P1 1
(6)     #define P2 2
(7)     #define P3 3
(8)     #define P4 4
(9)     #define P5 5
(10)    #define P6 6
(11)    #define P7 7
(12)    #define P8 8
(13)    #define P9 9
(14)    typedef int PROTOCOL;
(15)    #define UPDATED 0
(16)    #define OUTDATED 1
(17)    class Protocol :virtual public FObject{
(18)       protected:
(19)          // Variáveis do objecto
(20)          PROTOCOL ___protocol;
```

```
(21)        // Mátodos do objecto
(22)        virtual bool execute0(); // Protocolo P0 (omissão)
(23)        virtual bool execute1(); // Protocolo P1
(24)        virtual bool execute2(); // Protocolo P2
(25)        virtual bool execute3(); // Protocolo P3
(26)        virtual bool execute4(); // Protocolo P4
(27)        virtual bool execute5(); // Protocolo P5
(28)        virtual bool execute6(); // Protocolo P6
(29)        virtual bool execute7(); // Protocolo P7
(30)        virtual bool execute8(); // Protocolo P8
(31)        virtual bool execute9(); // Protocolo P9
(32)        // Construtor
(33)        Protocol();
(34)      public:
(35)        // Variáveis de classe
(36)        static const char *id;
(37)        // Destrutor
(38)        virtual ~Protocol();
(39)        // Métodos do objecto
(40)        virtual PROTOCOL getProtocol();
(41)        virtual void setProtocol(PROTOCOL);
(42)        virtual bool execute();
(43)        // Implementação das interfaces:
(44)        // Para DObject
(45)        const char *getClass();
(46)    };
(47)    #endif
```

A.2 Interface *Component*

```
(1)     #ifndef _TCOMPONENT
(2)     #define _TCOMPONENT
(3)     #include "Protocol.h"
(4)     class compManager;
(5)
(6)     typedef enum {UNDEFTYPE,FRONTEND,ANALYSES,
(7)       OPTIMIZATION,BACKEND,SUPPORT,INSPECTION} COMPTYPE;
(8)
(9)     class Component :public Protocol{
(10)      protected:
(11)        // Variáveis do objecto
(12)        COMPTYPE ___cptype;
(13)        compManager *___cpm;
(14)        unsigned short ___state;
(15)        // Métodos do objecto
(16)        void setType(COMPTYPE);
```

```
(17)          virtual void setElem(compManager*);
(18)          void setState(unsigned short);
(19)          // Construtores
(20)          Component();
(21)          Component(compManager*);
(22)          friend class compManager;
(23)       public:
(24)          // Variáveis de classe
(25)          static const char *id;
(26)          // Destrutor
(27)          virtual ~Component();
(28)          // Métodos do objecto
(29)          COMPTYPE getType();
(30)          virtual compManager* getElem();
(31)          unsigned short getState();
(32)          bool update();
(33)          // Implementação de interfaces:
(34)          // Para DObject
(35)          const char *getClass();
(36)    };
(37)    #endif
```

A.3 Interface *compManager*

```
(1)     #ifndef _TCOMPMANAGER
(2)     #define _TCOMPMANAGER
(3)     #include "Set.h"
(4)     #include "FObject.h"
(5)     class Component;
(6)
(7)     class compManager:virtual public FObject{
(8)        protected:
(9)          // Variáveis do objecto
(10)         Set<Component*> ___sComp;
(11)         // Construtor
(12)         compManager();
(13)       public:
(14)         // Variáveis de classe
(15)         static const char *id;
(16)         // Destrutor
(17)         virtual ~compManager(void);
(18)         // Métodos do objecto
(19)         int regComp(Component*);
(20)         bool hasComp(const char*);
(21)         bool hasComp(Component*);
(22)         Component *getComp(const char*);
```

```
(23)          bool remComp(const char*);
(24)          bool remComp(Component*);
(25)          int howManyComp();
(26)          Component *update(const char*);
(27)          unsigned short getState(const char*);
(28)          int update();
(29)          bool buildQueue(QueueL<Component*>*,int st=0);
(30)
(31)          template <class C>
(32)            C *reqComp(){
(33)            C *c = (C*)(getComp(C::id));
(34)            if(!c) c=new C(this);
(35)            return c;
(36)          }
(37)          // Implementação de interfaces:
(38)          // Para DObject
(39)          const char *getClass();
(40)          // Métodos amigáveis
(41)          friend ostream &operator<<(ostream&,compManager&);
(42)      };
(43)      #endif
```

A.4 Interface *regObserver*

```
(1)   #ifndef _TREGOBSERVER
(2)   #define _TREGOBSERVER
(3)   #include "DictT.h"
(4)   #include "List.h"
(5)   #include "FObject.h"
(6)   class Observer;
(7)   class Observed;
(8)   class Report;
(9)
(10)  class regObserver :virtual public FObject{
(11)    protected:
(12)      // Métodos do objecto
(13)      // Métodos responsáveis pelo reencaminhamento das mensagens
(14)      // Estes métodos fazem parte do padrão de desenho:
(15)      // Chain of Responsability
(16)      // Método para registar um Observer
(17)      virtual long long regChain(Observer*,long long,bool);
(18)      // Método para remover um Observer
(19)      virtual long long remChain(Observer*,long long);
(20)      // Método para testar se existe um dado observadorObserver
(21)      virtual long long hasChain(Observer*,long long);
(22)      // Método para pesquisar os elementos observados
```

```
(23)     virtual long long getChain(Observer*,List<Observed*>*);
(24)     // Método para obter os Reports  dos Observed
(25)     virtual long long getRChain(Observer*,DictT<Observed*,Report*>*);
(26)     // Método para certificar um Component
(27)     virtual long long certifyChain(Observer*);
(28)     // Método para alterar o status do Observed
(29)     virtual long long setSRChain(long long,bool);
(30)     // Construtor
(31)     regObserver()
(32)   public:
(33)     // Variáveis de classe
(34)     static const char *id;
(35)     // Destrutor
(36)     virtual ~regObserver();
(37)     // Métodos do objecto
(38)     // Métodos utilizados pelos observadores para
(39)     // submeterem mensagens aos elementos observados
(40)     // Estes métodos fazem parte do padrão de desenho:
(41)     // Chain of Responsability
(42)     // Método para registar um Observer
(43)     virtual long long regObs(Observer*,long long,bool);
(44)     // Método para remover um Observer
(45)     virtual long long remObs(Observer*,long long);
(46)     // Método que determina o tipo de elementos em que
(47)     // um Observer está registado
(48)     virtual long long hasObs(Observer*long long);
(49)     // Método para obter os Observed de um dado Observer
(50)     virtual long long getObs(Observer*,List<Observed*>*);
(51)     // Método para obter os Reports  dos Observed
(52)     virtual long long getReport(Observer*,DictT<Observed*,Report*>*);
(53)     // Método para certificar um Component
(54)     virtual long long certify(Observer*);
(55)     // Método para alterar o status de um Observed
(56)     virtual long long setRStatus(long long,bool);
(57)     // Implementação de interfaces:
(58)     // Para DObject
(59)     const char *getClass();
(60)     // Métodos amigáveis
(61)     friend ostream &operator<<(ostream&,regObserver&);
(62)   };
(63)   #endif
```

A.5 Interface *Observed*

```
(1)    #ifndef _TOBSERVED
(2)    #define _TOBSERVED
```

```
(3)   #include "Set.h"
(4)   #include "Messages.h"
(5)   #include "regObserver.h"
(6)   class Observer;
(7)   class Report;
(8)
(9)   class Observed :virtual public regObserver{
(10)    protected:
(11)      // Variáveis do objecto
(12)      long long ___msg;
(13)      Report *___rep;
(14)      DictT<Observer*,int> ___sObs;
(15)      QueueL<CellKey<Observer*,int> *> ___qnot;
(16)      // Métodos do objecto
(17)      void createReport(int);
(18)      void createSState(int);
(19)      bool notify();
(20)      void setMSG(long long);
(21)      // Construtor
(22)      Observed();
(23)    public:
(24)      // Variáveis da classe
(25)      static const char *id;
(26)      // Destrutor
(27)      virtual ~Observed();
(28)      // Métodos do objecto
(29)      long long getMSG();
(30)      // Métodos utilizados pelos observadores para
(31)      // submeterem mensagens aos elementos observados
(32)      // Estes métodos fazem parte do padrão de desenho:
(33)      // Chain of Responsability
(34)      // Método para registar um Observer
(35)      long long regObs(Observer*,bool);
(36)      long long regObs(Observer*,long long,bool);
(37)      // Método para remover um Observer
(38)      long long remObs(Observer*);
(39)      long long remObs(Observer*,long long);
(40)      // Método que determina o tipo de elementos em que
(41)      // um Observer está registado
(42)      long long hasObs(Observer*);
(43)      long long hasObs(Observer*,long long);
(44)      // Método para obter os Observed de um dado Observer
(45)      long long getObs(Observer*,List<Observed*>*);
(46)      // Método para obter os Reports  dos Observed
(47)      Report *getReport();
(48)      long long getReport(Observer*,DictT<Observed*,Report*>*);
(49)      // Método para certificar um Component
(50)      long long certify(Observer*);
(51)      // Método para alterar/aceder o status de um Observed
```

```
(52)        int getRStatus(Observer*);
(53)        long long setRStatus(int);
(54)        // Método que permite obter o estado do elemento observado
(55)        virtual AbstractState *getState();
(56)        // Método para testar se um observador já foi notificado
(57)        bool wasNotified(char*);
(58)        // Método que permite passar a vez na notificação
(59)        bool passNotification(Observer*);
(60)        // Método que permite passar a vez se um dado observador
(61)        // ainda não tiver sido notificado
(62)        boolpassNotification(Observer*,char*);
(63)        // Métodos de acesso ao conjunto de observadores
(64)        bool buildQueue(QueueL<Observer*>*,int s=0);
(65)        bool buildQueue(QueueL<CellKey<Observer*,bool>*>*,int s=0);
(66)// Implementação das interfaces:
(67)        // Para DObject
(68)        const char *getClass();
(69)        // Métodos amigáveis
(70)        friend ostream &operator<<(ostream&,Observed&);
(71)   };
(72)   #endif
```

A.6 Interface *Observer*

```
(1)     #ifndef _TOBSERVER
(2)     #define _TOBSERVER
(3)     class Observed;
(4)     class Report;
(5)
(6)     class Observer{
(7)       protected:
(8)         // Métodos do objecto
(9)         virtual bool procReport();
(10)        // Construtor
(11)        Observer();
(12)      public:
(13)        // Destrutor
(14)        virtual ~Observer();
(15)        // Métodos do objecto
(16)        virtual bool notify(Observed*);
(17)        virtual bool notify(Observed*,Report*)
(18)    };
(19)    #endif
```

APÊNDICE B

Templates de adaptação da arquitectura à DIR

Este apêndice contém a *template* que descreve detalhadamente como implementar as interfaces da arquitectura às classe base da DIR e assim definir as diversas variantes. Para além da classe base, à qual se acrescenta o prefixo *_B_*, a *template* descreve ainda como definir:

- A implementação da interface *regObserver* (classe com prefixo *_R_*);
- A implementação da interface *Observed* (classe com prefixo *_O_*);
- A implementação da interface *compManager* (classe com prefixo *_C_*);
- A implementação conjunta das interfaces *compManager* e *regObserver* (classe com prefixo *_CR_*);
- A implementação conjunta das interfaces *compManager* e *Observed* (classe com prefixo *_CR_*).

B.1 Classe base

```
(1)     // ////////////////////////////////////////////////////////
(2)     // Template representa uma qualquer classe DIR.
(3)     // Há várias classes derivadas que implementam
(4)     // as interfaces:  compManager, regObserver and Observed
```

```
(5)   // PARENT representa a classe pai da variante base
(6)   // ////////////////////////////////////////////////////////
(7)   #ifndef _TTEMPLATE
(8)   #define _TTEMPLATE
(9)
(10)  class _CO_Template;
(11)  class _CR_Template;
(12)  class _C_Template;
(13)  class _O_Template;
(14)  class _R_Template;
(15)  class _B_Template;
(16)
(17)  // ////////////////////////////////////////////////////////
(18)  // _B_Template é a variante base que contém
(19)  // os objectos e métodos da classe base
(20)  // ////////////////////////////////////////////////////////
(21)  #include "PARENT.h"
(22)
(23)  class _B_Template :public PARENT{
(24)    protected:
(25)      // Atenção:  As variáveis devem ser definidas
(26)      // utilizando os tipos genéricos (e não as variantes)
(27)      ...
(28)      // Métodos do objecto
(29)      virtual void init();
(30)    public:
(31)      // Variáveis da classe
(32)      static const char *id;
(33)      // Construtor + Destrutor
(34)      _B_Template(void);
(35)      virtual ~_B_Template();
(36)      // Operadores do objecto
(37)      _B_Template &operator=(_B_Template&);
(38)      // Métodos de conversão
(39)      _CO_Template *toCO();
(40)      _CR_Template *toCR();
(41)      _C_Template *toC();
(42)      _O_Template *toO();
(43)      _R_Template *toR();
(44)      _B_Template *toB();
(45)      // Métodos estáticos para operações de cast
(46)      static _B_Template* cast(_CO_Template*);
(47)      static _B_Template* cast(_CR_Template*);
(48)      static _B_Template* cast(_C_Template*);
(49)      static _B_Template* cast(_O_Template*);
(50)      static _B_Template* cast(_R_Template*);
(51)      static _B_Template* cast(_B_Template*);
(52)      // Implementação de interfaces
(53)      // Para DObject
```

```
(54)          const char *getClass();
(55)          // Métodos amigáveis
(56)          friend ostream &operator<<(ostream&,_B_Template&);
(57)      };
```

B.2 Implementação da interface *regObserver*

```
(1)       // ////////////////////////////////////////////////////
(2)       // _R_Template é a variante que deriva de _B_Template
(3)       // e que implementa a interface regObserver
(4)       // ////////////////////////////////////////////////////
(5)       #include "regObserver.h"
(6)
(7)       class _R_Template :virtual public _B_Template,
(8)       virtual public regObserver{
(9)         protected:
(10)          // Atenção:  Se a classe contiver objectos que tenham
(11)          // que processar mensagens dos observadores então
(12)          // os seguintes métodos devem ser redefinidos:
(13)          long long regChain(Observer*,long long,bool);
(14)          long long remChain(Observer*,long long);
(15)          long long hasChain(Observer*,long long);
(16)          long long getChain(Observer*,List<Observed*>*);
(17)          long long getRChain(Observer*,DictT<Observed*,Report*>*);
(18)          long long certifyChain(Observer*);
(19)          long long setSRChain(long long,bool);
(20)        public:
(21)          // Variáveis da classe
(22)          static const char *id;
(23)          // Construtor+Destrutor
(24)          _R_Template();
(25)          virtual ~_R_Template();
(26)          // Operadores do objecto
(27)          _R_Template &operator=(_R_Template&);
(28)          // Métodos de conversão
(29)          _CO_Template *toCO();
(30)          _CR_Template *toCR();
(31)          _C_Template *toC();
(32)          _O_Template *toO();
(33)          _R_Template *toR();
(34)          _B_Template *toB();
(35)          // Métodos estáticos para operações de cast
(36)          static _R_Template* cast(_CO_Template*);
(37)          static _R_Template* cast(_CR_Template*);
(38)          static _R_Template* cast(_C_Template*);
(39)          static _R_Template* cast(_O_Template*);
```

```
(40)        static _R_Template* cast(_R_Template*);
(41)        static _R_Template* cast(_B_Template*);
(42)        // Implementação de interfaces
(43)        // Para DObject
(44)        const char *getClass();
(45)        // Métodos amigáveis
(46)        friend ostream &operator<<(ostream&,_R_Template&);
(47)    };
```

B.3 Implementação da interface *Observed*

```
(1)     // ////////////////////////////////////////////////////
(2)     // _O_Template é a variante que deriva de _R_Template
(3)     // e que implementa a interface Observed
(4)     // ////////////////////////////////////////////////////
(5)     #include "Observed.h"
(6)
(7)     class _O_Template :public _R_Template,
(8)     public Observed{
(9)       protected:
(10)        // Incluir:  Métodos que dêem directo acesso às variáveis
(11)        // do objecto (de forma a impedir o acesso público)
(12)        // Métodos do objecto
(13)        ...
(14)      public:
(15)        // Variáveis da classe
(16)        static const char *id;
(17)        // Construtor + Destrutor
(18)        _O_Template();
(19)        virtual ~_O_Template();
(20)        // Operadores do objecto
(21)        _O_Template &operator=(_O_Template&);
(22)        // Incluir:  Métodos que alterem ás variáveis do objecto
(23)        // Métodos do objecto (controlados)
(24)        ...
(25)        // Métodos de conversão
(26)        _CO_Template *toCO();
(27)        _CR_Template *toCR();
(28)        _C_Template *toC();
(29)        _O_Template *toO();
(30)        _R_Template *toR();
(31)        _B_Template *toB();
(32)        // Métodos estáticos para operações de cast
(33)        static _O_Template* cast(_CO_Template*);
(34)        static _O_Template* cast(_CR_Template*);
(35)        static _O_Template* cast(_C_Template*);
```

```
(36)          static _O_Template* cast(_O_Template*);
(37)          static _O_Template* cast(_R_Template*);
(38)          static _O_Template* cast(_B_Template*);
(39)          // Implementação de interfaces
(40)          // Para DObject
(41)          const char *getClass();
(42)          // Métodos amigáveis
(43)          friend ostream &operator<<(ostream&,_O_Template&);
(44)      };
```

B.4 Implementação da interface *compManager*

```
(1)       // ////////////////////////////////////////////////////
(2)       // _C_Template é a variante que deriva de _B_Template
(3)       // e que implementa a interface compManager
(4)       // ////////////////////////////////////////////////////
(5)       #include "compManager.h"
(6)
(7)       class _C_Template :virtual public _B_Template,
(8)       virtual public compManager{
(9)         public:
(10)          // Variáveis da classe
(11)          static const char *id;
(12)          // Construtor + Destrutor
(13)          _C_Template();
(14)          virtual ~_C_Template();
(15)          // Operadores do objecto
(16)          _C_Template &operator=(_C_Template&);
(17)          // Métodos de conversão
(18)          _CO_Template *toCO();
(19)          _CR_Template *toCR();
(20)          _C_Template *toC();
(21)          _O_Template *toO();
(22)          _R_Template *toR();
(23)          _B_Template *toB();
(24)          // Métodos estáticos para operações de cast
(25)          static _C_Template* cast(_CO_Template*);
(26)          static _C_Template* cast(_CR_Template*);
(27)          static _C_Template* cast(_C_Template*);
(28)          static _C_Template* cast(_O_Template*);
(29)          static _C_Template* cast(_R_Template*);
(30)          static _C_Template* cast(_B_Template*);
(31)          // Implementação de interfaces
(32)          // Para DObject
(33)          const char *getClass();
(34)          // Métodos amigáveis
```

```
(35)          friend ostream &operator<<(ostream&,_C_Template&);
(36)      };
```

B.5 Implementação das interfaces *compManager* e *regObserver*

```
(1)       // ////////////////////////////////////////////////////////////
(2)       // _CR_Template é a variante que deriva de _R_Template
(3)       // e que implementa a interface compManager (e regObserver)
(4)       // ////////////////////////////////////////////////////////////
(5)       class _CR_Template :public _R_Template,
(6)       public _C_Template{
(7)          public:
(8)             // Variáveis da classe
(9)             static const char *id;
(10)            // Construtor + Destrutor
(11)            _CR_Template();
(12)            virtual ~_CR_Template();
(13)            // Operadores do objecto
(14)            _CR_Template &operator=(_CR_Template&);
(15)            // Métodos de conversão
(16)            _CO_Template *toCO();
(17)            _CR_Template *toCR();
(18)            _C_Template *toC();
(19)            _O_Template *toO();
(20)            _R_Template *toR();
(21)            _B_Template *toB();
(22)            // Métodos estáticos para operações de cast
(23)            static _CR_Template* cast(_CO_Template*);
(24)            static _CR_Template* cast(_CR_Template*);
(25)            static _CR_Template* cast(_C_Template*);
(26)            static _CR_Template* cast(_O_Template*);
(27)            static _CR_Template* cast(_R_Template*);
(28)            static _CR_Template* cast(_B_Template*);
(29)            // Implementação de interfaces
(30)            // Para DObject
(31)            const char *getClass();
(32)            // Métodos amigáveis
(33)            friend ostream &operator<<(ostream&,_CR_Template&);
(34)      };
```

B.6 Implementação das interfaces *compManager* e *Observed*

```
(1)     // ////////////////////////////////////////////////////////
(2)     // _CO_Template é a variante que deriva de _O_Template
(3)     // e que implementa a interface compManager (e Observed)
(4)     // ////////////////////////////////////////////////////////
(5)     class _CO_Template :public _O_Template,
(6)     public _C_Template{
(7)       public:
(8)         // Variáveis da classe
(9)         static const char *id;
(10)        // Construtor + Destrutor
(11)        _CO_Template();
(12)        virtual ~_CO_Template();
(13)        // Operadores do objecto
(14)        _CO_Template &operator=(_CO_Template&);
(15)        // Métodos de conversão
(16)        _CO_Template *toCO();
(17)        _CR_Template *toCR();
(18)        _C_Template *toC();
(19)        _O_Template *toO();
(20)        _R_Template *toR();
(21)        _B_Template *toB();
(22)        // Métodos estáticos para operações de cast
(23)        static _CO_Template* cast(_CO_Template*);
(24)        static _CO_Template* cast(_CR_Template*);
(25)        static _CO_Template* cast(_C_Template*);
(26)        static _CO_Template* cast(_O_Template*);
(27)        static _CO_Template* cast(_R_Template*);
(28)        static _CO_Template* cast(_B_Template*);
(29)        // Implementação das interfaces
(30)        // Para DObject
(31)        const char *getClass();
(32)        // Métodos amigáveis
(33)        friend ostream &operator<<(ostream&,_CO_Template&);
(34)    };
(35)    #endif // do #ifndef inserido antes da definição da variante base
```

APÊNDICE C

Framework Dolphin

C.1 Componentes de *front-end*

cint

Versão:	1.2
Objectivo:	*Front-end* de C para inteiros.
Descrição:	Componente para compilar um sub-conjunto da linguagem C que é restrito à utilização de variáveis e constantes do tipo inteiro.

littleC

Versão:	3.3
Objectivo:	Pseudo *front-end* de C.
Descrição:	Componente para compilar código "correcto" de linguagem C. Este *front-end* ainda é bastante limitado no que diz respeito à detecção e controlo de erros.

littleF

Versão: 1.0
Objectivo: Pseudo *front-end* de Fortran.
Descrição: Componente para compilar código de linguagem Fortran que está actualmente em fase inicial de desenvolvimento.

C.2 Componentes de análise

Dominators

Versão: 1.3
Objectivo: Cálculo dos dominadores.
Descrição: Dado o GFC determina os dominadores de cada nodo.
Elemento(s) de registo: CFG
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

Dominated

Versão: 1.3
Objectivo: Cálculo dos dominados.
Descrição: Dado o GFC determina os dominados de cada nodo.
Elemento(s) de registo: CFG
Componentes de suporte: Dominators
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

IDominator

Versão: 1.3
Objectivo: Cálculo dos dominadores imediatos.
Descrição: Dado o GFC calcula os dominador imediato de cada nodo.
Elemento(s) de registo: CFG
Componentes de suporte: Dominators
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

IDominated

Versão: 1.3
Objectivo: Calcular para cada nodo, os nodos dos quais é dominador imediato.
Descrição: Dado o GFC calcula para cada nodo, o conjunto de nodos que o têm por dominador imediato.
Elemento(s) de registo: CFG
Componentes de suporte: IDominator
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

DFrontiers

Versão: 1.3
Objectivo: Cálculo das fronteiras de dominância.
Descrição: Dado o GFC calcula as fronteiras de dominância de cada nodo.
Elemento(s) de registo: CFG
Componentes de suporte: IDominated
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

IDFrontiers

Versão: 1.3
Objectivo: Cálculo do conjunto de nodos que têm um dado nodo por fronteira de dominância.
Descrição: Dado o GFC para cada nodo, o conjunto de nodos que o têm como fronteira de dominância.
Elemento(s) de registo: CFG
Componentes de suporte: DFrontiers
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

nodeDominated

Versão: 1.0
Objectivo: Cálculo dos dominados.
Descrição: Dado o GFC calcula para cada nodo, o conjunto de nodos que o têm por dominador. Esta solução difere de *Dominated* por permitir que o cálculo se faça tendo por nodo raiz um qualquer nodo.
Elemento(s) de registo: CFG
Componentes de suporte:
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

CFTDominators

Versão: 1.0
Objectivo: Cálculo dos dominadores com base em *IntervalNodes*.
Descrição: Este componente permite o cálculo dos dominadores utilizando *IntervalNodes*.
Elemento(s) de registo: CFG
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

CFTDominated

Versão: 1.3
Objectivo: Cálculo dos nodos dominados com base em *IntervalNodes* (CFT).
Descrição: Este componente permite o cálculo dos nodos dominados utilizando *IntervalNodes*.
Elemento(s) de registo: CFG
Componentes de suporte: CFTDominators
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

CFTIDominator

Versão: 1.0
Objectivo: Cálculo dos dominadores imediatos com base em *IntervalNodes* (CFT).
Descrição: Este componente permite o cálculo dos dominadores imediatos utilizando *IntervalNodes*.
Elemento(s) de registo: CFG
Componentes de suporte: CFTDominators
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

StructuralAnalyses

Versão: 1.3
Objectivo: Análise estrutural - Calcula da árvore de fluxo de controlo.
Descrição: Este componente permite construir a árvore de fluxo de controlo.
Elemento(s) de registo: CFG
Componentes de suporte: CFTDominated
Requisitos:

Elemento	Variante	Interface(s)
Jump	_O_Jump	Observed
CJump	_O_CJump	Observed
CFG	_CO_CFG	Observed compManager
FlowNode	_R_FlowNode	regObserver
LDT	_R_LDT	regObserver
DT	_R_DT	regObserver

varIndex

Versão: 1.2
Objectivo: Indexação da variáveis do programa.
Descrição: Este componente permite indexar as variáveis do programa, associando a cada uma um índice.
Elemento(s) de registo: Function
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
IdentTable	_O_IdentTable	Observed

indexVar

Versão: 1.2
Objectivo: Permite aceder às variáveis pelo seu índice.
Descrição: Este componente permite aceder às variáveis utilizando o índice calculado pelo varIndex.
Elemento(s) de registo: Function
Componentes de suporte: varIndex
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
IdentTable	_O_IdentTable	Observed

varSeriation

Versão: 1.2
Objectivo: Gestão da indexação das variáveis.
Descrição: Este componente visa disponibilizar todas as funcionalidades relacionadas com a indexação de variáveis.
Elemento(s) de registo: Function
Componentes de suporte: varIndex
indexVar
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
IdentTable	_O_IdentTable	Observed

LstAssigns

Versão: 1.0
Objectivo: Gestão dos *assignments* das variáveis.
Descrição: Este componente permite gerir os *assignments* das variáveis do programa.
Elemento(s) de registo: Function
Requisitos:

Elemento	Variante	Interface(s)
Function	_CO_Function	compManager Observed
IdentTable	_O_IdentTable	Observed

NodeDefVar

Versão: 1.1
Objectivo: Determina as definições das variáveis num nodo.
Descrição: Este componente permite determinar todas as definições que ocorrem num dado nodo do GFC.
Elemento(s) de registo: FlowNode
Requisitos:

Elemento	Variante	Interface(s)
FlowNode	_CR_FlowNode	compManager regObserver
LDT	_O_LDT	Observed

DefVar

Versão: 1.1
Objectivo: Determina as definições das variáveis.
Descrição: Este componente permite determinar todas as definições que ocorrem numa dada função.
Elemento(s) de registo: Function
Componentes de suporte: NodeDefVar
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
CFG	_O_CFG	Observed
LDT	_O_LDT	Observed

DefReach

Versão: 1.1
Objectivo: Análise do alcance das definições (atribuições) das variáveis.
Descrição: Este componente permite determinar que definições das variáveis alcançam cada posição do programa.
Elemento(s) de registo: Function
Componentes de suporte: varSeriation
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
CFG	_O_CFG	Observed
LDT	_O_LDT	Observed

AliveVar

Versão: 1.1
Objectivo: Análise do período de vida útil das variáveis.
Descrição: Este componente permite determinar para cada posição do programa que variáveis é que estão ”vivas”, isto é, que foram previamente definidas e são utilizadas posteriormente.
Elemento(s) de registo: Function
Componentes de suporte: varSeriation
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
CFG	_O_CFG	Observed
LDT	_O_LDT	Observed

ExprReach

Versão: 1.1
Objectivo: Análise do alcance das sub-expressões de um programa.
Descrição: Este componente permite determinar para cada posição do programa quais as expressões que podem ser reutilizadas.
Elemento(s) de registo: Function
Componentes de suporte: varSeriation
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
CFG	_O_CFG	Observed
LDT	_O_LDT	Observed

genDDG

Versão: 1.1
Objectivo: Disponibiliza explicitamente o GDD.
Descrição: Este componente disponibiliza de uma forma mais amigável a informação contida na RIC sobre as dependências entre dados, construindo explicitamente o Grafo de Dependências de Dados.
Elemento(s) de registo: Function
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

C.3 Componentes de conversão

cnv2SSA

Versão: 1.7
Objectivo: Conversão para a forma SSA.
Descrição: Componente que permite passar a RIC da forma normal para a forma SSA.
Elemento(s) de registo: Function
Componentes de suporte: Dominators
IDominator
DFrontiers
IDFrontiers
indexVar
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

cnv2F

Versão: 1.7
Objectivo: Conversão da forma SSA para a forma normal.
Descrição: Componente que permite passar a RIC da forma SSA para a forma normal.
Elemento(s) de registo: Function
Componentes de suporte:
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

C.4 Componentes de optimização

elimUnreachCode

Versão: 1.2
Objectivo: Elimina código inalcançável.
Descrição: Componente que permite eliminar código inalcançável.
Elemento(s) de registo: CFG
Componentes de suporte: Dominated
Requisitos:

Elemento	Variante	Interface(s)
CFG	_C_CFG	compManager

elimJumpChain

Versão: 1.1
Objectivo: Redução de cadeias de salto.
Descrição: Componente que permite minimizar o número de operações de salto incondicional.
Elemento(s) de registo: CFG
Requisitos:

Elemento	Variante	Interface(s)
CFG	_C_CFG	compManager

elimLComSubExpr

Versão: 1.0
Objectivo: Elimina sub-expressões locais.
Descrição: Este componente permite eliminar sub-expressões ao nível de um nodo do GFC.
Elemento(s) de registo: FlowNode
Requisitos:

Elemento	Variante	Interface(s)
FlowNode	_C_FlowNode	compManager

elimComSubExpr

Versão: 1.3
Objectivo: Elimina sub-expressões.
Descrição: Este componente permite eliminar sub-expressões ao nível de uma função.
Elemento(s) de registo: Function
Componentes de suporte: elimLComSubExpr
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

elimLoads

Versão: 1.2
Objectivo: Elimina operações de acesso a memória.
Descrição: Este componente permite reduzir o número de operações de acesso a memória para leitura (*loads*).
Elemento(s) de registo: Function
Componentes de suporte: ExprReach
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

cnstPropagation

Versão: 1.3
Objectivo: Propagação de constantes.
Descrição: Este componente permite simplificar as expressões através da propagação de constantes.
Elemento(s) de registo: Function
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

copyPropagation

Versão: 1.2
Objectivo: Propagação de cópias.
Descrição: Este componente permite simplificar as expressões através da propagação de cópias de variáveis.
Elemento(s) de registo: Function
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

C.5 Componentes de inspecção

cntExpr

Versão: 1.1
Objectivo: Quantifica o número de expressões/DTs.
Descrição: Componente que permite apurar várias métricas relacionadas com o número de expressões de um programa.
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

cntJmp

Versão: 1.1
Objectivo: Quantifica instruções de salto.
Descrição: Componente que permite apurar várias métricas relacionadas com instruções de salto.
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

cntMRef

Versão: 1.1
Objectivo: Quantifica referências a posições de memória.
Descrição: Componente que permite apurar várias métricas relacionadas com operaçóes de acesso a memória.
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

C.6 Componentes de suporte ao *back-end*

inferenceGraph

Versão: 1.1
Objectivo: Cálculo do grafo de interferências.
Descrição: Este componente permite calcular o grafo de interferências, o qual suporta o processo de atribuição global de registos dos *back-ends*.
Elemento(s) de registo: Function
Componentes de suporte: AliveVar
Requisitos:

Elemento	Variante	Interface(s)
Function	_CR_Function	compManager regObserver
Function	_CR_Function	compManager regObserver
CFG	_O_CFG	Observed
LDT	_O_LDT	Observed

serialize

Versão: 1.1
Objectivo: Serializa a RIC.
Descrição: Este componente permite serializar a RIC gerando assim uma lista de árvores de expressões.
Elemento(s) de registo: Function
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

C.7 Componentes de *back-end*

genHTML

Versão: 2.1
Objectivo: Conversão da RIC para HTML.
Descrição: Componente que permite colocar em formato HTML o conteúdo da RIC.
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

genXML

Versão: 1.8
Objectivo: Conversão da RIC para XML.
Descrição: Componente que permite colocar em formato XML o conteúdo da RIC.
Elemento(s) de registo: Program
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

genPseudoCode

Versão: 1.5
Objectivo: Conversão da RIC para pseudo-código (assembly genérico).
Descrição: Componente que permite gerar pseudo-código, uma forma genérica de assembly
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

genX86

Versão: 2.1
Objectivo: Geração de assembly x86.
Descrição: Componente que permite gerar código assembly para arquitecturas x86.
Elemento(s) de registo: Program
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Program	_C_Program	compManager

genExprLst

Versão: 2.1
Objectivo: Geração de árvores de expressões.
Descrição: Componente que permite gerar em formato texto todas as árvores de expressões de uma função.
Elemento(s) de registo: Function
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

genDTLst

Versão: 2.1
Objectivo: Geração da lista de DT.
Descrição: Componente que permite gerar em formato texto uma lista devidamente ordenada com todos os DTs de uma função.
Elemento(s) de registo: Function
Componentes de suporte: StructuralAnalyses
Requisitos:

Elemento	Variante	Interface(s)
Function	_C_Function	compManager

APÊNDICE D

Rotinas de teste

Este anexo contém as rotinas que foram utilizadas para efectuar os testes da Secção 8.3.

D.1 Multiplicação de matrizes

```
(1)  // ////////////////////////////////////////////////////////
(2)  // Mutiplicação de matrizes
(3)  // ////////////////////////////////////////////////////////
(4)  void mulmatriz(int ma[100][100],int mb[100][100]){
(5)     int i,j;
(6)     for(i=0;i<100;i=i+1)
(7)        for(j=0;j<100;j=j+1)
(8)           ma[i][j]=ma[i][j]*mb[i][j];
(9)  }
```

D.2 Pesquisa numa lista ligada simples

```
(1)  // ////////////////////////////////////////////////////////
(2)  // Pesquisa numa lista ligada simples
```

```
(3)   // ////////////////////////////////////////////////////////
(4)   typedef int CHAVE;
(5)   typedef int INFO;
(6)   typedef struct nodos{
(7)     CHAVE ch;
(8)     INFO inf;
(9)     struct nodos *next;
(10)  } NODOS;
(11)
(12)  INFO *peslls(NODOS *l,CHAVE ch){
(13)    INFO *inf=0;
(14)    while(l)
(15)      if(l->ch<ch)
(16)        l=l->next;
(17)    if(l)
(18)      if(l->ch==ch) inf=&(l->inf);
(19)    return inf;
(20)  }
```

D.3 Pesquisa em árvore binária

```
(1)   // ////////////////////////////////////////////////////////
(2)   // Pesquisa num árvore binária
(3)   // ////////////////////////////////////////////////////////
(4)   typedef int CHAVE;
(5)   typedef int INFO;
(6)   typedef struct nodot{
(7)     CHAVE ch;
(8)     INFO inf;
(9)     struct nodot *esq, *dir;
(10)  }NODOT;
(11)
(12)  INFO *pesquisaABP(NODOT *t,CHAVE chx){
(13)    INFO *infx;
(14)    if(t){
(15)    if(t->ch==chx) infx=&(t->inf);
(16)    else if(t->ch<chx) infx=pesquisaABP(t->dir,chx);
(17)    else infx=pesquisaABP(t->esq,chx);
(18)    }else infx=0;
(19)    return infx;
(20)  }
```

D.4 Pesquisa em árvore binária

```
(1)   // ////////////////////////////////////////////////////////
(2)   // Máquina de estados
(3)   // ////////////////////////////////////////////////////////
(4)   int maqestados(int s,int t){
(5)     switch(s){
(6)       case 1:  if(t)s=2;else s=3;break;
(7)       case 2:  if(t)s=4;else s=5;break;
(8        case 3:  if(t)s=6;else s=7;break;
(9)       case 4:  if(t)s=8;else s=9;break;
(10)      case 5:  if(t)s=10;else s=11;break;
(11)      case 6:  if(t)s=12;else s=13;break;
(12)      case 7:  if(t)s=14;else s=15;break;
(13)      case 8:  if(t)s=16;else s=17;break;
(14)      case 9:  if(t)s=18;else s=19;break;
(15)      case 10:  if(t)s=0;else s=1;break;
(16)      case 11:  if(t)s=2;else s=3;break;
(17)      case 12:  if(t)s=4;else s=5;break;
(18)      case 13:  if(t)s=6;else s=7;break;
(19)      case 14:  if(t)s=8;else s=9;break;
(20)      case 15:  if(t)s=10;else s=11;break;
(21)      case 16:  if(t)s=12;else s=13;break;
(22)      case 17:  if(t)s=14;else s=15;break;
(23)      case 18:  if(t)s=16;else s=17;break;
(24)      case 19:  if(t)s=18;else s=19;break;
(25)      default:  s=-1;
(26)    }
(27)    return s;
(28)  }
```

D.5 Dominadores imediatos

```
(1)   // ////////////////////////////////////////////////////////
(2)   // Dominadores imediatos
(3)   // ////////////////////////////////////////////////////////
(4)   typedef enum FALSE,TRUE BOOL;
(5)   typedef struct nodo{
(6)     int v;
(7)     struct nodo *n;
(8)   }NODO;
(9)   typedef struct grafo{
(10)    int nv, root;
(11)    int ma[100][100];
```

```
(12)    }GRAFO;
(13)    BOOL belong(int,NODO*);
(14)    void bfpo(GRAFO*,int*);
(15)    int copiarL2A(int*,NODO*);
(16)    NODO *copiarL2L(NODO*,NODO*);
(17)    void free(void*);
(18)    int getFirst(NODO*);
(19)    void ins(NODO*,int);
(20)    void *malloc(int);
(21)    void rem(NODO*,int);
(22)    int remFQ(int*,int);
(23)
(24)    int *main(GRAFO *g,NODO **dom){
(25)       int *q0=(int*)malloc(sizeof(int)*(g->nv));
(26)       int *q1=(int*)malloc(sizeof(int)*(g->nv));
(27)       int *q2=(int*)malloc(sizeof(int)*(g->nv));
(28)       int *idom=(int*)malloc(sizeof(int)*(g->nv));
(29)       NODO **Tmp=(NODO**)malloc(sizeof(NODO*)*(g->nv));
(30)       int i,j,l,n,s,t,sq1,sq2;
(31)       bfpo(g,q0);
(32)
(33)       for(i=0;i<g->nv;i++)
(34)          Tmp[i]=copiarL2L(Tmp[i],dom[i]);
(35)
(36)       for(i=0;i<g->nv;i++){
(37)          n=q0[i];
(38)          if(n!=g->root){
(39)             sq1=copiarL2A(q1,Tmp[n]);
(40)             for(j=0;j<sq1;j++){
(41)                s=q1[j];
(42)                rem(Tmp[n],s);
(43)                sq2=copiarL2A(q2,Tmp[n]);
(44)                ins(Tmp[n],s);
(45)                for(l=0;l<sq2;l++){
(46)                   t=q2[l];
(47)                   if(belong(t,Tmp[s])==TRUE){
(48)                      rem(Tmp[n],t);
(49)                      sq1=remFQ(q1,t);
(50)                   }
(51)                }
(52)             }
(53)          }
(54)       }
(55)       for(i=0;i<g->nv;i++)
(56)          idom[i]=getFirst(Tmp[i]);
(57)       free(q1);
(58)       free(q2);
(59)       free(Tmp);
(60)       return idom;
```

```
(61)    }
```

APÊNDICE E

XML Schema dos componentes

```
(1)  <?xml version="1.0"encoding="UTF-8"?>
(2)  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema»
(3)   <xs:element name="component»
(4)    <xs:annotation>
(5)     <xs:documentation>
(6)      Schema for XML documents describing
(7)      the components of the Dolphin framework
(8)     </xs:documentation>
(9)    </xs:annotation>
(10)   <xs:complexType>
(11)    <xs:all>
(12)     <xs:element name="id" type="xs:string"/>
(13)     <xs:element name="goal" type="xs:string"/>
(14)     <xs:element name="date" type="xs:date"/>
(15)     <xs:element name="version" type="xs:string"/>
(16)     <xs:element name="description" type="xs:string"/>
(17)     <xs:element name="algorithm" type="xs:string" minOccurs="0"/>
(18)     <xs:element name="regelements" minOccurs="0»
(19)      <xs:complexType>
(20)       <xs:sequence>
(21)        <xs:element name="regelement"
(22)        type="xs:string" maxOccurs="unbounded"/>
```

```
(23)          </xs:sequence>
(24)         </xs:complexType>
(25)        </xs:element>
(26)        <xs:element name="requirements" minOccurs="0»
(27)         <xs:complexType>
(28)          <xs:sequence>
(29)           <xs:element name="element" maxOccurs="unbounded»
(30)            <xs:complexType>
(31)             <xs:sequence>
(32)              <xs:element name="id" type="xs:string"/>
(33)              <xs:element name="interfaces»
(34)               <xs:complexType>
(35)                <xs:sequence>
(36)                 <xs:element name="interface"
(37)                 type="xs:string" maxOccurs="unbounded"/>
(38)                </xs:sequence>
(39)               </xs:complexType>
(40)              </xs:element>
(41)             </xs:sequence>
(42)            </xs:complexType>
(43)           </xs:element>
(44)          </xs:sequence>
(45)         </xs:complexType>
(46)        </xs:element>
(47)        <xs:element name="methods" minOccurs="0»
(48)         <xs:complexType>
(49)          <xs:sequence>
(50)           <xs:element name="method" maxOccurs="unbounded»
(51)            <xs:complexType>
(52)             <xs:sequence>
(53)              <xs:element name="id" type="xs:string"/>
(54)              <xs:element name="goal" type="xs:string"/>
(55)              <xs:element name="rettype"type="xs:string"/>
(56)              <xs:element name="parameters"minOccurs="0»
(57)               <xs:complexType>
(58)                <xs:sequence>
(59)                 <xs:element name="parameter"
(60)                 type="xs:string" maxOccurs="unbounded"/>
(61)                </xs:sequence>
(62)               </xs:complexType>
(63)              </xs:element>
(64)              <xs:element name="affectedvariables" minOccurs="0»
(65)               <xs:complexType>
(66)                <xs:sequence>
(67)                 <xs:element name="affectedvariable"
(68)                 type="xs:string" maxOccurs="unbounded"/>
(69)                </xs:sequence>
(70)               </xs:complexType>
(71)              </xs:element>
```

```
(72)            </xs:sequence>
(73)           </xs:complexType>
(74)          </xs:element>
(75)         </xs:sequence>
(76)        </xs:complexType>
(77)       </xs:element>
(78)       <xs:element name="samples" minOccurs="0»
(79)        <xs:complexType>
(80)         <xs:sequence>
(81)          <xs:element name="sample" maxOccurs="unbounded»
(82)           <xs:complexType>
(83)            <xs:sequence>
(84)             <xs:element name="ln" type="xs:string"
(85)             maxOccurs="unbounded"/>
(86)            </xs:sequence>
(87)           </xs:complexType>
(88)          </xs:element>
(89)         </xs:sequence>
(90)        </xs:complexType>
(91)       </xs:element>
(92)       <xs:element name="supcomponents" minOccurs="0»
(93)        <xs:complexType>
(94)         <xs:sequence>
(95)          <xs:element name="supcomponent"
(96)          type="xs:string"maxOccurs="unbounded"/>
(97)         </xs:sequence>
(98)        </xs:complexType>
(99)       </xs:element>
(100)      <xs:element name="observed" minOccurs="0»
(101)       <xs:complexType>
(102)        <xs:sequence>
(103)         <xs:element name="observed"
(104)         type="xs:string" maxOccurs="unbounded"/>
(105)        </xs:sequence>
(106)       </xs:complexType>
(107)      </xs:element>
(108)      <xs:element name="historic" minOccurs="0»
(109)       <xs:complexType>
(110)        <xs:sequence>
(111)         <xs:element name="event" maxOccurs="unbounded»
(112)          <xs:complexType>
(113)           <xs:sequence>
(114)            <xs:element name="date" type="xs:date"/>
(115)            <xs:element name="description" type="xs:string"/>
(116)           </xs:sequence>
(117)          </xs:complexType>
(118)         </xs:element>
(119)        </xs:sequence>
(120)       </xs:complexType>
```

```
(121)       </xs:element>
(122)       <xs:element name="affectedelements" minOccurs="0»
(123)        <xs:complexType>
(124)         <xs:sequence>
(125)          <xs:element name="affectedelement" type="xs:string"
(126)          maxOccurs="unbounded"/>
(127)         </xs:sequence>
(128)        </xs:complexType>
(129)       </xs:element>
(130)       <xs:element name="compability" minOccurs="0»
(131)        <xs:complexType>
(132)         <xs:sequence>
(133)          <xs:element name="system" maxOccurs="unbounded»
(134)           <xs:complexType>
(135)            <xs:sequence>
(136)             <xs:element name="id" type="xs:string"/>
(137)             <xs:element name="version" type="xs:string"/>
(138)             <xs:element name="observation" type="xs:string"/>
(139)            </xs:sequence>
(140)           </xs:complexType>
(141)          </xs:element>
(142)         </xs:sequence>
(143)        </xs:complexType>
(144)       </xs:element>
(145)     </xs:all>
(146)    </xs:complexType>
(147)   </xs:element>
(148)  </xs:schema>
```

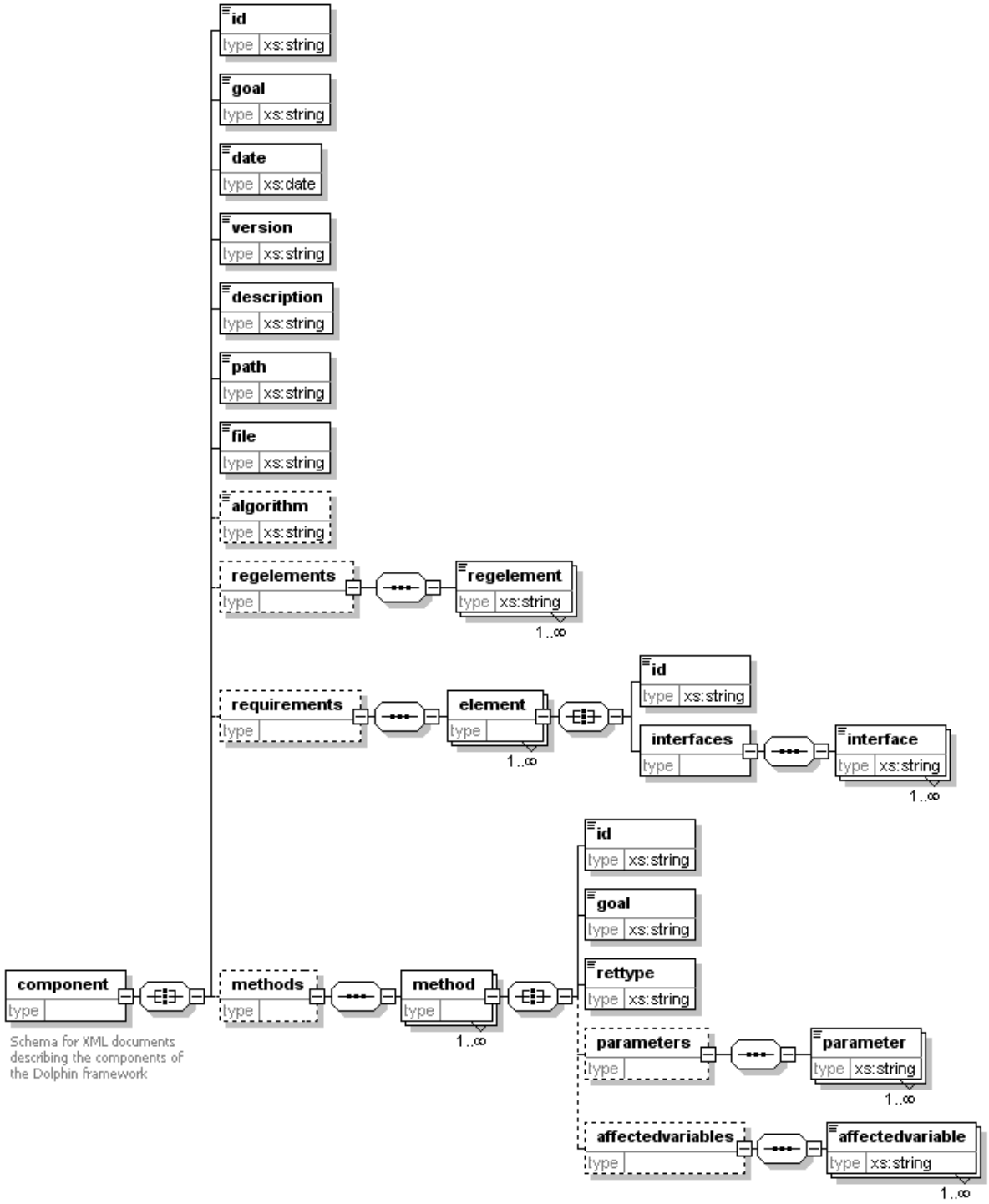


Figura E.1: Diagrama esquemático do Schema - parte 1.

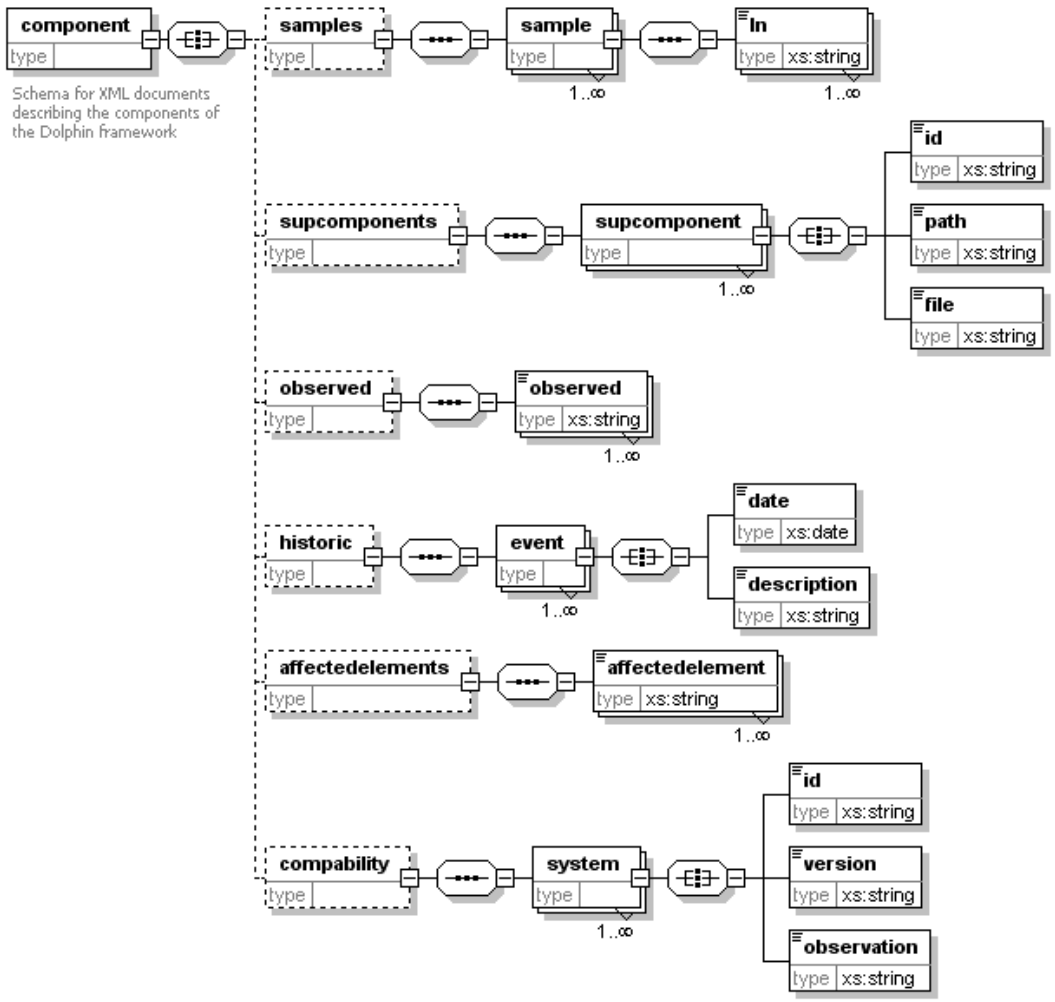


Figura E.2: Diagrama esquemático do Schema - parte 2.