Rodrigues, Nuno F.

Slicing techniques applied to architectural analysis of legacy software

http://hdl.handle.net/1822/9078

Metadados

Data de Publicação 2009-02-09

Resumo

Program understanding is emerging as a key concern in software engineering. In a situation in which the only quality certificate of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on - i.e., the level of understanding of - their (otherwise untouchable) code...

A compreensão de sistemas de software reveste-se de uma cada vez maior importância no campo da engenharia de software. Numa situação em que a única garantia de funcionamento dos diversos componentes de software reside apenas na metodologia de desenvolvimento adoptada, tanto clientes bem como produtores de software encontram-se pouco preparados para modificar ou melhorar os seus programas. No entanto, face a uma tão grande dependência em relação ao código legado, os gestores estão cada vez mais r...

Tipo doctoralThesis

Universidade do Minho
Escola de Engenharia

Nuno Miguel Feixa Rodrigues

Slicing Techniques Applied to Architectural Analysis of Legacy Software

Outubro de 2008

Nuno Miguel Feixa Rodrigues **Slicing Techniques Applied to Architectural Analysis of Legacy Software**

UMinho | 2008

Universidade do Minho
Escola de Engenharia

Nuno Miguel Feixa Rodrigues

Slicing Techniques Applied to Architectural Analysis of Legacy Software

Tese de Doutoramento em Informática
Ramo de Fundamentos da Computação

Trabalho efectuado sob a orientação do
Professor Doutor Luís Soares Barbosa

Outubro de 2008

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/______

Assinatura: ____

Acknowledgements

First of all, I would like to thank Luís Soares Barbosa for supervising my research, a task in which he remarkably managed to give me the freedom to choose every investigation path I pleased, while at the same time, providing a clear view of the headings I was about to undertake. My gratitude to Luís goes far beyond the supervision of this thesis, he has been a truly friend.

For accompanying me during my long investigation periods at the Informatics Department and for making it such a pleasant and funny experience, I would like to thank José Bernardo Barros.

To the Department people, who contributed to this work with their experience and ideas, I wish to especially thank José Nuno, Alcino, Jorge, Manuel, Bacelar, João Paulo and Miguel Vilaça.

I wish to express my deepest gratitude to my parents, António and Isabel, who always believed and supported me in every moment of my life, to my brother, João Pedro, for being my biggest fan and friend.

For having always set the course of my life by living her own, I wish to especially thank my aunt, Maria João.

Finally, I would like to dedicate this thesis to Sónia. You are very special to me.

This thesis was supported by Fundação para a Ciência e a Tecnologia (FCT) under Doctoral Grant SFRH/BD/19127/2004.

Abstract

Program understanding is emerging as a key concern in software engineering. In a situation in which the only quality certificate of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *i.e.*, the level of understanding of — their (otherwise untouchable) code. In fact the technological and economical relevance of *legacy* software as well as the complexity of their re-engineering entails the need for rigour.

Addressing such a scenario, this thesis advocates the use of direct source code analysis for both the process of understanding and transformation of software systems. In particular, the thesis focuses on the development and application of slicing techniques at both the "micro" and "macro" structural levels of software.

The former, deals with fine-grained structures of programs, slicing operating over elementary program entities, such as types, variables or procedure identifiers. The latter, on the other hand, addresses architectural issues and interaction modes across modules, components or services upon which a system is decomposed. At the "micro" level this thesis delves into the problem of slicing functional programs, a paradigm that is gaining importance and was generally neglected by the slicing community. Three different approaches to functional slicing are proposed, accompanied by the presentation of the HaSlicer application, a software tool developed as a proof-of-concept for some of the ideas discussed. A comparison between the three approaches, their practical application and the motivational aspects for keeping investigating new functional slicing processes are also discussed.

Slicing at a "macro" level is the theme of the second part of this thesis, which addresses the problem of extracting from source code the system's coordination model which governs interaction between its components. This line of research delivers two approaches for abstracting software systems coordination models, one of the most vital structures for software architectural analysis. Again, a software tool – CoordInspector – is introduced as a proof-of-concept.

Resumo

A compreensão de sistemas de software reveste-se de uma cada vez maior importância no campo da engenharia de software. Numa situação em que a única garantia de funcionamento dos diversos componentes de software reside apenas na metodologia de desenvolvimento adoptada, tanto clientes bem como produtores de software encontram-se pouco preparados para modificar ou melhorar os seus programas. No entanto, face a uma tão grande dependência em relação ao código legado, os gestores estão cada vez mais receptivos a gastar recursos de forma a aumentar a confiança - i.e., o nível de compreensão - dos seus (de outra forma intocáveis) programas. De facto, a relevância tecnológica e económica do software legado bem como a complexidade associada á sua reengenharia provocam uma urgente necessidade de rigor.

Tendo este cenário como contexto, esta tese advoga o uso de uma análise directa de código fonte com o objectivo de compreender e transformar sistemas de software. Em particular, esta tese debruça-se sobre o desenvolvimento e a aplicação de técnicas de *slicing* aos níveis “micro” e “macro” das estruturas de software.

A análise efectuada ao nível “micro” lida com estruturas de programas de pequena granularidade, onde o *slicing* opera sobre entidades elementares dos programas, tais como tipos, variáveis ou identificadores de procedimentos. Por outro lado, o nível de análise “macro” aborda questões arquitecturais, tais como as interacção entre módulos, componentes ou serviços sobre os quais um sistema de software pode ser decomposto.

Ao nível “micro”, esta tese aborda o problema de efectuar *slicing* a programas funcionais, um paradigma que se reveste de uma cada vez maior importância e o qual tem sido negligenciado pela comunidade de *slicing*. Neste sentido, esta tese apresenta três diferentes abordagens ao *slicing* funcional, acompanhadas pela apresentação da aplicação HaSlicer, uma ferramenta de software desenvolvida como prova de conceito para algumas das ideias expostas. No decorrer da apresentação destas propostas de abordagem ao *slicing* funcional, efectua-se ainda uma comparação entre os diversos processos, as suas aplicações práticas bem como os aspectos motivacionais que levaram á investigação de novos processos de *slicing* funcional.

As operações de *slicing* ao nível "macro" constituem o tema da segunda parte desta tese, onde se aborda o problema específico da extracção de arquitecturas de sistemas de software. Neste sentido, são desenvolvidas duas abordagens distintas para a abstracção do modelo de coordenação de um sistema de software, o que constitui uma das mais vitais estruturas para a análise de sistemas de software. Mais uma vez, é apresentada uma ferramenta de software – CoordInspector – como prova de conceito.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Objectives

1.1.1 Context and Motivation

By the end of the century *program understanding* and *reverse engineering* emerged as key concerns in software engineering, attracting an ever-increasing attention both in Industry and Academia. Actually, the increasing relevance and exponential growth of software systems, both in size and quantity, lead to an equally growing amount of legacy code that has to be maintained, improved, replaced, adapted and assessed for quality every day.

The high dependence of modern societies on such legacy systems and the incredibly fast rate of evolution which characterises software industry, make companies and managers willing to spend resources to increase confidence on — *i.e.*, the level of understanding of — their running code. In fact the technological and economical relevance of *legacy* software, as well as the complexity of its re-engineering and the (often exponential) costs involved, justifies this technical "*movida*", witnessed by the volume of publications, projects and dedicated conferences, as well as by the number and diversity of approaches, methods and tools announced. Moreover, such factors entail the need for rigour, *i.e.*, for precise engineering methods and solid foundations.

Such is the context for this thesis, which grew up while the author integrated the research team of a broader project on *program understanding and* *re-engineering* of legacy code supported by formal methods[1]. The project aimed at combining program analysis techniques with theoretical results developed in the area of formal methods and program calculi to meet the challenges of this novel domain of application. Furthermore, it was organised around two main research axes: the *micro*, algorithmic one, concerned with program understanding and re-engineering at code level, and the *macro*, architectural one, intended to pursue similar goals but at the level of system's macro structuring and software architectures.

Our own contribution to the PURE project, which directly leads to the present thesis, focused mainly on the *program analysis* side, understood as the broad range of techniques to extract, from source code, specific and rigorous knowledge, to be suitably represented and visualised, and to provide a basis for systems analysis, classification and reconstruction. We have concentrated in a particular family of techniques, that of *program slicing* — a decomposition technique to extract from a program, information relevant to a given computation, originally proposed by Mark Weiser, 30 years ago, in his PhD thesis [Wei79].

If slicing is the basic technique addressed in this thesis, our research developed itself along the two axes mentioned above:

- At the *micro* level, we have considered the problem of slicing *functional* programs, and developed a number of techniques and a prototype tool for this programming paradigm. Actually, mainstream research in the area targets imperative or object-oriented languages, and was developed around well characterised notions of computational variable, program statement and control flow behaviour, which are alien to functional programs.

- At the *macro* level, on the other hand, we addressed the problem of extracting from source code, and through suitable slicing techniques,

[1] The PURE project, funded by FCT, the Portuguese Foundation for Science and Technology, under contract `POSI/CHS/44304/2002`, was hosted by the Theory and Formal Methods group of the Informatics Department, at Minho University, from November 2004 to January 2007.

the underlying *coordination model*, which abstracts the behavioural interplay between the various services, components, and the (more or less explicit) independent *loci* of computation from which a system is composed of.

Such an application area was, in fact, our original motivation for this research because our own experience in industry[2] singled out the reconstruction of *software architectures* as a difficult but highly relevant issue for the working software developer.

By the expression *software architecture* we understand, following [BCK98], *the set of specific scoped models that expose particular aspects of parts (possibly components, modules, services, processes) of the system and the interactions between them.* Or, according to norm ANSI/IEEE Std 1471-2000, which is part of an on-going standardisation effort, *the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*

Several approaches have been proposed for reverse architectural analysis. For example, in the context of model-driven engineering [Sch06], generators for UML diagrams became rather popular. *Class Diagram* generators, which extract class diagrams from object oriented source code, *Module Diagram* generators that construct box-line diagrams from system's modules, packages or namespaces, *Uses Diagram* generators which reflect the import dependencies of the system and *Call Diagram* generators which expose the direct calls between system parts, are but a few examples.

However, none of these techniques/tools make it possible to answer a critical question about the dynamics of a system: *how does it interact with its own components and external services and coordinate them to achieve its goals*? From a *Call Diagram*, for example, one may identify which parts of a system (and, sometimes, even what external systems) are called during the execution of a particular procedure. However, no answers are provided to questions like: Will the system try to communicate indefinitely if an external

[2] From April, 2003 to March, 2004 we have worked on software components and repositories in a development project at SIDEREUS, S. A., in Porto. Reference [RB03] and the last section of [BSAR05] report on research conducted in such a context.

resource is unavailable? If a particular process is down, will it cause the entire system to halt? Can the system enter in a deadlock situation? And what is the sequence of actions for such a deadlock to take place?

This sort of questions belongs to what can be called the *coordination* architectural layer, which captures the system's behaviour with respect to its network of interactions. The qualifier is borrowed from research on *coordination* models and languages [JMA96, Arb98], which emerged a decade ago to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

It is not surprising that the questions above cannot be answered within most of the models built from source code, because behavioural analysis is placed at a much higher abstraction level than most of other architectural structures. Actually, recovering a *coordination model* is a complex process dealing with multiple activities, roles and primitives, which in turn are influenced by multiple constrains, such as exceptional situations, interrupts and failures.

On the other hand, however, the need for methods and tools to identify, extract and record the coordination layer of running applications is becoming more and more relevant as an increasing number of software systems rely on non trivial coordination logic for combining autonomous services, typically running on different platforms and owned by different organisations. This is why this thesis adopts a *coordination-driven view* of software architecture, which underlies, in the sequel, the use of the adjective *architectural* qualifying *analysis*, *slicing* or *extraction* processes.

We claim that, if coordination policies can be extracted from source code and made explicit, it becomes easier to understand the system's *emergent behaviour* (which, by definition, is the behaviour which cannot be inferred directly from the individual components) as well as to verify the adequacy of the software architecture (and of the code itself) with respect to expected interaction patterns and constraints. We would like to regard this thesis, especially its second part, as a step in that direction.

1.1.2 Objectives

In the context detailed above, we defined the following research objectives which guided the development of this thesis:

- Investigate slicing techniques, in particular their application to different programming paradigms and potential to support re-engineering of legacy code.

- Develop new slicing techniques specifically designed for application to functional programs.

- Develop methods for extraction of architectural information from source code, based on application of slicing techniques and aimed at recovering the underlying, often implicit coordination policies.

- Test and validate the applicability of such methods and techniques through the development of "proof-of-concept" prototypes.

Our research intersects, therefore, two well-established areas in Software Engineering: *slicing* and *coordination* models in *software architecture*. The former provides our basic tools for analysis, the latter a major application challenge.

Therefore, the following two sections provide a brief introduction to both areas, as a background for the thesis. Our specific contributions are detailed in section 1.3 and traced back to the publications in which they were first introduced. This last section also provides an overview of the thesis structure.

1.2 Background

1.2.1 Slicing

Slicing Techniques and Applications

Slicing was first proposed by Mark Weiser [Wei79, Wei84] as a technique for program debugging [Wei82]. In a broad definition it stands for a decomposition technique that extracts from a program those pieces (typically, statements) relevant to a particular computation. A *slice*, in Weiser's original definition, is then a reduced, executable program obtained from another program by removing statements, such that it replicates part of the behaviour of the original program. A classical way of formulating the question program slicing is supposed to answer is as follows: *what statements in the program can potentially affect the value of a particular (set of) variable(s) at a particular execution point?* The answer is typically given in terms of an executable program whose execution is indistinguishable from the execution of the original program whenever the observer concentrates his attention on the value(s) of the variable(s) of interest (the *slicing criterion*). In Weiser's view, program slicing is an operation already being performed much before the term had been coined, in fact it is an abstraction exercise that every programmer has gone through, aware of it or not, every time he undertakes source code analysis.

Weiser approach corresponds to what would now be classified as an *executable*, *backward*, *static* slicing method. A dual concept is that of *forward slicing* introduced by Horwitz et. al. [HRB88]. In forward slicing one is interested on what depends on or is affected by the entity selected as the *slicing criterion.* Note that, combining the two methods also gives interesting results. In particular the union of a backward with a forward slice for the same criterion n provides a sort of a selective window over the code, highlighting the *code region* relevant for entity n. On the other hand, the intersection of a backward slice for a given slicing criterion with a forward slice with respect to another slicing criterion, retrieves what is called a *chop* [RR95, JR94], exposing the program elements contain within the two slicing criterions that may affect the second slicing criterion.

Another duality emerges between *static* and *dynamic* slicing. In the first case, only static program information is used, which typically consists on the source code of the program to be sliced and a slicing criterion. On the second case [KL88, KL90], one also considers input values of the program, leading frequently, due to the extra information used, to smaller and easier to analyse slices, although with a validity restricted to the values employed. References [Tip95], [BG96] and [HG98] provide comprehensive surveys and include extensive bibliographies.

What can be achieved by slicing, *i.e.* the isolation of a particular sub-computation of interest inside an entire program, goes far beyond its initial purpose of error detection. In fact, program slicing techniques became relevant to a large number of areas, such as, reverse engineering [CCM94, SVM+93], program understanding [dLFM96, HHD+01], debugging [ADS93, WL86], software integration [BHR95, HPR89], software maintenance [GL91, CCLL94, CLM96], testing and test planning [HH99, HD95], among others.

On the other hand, since the publication of Weiser´s paper, a myriad of slicing techniques, algorithms and variants have been proposed in the literature: slicing for declarative languages [RT96, Bis97], object oriented programs and multithreading [LH98, LH96, RH07, Kri03, NR00], conditional slicing [CCL98], slicing for abstract interpretation [HLS05] and monadic slicing [ZXS+04], just to mention but a fraction. Some of these approaches have come to play an important role in the course of our own work, in particular the so-called interprocedural slicing techniques [HRB88], as well as the extensions to multithreading and the declarative and object-oriented paradigms mentioned above.

Specifying the Slicing Problem

Most of the papers mentioned above give an informal definition of the meaning of a program slice and focus on defining and computing different sorts of program dependencies (e.g. related to data, control, etc.). As Martin Ward puts it in a recent paper [WZ07], "*this focus on dependency calculations confuses the definition of a slice with various algorithms for computing slices*".

On the other hand, a number of attempts have been made to formally characterise the slicing problem and frame it as some sort of transformation inside suitable models for program semantics. Actually, as a slice always corresponds to a fragment of a program and, being executable, to a program itself, one may talk about the associated semantics also as a sub-object (i.e., a subset, a sub-domain, a sub- whatever-semantical-structure is taken) of the semantics of the original program. As early as [GL91] it was observed that the set of slices of a given program, ordered by inclusion in the associates semantics, form a semi-lattice where the meet operation corresponds to code sharing.

Formalising the slicing problem makes it possible to compare and classify different forms of program slicing as done, for example, in [BDG$^+$06], and to assess how semantically sound are they as code decomposition techniques. The latter problem is addressed in [Oli01b] which proposes to resort to well-known program calculi laws (namely from the so-called *mathematics of program construction* community [BM97]) to compose slices and reconstruct a program proved by construction to be semantically equivalent to the original one.

Slicing is often formalised as a *program transformation* in a restriction (or projection) of the original program. The *rationale* underlying this characterisation is that, if the original program is restricted to (i.e., observed through) the variables of interest, then it should be semantically *equivalent* to the corresponding slice computed by statement deletion.

In a landmark paper [HBD03], published in 2003, Harman, Binkley and Danicic define a slice as a combination of a syntactic ordering (any computable, transitive, reflexive relation on programs), intended to capture the reduction of syntactic complexity, and a semantic equivalence on a projection of the program semantics. It should be noted that the authors go a step forward to emphasise the essentially semantic character of the slicing problem, and even coined the term *amorphous slice* to denote a slice which is not required to preserve a projection of the syntax of the original program. This makes the task of amorphous slice construction harder, but it also often makes the result thinner and thereby preferable in applications where syntax preservation is unimportant. In particular, amorphous slicing can be guided by the original program syntax tree and does not require the construction of control flow nor program dependence graphs.

From the point of view of formalising the slicing problem, however, what is relevant is to retain the concept of slicing as a combination of two relations:

- a syntactic relation, corresponding to some form of syntactic pruning

(e. g., statement deletion)

- a semantic relation, intending to show what subset of the original program semantics has been preserved through syntactic pruning.

Whether such semantic relation is an equivalence (as in [HBD03] and [GB03]) or a refinement (i.e., yielding a semantically less partial and more deterministic program [BvW98]), is still a matter of controversy. As almost all research on program slicing, the problem is being discussed in the context of sequential, imperative programs.

A noteworthy approach has been put forward by Martin Ward in a series of papers from [War03], to the recent detailed account in [WZ07]. Ward claims that in practice semantic equivalence is too strong a requirement to place on the definition of a slice. He proposes, instead, the notion of a *semi-refinement*: program S is a *semi-refinement* of P if it is semantically equivalent to P on all the domain of definition of P. This means that the behaviour of S can be arbitrary out of such a domain. Note that a theory of slicing based on semi-refinement allows for an infinite number of slices for a given program, introducing slices that are actually larger than the original program, something which is precluded by simple syntactic criteria. Such a view of slicing as a program transformation is formalised, for sequential, non deterministic programs, in FermaT, a workbench for program transformation supported by semi-automatic program manipulation (see in [War02, WZH05] for details).

1.2.2 Software Architecture and Coordination

Software Architecture

The term *architecture* became popular in Software Engineering as a way to refer to the high-level structure of a system, *i.e.* to its gross structure, main guidelines and constraints in which it is based. However, if, in a concrete development project, one enquires on what the systems' architecture is after all, the number of different answers will probably be equal to the number of people involved. Moreover each of the answers will be deeply influenced by each one specific role within the project.

In a sense, the popularity of expression *software architecture* (statements like "*the architecture does not allow so and so*" or "*let's stick to the original client-server architecture*" are quite common within development teams) goes hand in hand with its informality and the lack of a precise, consensual definition. Surprisingly enough, the set of concepts and models in this area turn out to be extremely useful in practice.

Actually, software architecture emerged in the early nineties, as a proper discipline in Software Engineering, from the need to explicitly consider, in the development of increasingly bigger and more complex systems, built of many components, the effects, problems and opportunities of the system's overall structure, organisation and emergent behaviour. The seminal work of Shaw and Garlan [SG96], which put forward many of the concepts and vocabulary still in use today, must be mentioned at this stage. But, let us concentrate on possible definitions of software architecture, to highlight later its view from a coordination perspective.

In a broad definition, the architecture of a system describes its fundamental organisation, which illuminates the top level design decisions, namely

- how is it composed and of which interacting parts?
- which are the interactions and communication patterns present?
- which are the key properties of parts the overall system relies and/or enforces?

Other definitions stress,

- *the systematic study of the overall structure of software systems* [GS93];
- *the structure of the components of a program/system, their interrelationships, principles and guidelines governing their design and evolution over time.* [GP94];

- *the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution* (norm ANSI/IEEE Std 1471-2000);

- *a set of architectural (or, if you will, design) elements that have a particular form; (...) we distinguish three different classes of architectural element: processing elements, data elements and connecting elements* [PW92];

- *the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.* [BCK03]

As a model, the software architecture abstracts away details of elements that do not affect how they use, are used by, relate to or interact with other elements. Therefore it focuses on the system structural elements and their interfaces, their interactions and composition into larger subsystems.

Quickly this area of concern became a mature discipline in Software Engineering, part of main curricula and object of popular textbooks (of which [BCK03] is a well-known example). In its rapid evolution along the last 15 years, it should be singled out both the classification of architectural *styles* [MKMG97], characterising families of software organisation, after the pioneering work of David Garlan [GS93, AG97, Gar03], and the impact of several proposals for so-called *architecture definition languages.* The latter provides both a conceptual framework and a concrete syntax to describe architectures, and often tools for parsing, displaying, analysing or simulating architectural descriptions. Well-known representatives are, among others, Wright [All97], Rapide [LAK$^+$95], Darwin [MDEK95], C2 [MORT96], Piccola [SN99, NA03] and ACME [GMW97].

Reference [Gar03] provides a comprehensive survey of the software architecture and related tools. Interesting application areas include software analysis and documentation [GS06], performance analysis [SG98], architectural recovery from legacy code [HRY95, KC98, Bou99, MMCG02] and architectural models for new computing paradigms, including mobility [WLF01, GS04, Oqu04] and web-service applications [ZKG02].

Coordination

The increasing dependency of society and economy on software, allied to their exponential growth, both in size and complexity, is pushing the adoption of componentware and service oriented architectures. In despite of everyday publicity, the complexity associated to the design, implementation and maintenance of component-based or service-oriented solutions cannot be underestimated, especially if one is demanding for rigorous and flexible solutions. The idea that, for example, a real service oriented system is just a series of instructions invoking foreign components or services which perform the entire complex work, is actually far from reality.

Problems arise because such systems have to deal with multiple *loci* of computation, providing functionalities to multiple participants at the same time, which in turn are influenced by multiple and different constraints, typically enforced by other services or components. Moreover, professional service oriented systems often live in multithreaded environments, because users have to be informed while the system is performing other tasks, or because the latency introduced by relying on external services, instead of local components, requires the developer to program asynchronous calls to external services and proceed execution only, and if, the service returns. So, correct, responsive service-oriented systems are required to be highly multithreaded and complex to orchestrate, due to the myriad of external services they may depend upon. To support their development, the software architecture should be able to provide specific coordination knowledge, *i.e.*, the structure and nature of interactions among the different sub-systems in presence.

The concept of *coordination* is not that new in Computer Science. Actually, the coordination paradigm [JMA96, Arb98], claiming for a strict separation between effective computation and its control, emerged, more than a decade ago, from the need to exploit the full potential of massively parallel systems. This entails the need for models able to deal, in an explicit way, with the concurrency of cooperation among a very large number of heterogeneous, autonomous and loosely-coupled components. Coordination models [GC92, PA98, Arb03] make a clear distinction between such components and their interactions, and focus on their joint *emergent* behaviour.

Traditionally, coordination models and languages have evolved around the notion of a shared *dataspace* — a memory abstraction accessible, for data exchanging, to all processes cooperating towards the achievement of a common goal. The first coordination language to introduce such a notion was Linda [ACG86]; many related models evolved later around similar notions [JMA96, BCG97]. The underlying model was *data-driven*, in the sense that processes can actually examine the nature of the exchanged data and act accordingly. An alternative family of models, called *event-driven* or *control-driven*, more suitable to systems whose components interact with each other by posting and receiving messages which trigger some activity. A pioneer model in this family is MANIFOLD [AHS93], which implements the IWIM model [Arb96]. Contrary to the case of the data-driven family where coordinators directly handle data values, in these models processes are regarded as black boxes and communicate with their environment by means of clearly defined interfaces (often referred to as input and output ports).

Another typical distinction is drawn between *endogenous* and *exogenous* coordination. The former treats glue code as a first-class modelling entity that resides outside of any of the components it coordinates, while the later, distributes control over the coordinated entities themselves, thus making impossible to single out a piece of code identifiable as the coordination module. REO [Arb03, Arb04] is a recent example of a model for *exogenous* coordination, while LINDA [ACG86], which provides a number of coordination primitives to be incorporated within a programming language, remains a prototypical example of an *endogenous* coordination language.

A Coordination-driven View of Software Architecture

Coordination models and architectural descriptions were born within different contexts, concerns and typical application domains. However their focuses are similar and recent trends in the software industry stresses the relevance of such common underlying principles. Recall, for example, the challenges entailed by the move from the *programming-in-the-large* paradigm of two decades ago, to the recent *programming-in-the-world* where not only one has to master the complexity of building and deploying a large application in time and budget, but also of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous. Or the related shift from the traditional understanding of software as a *product* to *software as a service* [Fia04], emphasising its open, dynamic re-configurable and evolutive structure. Terms like service *orchestration* and *choreography*, and the associated intensive research effort (see [BGG$^+$05, AF04, BCPV04, ZXCH07], among many others), stress the relevance of main themes in both coordination and architectural research to modern Software Engineering. In a sense, an early definition of coordination which emphasises its goal of *finding solutions to the problem of managing the interaction among concurrent programs* [Arb98], could be taken as a main challenge to this Engineering domain. As Farhad Arbab puts it, in his inaugural lecture at Leiden University,

> *We have been composing software since the inception of programming. Recognising the need to go beyond the success of available tools is sometimes more difficult than accepting to abandon what does not work. Our software composition models have served us well-enough to bring us up to a new plateau of software complexity and composition requirements beyond their own effectiveness. In this sense, they have become the victims of their own success. Dynamic composition of behavior by orchestrating the interactions among independent distributed subsystems or services has quickly gained prominence. We now need new models for software composition, on par with those commonly used in more mature engineering disciplines, such as mechanical or electrical engineering. This deficiency is one of the reasons why software engineering is sometimes criticised as not-truly-engineering.*

Actually, coordination models [GC92, PA98, Arb03] aim at finding solutions to the problem of managing the interaction among concurrent activities in a system. For the last 15 years, the emergence of massive concurrent, heterogeneous systems and the growing complexity of interaction protocols and concurrency relationships between different processes, often developed and deployed in a distributed way, have brought coordination to a central place in software development. Such development contributed to broadening its scope of application and entailed the development of a number of specific models and languages [Arb04].

Software architecture [GS93, PW92, FL97, BCK03, Gar03], on the other hand, describes the fundamental assembly structure of a system, and, a now mature discipline, plays a significant role in improving the dependability, documentation and maintainability of large, complex software systems [GS06].

Both of them tackle component interaction, abstracting away the details of computation and focussing on the nature and form of interactions. Synchronisation, communication, reconfiguration, creation and termination of computational activities are, thus, primary issues of concern.

It should also be remarked that, despite remarkable progress in the representation and use of software architecture, specification of architectural designs remain, at present, largely informal. Typically, they rely on graphical notations with poor semantics, and often limited to express only the most basic structural properties. Recent coordination models and languages, on the other hand, present a higher degree of formality — see, for example, the cases of Reo [Arb03, Arb04] or Orc [KCM06, MC07] — which stresses the case for a *coordination-driven view* of systems' architecture.

1.3 Contributions and Thesis Structure

The thesis contributions are organised in main two areas, corresponding, respectively, to Parts I and II,

- Slicing techniques for the functional programming paradigm;

- Slicing-based approaches to the identification and extraction of architectural, coordination specifications from source code.

The following enumeration details such contributions and relates each of them both to the corresponding chapter and the publications in which it was originally presented. Both Parts I and II end with a brief concluding chapter, chapters 7 and 12 respectively, which also discuss some relevant related work.

- Slicing for functional programming:
  - Specific techniques based on dependence graphs for functional programs, a line of enquiry closely related to mainstream slicing research for imperative languages. Basic results appeared in [BSR06, RB06a] and are reported in chapter 2 of this thesis.
  - An alternative approach to functional slicing by calculation, resorting to a well-known calculus of functions and addressing low-level programmatic entities. The approach first appeared in [RB06b] and is the subject of chapter 5.
  - A semantic based approach, which overcomes some limitations of the calculational techniques and addresses higher-order lazy functional languages. This was published in [RB07] and is detailed here along chapter 6.
  - An application of the graph-based approach to functional slicing to the problem of software component discovery in the context of Haskell programs. This case study appeared in [RB06a] and is reported in chapter 4.
  - A prototype tool, HaSlicer, developed as a proof-of-concept for some of the slicing techniques proposed. This was not object of an independent publication, but is mentioned in [RB06a]. The tool is presented in chapter 3 and available from `http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx`.
- Slicing-based approaches to discovering and extracting coordination specifications:

- New graph structures to represent coordination related information extracted from legacy code, and the algorithms to build them: the *Managed System Dependence Graph* (MSDG) and the *Coordination Dependence Graph* (CDG). These structures, introduced in chapter 8, are at the base of two different methods for coordination discovery discussed in chapters 8 and 9.
- A technique for generating coordination specifications from the CDG in Orc, which appeared in [RB08a] and is detailed in section 8.5 of chapter 8.
- A technique for generating coordination specifications from the CDG in Ws-Bpel, first published in [RB08c] and discussed here in section 8.6 of chapter 8.
- An alternative pattern-oriented discovery technique based on subgraph search over the CDG to detect instances of specific coordination patterns. This approach was proposed in [Rod08]. It is detailed in chapter 9 and applied, in chapter 11, to a real case study in the area of systems integration.
- A prototype tool, CoordInspector, developed as a proof-of-concept for both approaches to coordination discovery. The tool processes *Common Intermediate Language* (CIL) code, which makes it potentially able to analyse systems developed in more than 40 programming languages. CoordInspector, discussed in chapter 10, was first introduced in [RB08b]. It is available from `http://alfa.di.uminho.pt/~nfr/Tools/CoordInspector.zip`.

Part I

Functional Program Slicing

Chapter 2

A Graph-Oriented Approach

Mainstream research on program slicing targets imperative languages and, therefore, it is oriented towards particular, well characterised notions of computational variable, program statement and control flow behaviour. On the other hand, slicing functional programs requires a completely different perspective.

In a functional program, functions, rather than program statements, are the basic computational units and functional composition replaces statement sequencing. Moreover there is no notion of assignable variable or global state whatsoever. Even techniques like the use of the state monad [Wad92, Mog91] just simulate an underlying state: what is really happening behind the scenes, is that functions are being composed in a way that maintains an artificial state. Besides, in modern functional languages encapsulation constructs, such as HASKELL [Bir98] *modules* or ML [HM86] *abstract data types*, provide powerful structuring mechanisms which cannot be ignored in program understanding.

What are then suitable notions of slicing for functional programs? With respect to what are functional slices computed? How is program data extracted and in which form should it be stored? Such are the questions set in this chapter.

Although different approaches will be discussed later, namely in chapters 5 and 6, the questions just stated are addressed here from the point of view of traditional graph based approaches to the slicing problem. Moreover, our approach regards an architectural view of functional programs, where one is interested in high level code entities such as functions, modules and data types rather than more fine grained entities like functional expressions upon which functions are built upon. Focusing on this high level code entities, one is concerned about analysing the different interactions these entities may hold as well as the development of program transformation algorithms, like slicing, which explore the discovered interactions.

2.1 Functional Dependence Graphs

Most slicing techniques are based on some kind of *dependence graph*, where in general, nodes represent slicing units of program entities and different kinds of edges are used to store a number of types of dependencies among such units. Typical such structures are extracted from the program's control flow graph which is based on a precise notion of *program statement*. Such program statements are usually regarded by graph based slicing techniques as its slicing units and are often defined as expressions manipulating a shared variable state.

Unlike programs from other paradigms, functional programs do not identify a precise and well defined notion of program statement. Rather, they define a programming logic by using rich data structures, often recursive, and by composing functional expressions in diverse and complex ways. Thus, one needs to adapt the definition of whatever a dependence graph is, shift it to the functional paradigm, and replace program statements by functional program entities such as constructors, destructors, data types, functions and modules. Such a program representation structure can form the basis of meaningful slicing criteria, which leads us to defining a *Functional Dependence Graph* (FDG) as a directed graph

$$G = (E, N) \tag{2.1}$$

where N is a set of nodes and $E \subseteq N \times N$ a set of edges represented as a binary relation between nodes. A node $N = (t, s, d)$ consists of a node type t, of type $NType$, a source code location s, of type $SrcLoc$, and a description d of type $Descr$.

A source code location is simply an index of the node contents in the actual source code. The type $SrcLoc$ is a product composed by the source file name and the line-column code coordinates of a particular program element, *i.e.*,

$$\begin{aligned} SrcLoc \;&= SrcFileName \times SrcBgnLine \times SrcBgnColumn \times \\ &\quad SrcEndLine \times SrcEndColumn \end{aligned} \tag{2.2}$$

More interesting is the definition of a node type which captures the information diversity mentioned above and is the cornerstone of FDG's flexibility. The type of a FDG node is given by the following union type

$$\begin{aligned} NType \;&= N_m(\text{module}) \mid N_f(\text{function}) \mid N_{dt}(\text{data type}) \\ &\mid N_c(\text{constructor}) \mid N_d(\text{destructor}) \end{aligned} \tag{2.3}$$

Let us explain in some detail the intuition behind these types.

Nodes bearing the N_m (Module) type, represent software modules, which, from the program analysis point of view, correspond to the highest level of abstraction over source code. Note that Haskell has a concrete definition of module, which makes the identification of N_m nodes straightforward. Modules encapsulate several program entities, in particular code fragments that give rise to other FDG nodes. Thus, a N_m node depends on every other node representing entities defined inside the module as well as on nodes corresponding to modules it may import.

Nodes of type N_f represent functions, *i.e.*, abstractions of computational processes which transform some kind of input information (eventually void) into an output. Functions are the building blocks of functional programs, which in most cases, decorate them with suitable type information, making extraction simpler. More complex is the task of relating a function node to the nodes corresponding to computational entities in its body — data type references, other functions or what we shall call below *functional statements*.

Constructor nodes (N_c) are used to explicitly represent the implicit constructors, usually one for each alternative in the defining sum type underlying a typical data type declaration. These are especially relevant to functional languages admitting several constructors for a given data type (such as the ones associated to *datatype declarations* in HASKELL).

Destructor nodes (N_d) store data type selectors, which are dual to constructors. Unlike constructor nodes, for which a data type must always have a dependence to at least one of these nodes, destructor nodes may not be present in a data type definition. Actually, in practice, what usually happens is that the destruction of the data types values is made by pattern matching clauses involving the constructors of the data type. In any of these situations one can always capture a dependence between any program entity and a data type by tracing the dependence between the program entity and a direct use of one of its destructors or the indirect use, through pattern matching, of one of its constructors.

Note that, similar notions to these constructor and destructor operators may, however, be found in other contexts note necessarily functional. Recall, for example, the C selector operator "`.`" which retrieves specific fields from a `struct` value and class constructors from object oriented languages.

All sets of nodes in a FDG are interconnected by edges. In all cases an edge from a node n_1 to a node n_2 witnesses a dependence relation of n_2 on n_1. The semantics of such a relation, however, depends on the types of both nodes. For example, an edge from a N_f (function) node n_1 to a N_m (module) node n_2 means that the module represented by n_2 depends on the function associated to n_1, that is, in particular, that the function in n_1 is defined inside the module in n_2. On the other hand, an edge from a node n_3 to n_4, both of type N_f, witnesses a dependence of the function in n_4 on the one in n_3. This means, in particular, the latter is called by the former.

Table 2.1 introduces the intended semantics of edges with respect to the types of nodes they connect. Also note that a FDG represents only *direct* dependencies. For example there is no node in a FDG to witness the fact

Target	**Source**	**Edge Semantic**
N_m	$\{N_m\}$	Target node imports source node
N_m	$\{N_f,\ N_c,\ N_d,\ N_{dt}\ \}$	Source node contains target node definition
N_f	$\{N_c, N_d,\ N_{dt}, N_f\}$	Function is using target node functionality
N_{dt}	$\{N_{dt}\}$	Source data-type is using target data-type
N_{dt}	$\{N_c\}$	Data-type is constructed by target node
N_{dt}	$\{N_d\}$	Data-type is destructed by target node

Table 2.1: *FDG edge description*

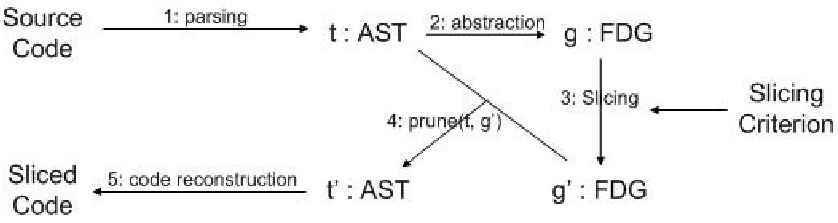


Figure 2.1: The slicing process

that a module uses a function defined elsewhere. What would be represented in such a case is a relationship between the external function and the internal one which calls it. From there, the indirect dependence could be retrieved by a particular slicing criterion. As it can be easily anticipated, slicing criteria in this approach corresponds to nodes in a FDG.

2.2 The Slicing Process

Program slicing based on *Functional Dependence Graphs* is a five phase process, as illustrated in Figure 2.1.

As expected, the first phase corresponds to the parsing of the source code to produce an *abstract syntax tree* (AST) instance t. This is followed by an abstraction process that extracts the relevant information from t, constructing a FDG instance g according to the different types of nodes found.

The third phase is where the actual slicing takes place. Here, given a slicing criterion, composed by a node from t and a specific slicing algorithm, the original FDG g is sliced, originating the sub-graph g'. Note that, slicing takes place over the FDG, and that the result is always a sub-graph of the original graph.

The fourth phase, is responsible for pruning the AST t, based on the sliced graph g'. At this point, each program entity that is not present in graph g', is used to prune the correspondent syntactic entity in t, giving origin to a subtree t' of t. Finally, code reconstruction takes place, where the pruned tree t' is consumed to generate the sliced program by a process inverse to the one of phase 1.

In the next section, a number of what we have called *slicing combinators* is formally defined, as operators in the relational calculus [BH93], on top of which the actual slicing algorithms, underlying phases three and four above, are implemented. This provides a basis for an algebra of static program slicing over graph structures. This formalisation of the functional slicing algorithms helped considerably not only at verifying the correctness of the algorithms, but also in deriving the implementation of the slicing algorithms, as explained in chapter 3.

2.3 Slicing Combinators

Given that both the *extraction* and *code reconstruction* phases amount basically to a language engineering problem, we shall concentrate now on the specification of the slicing algorithms. Actually, the combinators can be defined over any directed graph G. Therefore, in the sequel, we shall abstract from the node/edge type information as introduced above.

Let us consider first the *forward slicing* process, taking as slicing criteria a single FDG node n.

A forward slice computed over a graph G from a node n consists of the G sub-graph including all nodes which depend, either directly or indirectly, on

n. It is convenient to rephrase the definition of a graph G to the equivalent form of a relation $G : N \longleftarrow N$, where nGm means *node n depends on node m*. Thus we may express the forward slicing operation, represented by $G \otimes n$, as the least fixed point of the following equation

$$x \;=\; G \cdot \lfloor n \rceil \,\cup\, \mathsf{next}_G\, x \tag{2.4}$$

where $\mathsf{next}_G\, x = G \cdot \mathit{rng}\, x$ and $\lfloor n \rceil : N \longleftarrow N$ is the singleton coreflexive associated to node n. Recall (from e.g. [Oli08]), that $\mathit{rng}\ S$ denotes the range of relation S. Therefore,

$$G \otimes n \;\triangleq\; \mu\, x \,.\, (G \cdot \lfloor n \rceil \,\cup\, \mathsf{next}_G\, x) \tag{2.5}$$

where μ is the least fixed point operator [Bac02].

Applying the *rolling rule*[1] with $g\, x \;=\; G \cdot \lfloor n \rceil \cup x$ and $h\, x \;=\; \mathsf{next}_G\, x$, yields

$$\begin{array}{cl}
 & G \otimes n \\
\Leftarrow & \qquad \{\text{definition, rolling rule }\} \\
 & G \cdot \lfloor n \rceil \,\cup\, \mu\, x \,.\, (G \cdot \mathit{rng}\,(G \cdot \lfloor n \rceil \,\cup\, x)) \\
\Leftarrow & \qquad \{\mathit{rng}\ \ \text{preserves} \cup \} \\
 & G \cdot \lfloor n \rceil \,\cup\, \mu\, x \,.\, (G \cdot (\mathit{rng}\,(G \cdot \lfloor n \rceil) \,\cup\, \mathit{rng}\, x)) \\
= & \qquad \{\cdot \text{ distributes over } \cup \} \\
 & G \cdot \lfloor n \rceil \,\cup\, \mu(G \cdot \mathit{rng}\,(G \cdot \lfloor n \rceil) \,\cup\, G \cdot \mathit{rng}\, x)) \\
= & \qquad \{\text{definition}\} \\
 & G \cdot \lfloor n \rceil \,\cup\, \mu\, x \,.\, (\mathsf{next}_G\,(G \cdot \lfloor n \rceil) \,\cup\, \mathsf{next}_G\, x\,)
\end{array}$$

which may help to build up the correct intuition about the definition: in each iteration a new level of *descendent* nodes is added to the incremental slice.

The transformation algorithm specified by $G \otimes n$ can be characterized

[1] In the fixed point calculus, the rolling rule — $\mu(g \cdot h) \;=\; g(\mu(h \cdot g))$ — provides a way of unfolding fixed point definitions [Bac02].

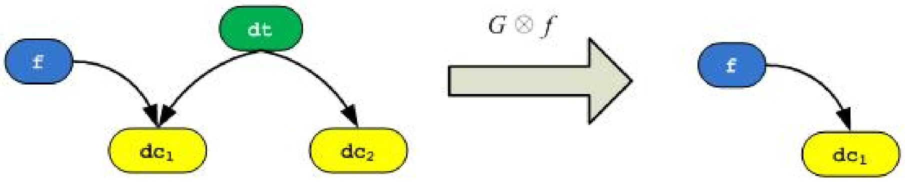


Figure 2.2: Non-executable forward slice

as a static forward slicing operation. Static because it is only based on the FDG instance, which is computed by just taking the program source code as input. Although it may seem that the slices produced by this specification are executable functional programs, such is not the case, because of the way functions may depend on data types and how these data type dependencies are captured in what respects to its constructor and destructor nodes.

Actually, non-executable forward slices are indeed produced by this specification whenever one performs a forward slice with respect to a function f that makes use of a data-type constructor dt_1 (or destructor) without making any explicit reference to the use of the data type dt for which the constructor (or destructor) belongs. Such a situation would lead to the construction of a slice containing function f definition and the data type constructor used dc_1, not including the remaining definition of the used data type which is crucial to obtain an executable slice. Figure 2.2 illustrates this case in FDG terms.

Although this forward slicing algorithm may produce non-executable slices, it is still useful not only from a program understanding point of view but also as a basis for other interesting program analysis operations. Such a notion of forward slicing over FDG's can be used to compute the components of a data type effectively used in a program, by inspecting the union of all forward slices obtained using every function that uses the data type as a slicing criterion. With such an operation, one could, for instance, simplify the data type in question by removing the parts not present in the computed slices. In other cases, the detection of unused parts of a data type may well indicate that the program is not working as expected and thus lead to an early error discovery. We shall come later to the problem of specifying an executable forward slice, but first we have to set the scene by introducing some other slicing operations that are needed for such a specification.

To begin with, consider the dual slicing operation: *backward slicing*, denoted by $n \oplus G$, can be regarded as forward slicing over the *converse* FDG, *i.e.*,

$$n \oplus G \ \triangleq \ G^{\circ} \otimes n \tag{2.6}$$

where R° denotes relational converse.

This time, the obtained slice may not be executable because it collects, form the initial program, every program entity x that depends on the slicing criterion n, without necessarily including in this collection every program entity for which x depends upon to operate.

As an example of a program that may originate a non executable backward slice, consider the case where a function f_1 is dependent on a given slicing criterion and also on another function f_2 that does not depend on this slicing criterion. In such a program, our backward slicing algorithm collects the program entities visited by a reverse traversal over the dependencies edges with origin in the slicing criterion. Clearly, the obtained backward slice contains function f_1 but not function f_2 which is crucial for the execution of f_1 and thus to the slice to be qualified as executable.

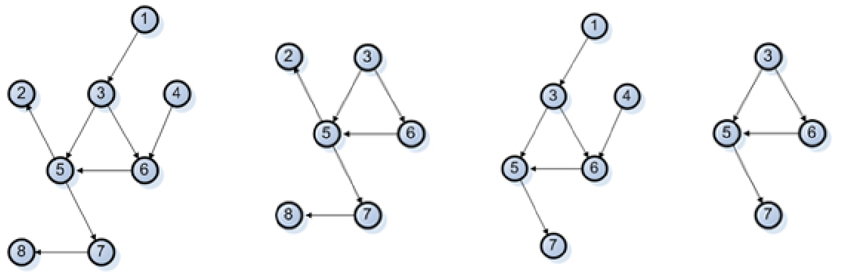
Any of the combinators — $\otimes$ or $\oplus$ — can be taken as the "building blocks" of an entire slicing algebra. For example, given two, not necessarily distinct nodes, n and m, one may define what is called a *chop* limited by n and m [JR94], as

$$\mathsf{chop}(n, m) \ \triangleq \ (G \otimes n) \cap (m \oplus G) \tag{2.7}$$

Figure 2.3 illustrates a chop limited by nodes labelled 3 and 7.

Note that a chop can be quite useful in program analysis. A typical situation arises when an error is identified between two points in the source code but its exact occurrence is not obvious. In such a case the corresponding *chop* would isolate the relevant code between the two points.

By reusing the previous definitions, we can now define a combinator that



(a) G (b) $G \otimes 3$ (c) $7 \oplus G$ (d) $\mathsf{chop}(3,7)$

Figure 2.3: Chopping with FDG

computes the whole *influence area* of a given program entity. Such a combinator can be defined as the union of the forward and backward slices of the given program entity node:

$$\mathsf{area}(n) \triangleq (G \otimes n) \cup (n \oplus G) \tag{2.8}$$

In practice, slicing a whole FDG often leads to very large slices, difficult to manage and analyse. An expedite method to substantially reduce the size of a slice consists of restricting the nodes in the computed sub-graph to a specified subset of node types. For example, if the focus is placed on finding dependencies between functions and modules, the calculated slice should be a sub-graph restricted to nodes of type N_f or N_m. Following this approach, slicing operators become *parametric* on a set T of *node types of interest*. This means that the slicing criterion adds to the *seed node* a subset of *NType* to specify the types of nodes allowed to appear in the slice. The definition in equation (2.5) extends smoothly to this case. Let $\phi_T = \in_T \cdot \pi_1$ [2]. Then,

$$G \otimes_T n \triangleq \mu x . (G \cdot \lfloor n \rceil \cup G \cdot (rng\, x \cap [\![\phi_T]\!])) \tag{2.9}$$

and, of course, $n \oplus_T G = G^\circ \otimes_T n$. In fact, the whole slicing algebra becomes parametric on T. For example the inter-relations of a datatype with other

[2] In pointwise notation predicate ϕ_T reads $\phi_T\, n = \pi_1(n) \in T$, where π_1 is the first projection in a cartesian product.

datatypes, constructors and selectors can be computed by a generic version of the area combinator parametric on $T = \{N_{dt}, N_c, N_d\}$.

We are now able to define an executable forward slicing algorithm which we shall call *complete forward slicing.* The difference between this algorithm and the previously presented forward slicing operator is that, when faced with a situation where a program entity depends upon a constructor or destructor node, the new algorithm collects not only the constructor or destructor node in question, but also the entire data type definition to which it belongs.

The result of applying this algorithm, with slicing criterion f, to the example of Figure 2.2, is a slice exactly equal to the initial program. So, as expected, obtaining executable slices comes to the price of increased length and complexity.

Again, this operator of complete forward slicing can be formalised as

$$G \,\overline{\otimes}\, n \;\triangleq\; G \otimes n \cup \Big(\bigcup_{u \in G \otimes n} u \oplus_{\{N_d, N_c, N_dt\}} G \Big) \tag{2.10}$$

One may argue that, even if computed by complete forward slicing, slices may not be executable programs, since it may happen that the `main` function is not present in the slice. This is usually taken as the entry point of the compiled code and often regarded as a requirement in the definition of an executable slice, at least in imperative settings. Nevertheless, we propose that, in the context of functional programming, the suitable interpretation of executable slice is that of a functional program, where each function defined in it has access to every program entity it depends upon to operate. On the contrary, if the calculated slices contained functions that depend on some other sliced program entity (like a data type, or another function), then, one would consider such slices as non executable.

Moreover there are other program analysis operations which, although only indirectly regarded as members of the slicing family, can be defined over a FDG. A typical example is testImport which, given a FDG G and two nodes of type N_m, n_1 and n_2, such that $n_2 \, G \, n_1$ (*i.e.*, module in n_2 imports the one in n_1), returns true if there is at least an entity defined in n_2 which depends on an entity defined in n_1. Otherwise one may conclude that the import statement is redundant, *i.e.*, no services of the imported module are really used. Of course nodes of type N_m are excluded from this test. Formally,

$$\mathsf{testImport}(n_1, n_2) \;\; \triangleq \mathsf{top}(dom\,(G_m^\circ \cdot \lfloor n_1 \rceil), dom\,(G_m^\circ \cdot \lfloor n_2 \rceil)) \cap G \;=\; \emptyset$$

where G_m is the restriction of FDG G to nodes of type different from N_m and $\mathsf{top}(S, T)$, for coreflexives S and T, builds relation $\{(x, y), (y, x) |\ (x, x) \in S \wedge (y, y) \in T\}$.

Chapter 3

HaSlicer

This chapter introduces HaSlicer a prototype tool built as a proof-of-concept for the ideas presented in the previous chapter.

Although functional languages have recently witnessed an enormous growth in popularity, suitable tools for (functional) program comprehension are still lacking. Also lacking are useful visualisations of program entities, upon which programmers can support their understanding of the system aspects being inspected.

HaSlicer is a step in such a direction: not only a tool able to perform diverse kinds of slicing, but also to deliver a useful visualisation of the high-level program entities under analysis, as well as of the calculated slices. Even more, the tool was designed in a way that provides the user with a functional framework upon which he can easily develop other FDG based operations, as suggested above.

The first version of HaSlicer was released to the public in March 2006 as a *Web Application* available from http://labdotnet.di.uminho.pt/HaSlicing/HaSlicing.aspx. The second version of the tool, with improved slicing operations as well as better FDG navigation and visualisation possibilities, was released in February 2007, again as a *Web Application* at http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx. This `url` still hosts the latest version of HaSlicer.

Since February 2007, all programs submitted for analysis through the web, as well as the computed slices, have been recorded by the tool in a specific server. A quick examination of the data submitted from February 2007 to October 2008, shows that HaSlicer has been used to analyse 190 programs, corresponding to 11325.7 KB of Haskell source code, distributed through 1087 files. In 150 cases, out of the 190 submissions, slicing was effectively carried on, while in the remaining cases the tool was mainly used for visualising and navigating through the generated FDG.

3.1 The HaSlicer Prototype

HaSlicer is a faithful implementation of the processing schema of Figure 2.1, with each phase isolated in a different software component. The tool implements the previously presented specifications for *forward*, *backward* and *complete forward* slicing.

For the first phase, which consists of the parsing of the source code, HaSlicer uses the Haskell parser from the GHC (Glasgow Haskell Compiler) libraries which delivers an abstract syntax tree (AST) as a Haskell data type instance.

This choice of resorting to the built-in GHC Haskell parser brings, however, some limitations to the range of Haskell programs that the tool will be able to deal with. The specific limitation introduced by this parser is its restriction to handle only “pure” Haskell programs, *i.e.*, programs that do not make any use of pre-processing instructions nor C-like foreign functions calls. This may not seem to constitute such a great limitation to the tool, given that most functional programs do not make use pre-processing instructions nor use foreign functions calls. Nevertheless, in order to be useful the tool must take into consideration all the libraries used by the program under analysis and, in particular, the GHC libraries (or the libraries from the compiler being used) which most often contain pre-processor and foreign function calls. This problem is addressed in the following phase of the analysis process implemented by HaSlicer.

The second phase consists of an extensive analysis of the parsed AST in order to construct the corresponding FDG. For this, one could have used one of the several program transformations and visitor pattern libraries available in the literature [Vis01, LV03], in order to strategically consume the AST and generate the correspondent FDG. However, the transformation of an AST to a FDG contains specific details that greatly complicate its implementation using one of the strategical traversal libraries. Among other implementation details, this operation needs to have several values of temporal states capturing the dependence's in a given scope of the program, which must then be combined in an overall dependency structure. Thus, we decided to implement this transformation directly in Haskell, resorting to the language constructs and some standard libraries.

The result of having performed this transformation directly in Haskell is a component that makes heavy use of pattern matching and polymorphism in order to extract not only the program entities but also every dependency between them. This is one of the most important and difficult phases in the entire process because of the great abstraction distance between the source code, represented in the form of an AST, and the FDG model.

One of the most relevant issues that had to be resolved during the implementation of HaSlicer was the problem of capturing the imports between functional modules, especially the imports of system libraries that one does not usually have direct access to the source code. This is a critical issue, since every program relies on some imports (even when not explicitly importing anything, as Haskell programs import, by default, the prelude library), and those dependencies are often quiet relevant from a program analysis point of view. In particular the previous chapter formula for calculating the unnecessary imports of a module (or of an entire project), can only be implemented if the code analysis is sensible about the several imports that are taking place in the code being analysed. Moreover, every program analysis based on dependencies between programmatic entities, like slicing operations, are extremely sensible to the imports and the uses of the definitions from such imports by the program under analysis. The main reason for this impact in program analysis is the fact that, by using imported definitions, a program may actually carry underlying and entirely new dependency graphs involving the imports of the imported modules.

To overcome this problem of having to deal with imports, HaSlicer was given access to the source code of all GHC libraries as user defined code, each time they are referred in some import from the program under scrutiny.

This decision, imposed a careful pre-processing of all the GHC libraries in order to remove every foreign function call and other language extensions that the GHC parser does not interpret. However, we are aware that in doing so we, one may remove some dependencies between the libraries code entities that should have been taken into consideration on the final FDG instance. Because of this, it may be possible that some slices produced by HaSlicer are not as accurate as one would like, in what respects to programs using libraries containing language extensions. Nevertheless, this impact over the precision of slices is largely reduced in practice, because, although HaSlicer takes into consideration the libraries used, it does not outputs by default the sliced libraries in the calculation of the final slice.

However, even by using the pre-processed libraries, this solution comes with a performance problem *i.e.*, HaSlicer has to analyse the same static libraries, and produce the correspondent FDG's, each time it is invoked on a program that refers to those libraries. In order to overcome this, we have introduced an incremental behaviour in the analysis of libraries. Thus, each time a new library is imported in some program, the HaSlicer performs phases one and two over the library, obtaining a FDG instance of that library which is then stored permanently in a suitable XML format.

When analysing a program that refers to a library that happens to have been referred to by another program previously analysed by the tool or by some other module of the program under analysis, HaSlicer just reads the previously stored FDG instance and merges it into the overall FDG of the program.

Once computed the entire FDG of the program, that is the FDG of the user defined modules and all the modules it imports, the program analysis process demands a suitable visualisation of the obtained graph structure. Here, several visualisation techniques were tried in order to give to the user the best overview of the program entities in the graph as well as a good graph navigation mechanism.

Among other approaches, we have tried a static display of the graph with tools like Graphviz, which delivered a plain display of the graph with navigation capabilities[1], and an elliptical display of the graph with smooth navigation facilities, which is the technique used in the current version of HaSlicer.

This kind of visualisation is currently implemented as a Java applet running on the client browser, but it can easily be adapted in the future to a standalone program analysis application. It displays the graph in an elliptical area where the nodes nearest to the centre of the display are enlarged with respect to the ones in the peripheral zones of the ellipse, where nodes are presented with smaller dimensions or even completely hidden. This is illustrated in Figure 3.1. This visualisation structure is preserved during the navigation process, where the user can drag the entire graph in order to analyse particular parts of the FDG. Also concerning graph navigation of large FDGs, HaSlicer is able to perform searches for particular nodes in the graph by positioning a node that matches a particular string in the centre of the displaying area.

The third phase of the slicing process is concerned with the pruning of the FDG based on a slicing criterion. This slicing criterion usually consists of a FDG node, though, as presented in the previous chapter, this definition may vary depending on the algorithm used to perform the desired program analysis.

This phase is entirely based on the specification of the slicing operators introduced in the previous chapter. Actually, as these are entirely developed in Haskell, the implementation is quite straightforward, amounting almost to a direct translation process. For example, the relational calculus formula

$$G \otimes n \ \triangleq\ \mu\, x\, .\, (G \cdot \lfloor n \rceil \cup \mathsf{next}_G\, x) \tag{3.1}$$

is implemented as

```
fs :: Ord a => Rel a a -> a -> Rel a a -> Rel a a
```

[1] Version beta 1 available at http://labdotnet.di.uminho.pt/HaSlicing/HaSlicing.aspx uses this kind of visualisation technique

Node Colour	Node Type
🟥	N_m
🟦	N_f
🟩	N_{dt}
🟨	N_c
⬛	N_d

Table 3.1: FDG edge codes

```
fs g n x = (g ‘comp‘ (singRel n)) ‘union‘
           (g ‘comp‘ (rngRel x))

(|+|) :: Ord a => Rel a a -> a -> Rel a a
g |+| n = relFix $ fs g n
```

where `relFix` is the fixed point on relations. All other relational formulas over FDG's are implemented in a similar way.

The fourth phase consists of a pruning traversal of the abstract syntax tree, obtained during the parsing phase, in order to remove every expression that is not present in the sliced FDG.

The fifth, and final, phase concerns code reconstruction of the obtained sub-AST from the previous phase. For this, HaSlicer uses the GHC pretty printer that works over the same data type of the AST's returned by the parser. In order to be able to use this pretty printer, the AST pruning must be performed in a way to deliver a reduced, but valid, AST instance.

Because of this use of GHC pretty printer, the sliced code may have a different aspect from the original *i.e.*, some tabs and white spaces may be missing as well as some line breaks. Nevertheless, the code itself, *i.e.* the functional expressions, is the same as the one present in the original code.

Figure 3.1 shows a snapshot of HaSlicer working over a Haskell program. Note that the differently coloured nodes indicate different program entity types according to Table 3.1.

Figure 3.2 reproduces the sub-graph corresponding to a slice over one of the nodes of the graph in Figure 3.1. Once a slice has been computed, the user may retrieve the corresponding sliced code. The whole process can also be undone or launched again with different criteria or object files. This and other usability aspects of the tool are presented in the next section in the format of a mini-tutorial.

Although its current version only accepts HASKELL code, plug-ins for other functional languages as well as for the VDM [FLM$^+$05] metalanguage [FL98] are currently under development.

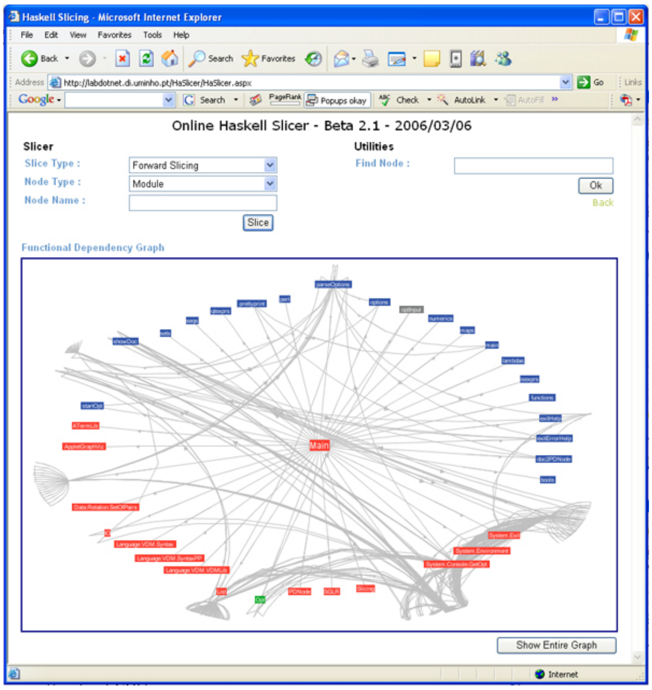
3.2 Working With HaSlicer

To illustrate HASLICER in action, we will analyse an HASKELL project which extends the capabilities of HASLICER to analyse VDM-SL code [FL98, Jon86]. Note that this is a kind of "meta-analysis", since one is using the already implemented HASLICER to analyse an extension to this same tool. In particular, the module that one will analyse resorts to the calculation of the FDG from a VDM-SL code file. The project is composed of 13 files containing user defined modules and about 5000 lines of code. It is advisable for the reader to follow this small tutorial while performing the correspondent actions in HASLICER.

The first step in using HASLICER consists of submitting the source code of the project to be analysed to the tool. In this particular case, the source code for project VDM2FDG is available as sample 3 (file `sample3.zip`) from the samples list in the tool.

Once this sample file is downloaded, the user has to submit it to the tool by locally browsing the file and then clicking the `Submit File` button. Here the user can either submit the source files as a zip file containing all the project source code files, or every source code file separately. Once the entire project source code is submitted to HASLICER the user must select the project main file from the drop down control labelled `Main File`. In the case of the VDM2FDG project the user should select file `SlicingVDM.hs`, and then click the `Generate Graph` button so that HASLICER calculates and presents the correspondent FDG instance.

Figure 3.1 shows the aspect of HASLICER once the project is loaded and



Online Haskell Slicer - Beta 2.1 - 2006/03/06

Slicer

Slice Type : Forward Slicing

Node Type : Module

Node Name :

Utilities

Find Node :

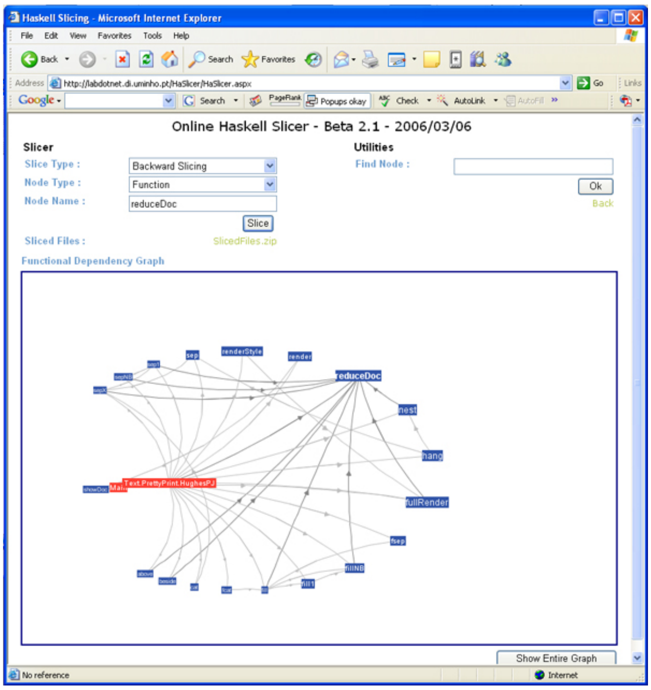
Functional Dependency Graph

Figure 3.1: VDM2FDG loaded in HaSlicer

the FDG is computed and visualised. Note that only the nodes closest to the main module entities are visible and that all others are hidden in the periphery of the visualiser ellipse. This kind of visualisation technique has proven to be very useful, especially when dealing with very large projects.

Just by looking at the FDG of the entire project, the analyst gets already an overall view of the dependency layout of the entire system. He can also navigate towards particular areas in the FDG. A useful functionality provided by HaSlicer is the ability to search a particular node in the entire graph by introducing the node name in the upper right utilities area.

Continuing the tool demonstration, suppose now that the user wants to analyse the potential impact of changing function `reduceDoc` in the rest of the project. In such a case, the user first has to locate the node corresponding to function `reduceDoc`. For this, he could use the search utility of HaSlicer. In return HaSlicer would display the node for function `reduceDoc` in the centre of the display area, enabling the user to perform a preliminary visual


Haskell Slicing - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx
Go Links
Google Search PageRank Popups okay Check AutoLink AutoFill
Slice
Ok
Back
sep
renderStyle
render
sep1
sepNB
sepX
reduceDoc
nest
hang
fullRender
showDoc
Main
Text.PrettyPrint.HughesPJ
fsep
fillNB
fill1
above
beside
cat
fcat
fill
Show Entire Graph
No reference
Internet

Online Haskell Slicer - Beta 2.1 - 2006/03/06

Slicer

Slice Type : Backward Slicing

Node Type : Function

Node Name : reduceDoc

Sliced Files : SlicedFiles.zip

Functional Dependency Graph

Utilities

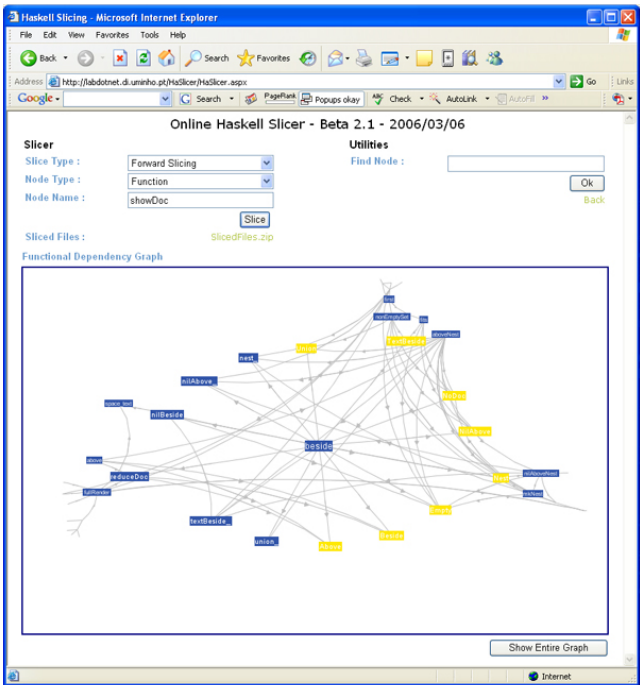
Find Node :

Figure 3.2: Backward slice w.r.t `reduceDoc`

analysis of its immediate dependant entities.

After having visually examined the part of the system in question, should he be interested in further calculating every possibly affected entity, he would then perform a backward slice with respect to function `reduceDoc`. This can be accomplished by using the upper left control in HaSlicer and selecting as `Slice Type` the option `Backward Slicing`, as `Node Type` the option `function`, as `Node Name` the string `reduceDoc` and finally clicking the slice button. The result of performing such a slice can be seen in Figure 3.2. Once the slice is performed a link bellow the slice button appears so that the analyst may download the code corresponding to the performed slice.

Now, we have the complete set of the entities that might be affected by changing function `reduceDoc`. Note that, in this example, function `showDoc`, defined in a different module from function `reduceDoc`, can also be affected by a change in function `reduceDoc` which could not have been predicted by a quick manual impact analysis. This and other complex dependencies can



http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx

Online Haskell Slicer - Beta 2.1 - 2006/03/06

Slicer

Slice Type : Forward Slicing

Node Type : Function

Node Name : showDoc

Slice

Sliced Files : SlicedFiles.zip

Utilities

Find Node :

Ok

Back

Functional Dependency Graph

Figure 3.3: Forward slicer w.r.t `showDoc`

be calculated by HaSlicer, giving the analyst a visual scenario of what is the impact of changing some code entity.

The impact analysis performed over function `reduceDoc` is useful not just to indicate what can be affected, but also tp point out what certainly is not going to be affected by a particular change in the code. This information assures the programmer undertaking modifications on the code, that whatever he might change in the program, it will certainly not affect some other clearly identified parts of the project.

Once accomplished a given program analysis, if desired, the user may navigate back and inspect the entire graph of the submitted project by clicking the button `Show Entire Graph` in the lower right side of the tool.

Suppose now that the VDM2FDG programmer has moved on to another project, and that he realises that he needs some of the functionalities he had developed for the first project. Let us assume that for the second project the programmer needs to implement a pretty printer for documents which he already did for the first project with function `showDoc`. An unpractical, but valid, solution would be to import the entire first project into the current project, with, of course, the obvious disadvantages of possibly incurring in name clashes and over charging the compiler with useless code. A better solution would be to use HaSlicer in a similar way to the previous `reduceDoc` case, but this time selecting the option `Forward Dep Slice` (due to previous versions compatibility reasons, in HaSlicer the option `Forward Dep Slice` corresponds to the previously presented complete forward slicing algorithm) with respect to `showDoc`. Such a slice is displayed in Figure 3.3 and it is not certainly trivial to compute manually. The code contained in this slice corresponds to the minimum subprogram from the original VDM2FDG project that implements function `showDoc`. Finally, the programmer could use the computed slice and import it in the second project, obtaining a much clearer solution without any unnecessary code.

Chapter 4

Component Discovery: A Case Study in Functional Slicing

A fundamental problem in system's re-engineering is the identification of coherent units of code providing recurrently used services or functionalities. Such units, which are typically organised around a collection of data structures or inter-related functions, can be wrapped around an interface and made available as software components in a modular architectural reconstruction of the original system. Moreover they can then be made available for reuse in different contexts.

This chapter introduces the use of software *slicing* techniques and its supporting program representation structures to carry out a component's identification process. The potential of program transformation techniques, like software slicing, for service or component identification is therefore quite obvious. In practice, however, this requires a flexible definition of what is understood by a *slicing criterion*, the ability to extract actual (executable) code fragments, and, of course, suitable tools that make this all possible in practice.

All these issues are addressed in this chapter where, however, our attention is restricted to *functional* programs [Bir98]. Such focus is explained not only by the research context of previous work, but also because we deliberately want to take an alternative path to mainstream research on component identification for which functional programming has been largely neglected. Therefore our research questions include how can slicing techniques be used in practice to accomplish component extraction, and what would be the most suitable criteria for component identification applied to functional monolithic code.

4.1 Component Discovery and Identification

There are basically two ways in which slicing techniques, and the HaSlicer tool, can be used in the process of component identification: either as a supporting procedure for manual component identification or as a basis for an automatic discovery process in which the whole system is searched for possible *loci* of services, and therefore potential components. In this section both approaches are discussed.

4.1.1 User Driven Approaches

The first approach deals with manual component identification guided by a process that iterates between analysing and slicing a suitable representation of the legacy code. In this context, the FDG seems to provide a suitable representation model as it concentrates, in a single representation form, information about the source code entities (and its dependencies) which constitute the basic aggregation units of software components. Through its analysis, the software architect can easily identify all the dependencies between the code entities and look for certain architectural patterns and/or undesired dependencies in the graph.

One of the most interesting operations in this category of manually driven approaches, is component identification by service. The idea is to isolate a component that implements a specific service which is provided by the overall system.

Our approach to isolate such a service in a component, starts by following a top-down approach to look for the top level functions that characterise the desired service. This identification must be performed manually by direct inspection of the source code or by analysing the FDG instance of the system. We regard the latter as a better alternative for this task, because most of the irrelevant code details are abstracted from the FDG, thus making it easier to inspect which system functions are taking care of the foreign service invocations. Even more, by using the FDG visualisation and search capabilities of HaSlicer this task gets even further simplified.

Once the functions implementing the desired services are found, *complete forward dependency slicing* is applied starting from the corresponding FDG nodes. This produces a series of sliced files (one per top level function), that have to be merged together in order to build the desired component. Note that a complete forward slice collects all the program entities that each top level function requires to operate correctly. Thus, by merging all the complete forward slices corresponding to a particular service one gets the least (derived) program that implements such a service. The problematic of merging the computed software slices is treated in section 4.2.

An alternative user driven approach to component identification, resorts to exploring the fact that many software systems are developed around data type definitions. These systems are composed by a series of core data types which are then decorated with functions and operations in order to deliver the overall system behaviour. Although being developed around easily identifiable data types, it is not generally trivial in practice to isolate the parts of the entire system that concern a particular data type. The are several reasons behind the difficulty in disentangling the system parts dealing with each data type, but the most important ones are the introduction of functions and operations dealing with several data types instead of relying on other functions retrieving the needed data type information as well as the evolution of systems by the intervention of different development teams, which may not always respect the original encapsulation design principles.

Another important reason to delve into this category of software systems is that they are among the most frequent systems developed and still being developed today. Just to name a few usual sub-categories and to give a notion of the range of systems one is referring to here, consider for instance Enterprise Resource Planning (ERP), Customer Relationship Management

(CRM), Manufacturing Resource Planning (MRP) and other tightly data dependent software's.

Our approach to perform component identification over such data centric systems is to manually select a particular data type from the set of systems data types and isolate both the selected data type and every program entity in the system that depends on it. Such an operation can be accomplished in a two phase process based on forward and backward slicing.

The first phase consists in performing a *backward slicing* using as slicing criterion the selected data type node from the FDG. This operation retrieves a slice containing the selected data type and every program entity that depends upon it, but as explained in chapter 2, the retrieved slice may not be an executable one, thus not always suitable to be considered as a component. This leads us to next phase, whose objective is to transform the slice obtained in phase one to the *minimal* executable program that contains it.

In the second phase one calculates a list of the complete forward slices with respect to every function contained in the slice obtained in phase one. Finally the process ends by merging the slice from phase one with all the slices from phase two (see section 4.2 for details).

One may wonder whether that this data type driven component discovery can be applied to every software system, since almost every system delivers functionality by using some kind of data type definition. This would indicate that this approach could be transformed into a completely automatised one, by applying it to all root data type definitions[1] or even to every system data type. Note, however that we have tried such an approach to a few software systems and, in practice, this led in most cases to a very low number of useful discovered components. The main reason for this resides in the fact that many of the data types used by software systems do not encapsulate real "functional" entities (*i.e.* meaningful system requirement subjects or relations) but rather serve as auxiliary structured repositories of information to be used internally in a myriad of ways. Thus, when applying such an approach to these auxiliary data types, one obtains a piece of software that, although being perfectly executable, does not deliver a real useful service or

[1] I.e., data type definitions that do not depend upon any other data type definition

set of services.

4.1.2 Automatic Component Discovery

Another possibility for using slicing and its underlying data representation structures, resorts to the application of such techniques to the automatic isolation of possible components. In our experience, the use of automatic approaches was found particularly useful when employed at early stages of component identification and especially for dealing with legacy software systems for which one cannot immediately tell where to start looking for potential components.

Such automatic procedures, however, must be used carefully, since they may lead to the identification of both false positives and false negatives. This means that there might be good candidates for components which are not discovered by such techniques as well as situations in which several possible components are identified which turn out to lack any practical or operational interest. We shall come later in this section to the problems behind this false positive and negative identification of components.

Before undertaking the implementation of an automatic component discovery process, one must first understand what to look for, since there is no universal way of stating which characteristics correspond to a potential software component, nor what is the best way to have a system organised in term of its constituents. In practice this means that one has to look for components by indirect means, that certainly include the identification of a number of characteristics that components usually present, but also some filtering criteria.

A well known method used to characterise “interesting” software components is based on the notion of *coupling* and *cohesion* [FP97, YC79, SvdMK$^+$04].

Coupling is a metric to assess how mutually dependable two components are, *i.e.*, it tries to measure how much a change in one component affects another one in a system. In practice coupling of a system’s software parts can be measured in different ways, depending on how the software parts in question interact among each other. Among the many ways available for this interactions to take place, one may list interactions using shared files in the disk, shared memory spaces (e.g. Linda based systems), direct procedure calls, distributed objects and web-services calls.

Once identified the particular interaction types one is interested in, coupling can be measured by the number of dependencies a software component has with respect to other systems parts outside the component. With HaSlicer, one can compute such a metric based on the diverse kinds of dependencies captured in the FDG. We will come later on this section to the specific details for calculating coupling metrics with the FDG.

It is widely accepted that, in general, a software component should not be dependent on any parts of the system where it is being used. On contrary, it is the "glue" code of the system that should be dependent on the components being used. Since, coupling measures the dependencies of the component towards the rest of system, the minimisation of this metric seems to be useful as a search criterion in the automatic discovery of software components.

On the other hand, *cohesion* measures how internally related are the entities of a specific component. Like in the coupling case, cohesion may be accessed based on several kinds of interactions the component presents, not to the outside entities, but inside himself. Again, HaSlicer seems to be a good alternative to compute such a metric from computed FDGs.

Generally, components with low levels of cohesion are usually difficult to debug and to detect the specific logic behind undesirable behaviour. The reason for this characteristics relies on the fact that other entities of such components, especially functions, are weakly related, thus opening space for errors to "hide" themselves in rarely used areas of the code. This indicates that, in general, one can use the *maximisation* of the cohesion metric as a quality measure in the discovery of components.

The conjunction of these two metrics leads to a discovery criteria which explores the FDG to look for specific clusters of functions, *i.e.*, sets of strongly related functions, with reduced dependencies on any other program entity outside this set. Such function clusters cannot be identified by program slicing techniques, but the FDG is still very useful in determining these clusters. In fact these metrics can be computed on top of the information represented in the FDG. The HASLICER framework, in particular, can be used to compute their combined value through the implementation of the following operator.

$$\begin{aligned}
\mathsf{coupling}(G, F) &\triangleq \sharp\{(x, y) \mid \exists x, y.\ yGx \wedge x \in F \wedge y \notin F\} \quad (4.1)\\
\mathsf{cohesion}(G, F) &\triangleq \sharp\{(x, y) \mid \exists x, y.\ yGx \wedge x \in F \wedge y \in F\} \quad (4.2)\\
\mathsf{ccanalysis}(G, F) &\triangleq (\mathsf{coupling}(G, F),\ \mathsf{cohesion}(G, F)) \quad (4.3)
\end{aligned}$$

Where G is a FDG, $\sharp$ denotes the cardinal of a set and F is the set of functions under scrutiny.

This definition, however, leads us to another problem: what sets of functions F should be considered as potential candidates for further inspection of component quality. A complete evaluation of all functions defined in the system is certainly out of the question, since the amount of cases to be inspected turns the process infeasible from a practical computational point of view. This was, however, implemented as a quality control for the different improvements made to the automatic component discovery.

Experimental result confirmed that one has always to limit the size of the sets of functions under scrutiny, since the closer they get to the size of the set of all functions defined in the system, the better the results achieved for coupling and cohesion. Ultimately, the set of all functions in the system gets, of course, the maximum level of cohesion and the minimum level of coupling, thus making it the "best" candidate for a software component.

Therefore, the maximum size of the sets of functions to be inspected becomes a relevant parameter in this process. It may vary depending on the size and specific details of the system under analysis. By limiting the overall size of components under search one is already reducing, in an exponential way, the amount of time the discovery process takes to point out potential component candidates. However, the algorithm can be improved by resorting to HASKELL lazy evaluation. The following recursive definitions of the previous coupling and ccanalysis operators reflect such improvements.

$$
\begin{aligned}
\mathsf{coupling}(G, \{\}) &\triangleq 0 && (4.4)\\
\mathsf{coupling}(G, \{f\} \cup F) &\triangleq \text{if } \exists x \notin F.\ fGx \text{ then } 1 + \mathsf{coupling}(G, F) && (4.5)\\
&\qquad\qquad\qquad\qquad \text{else } \mathsf{coupling}(G, F)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ccanalysis}(G, F) &\triangleq \ \text{let } x = \mathsf{coupling}(G, F) && (4.6)\\
&\quad\ \text{in if } x < \mathbf{mc} \text{ then } (x,\ \mathsf{cohesion}(G, F))\\
&\qquad\qquad\qquad \text{else } \ \bot
\end{aligned}
$$

Note that now formula ccanalysis returns $\bot$ whenever the coupling value exceeds a user defined level **mc**. Even more, by unfolding the definition of coupling, the HASKELL implementation improves since in every recursive step of the calculation it inspects if the accumulated value exceeds the maximum coupling value, eventually making the process to end.

In practice, the introduction of this upper bound for the coupling value is quite useful since most of the times one is looking for components with very small values of coupling, typically close to 0. There is also space to apply a similar improvement to the computation of the cohesion metric. Nevertheless, this seems to be less effective because it is based on a lower bound for cohesion which implies that the algorithm has to proceed until achieving such lower bound, which, as expected, is most of the times a large value.

Once such function clusters are identified, the process continues by applying *complete forward slicing* on every function in the cluster and merging the resulting code.

The problem of false negatives, *i.e.*, the existence of software components inside a system which are not discovered by this method, is most of the times explained by the fact that such components are larger than the size of the maximum component to be found. Has expected, this can be solved by increasing the minimum size of the components to find.

On the other hand, the problem with false positives, *i.e.*, software parts that are pointed out by this process as software components which turn out not be real software components, is most often related to non considered dependencies between program entities. In fact this is not a problem of the method but of its implementation which was carried out using HaSlicer. The problem is that HaSlicer only captures dependencies that arise from the semantics of the basic programming language (*i.e.*, Haskell) and it does not take into consideration other sources of dependencies like file sharing, memory share, foreign function calls, etc. Thus, such false components may have dependencies to other system parts that were taken into consideration, therefore leading to a miscalculation of the cohesion and coupling metrics.

One way to overcome this problem, is to manually feed the dependencies in the system that HaSlicer cannot compute, before the discovery process takes place. This can easily be accomplished by using the HaSlicer accompanying framework, n particular the functions dedicated to the manipulation of the FDG instance.

Although the main focus of this case-study is to introduce an automatic discovery process, based on coupling and cohesion, note that these metrics can also be of use for assessing the quality of component based systems. In such a scenario, the program analyst is given a system where the components are already identified so that he can compute and assess the values of coupling and cohesion for the identified components. In case the coupling and cohesion values do not lie between the acceptable values for the system in question, then the dependencies responsible for such diversion should be inspected, eventually using HaSlicer again.

4.2 Isolating Software Components

After a software component is identified by one of the approaches introduced above, the corresponding isolation process takes place. For most cases, as depicted in Figure 4.1, this process is divided in two phases, consisting of the extraction of the component code fragments and their merging, respectively. The specific details of the first phase vary according to the technique employed, but it often resorts to using different slicing techniques as explained in the two previous sections.

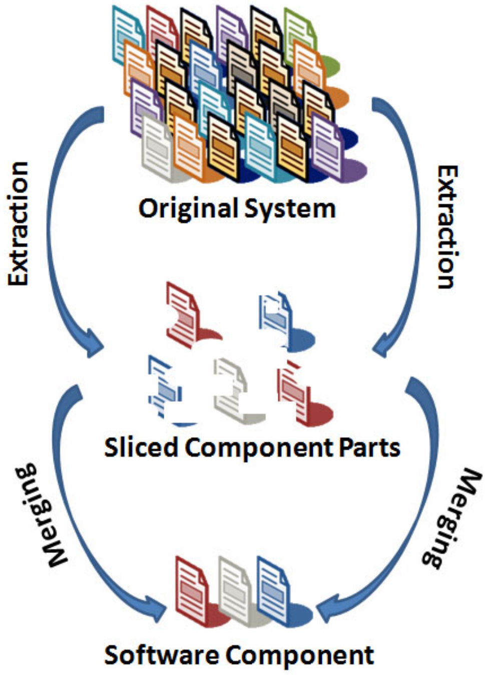


Figure 4.1: Component isolation process

The second phase consists in merging the code fragments identified in phase one, so that one gets a single and unified piece of software, executable and reusable in other contexts. Due to the details of this operation, the merging problem goes far beyond syntactic code cut and paste. In particular, we must take into consideration that the code fragments to be merged are scattered across different files, whose structure and existence must be preserved in order to keep the physical modularity structure of the component. Even more, there are certainly overlapping definitions in each version of the same file which have to be resolved in order to accomplish an executable software component.

The merge of the different slices, that constitute the component to be isolated, can be performed manually by combining the different versions of the same module in each slice. Nevertheless, for cases where there is a great overlapping of programming entities across slices, such a process, although not complex, can become quite time consuming and resort to a try-failure iteration for merging the different modules in a compilable version. Moreover, in cases where the size of the component being isolated is really big, the time to perform each compilation of the manually merged slices can take up to minutes, thus turning this approach uninteresting.

A better choice for merging this set of slices, can be indirectly accomplished by the use of HaSlicer. Actually, although this functionality is not directly available from the tool user interface, one can still use the underlying framework to merge FDG instances. Recall, from the previous chapter, that each slice is obtained by an FDG instance corresponding to the sliced program representation. Therefore, it is easy to use the HaSlicer framework to invoke the complete forward slicing operation over the target system and store the FDG instances in a list, which later can be consumed to compute the desired component.

HaSlicer represents FDG's internally as sets of pairs of nodes. So, slice merging is achieved simply by set union of the corresponding FDGs, without having to worry about overlapping nor duplicate program entities representations, since, by definition, set structures do not allow repetitions of elements. Finally, once computed the FDG associated to the merged slices, one just has to call the FDG program construction function, contained in the HaSlicer framework, to transform it into the code corresponding to the desired isolated software component.

Once accomplished the isolation process, another question arises: in what direction should such system be reorganised to make it use the identified service as a (now) independent component? This would require an operation upon the FDG which is, in a sense, dual to slicing. It consists of extracting every program entity from the system, but for the ones already collected in the computed slices.

Such operation, which is, at present, only partially supported by HaSlicer, produces typically a program which cannot be immediately executed, but may be transformed in that direction. This amounts to identify potential broken function calls in the original code and re-direct them to the new component's services.

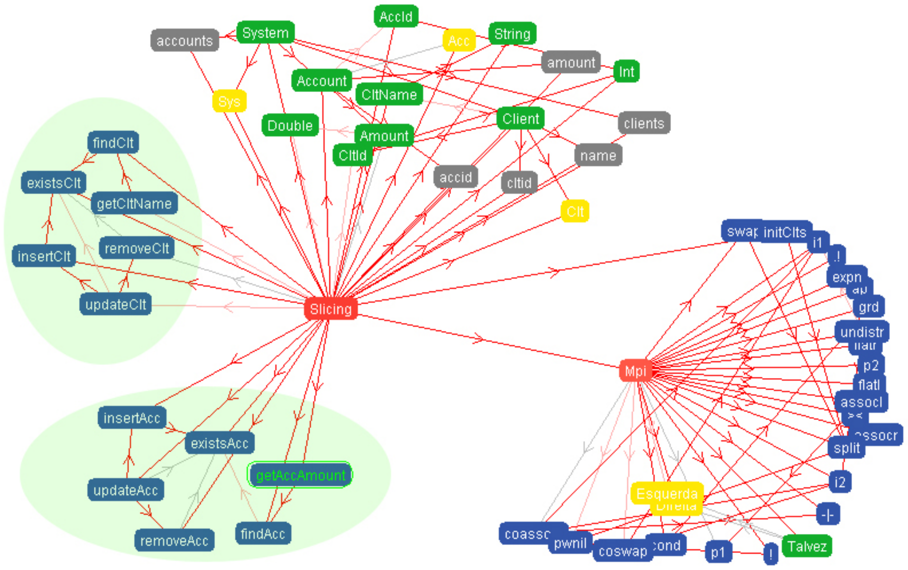


Figure 4.2: FDG for the toy bank account system

4.3 Component Discovery with HaSlicer

To illustrate the component discovery process introduced in this chapter, we present a brief case-study on top of a Haskell implementation toy bank account system. The entire code is given in appendix A. The corresponding FDG, as computed by HaSlicer, is depicted in Figure 4.2[2]. Note that, for a clearer presentation of the function clusters that will constitute the components to be found, dependencies from functions to data-types are omitted.

If one tries to apply an automatic component discovery method to this code, based, for example, in the combined cohesion-coupling metric, the number of cases to consider soon becomes very large. Even more, considering all of the cases in the powerset of the functions set, will lead to the previously mentioned problem: the set containing all functions is certainly the one with better results for coupling and cohesion but it is certainly not the case for identifying a useful component decomposition. Thus, one can use the above mentioned strategy for optimizing the discovery process based on limiting

[2] The reader is invited to try this example at http://alfa.di.uminho.pt/~nfr/PhdThesis/ToyBank.hs

Functions clusters	Cohesion	Coupling
getAccAmount findAcc existsAcc insertAcc updateAcc removeAcc	7	0
getCltName findClt existsClt insertClt updateClt removeClt	7	0

Table 4.1: Cohesion and coupling metric

the size (in terms of program entities contained) of the components as well as on the coupling value. For this particular example, given that the entire program is composed of 13 functions, one has chosen to look for components with less than 10 functions and with a limit of coupling set to 0. The results of applying such a component discovery criteria to the example are listed in Table 4.1.

Clearly, two components have been identified (corresponding to the light green area of the FDG in Figure 4.2): a component for handling *Client* information and another one for managing *Accounts* data. As explained, the process continues by applying complete forward dependency slicing over the nodes corresponding to the functions in the identified sets, followed by slice merging.

One of the advantages of this automatic component discovery method is that it helps significantly in early stages of program comprehension, that usually precede other more sophisticated layers of analysis. In this example, for instance, one could ignore completely what the code is supposed to do or what the entities upon which the program executes are. But, after having performed this automatic analysis, one could clearly identify not only the main entities, *Clients* and *Accounts*, but also the functionality defined around each one. In this sense, we believe that this process can play a major role in (functional) program comprehension by giving a first logical division of the program parts that together implement the entire program behaviour.

Chapter 5

Slicing by Calculation

This chapter is an attempt to reframe slicing of functional programs as a calculational problem in the *algebra of programming* [BM97]. More specifically, to compute program *slices* by solving an equation on the program denotational domain.

The main motivation for investigating this alternative approach to functional program slicing, is that the graph-based method discussed in previous chapters is unable to slice inside what we could be called high level program entities, *i.e.*, functions, data-types, constructors and destructors. So, the computed slices never go inside a function definition, or any other high level program entity, even though they may contain components which are not relevant according to the slicing criterion.

Slicing over high level entities, as done before, has the disadvantage that, for programs containing large functions, the slices tend to be rather large. The main reason for this is that, even in cases where only a single clause[1] of a large function is of relevance to the slice, the process will make the entire function to be showed up in the final slice. Even more, since the slicing process is defined as a recursive fixed point calculation, the unnecessarily collected function clauses will contribute with spurious dependencies that will trigger the inclusion of further unnecessary program entities.

The approach presented in this chapter takes a completely different path

[1] We use the term functional clause to refer to a function definition over a particular kind of input

from the traditional approaches to slicing. Instead of extracting program information to build an underlying dependencies' structure, we resort to standard program calculation strategies, based on the so-called Bird-Meertens formalism. The slicing criterion is specified either as a *projection* or a *hiding* function which, once composed with the original program, leads to the identification of the intended slice.

The process is driven by the denotational semantics of the target program, as opposed to more classical syntax-oriented approaches documented in the literature. To make calculation effective and concise we adopt the *pointfree* style of expression [BM97] popularised among the functional programming community.

This approach seems to be particularly suited to the analysis of *functional* programs. Actually, it offers a way of going inside function definitions and, in some cases, to extract new functions with a restricted input or output. Note that with approaches based on dependencies' graphs, as the ones presented in the previous chapters, one usually works at an "external" level, for example collecting references to an identifier or determining which functions make use of a particular reference. Here, however, we take a completely different path.

5.1 A Glimpse on the Laws of Functions

Composition. In order to maintain this thesis as self contained as possible, this section provides a brief review of the algebra of functions, recalling the basic constructions and laws that will be used throughout this chapter (see [BM97, Bac03]). We begin mentioning some functions which have a particular role in the calculus: namely, *identities* denoted by $\mathsf{id}_A : A \longleftarrow A$ or the so-called *final* functions $!_A : \mathbf{1} \longleftarrow A$ whose codomain is the singleton set denoted by $\mathbf{1}$ and consequently map every element of A into the (unique) element of $\mathbf{1}$. Elements $x \in X$ are represented as *points*, *i.e.*, functions $\underline{x} : X \longleftarrow \mathbf{1}$, and therefore function application $f\,x$ can be expressed by composition $f \cdot \underline{x}$.

Functions can be *glued* in a number of ways which bare a direct correspondence with the ways programs may be assembled together. The most obvious one is *pipelining* which corresponds to standard functional composition denoted by $f \cdot g$ for $f : C \longleftarrow B$ and $g : B \longleftarrow A$. Functions with a common domain can be glued through a *split* $\langle f, g\rangle$, the universal function associated to cartesian product of sets as shown in the following diagram:

$$\begin{tikzcd}
 & Z \arrow[ld, "f"'] \arrow[d, "\langle f,g\rangle"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

Actually, the product of two sets A and B can be characterised either concretely (as the set of all pairs that can be formed by elements of A and B) or in terms of a universal specification. In this case, we say set $A \times B$ is defined as the source of two functions $\pi_1 : A \longleftarrow A \times B$ and $\pi_2 : B \longleftarrow A \times B$, called the *projections*, such that for any other set Z and arrows $f : A \longleftarrow Z$ and $g : B \longleftarrow Z$, there is a unique arrow $\langle f, g\rangle : A \times B \longleftarrow Z$, called the *split* of f and g, that makes the diagram above to commute. This can be expressed in quite concise way through the following equivalence which entails both an *existence* ($\Rightarrow$) and a *uniqueness* ($\Leftarrow$) assertion:

$$k = \langle f, g\rangle \quad \equiv \quad \pi_1 \cdot k = f \ \wedge \ \pi_2 \cdot k = g \tag{5.1}$$

Such an abstract characterisation turns out to be more generic and suitable for conducting calculations than the usual pointwise formulation. Let us illustrate this claim with a very simple example. Suppose we want to show that pairing projections of a cartesian product has no effect, *i.e.*, $\langle \pi_1, \pi_2\rangle = \mathsf{id}$. If we proceed in a concrete way we first attempt to convince ourselves that the unique possible definition for *split* is as a pairing function, *i.e.*, $\langle f, g\rangle\ z = \langle f\ z, g\ z\rangle$. Then, instantiating the definition for the case at hands, conclude

$$\langle \pi_1, \pi_2\rangle\ \langle x, y\rangle = \langle \pi_1\ \langle x, y\rangle, \pi_2\ \langle x, y\rangle\rangle = \langle x, y\rangle$$

Using the universal property (5.1) instead, without any reference to points:

$$\mathsf{id} = \langle \pi_1, \pi_2 \rangle \;\equiv\; \pi_1 \cdot \mathsf{id} = \pi_1 \wedge \pi_2 \cdot \mathsf{id} = \pi_2$$

Equation

$$\langle \pi_1, \pi_2 \rangle \;=\; \mathsf{id}_{A \times B} \tag{5.2}$$

is called the *reflection* law for products. Similarly the following laws (known respectively as × *cancellation*, *fusion* and *absorption*) are derivable from (5.1):

$$\pi_1 \cdot \langle f, g \rangle = f \;,\; \pi_2 \cdot \langle f, g \rangle = g \tag{5.3}$$

$$\langle g, h \rangle \cdot f \;=\; \langle g \cdot f, h \cdot f \rangle \tag{5.4}$$

$$(i \times j) \cdot \langle g, h \rangle \;=\; \langle i \cdot g, j \cdot h \rangle \tag{5.5}$$

The same applies to *structural equality*:

$$\langle f, g \rangle = \langle k, h \rangle \;\equiv\; f = k \wedge g = h \tag{5.6}$$

Finally note that the product construction applies not only to sets but also to functions, yielding, for $f : B \longleftarrow A$ and $g : B' \longleftarrow A'$, function $f \times g : B \times B' \longleftarrow A \times A'$ defined as the split $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$. This corresponds to the following pointwise definition: $f \times g = \lambda \langle a, b \rangle \;.\; \langle f\ a, g\ b \rangle$.

Notation B^A is used to denote *function space*, *i.e.*, the set of (total) functions from A to B. It is also characterised by a universal property: for all function $f : B \longleftarrow A \times C$, there exists a unique $\overline{f} : B^C \longleftarrow A$, called the *curry* of f, such that $f = \mathsf{ev} \cdot (\overline{f} \times C)$. Diagrammatically,

$$\begin{tikzcd}
A \arrow[d, "\overline{f}"'] & & A \times C \arrow[d, "\overline{f} \times \mathsf{id}_C"'] \arrow[dr, "f"] & \\
B^C & & B^C \times C \arrow[r, "\mathsf{ev}"'] & B
\end{tikzcd}$$

i.e.,

$$k = \overline{f} \quad \equiv \quad f = \mathsf{ev} \cdot (k \times \mathsf{id}) \tag{5.7}$$

Dually, functions sharing the same codomain may be glued together through an *either* combinator, expressing alternative behaviours, and introduced as the universal arrow in a datatype sum construction.

The *sum* $A + B$ (or *coproduct*) of A and B corresponds to their disjoint union. The construction is dual to the product one. From a programming point of view it corresponds to the aggregation of two entities in *time* (as in a `union` construction in C), whereas product entails an aggregation in *space* (as a `record`). It also arises by universality: $A + B$ is defined as the target of two arrows $\iota_1 : A + B \longleftarrow A$ and $\iota_2 : A + B \longleftarrow B$, called the *injections*, which satisfy the following universal property: for any other set Z and functions $f : Z \longleftarrow A$ and $g : Z \longleftarrow B$, there is a unique arrow $[f, g] : Z \longleftarrow A + B$, usually called the *either* (or *case*) of f and g, that makes the following diagram to commute:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f,g]}"] & B \arrow[l, "\iota_2"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}$$

Again this universal property can be written as

$$k = [f, g] \quad \equiv \quad k \cdot \iota_1 = f \ \wedge \ k \cdot \iota_2 = g \tag{5.8}$$

from which one infers correspondent *cancellation*, *reflection* and *fusion* results:

$$[f,g] \cdot \iota_1 = f \; , \; [f,g] \cdot \iota_2 = g \tag{5.9}$$

$$[\iota_1, \iota_2] \; = \; \mathsf{id}_{X+Y} \tag{5.10}$$

$$f \cdot [g,h] \; = \; [f \cdot g, f \cdot h] \tag{5.11}$$

Products and sums interact through the following *exchange* law

$$[\langle f,g\rangle, \langle f',g'\rangle] \; = \; \langle [f,f'], [g,g'] \rangle \tag{5.12}$$

provable by either product (5.1) or sum (5.8) universality. The *sum* combinator also applies to functions yielding $f+g : A'+B' \longleftarrow A+B$ defined as $[\iota_1 \cdot f, \iota_2 \cdot g]$.

Conditional expressions are modelled by coproducts. In this chapter we adopt the McCarthy conditional constructor written as $(p \rightarrow f,\, g)$, where $p : \mathbb{B} \longleftarrow A$ is a predicate. Intuitively, $(p \rightarrow f,\, g)$ reduces to f if p evaluates to true and to g otherwise. The conditional construct is defined as

$$(p \rightarrow f,\, g) \; = \; [f,g] \cdot p?$$

where $p? : A+A \longleftarrow A$ is determined by predicate p as follows

$$p? = \qquad A \xrightarrow{\langle \mathsf{id}, p\rangle} A \times (\mathbf{1}+\mathbf{1}) \xrightarrow{\mathsf{dl}} A \times \mathbf{1} + A \times \mathbf{1} \xrightarrow{\pi_1 + \pi_1} A + A$$

where dl is the distributivity isomorphism. The following laws are useful to calculate with conditionals [Gib97].

$$h \cdot (p \rightarrow f,\, g) \; = \; (p \rightarrow h \cdot f,\, h \cdot g) \tag{5.13}$$

$$(p \rightarrow f,\, g) \cdot h \; = \; (p \cdot h \rightarrow f \cdot h,\, g \cdot h) \tag{5.14}$$

$$(p \rightarrow f,\, g) \;=\; (p \rightarrow (p \rightarrow f,\, g),\, (p \rightarrow f,\, g)) \tag{5.15}$$

Recursion. Recursive functions over inductive datatypes (such as finite sequences or binary trees) are induced by their *genetic* information, *i.e.*, the specification of what is to be done in an instance of a recursive call. Consider, for example, the pointfree specification of the function which computes the length of a list $\mathsf{len} : \mathbb{N} \longleftarrow A^*$. A^* is an example of an inductive type: its elements are built by one of the following *constructors*: $\mathsf{nil} : A^* \longleftarrow \mathbf{1}$, which builds the empty list, and $\mathsf{cons} : A^* \longleftarrow A \times A^*$, which appends an element to the head of the list. The two constructors are glued by an *either* $\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]$ whose codomain is an instance of polynomial functor $\mathsf{F}\,X = \mathbf{1} + A \times X$. The algorithmic contents of function len are exposed in the following diagram:

$$\begin{tikzcd}
\mathbf{1} + A \times \mathbb{N} \arrow[r, "{[\underline{0}, \mathsf{succ} \cdot \pi_2]}"] & \mathbb{N} \\
\mathbf{1} + A \times A^* \arrow[u, "\mathsf{id} + \mathsf{id} \times \mathsf{len}"] \arrow[r, "{\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]}"] & A^* \arrow[u, "\mathsf{len}"']
\end{tikzcd}$$

where the "genetic" information is given by $[\underline{0}, \mathsf{succ} \cdot \pi_2]$: either return 0 or the successor of the value computed so far. Function len, being entirely determined by its "gene" is said its *inductive extension* or *catamorphism* and represented by $(\![[\underline{0}, \mathsf{succ} \cdot \pi_2]]\!)$.

Catamorphisms extend to any polynomial functor F and possess a number of remarkable properties, *e.g.*,

$$(\![in]\!) \;=\; \mathsf{id} \tag{5.16}$$

$$(\![g]\!) \cdot \mathsf{in} \;=\; g \cdot \mathsf{F}\,(\![g]\!) \tag{5.17}$$

$$f \cdot (\![g]\!) \;=\; (\![h]\!) \;\Leftarrow\; f \cdot g \;=\; h \cdot \mathsf{F}\,f \tag{5.18}$$

$$(\![g]\!) \cdot \mathsf{T}\,f \;=\; (\![g \cdot \mathsf{F}\,(f, \mathsf{id})]\!) \tag{5.19}$$

where T is the functor that assigns to a set X the corresponding inductive type for F (in the example above, $\mathsf{T}\,X = X^*$). Laws above are called, respectively, cata-reflection, -cancellation, -fusion and -absorption.

5.2 Slicing Equations

Algebra of Programming. In his Turing Award lecture J. Backus [Bac78] was among the first to advocate the need for programming languages which exhibit an *algebra* for reasoning about their own objects, leading to the development of program calculi directly based on, actually driven by, type specifications. Since then this line of research has witnessed significant advances based on the *functorial* approach to data types [MA86] and reached the status of a program calculus in [BM97], building on top of a discipline of algorithm derivation and transformation which can be traced back to the so-called *Bird-Meertens formalism* [Bir87, Mal90, MFP91] and the foundational work of T. Hagino [Hag87] on induction and coinduction.

In this chapter we intend to build on this collection of *programming laws* to solve what we shall call *slicing equations.* Pointwise notation, as used in classical mathematics, involving operators as well as variable symbols, logical connectives, quantifiers, etc, is however inadequate to reason about programs in a concise and precise way. This justifies the introduction of a *pointfree* program denotation in which elements and function application are systematically replaced by functions and functional composition. The translation of the target program into an equivalent pointfree formulation is well studied in the program calculi community and shown to be made automatic to a large extent. In [Oli01a, VO01] its role is compared to one played by the Laplace transform to solve differential equations in a linear space. The remaining of this section provides a quick introduction to the pointfree algebra of functional programs.

5.2.1 Slicing Equations

Our starting point is a very simple idea: to identify the "component" of a function $\Phi : A \longleftarrow B$ affected by a particular argument or contributing to a particular result all one has to do is to *pre-* or *post-*compose Φ with an appropriate function, respectively. In the first case the contribution of an argument is propagated through the body of Φ, forgetting about the role of other possible arguments: σ is a called a *hiding* function and equation

$$\Phi \cdot \sigma \;=\; \Phi' \tag{5.20}$$

captures the *forward* slicing operation. In such a scenario, one regards Φ' as the forward slice of Φ with respect to *slicing criterion* σ.

The dual problem corresponds to *backward* slicing: an output, selected through some sort of projection π, is traced back through the body of Φ. The equation combining the projection function and the function under analysis would then be

$$\pi \cdot \Phi \;=\; \Phi' \tag{5.21}$$

But, how far can this simple idea be pushed in order to actually compute functional slices? The simplest case arises whenever Φ is canonical, *i.e.*, defined as an *either* or a *split*. In the first case one gets $\Phi = [f, g] : A \longleftarrow B_1 + B_2$. The slicing criterion is simply an embedding, *e.g.*, $\iota_1 : B_1 + B_2 \longleftarrow B_1$ and the forward slice becomes

$$[f, g] \cdot \iota_1 \;=\; f \tag{5.22}$$

Dually, for $\langle f, g\rangle : A_1 \times A_2 \longleftarrow B$, one may compute a *backward* slice, by post-composition with a projection, *e.g.*, $p1 : A_1 \longleftarrow A_1 \times A_2$ and conclude

$$\pi_1 \cdot \langle f, g\rangle \;=\; f \tag{5.23}$$

The dual cases of computing a forward slice of a function with a *multiplicative* domain or a backward slice of a function with a *additive* codomain, amounts to composing Φ with the *relational converses* of a projection or an embedding, respectively, leading to the following relational composition $\Phi \cdot \pi_1^\circ$ or $\iota_1^\circ \cdot \Phi$.

From a formal point of view this entails the need to pursue calculation in the *relational* calculus [BH93]. For the language engineer, however, this means that, in the general case, there is no unique solution to the slicing problem: one may end with a set of possible slices, corresponding to different views over the "theoretical", relational, non executable, slice.

We will not explore this relational counterpart in this thesis. Instead our aim is to discuss how far one can go keeping within the functional paradigm in analysing slicing of a particularly important class of functions: the *inductive* ones. I.e., functions whose domain is the carrier of an initial algebra for a regular functor, usually called an *inductive type*. In this thesis, however, we will further restrict ourselves to structural recursive functions, *i.e.*, specified by *catamorphisms* [BM97], *i.e.*, $\Phi = (\!|f|\!)_{\mathsf{T}} : A \longleftarrow \mu_{\mathsf{T}}$, where μ_{T} is the inductive type for functor T and $f : A \longleftarrow \mathsf{T}A$ is the recursion *gene* algebra. Such will be our case-study through the following section.

5.3 Slicing Inductive Functions

This section is organised around four different slicing cases whose target is always an inductive function $\Phi : A \longleftarrow \mu_{\mathsf{T}}$. Each subsection discusses one of these cases: *product backward*, *sum forward*, *sum backward* and *product forward* slicing. In order to facilitate the understating of the application of each slicing operation to real practical cases, we provide a simple example for each one.

5.3.1 Product Backward Slicing

Product backward slicing fits in what we call a "well-behaved" case, *i.e.*, the codomain of Φ is a product and, therefore, the slicing criterion is just an appropriate projection function. Even more, as Φ is recursive, the solution to the slicing problem should be a new *gene* algebra f' such that $\pi_k \cdot \Phi = (\!|f'|\!)$, as explained in the following diagram:

$$\begin{tikzcd}
A_k & \prod_i A_i \arrow[l, "\pi_k"'] & \mu\mathsf{T} \arrow[l, "\Phi"'] \arrow[ll, bend right, "\Phi' = (\![f']\!)"'] \\
\mathsf{T}A_k \arrow[u, "f'"] & \mathsf{T}\prod_i A_i \arrow[l, "\mathsf{T}\pi_k"'] \arrow[u, "f"] & \mathsf{T}\mu\mathsf{T} \arrow[l, "\mathsf{T}\Phi"'] \arrow[u, "\mathsf{in}_\mathsf{T}"']
\end{tikzcd}$$

Solving the slicing equation $\Phi' = \pi_k \cdot \Phi$ reduces, by the fusion law for catamorphisms, to verifying the commutativity of the leftmost square. This becomes quite clear through an example.

Example. Consider the problem of identifying a *slice* in the following functional version of the Unix word-count utility (`wc`), with the `-lc` flag, which calculates both the number of lines and characters of a given file. We assume that the file contents are passed to our function as a list of `Char` values.

```
wc :: [Char] -> (Int, Int)
wc [] = (1, 0)
wc (h:t) = let (lc, cc) = wc t
           in  if h == '\n' then (lc+1, cc+1)
                            else (lc, cc+1)
```

This definition can easily be translated into the following catamorphism

$$(\![[\langle \underline{1}, \underline{0} \rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?]]\!)_\mathsf{F}$$

where $p = (('\backslash n' ==) \cdot \pi_1)$ and $\mathsf{F}X = \mathbf{1} + \mathtt{String} \times X$ is the relevant functor.

Our goal is to identify a slice of `wc` which isolates the parts of the program involved in the computation of the number of lines, and thus eliminating everything else. The number of lines is given by the first component of the pair returned by the original `wc` program. Thus, it is expectable that a function which selects the first element of a pair constitutes a good candidate for a slicing criterion. Indeed we shall use π_1 as the slicing criterion function which reduces the slicing problem to solving the following equation:

$$\llparenthesis f' \rrparenthesis_{\mathsf{F}} \;=\; \pi_1 \cdot \llparenthesis [\langle \underline{1}, \underline{0} \rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?] \rrparenthesis_{\mathsf{F}}$$

This is solved within the functional calculus, as follows

$$\begin{array}{ll}
 & \llparenthesis f' \rrparenthesis_{\mathsf{F}} \;=\; \pi_1 \cdot \llparenthesis [\langle \underline{1}, \underline{0} \rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?] \rrparenthesis_{\mathsf{F}} \\
\Leftarrow & \qquad \{\text{cata-fusion}\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; \pi_1 \cdot [\langle \underline{1}, \underline{0} \rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?] \\
\Leftrightarrow & \qquad \{\text{absorption-}+\text{, cancelation-}\times\text{, natural-}id\text{, definition of }\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot p?] \\
\Leftrightarrow & \qquad \{\text{definition of }\times\text{, cancelation-}\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2 \cdot (id \times \pi_1), \pi_2 \cdot (id \times \pi_1)] \cdot p?] \\
\Leftrightarrow & \qquad \{\text{absorption-}+\text{, } p = p \cdot (\mathsf{id} \times \pi_1)\text{, definiton of }\times\text{, cancelation.}\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot (id \times \pi_1 + id \times \pi_1) \cdot (p \cdot (id \times \pi_1))?] \\
\Leftrightarrow & \qquad \{\text{predicate fusion}\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p? \cdot (id \times \pi_1)] \\
\Leftrightarrow & \qquad \{\text{natural-}id\text{, absortion-}+\text{, }\mathsf{F}\text{ definition}\} \\
 & f' \cdot (id + id \times \pi_1) \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] \cdot (id + id \times \pi_1) \\
\Leftrightarrow & \qquad \{id + id \times \pi_1 \text{ is surjective}\} \\
 & f' \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?]
\end{array}$$

Such a calculation leads to the identification of the *gene* algebra for the intended slice. The slice, on its turn, can easily be translated back to HASKELL as follows

```
wc = foldr (\c -> if c == '\n' then succ else id) 1
```

or, going pointwise,

```
wc' :: [Char] -> Int
wc' [] = 1
```

```
wc' (h:t) = let lc = wc' t
            in if h == '\n' then lc+1
                             else lc
```

Note that, a similar approach, using π_2 as a slicing criterion, allows to isolate the character count computation inside `wc`.

5.3.2 Sum Forward Slicing

This is another "well-behaved" case, because where the slicing criterion reduces to an embedding. The slicing problem, however, has to be rephrased so that the domain of Φ becomes a sum. This is shown in the following diagram where the slicing criterion is $\sigma = \mathsf{in}_\mathsf{T} \cdot \iota_k$, *i.e.*, the relevant embedding composed with the initial algebra in_T (which is an isomorphism).

$$\begin{tikzcd}
A & \mu\mathsf{T} \arrow[l, "\Phi = (\![f]\!)"'] & \mathsf{T}\mu\mathsf{T} = \sum_i U_i \arrow[l, "\mathsf{in}_\mathsf{T}"'] & U_k \arrow[l, "\iota_k"'] \arrow[lll, bend right, "\Phi'"'] \arrow[ll, bend right, "\sigma = \mathsf{in}_\mathsf{T} \cdot \iota_k"']
\end{tikzcd}$$

The computation of Φ' proceeds by the cancellation law for catamorphisms, as illustrated in the following example.

Example. To illustrate a sum forward slicing calculation, consider a pretty printer for a subset of the XML language. We start with a data type encoding XML expressions:

```
data XML = SimpElem String [XML]
         | Elem String [(Att, AttValue)] [XML]
         | Text String
type Att = String
type AttValue = String
```

from which functor $\mathsf{F}\,X = S \times X^* + S \times AS \times X^* + S$ is inferred, where `String` and `[(Att, AttValue)]` are abbreviated to S and AS, respectively. Then consider the pretty printer program:

```
pXML (SimpElem e xmls)  = "<" ++ e ++ ">" ++ nl ++
                          (concat . map pXML $ xmls) ++
                          "</" ++ e ++ ">" ++ nl
pXML (Elem e atts xmls) = "<" ++ e ++ concat (map pAtts atts)
                          ++ ">" ++ nl ++
                          (concat . map pXML $ xmls) ++
                          "</" ++ e ++ ">" ++ nl
pXML (Text t)           = t ++ nl
pAtts (att, attvalue)   = " " ++ att ++ "=\"" ++
                          attvalue ++ "\""
nl                      = "\n"
```

whose pointfree definition reads

$$\begin{array}{rcl}
pXML & = & (\![[[pSElem, pElem], id \star \underline{nl}]]\!)_{\mathsf{F}} \\
pSElem & = & ob \star \pi_1 \star cb \star nl \star concat \cdot \pi_2 \star oeb \star \pi_1 \star cb \star nl \\
pElem & = & ob \star \pi_1 \cdot \pi_1 \star concat \cdot map\ pAtts \cdot \pi_2 \cdot \pi_1 \star cb \star nl \star \\
 & & concat \cdot \pi_2 \star oeb \star p1 \cdot \pi_1 \star cb \star nl \\
pAtts & = & \underline{"\ "} \star \pi_1 \star \underline{"=\backslash""} \star \pi_2 \star \underline{"\backslash""}
\end{array}$$

where

$$\begin{array}{rcl}
nl & = & \underline{"\backslash n"} \\
ob & = & \underline{"<"} \\
cb & = & \underline{">"} \\
oeb & = & \underline{"</"} \\
f \star g & = & \overline{++} \cdot \langle f, g \rangle \\
\overline{++}\ (x, y) & = & (++)\ x\ y
\end{array}$$

Note that operator $\star$ is a right associative operator and $\overline{++}$ denotes the uncurried version of the $(++)$ Haskell function for list concatenation.

The above pointfree definition may seem complex, but it becomes clear when the corresponding diagram is depicted:

$$\begin{tikzcd}
XML \arrow[r, "\mathsf{out}_{\mathsf{F}}"] \arrow[d, "pXML = (\!| f |\!)_{\mathsf{F}}"'] & (S \times XML^* + (S \times AS) \times XML^*) + S \arrow[d, "(id \times (\!| f |\!)_{\mathsf{F}}^* + (id \times id) \times (\!| f |\!)_{\mathsf{F}}^*) + id"] \\
A & (S \times A^* + (S \times AS) \times A^*) + S \arrow[l, "f = [[pSElem, pElem], id \,\star\, \underline{nl}]"]
\end{tikzcd}$$

Now let us suppose one wants to compute a slice with respect to constructor `SimpElem` of the `XML` data type. This amounts to isolate the parts of the pretty printer that deal with `SimpElem` constructed values.

To begin with, one has to define a slicing criterion to isolate arguments of the desired type. This is, of course, given by $\iota_1 \cdot \iota_1$ composed with the initial algebra of the underlying functor, *i.e.*, $\sigma = \mathsf{in}_{\mathsf{F}} \cdot \iota_1 \cdot \iota_1$. The calculation proceeds by cancellation in order to identify the impact of σ over $pXML$.

$$\begin{array}{ll}
 & pXML \cdot \sigma \\
\Leftrightarrow & \quad \{\text{definition of } pXML \text{, definition of } \sigma\} \\
 & (\!|[[pSElem, pElem], id \,\star\, nl]|\!)_{\mathsf{F}} \cdot in_{\mathsf{F}} \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{cata-cancelation}\} \\
 & [[pSElem, pElem], id \,\star\, nl] \cdot \mathsf{F}_{pXML} \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{definiton of } \mathsf{F}\} \\
 & [[pSElem, pElem], id \,\star\, nl] \cdot \\
 & ((id \times pXML^* + (id \times id) \times pXML^*) + id) \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{definition of } +\text{, cancelation-}+\} \\
 & [[pSElem, pElem], id \,\star\, nl] \cdot (\iota_1 \cdot (\iota_1 \cdot (id \times pXML^*)) \\
\Leftrightarrow & \quad \{\text{cancelation-}+ \text{ (twice)}\} \\
 & pSElem \cdot (id \times pXML^*) \\
\Leftrightarrow & \quad \{\text{definition of } pSElem\text{, result (5.24), constant function }\} \\
 & ob \,\star\, \pi_1 \cdot (id \times pXML^*) \,\star\, cb \,\star\, nl \,\star\, concat \cdot \pi_2 \cdot (id \times pXML^*) \,\star \\
 & oeb \,\star\, \pi_1 \cdot (id \times pXML^*) \,\star\, cb \,\star\, nl \\
\Leftrightarrow & \quad \{\text{definition of } \times\text{, cancelation-}\times\} \\
 & ob \,\star\, \pi_1 \,\star\, cb \,\star\, nl \,\star\, concat \cdot pXML^* \cdot \pi_2 \,\star\, oeb \,\star\, \pi_1 \,\star\, cb \,\star\, nl
\end{array}$$

The calculation above makes use of the following equality

$$(f \ \star\ g) \cdot h \ = \ f \cdot h \ \star\ g \cdot h \tag{5.24}$$

which is proved as follows:

$$\begin{array}{ll}
 & (f \ \star\ g) \cdot h \\
\Leftrightarrow & \qquad \{\text{definition of } \star\} \\
 & \overline{+\!+} \cdot \langle f, g \rangle \cdot h \\
\Leftrightarrow & \qquad \{\text{fusion-}\times\} \\
 & \overline{+\!+} \cdot \langle f \cdot h, g \cdot h \rangle \\
\Leftrightarrow & \qquad \{\text{definition of } \star\} \\
 & f \cdot h \ \star\ g \cdot h
\end{array}$$

The computed slice is a specialised version of function `pXML`, to deal with values built with `SimpleElem`. Such function can now be directly translated to Haskell, yielding the following program

```
pXML' (SimpElem e xmls)  = "<" ++ e ++ ">" ++ nl ++
                             (concat . map pXML \$ xmls) ++
                             "</" ++ e ++ ">" ++ nl
```

5.3.3 Sum Backward Slicing

The sum backward case is similar to the product backward case in the sense that both retrieve backward slices. This time, however, the co-domain of the original function $\Phi : \sum_i A_i \longleftarrow \mu_{\mathsf{T}}$ is a sum: therefore, each slice will be a function which produces values over a specific summand of the output type. This complicates the picture: we simply cannot *project* such a value from the output of Φ.

Let us take a different approach: if projecting is impossible, we may still *hide*, *i.e.*, use the universal $! : 1 \longleftarrow A_k$ to reduce to $\mathbf{1}$ the output components one wants to get rid of. Hiding functions are constructed by combining $+$, $\times$ and identities with $!$. Note that in this formulation the slicing criterion becomes *negative* — it specifies what is to be discarded. As we are dealing with inductive functions, the problem is again to find the *gene* for the slice, as documented in the following diagram.

$$\begin{tikzcd}
\sum_{i<k} A_i + 1_k + \sum_{i>k} A_i & & \sum_i A_i \arrow[ll, "\sigma = \sum_{i<k} \mathsf{id} + !_k + \sum_{i>k} \mathsf{id}"'] & \mu_{\mathsf{T}} \arrow[l, "\Phi = (\![f]\!)"'] \arrow[lll, bend right, "\Phi' = (\![f']\!)"'] \\
\mathsf{T}(\sum_{i<k} A_i + 1_k + \sum_{i>k} A_i) \arrow[u, "f'"] & & \mathsf{T} \sum_i A_i \arrow[ll, "\mathsf{T}\sigma"'] \arrow[u, "f"] & \mathsf{T}\mu_{\mathsf{T}} \arrow[l, "\mathsf{T}\Phi"'] \arrow[u, "\mathsf{in}_{\mathsf{T}}"']
\end{tikzcd}$$

This sort of slicing is particularly useful when the codomain of original Φ is itself an inductive type, say for a functor G. In such a case one has to compose Φ with the converse of the G-initial algebra in order to obtain an explicit sum in the codomain, *i.e.*

$$\sigma \;=\; (\sum_{i<k} \mathsf{id} + !_k + \sum_{i>k} \mathsf{id}) \cdot \mathsf{out}_{\mathsf{G}}$$

Such is the case discussed in the following example.

Example. Consider a program which generates the DOM tree of the (simplified) XML language introduced in the previous example. Let F be the corresponding polynomial functor. Note that DOM trees themselves are values of an inductive type for a functor $\mathsf{G}\,X \;=\; N + N \times X^*$, as one may extract from the following HASKELL declaration:

```
data DT a = Leaf NType a
          | Node NType a [DT a]
data NType = NText | NElem | NAtt
```

with N abbreviating `Ntype` $\times$ a. Suppose the program to be sliced is `dtree` : $\mu_{\mathsf{G}} \longleftarrow \mu_{\mathsf{F}}$, which is written, in pointfree, style as follows:

```
dTree = cata g
g = either (either g1 g2) (Leaf NText)
```

```
g1 = uncurry (Node NElem)
g2 = uncurry (Node NElem) . split (p1 . p1)
                                  (g3 . p2 . p1 <++> p2)
g3 = map (Leaf NAtt . uncurry (++) . (id >< ("="++)))
```

where `><` is the Haskell implementation of the × point-free operator, and `<++>` the implementation of the $\star$ operator.

Our aim is to calculate a slice with respect to values of type $Node$, *i.e.*, to isolate the program components which interfere with the production of values of this type. To do so, the slicing criterion must preserve the right hand side of data type `DT` and slice away everything else (in this case just the left hand side). Thus, we end up with the slicing function $\sigma = (! + id) \cdot \mathsf{out}_{\mathsf{G}}$. The situation is illustrated as follows:

$$\begin{tikzcd}
1 + N \times DT^* & N + N \times DT^* \arrow[l, "! + id"'] & DT \arrow[l, "\mathsf{out}_{\mathsf{G}}"'] & \mu_{\mathsf{F}} \arrow[l, "(\!|f|\!)_{\mathsf{F}} = \texttt{dTree}"'] \\
\mathsf{F}(1 + N \times DT^*) \arrow[u, "{[[g_1', g_2'], g_3']}"] & \mathsf{F}(N + N \times DT^*) \arrow[l, "\mathsf{F}(! + id)"] \arrow[u, "{[[g_1, g_2], g_3]}"] & \mathsf{F}(DT^*) \arrow[l, "\mathsf{Fout}_{\mathsf{G}}"] \arrow[u, "f"] & \mathsf{F}\mu_{\mathsf{F}} \arrow[l, "\mathsf{F}\texttt{dTree}"] \arrow[u, "\mathsf{in}_{\mathsf{F}}"']
\end{tikzcd}$$

The process proceeds by calculating the new genes g_1', g_2' and g_3' which define the desired slice.

$$\begin{array}{cl}
& [[g_1', g_2'], g_3'] \cdot (id \times (! + id) + id \times (! \times id) + id) = (! + id) \cdot [[g_1, g_2], g_3] \\
\Leftrightarrow & \qquad \{\text{absorption-+, fusion-+}\} \\
& [[g_1' \cdot (id \times (! + id)), g_2' \cdot (id \times (! \times id))], g_3' \cdot id] = \\
& [[(! + id) \cdot g_1, (! + id) \cdot g_2], (! + id) \cdot g_3]
\end{array}$$

Let us concentrate in the first component of this equality (the remaining cases follow obviously a similar pattern). Thus, our goal is to find g_1' such that

$$g_1' \cdot (id \times (! + id)) = (! + id) \cdot g_1 \tag{5.25}$$

Note, however, that using the right distributivity isomorphism, g_1 can be further decomposed as follows

$$\begin{tikzcd}
S \times (N + N \times DT^*) \arrow[r, "distr"] \arrow[d, "g_1"] & S \times N + S \times (N \times DT^*) \arrow[dl, "{[h_1,h_2]}"] \\
N + N \times DT^* &
\end{tikzcd}$$

and similarly for $g'_1 = [h_3, h_4] \cdot distr$, one gets the following diagram

$$\begin{tikzcd}
S \times (1 + N \times DT^*) \arrow[r, "distr"] \arrow[d, "g'_1"] & S \times 1 + S \times (N \times DT^*) \arrow[dl, "{[h_3,h_4]}"] \\
1 + N \times DT^* &
\end{tikzcd}$$

Then, a substitution in (5.25) yields

$$\begin{aligned}
& [h_3, h_4] \cdot distr \cdot (id \times (! + id)) = (! + id) \cdot [h1, h2] \cdot distr \\
\Leftrightarrow \quad & \qquad \{\text{definition of } distr\text{, fusion-+}\} \\
& [h_3, h_4] \cdot (id \times ! + id \times id)) \cdot distr = [(! + id) \cdot h1, (! + id) \cdot h2] \cdot distr \\
\Leftrightarrow \quad & \qquad \{\text{absorption-+}\} \\
& [h_3 \cdot (id \times !), h_4 \cdot (id \times id)] \cdot distr = [(! + id) \cdot h1, (! + id) \cdot h2] \cdot distr
\end{aligned}$$

Hence

$$h_3 \cdot (id \times !) = (! + id) \cdot h1 \qquad \text{and} \qquad h_4 \cdot (id \times id) = (! + id) \cdot h2$$

Let us focus again the first equality (the other case is similar), that is, diagram

$$\begin{tikzcd}
S \times 1 \arrow[r, "h_3"] & 1 + N \times DT^* \\
S \times N \arrow[u, "id \times !"] \arrow[r, "h_1"] & N + N \times DT^* \arrow[u, "! + id"']
\end{tikzcd}$$

In the most general case, functions to a sum type are conditionals. Therefore, we may assume that $h_3 = p \rightarrow \iota_1 \cdot e_1, \iota_2 \cdot e_2$ and $h_1 = q \rightarrow \iota_1 \cdot d_1, \iota_2 \cdot d_2$, respectively. Then,

$$\begin{array}{ll}
 & (p \rightarrow \iota_1 \cdot e_1, \iota_2 \cdot e_2) \cdot (id \times !) = (! + id) \cdot (q \rightarrow \iota_1 \cdot d_1, \iota_2 \cdot d_2) \\
\Leftrightarrow & \qquad \{\text{conditional fusion}\} \\
 & p \rightarrow \iota_1 \cdot e_1 \cdot (id \times !), \iota_2 \cdot e_2 \cdot (id \times !) = q \rightarrow (! + id) \cdot \iota_1 \cdot d_1, (! + id) \cdot \iota_2 \cdot d_2 \\
\Leftrightarrow & \qquad \{\text{cancelation-+, natural } id\} \\
 & p \rightarrow \iota_1 \cdot e_1 \cdot (id \times !), \iota_2 \cdot e_2 \cdot (id \times !) = q \rightarrow \iota_1 \cdot !, \iota_2 \cdot d_2
\end{array}$$

which amounts to

$$\begin{array}{rcl}
p \cdot (id \times !) & = & q \\
e_1 \cdot (id \times !) & = & ! \\
e_2 \cdot (id \times !) & = & d_2
\end{array}$$

What can be concluded from here? That $p : \mathbb{B} \longleftarrow S$ is derived from $q : \mathbb{B} \longleftarrow S \times N$ as follows

$$p(s) = \mathsf{false} \equiv \bigvee_n q(s, n) = \mathsf{false}$$

Finally $e_2 : N \times DT^* \longleftarrow S$ comes from $d_2 : N \times DT^* \longleftarrow S \times N$. But what is the relation between them? Actually, abstracting from the second argument of d_2 gives rise to a powerset valued function

$$\begin{array}{rcl}
\gamma & : & S \rightarrow \mathbb{P}(N \times DT^*) \\
\gamma(s) & = & \{d_2(n, s) \mid n \in N \wedge p(n, s)\}
\end{array}$$

Therefore e_2 is just a possible *implementation* of γ. This means that the slice is *not* unique: we are back to the relational world. It should be stressed, however, that the advantage of this calculation process is to lead the program analyst as close as possible of the critical details. Or, putting it in a different way, directs the slice construction until human interaction becomes necessary to make a choice.

5.3.4 Product Forward Slicing

At first sight this is an awkward case as far as inductive functions are concerned. One may resort to $\mathsf{out_T}$ to unfold the inductive type, as we did in the sum forward case, but this leads always to a polynomial functor with sums as the main connective. So what do we mean by product forward slicing? Suppose the relevant functor is, say, $\mathsf{F}X = \mathbf{1} + A \times B \times X + B \times X^2$. Our aim is to compute a slice of $\Phi : A \longleftarrow \mu_\mathsf{F}$ corresponding to discarding the contribution of the B component.

Our first guess is to adopt the strategy of the previous case and define the slicing criterion as a *hiding* function:

$$\mathsf{in_F} \cdot (\mathsf{id} + \mathsf{id} \times ! \times \mathsf{id} + ! \times \mathsf{id}) : \mu_\mathsf{F} \longleftarrow \mathsf{F}\mu_\mathsf{F}$$

However, this is a *wrong* approach to the problem, because the hiding function changes the signature functor. The expression above would become correct if formulated in terms of functor $\mathsf{F}'X = \mathbf{1} + A \times \mathbf{1} \times X + \mathbf{1} \times X^2$. In such a case, expression $\mathsf{id} + \mathsf{id} \times ! \times \mathsf{id} + ! \times \mathsf{id}$ becomes a natural transformation from F to F'. However, during the calculational process the relational converse of this natural transformation would be required and making progress would depend, to a great extent, on the concrete definition of Φ.

Therefore, let us try a different solution: instead of getting rid of component B, by composition with !, we replace each concrete values by a mark still belonging to B. For that we resort, for the first time in this calculational approach, to the classical semantics of HASKELL in terms of pointed complete partial orders. The qualificative *pointed* means there exists for each type X a bottom element $\bot_X$ which can be used for our purposes as illustrated in the following diagram:

$$\begin{tikzcd}
 & & & \mu_{\mathsf{T}} \arrow[d, "\mathsf{out}_{\mathsf{T}}"] \\
A & \mu_{\mathsf{T}} \arrow[l, "\Phi"] & \sum_i \prod_j U_{i,j} \arrow[l, "\mathsf{in}_{\mathsf{T}}"] & \mathsf{T}\mu_{\mathsf{T}} = \sum_i \prod_j U_{i,j} \arrow[l, "\sigma=\sum_i(\prod_{j<k} \mathsf{id}\times\underline{\bot_k}\times\prod_{j>k} \mathsf{id})"] \arrow[ll, bend right, "\sigma"']
\end{tikzcd}$$

Care should be taken when calculating functional programs in an order-theoretical setting. In particular, as embeddings fail to preserve bottoms, the sum construction is no longer a coproduct and the either is not unique. The set-theoretical harmony, however, can be (almost) recovered if one restricts to *strict* functions (details can be checked in, *e.g.*, [MFP91]). Such is the case of the example below, whose derivation is, therefore, valid.

Example. Let us return to the pretty printer example. Suppose we want to slice away every recursive call in this function. Such a slice could be particularly interesting, for instance in the understanding of what is happening in each recursive iteration.

The calculation of this slice can be achieved by the following slicing criteria $\sigma = in_{\mathsf{F}} \cdot ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}}$, which “reduces” to the bottom value $\bot$ every recursively computed value. The calculation proceeds as follows.

$$\begin{array}{cl}
 & pXML \cdot \sigma \\
\Leftrightarrow & \qquad \{\text{definition of } pXML\text{, definition of } \sigma\} \\
 & (\!|[[pSElem, pElem], id \ \star \ nl]|\!)_{\mathsf{F}} \cdot in_{\mathsf{F}} \cdot \\
 & ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \qquad \{\text{cata-cancelation}\} \\
 & [[pSElem, pElem], id \ \star \ nl] \cdot \mathsf{F}_{pXML} \cdot \\
 & ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \qquad \{\text{definiton of } \mathsf{F}\text{, Functor-}+\text{, Functor-}\times\text{, natural-}id\}
\end{array}$$

$$
\begin{array}{ll}
 & [[pSElem, pElem], id \ \star \ nl] \cdot \\
 & ((id \times (\underline{\perp} \cdot pXML^*) + (id \times id) \times (\underline{\perp} \cdot pXML^*)) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \qquad \{\text{absorption-}+\text{, natural-}id\} \\
 & [\ [pSElem \cdot (id \times (\underline{\perp} \cdot pXML^*), pElem \cdot ((id \times id) \times (\underline{\perp} \cdot pXML^*)], \\
 & \ id \ \star \ nl\ ] \cdot out_{\mathsf{F}}
\end{array}
$$

The calculation continues by evaluating the impact of σ upon each parcel. We shall concentrate on the *psElem* function, given that other cases demand a similar treatment. Then,

$$
\begin{array}{ll}
 & pSElem \cdot (id \times (\underline{\perp} \cdot pXML^*) \\
\Leftrightarrow & \qquad \{\text{definition of } psElem\} \\
 & ob \ \star \ \pi_1 \ \star \ cb \ \star \ nl \ \star \ concat \cdot \pi_2 \ \star \ oeb \ \star \ \pi_1 \ \star \ cb \ \star \ nl \cdot \\
 & (id \times (\underline{\perp} \cdot pXML^*)) \\
\Leftrightarrow & \qquad \{\text{constant function, result (5.24)}\} \\
 & ob \ \star \ \pi_1 \ \star \ cb \ \star \ nl \ \star \ concat \cdot \pi_2 \cdot (id \times (\underline{\perp} \cdot pXML^*)) \ \star \ oeb \ \star \\
 & \pi_1 \ \star \ cb \ \star \ nl \\
\Leftrightarrow & \qquad \{\text{definition of } \times\text{, cancelation-}\times\} \\
 & cb \ \star \ \pi_1 \ \star \ cb \ \star \ nl \ \star \ concat \cdot (\underline{\perp} \cdot pXML^*) \cdot \pi_2 \ \star \ oeb \ \star \\
 & \pi_1 \ \star \ cb \ \star \ nl
\end{array}
$$

The above expression explicitly points out with $\underline{\perp}$, the places where input information is missing. Given these specific critical points, it is up to the user to decide how to deal with them, given the overall context of the expression. In this particular case, we have decided to remove all the elements of the concatenation polluted with this mark, giving rise to the following slice.

```
pXML (SimpElem e xmls) = "<" ++ e ++ ">" ++ nl ++
                         "</" ++ e ++ ">" ++ nl
```

Note, however, that in general, unlike product backward slicing which always yields executable solutions, in this case the final slice may not be executable. This does not come to a surprise, since we are filtering input that can be critical to the overall computation of the original function.

Chapter 6

Semantic-based Slicing

The slicing method described in the initial chapter of this thesis is oriented to "macro" entities (functions, modules, data types, etc) of a program, therefore, extending to the functional paradigm a notion of slicing typical of the imperative program analysis. Of course, in the latter setting, the relevant program entities can be as localised as computational variables, leading to the identification of quite fine-grained slices. Chapter 5 introduced a different approach to enable slicing to cut through those "macro" entities. Its applicability, however, is somewhat limited and any sort of automated support will heavily depend on suitable support for algebraic rewriting. Furthermore, we seek for more expressive power in the definition of slicing criteria, *i.e.*, instead of resorting to slicing functions to serve as "indirect" slicing criteria, one would like to state where and what specific functional expressions are to be.

This chapter goes a step forward in the direction of what may be called "low-level" slicing: the aim is to allow slicing criteria to be by *any* functional expression within the code, together with its precise occurrence point inside the program.

By the beginning of this work we thought that the development of low-level higher-order lazy functional slicing would be a more or less straightforward engineering problem, easily solved through some combination of parsing and syntax tree traversal operations. We have even tried to implement low-level functional slicing in HaSlicer resorting to techniques similar to the ones used for performing high-level entity slicing.

However, all attempts to build such a tool resorting to a direct implementation of these operations invariantly ended by the discovery of some particular case where the resulting slices did not correspond to the expected ones. Moreover, by performing minor changes in the implementations in order to correctly cover some special cases, one often ended up introducing new problems and special cases.

Soon, however, we realised that the problem complexity had been underestimated from the outset. This goal led to a completely different approach to functional slicing which builds on both the language semantics (as the calculational approach in the previous chapter did), and the *evaluation strategy*. The latter happens to play a significant role in the whole process.

In such a context, we decided to target in this chapter *functional programs with higher-order constructs sharing a lazy strategy evaluation*. Note that, as it will be shown in section 6.5, the strict version of the proposed technique can be easily derived from the lazy one. Additionally, the removal of higher-order constructs represents a trivial simplification of the method introduced here. Thus, our working restrictions do not constraint in a significant way the applicability of this approach.

6.1 The Functional Language

In order to introduce our method in a generic, language-independent way, we start by defining a prototypical higher-order lazy functional language, which abstracts several functional programming language implementations.

The choice of the language syntax had to fulfil two main requisites. The language could not be excessively broad since this would introduce an unnecessary notational burden in the representation. On the other hand it could not be excessively small because this would make translations from/to real functional languages too complex to achieve.

Thus, a trade-off was found in the form of language FL, whose syntax is given in Figure 6.1. FL notation is basically that of the λ-Calculus enriched

$$
\begin{array}{rcll}
z & ::= & \lambda x.e & \\
 & | & C\ x_1 \cdots x_a & a \geq 0 \\
e & ::= & z & \\
 & | & e\ x & \\
 & | & x & \\
 & | & \texttt{let}\ x_n = e_n\ \texttt{in}\ e & n > 0 \\
 & | & \texttt{case}\ e\ \texttt{of}\ \{C_j\ x_{1j} \cdots x_{aj}\ \texttt{->}\ e_j\}_{j=1}^{n} & n > 0,\ a \geq 0 \\
\mathit{prog} & ::= & x_1 = e_1, \ldots, x_n = e_n &
\end{array}
$$

Figure 6.1: The FL syntax

with `let` and `case` statements. It introduces the domain $z \in$ U of values, the domain e ∈ E of expressions, the domain prog ∈ P of programs and the domain of x ∈ V of variables. Note that values are also expressions by the first rule in the definition of expressions.

A very important detail about the FL language is that functional application cannot occur between two arbitrary functional expressions, but only between an expression and a variable previously defined. In practice this implies that at evaluation time, the argument expression must have been previously added to the heap so that it can be used on a functional application. This requisite may seem strange for now, but it is necessary to deal correctly with the semantics upon which we define the slicing process. In particular, this rule ensures that when evaluating an application one does not have to address the creation of new heap closures.

However, this way of defining functional application requires some care when converting concrete functional programs to FL. In practice, the translation is achieved by the introduction of a new free variable within a `let` expression and the subsequent substitution of the expression by the newly introduced variable. As an example of such a transformation, consider the following Haskell program that removes every negative value from a list of integers

```
removeNegative :: [Int] -> [Int]
removeNegative = filter (> 0)
```

Note that function `filter` is being directly applied to another function, a predicate function. The definition has then to be rephrased into

```
removeNegative :: [Int] -> [Int]
removeNegative = let x = (> 0) in filter x
```

Of course, to accommodate real functional languages, some other straightforward syntactic translations are in demand. These include the substitution of `if then else` conditionals by `case` expressions with the respective `True` and `False` values or the substitution of `where` constructions by `let` expressions.

Some of these syntactic transformations have been implemented, as a proof-of-concept, in a front end for Haskell. This means that not only Haskell programs can be taken as input to the slicing process, but also that, on slicing completion one is able to reconstruct the slice exactly like the original program, but for the removal of the sliced expressions. These transformations amount to an expression rewriting process with the particularity that one keeps track of the transformations performed as well as the expressions involved in each one.

Finally, we have to uniquely identify the functional expressions and sub-expressions of a program, such that the slicing process may refer to these identifiers in order to specify what parts of the program belong to a specific slice. Moreover, these identifiers are also needed for identifying the expressions involved in each syntactic transformation so that the above mentioned translation can be performed as an isomorphism.

Expression and sub-expression identifiers are collected in a set L, and introduced in language FL by expression labelling as shown in Figure 6.2, where $a \geq 0$ and $n > 0$.

For the moment, one may look at labels from L as simple unique identifiers of functional expressions. Later, these labels will be used to capture information about the source language representation of the expression they denote, so that, by the end of the slicing process, one becomes able to reconstruct the slice's source code.

$$
\begin{array}{rcl}
z & ::= & (\lambda x : l_1.e) : l \\
 & | & (C\ x_1 : l_1 \cdots x_a : l_a) : l \\
e & ::= & z \\
 & | & e\ (x : l') : l \\
 & | & x : l \\
 & | & \texttt{let}\ x_n = e_n : l_n\ \texttt{in}\ e : l \\
 & | & \texttt{case}\ e\ \texttt{of}\ \{(C_j\ x_{1j} : l_{1j} \cdots x_{aj} : l_{aj}) : l'\ \texttt{->}\ e_j\}_{j=1}^{n} : l \\
\mathit{prog} & ::= & x_1 = e_1, \ldots, x_n = e_n
\end{array}
$$

Figure 6.2: Labelled FL syntax

6.2 Slicing and Evaluation

Dynamic slicing of functional programs is an operation that largely depends on the underlying evaluation strategy for expressions. This can be exemplified in programs where strict evaluation introduces non termination whereas a lazy strategy produces a result. As an example, consider the following functional program.

```
fact :: Int -> Int
fact 0 = 1
fact k = k * fact (k-1)

ssuc :: Int -> Int -> Int
ssuc r y = y + 1

g :: Int -> [Int] -> [Int]
g x z = map (ssuc (fact x)) z
```

If we calculate the slice of the above program with respect to expression `g (-3) [1,2]`, taking into consideration that the program is being evaluated under a strict strategy, the evaluation will never terminate and the intended slice is never computed.

On the other hand, under a lazy evaluation strategy, this same evaluation is possible, because `succ` is not strict over its arguments, and therefore `(fact` x), which introduces the non terminating behaviour, is not computed. Thus, under lazy evaluation, slicing is now feasible and one would expect, for this particular case, to obtain the following slice:

```
ssuc :: Int -> Int -> Int
ssuc r y = y + 1

g :: [Int] -> [Int]
g z = map (ssuc (fact x)) z
```

Note that, strictly speaking, the computed slice is not executable. Actually, this would require a definition of function `fact` in order to the entire sliced program to be interpreted or compiled. This possibility of retrieving non executable slices, was a deliberate choice. Actually, in a functional framework, if one calculates executable slices (without using any further program transformation), it often happens that such slices take enormous proportions when compared to the original code. Nevertheless, and because the expressions to be sliced away do not interfere with the selected slicing criterion, a suitable program transformation for this case is to substitute the expression in question by some special value of the same type. In HASKELL, for instance, and because types have a complete partial order structure, one could use the bottom value (usually denoted by $\perp$) of the type in question to signal the superfluous expressions. These and other possible code transformations that target the execution of slices are, however, drifting from the main focus of this chapter and will not be considered in the sequel.

The approach to low level slicing of functional programs proposed here is mainly oriented (but see section 6) to lazy languages. An important aspect which motivated this choice was that slicing has never been treated under such an evaluation strategy (combined with higher-order constructs). Moreover, intuition suggests, as in the example above, that lazy slices tend to be smaller than their strict counterparts.

Therefore, our starting point was a lazy semantics for FL, which is presented in Figure 6.3. This semantics is strongly based on the lazy semantics introduced by Launchbury in [Lau93], but for the increased expressiveness of FL. Thus, FL amounts to the Launchbury language with the extensions for both user defined data types (constructors) and case expressions, as presented in Figure 6.3. As it will be made clear in the following sections, these two syntactic constructs play an important role in the definition of the semantics as well as on the different semantic based operations to be defined.

In the lazy semantics presented in Figure 6.3, expression $\Gamma \vdash e \Downarrow \Delta \vdash z$ states that expression e under heap Γ evaluates to value z producing heap Δ as a result. As expected the expressions comply with the FL language and the heap structure used is a mapping from variables to expressions, where the latter may not be completely evaluated.

$$\Gamma \vdash \lambda y.e \Downarrow \Gamma \vdash \lambda y.e \qquad \textbf{\textit{Lamb}}$$

$$\Gamma \vdash C\ x_1 \cdots x_a \Downarrow \Gamma \vdash C\ x_1 \cdots x_a \qquad \textbf{\textit{Con}}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash \lambda y.e' \qquad \Delta \vdash e'[x/y] \Downarrow \Theta \vdash z}{\Gamma \vdash e\ x \Downarrow \Theta \vdash z} \qquad \textbf{\textit{App}}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash z}{\Gamma[x \mapsto\ e\ ] \vdash x \Downarrow \Delta[x \mapsto\ z\ ] \vdash \hat{z}} \qquad \textbf{\textit{Var}}$$

$$\frac{\Gamma[x_n \mapsto\ e_n] \vdash e \Downarrow \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n\}\ \texttt{in}\ e \Downarrow \Delta \vdash z} \qquad \textbf{\textit{Let}}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash C_k\ x_1 \cdots x_{a_k} \qquad \Delta \vdash e_k[x_i/y_{ik}] \Downarrow \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{C_j\ y_1 \cdots y_{a_j}\ \texttt{->}\ e_j\}_{j=1}^{n} \Downarrow \Theta \vdash z} \qquad \textbf{\textit{Case}}$$

Figure 6.3: Lazy semantics for FL

In Figure 6.3 and throughout this chapter the following syntactic abbreviations are used: $\hat{z}$ stands for $\alpha-$conversion of every bound variable in expression z to fresh variables, $[x_i \mapsto e_i]$ for $[x_1 \mapsto e_1, \ldots, x_i \mapsto e_i]$, $\Gamma[x_i \mapsto e_i]$ expresses the update of mapping $[x_i \mapsto e_i]$ in heap Γ and $e[x_i/y_i]$ the substitution $e[x_1/y_1, \ldots, x_i/y_i]$.

The semantics presented in Figure 6.3 entails lazy evaluation in the sense that lambda terms are allowed as values of computations and expressions (closures) are added to the heap (rule ***Let***) without further evaluation. Laziness also takes place, when, by rule ***App***, one substitutes the lambda variable by the application expression variable without further evaluating the expression in the heap which corresponds to the variable in the expression. The use of closures as well as the keeping of the evaluated values for each variable in the heap provides the semantics with a sharing mechanism which much improves its performance.

6.3 Lazy Forward Slicing

Let us start by analysing the *lazy print* problem, a simplified version of the more general problem of higher-order lazy functional slicing. The calculation of this particular kind of slice is completely based on the lazy evaluation coverage of a program, without taking any extra explicit slicing criterion. This means that a *lazy print* calculation amounts to extracting the program fragments that have some influence on the lazy evaluation of an expression within that program. For an example, consider the following trivial Haskell program where `g` receives a pair, whose first element is a list of integers and the second is an integer, and delivers the sum of all elements in the list.

```
fst :: (a, b) -> a
fst (x, y) = x

sum :: [Int] -> Int
sum []    = 0
sum (h:t) = h + (sum t)
```

```
g :: ([Int], Int) -> Int
g z = sum (fst z)
```

The *lazy print* of this program with respect to the evaluation of `g ([], 3)` is

```
fst :: (a, b) -> a
fst (x, ) = x

sum :: [Int] -> Int
sum []    = 0

g :: ([Int], Int) -> Int
g z = sum (fst z)
```

Note that the second clause of function `sum` (`sum (h:t) = h + (sum t)`) is sliced from the original program, because, for the slicing criterion in question (`g ([], 3)`), the evaluation of the result does not depend on the functional expression of the `sum` function clause dealing with non-empty lists. Furthermore, variable `y` in function `fst` is never used, which indicates that it can also be sliced away from the original program.

Automating this calculation entails the need for deriving an augmented semantics from the lazy semantics presented in Figure 6.3. This augmented semantics, which is presented in Figures 6.4 and 6.5, extends Launchbury semantics with an extra output value of type set of labels (S), for the evaluation function $\Downarrow$. The purpose of this set S is to collect all the labels from the expressions that identify what constitutes the *lazy print* of a given evaluation. Note that, instead of using an alpha conversion in the original rule ***Var***, we introduce a fresh variable in rule ***Let*** to avoid variable clashing.

The *lazy print* semantics uses two auxiliary functions, namely $\varphi : \mathbb{P}\,\mathrm{L} \leftarrow \mathrm{E}\times\mathrm{V}$ and $\mathcal{L} : \mathbb{P}\,\mathrm{L} \leftarrow \mathrm{E}$. Function φ collects the labels from all the occurrences of a variable in an expression and function $\mathcal{L}$ returns all the labels in an expression.

The intuition behind this augmented semantics is that operationally it collects all the labels from the expressions as they are evaluated by the semantic rules. The only exception to this behaviour, is rule ***Let***, which does

Con

$$\Gamma \vdash (\lambda y : l_1.e) : l \Downarrow_{\{l_1,l\}} \Gamma \vdash (\lambda y : l_1.e)$$

Lamb

$$\Gamma \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_{\{l'_k,l'\}} \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'$$

$$\text{where} \quad k \in \{1, \ldots, a\}$$

Figure 6.4: Lazy print semantics for values

not collect all the expression labels immediately but rather relegates the label collection to a later stage in the evaluation. This behaviour of rule ***Let*** is explained by the fact that there is not sufficient information available when this rule is executed to decide which variable bindings will be needed in the remainder of the evaluation towards the computation of the final result. A possible solution for this problem is to have a kind of memory associating pending labels and expressions such that, if an expression really gets to be used, then not only such expression labels, but also the pending labels that were previously registered in the memory, are included in the lazy print evaluation labels set.

A straightforward implementation of such a memory mechanism is the heap that we are already using as a memory device for registering variables and their associated expressions. Thus, by extending the heap from a mapping between variables and expressions to a mapping from variables to pairs of expressions and sets of labels, makes us able to capture the "pending labels" introduced by the ***Let*** rule.

A problem is spotted however in slices computed on top of the *lazy print* semantics given in Figures 6.4 and 6.5. As an example, consider the following fragment which calls some complex and very cohesive functions `funcG` and `funcH` which, however, do indeed contribute to the computation of values in `x` and `y`:

App

$$\frac{\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_{S_2} \Theta \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_{S_1 \cup S_2 \cup \{l', l\}} \Theta \vdash z}$$

Var

$$\frac{\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash z}{\Gamma[x \mapsto\ < e, L >\ ] \vdash x : l \Downarrow_{S_1 \cup L \cup \{l\}} \Delta[x \mapsto\ < z, \varepsilon >\ ] \vdash z}$$

Let

$$\frac{\begin{array}{c}\Gamma[y_n \mapsto\ < e_n[y_n/x_n], \{l_n\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) \cup \mathcal{L}(e_n) >\ ] \vdash \\ e[y_n/x_n] \Downarrow_{S_1} \Delta \vdash z\end{array}}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_{S_1 \cup \{l\}} \Delta \vdash z}\ y_n \text{ fresh}$$

Case

$$\frac{\begin{array}{c}\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash (C_k\ x_1 : l_1^\star \cdots x_{a_k} : l_{a_k}^\star) : l_k^\sharp \\ \Delta \vdash e_k[x_i/y_{ik}] \Downarrow_{S_2} \Theta \vdash z\end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l'_1 \cdots y_{a_j} : l'_{a_j}) : l_j^\natural\ \texttt{->}\ e_j\}_{j=1}^{n} : l \Downarrow_S \Theta \vdash z}$$

$$\begin{array}{lcl}\text{where} \quad S & = & S_1 \cup S_2 \cup \{l^\star{}_{n_j} \mid 1 \leq n \leq a\} \cup \\ & & \{l'{}_{n_j} \mid 1 \leq n \leq a\} \cup \{l_k^\sharp, l_j^\natural, l\}\end{array}$$

Figure 6.5: Lazy print semantics for expressions

```
f z w = let x = funcG z w
            y = funcH x z
        in (x, y)
```

When computing the *lazy print* of such a program, no matter what values are chosen for `z w`, the returned slice is always the following

```
f z w =


        (x, y)
```

The interpretation of such a slice may suggest that the variables introduced by the `let` expression do not have any effect on the result of the overall function. However, this completely contradicts what one already knew about the behaviour of functions `funcG` and `funcH`, *i.e.*, that they do contribute to the calculation of the final tupple result, thus, they should be part of the slice.

The reason for such a deviating behaviour induced by the lazy print semantics is behind rule ***Con*** definition. In particular, this problem arises because $C\ x_1 : l_1 \cdots x_a : l_a$ expressions are considered primitive values in the language, thus making rule ***Con*** to simply collect the outer labels of such expressions without evaluating the arguments of the constructor involved. This explains the odd behaviour of the above example, where function f returns a pair which falls into the $C\ x_1 : l_1 \cdots x_a : l_a$ representation in FL. Therefore, the only semantic rule applied during the lazy print calculation was the ***Con*** rule which does not evaluate the constructor (Pair) arguments and their associated expressions. Hence, one may now understand why the only labels that the semantics yielded during the evaluation are the ones visible at the time of application of the *Con* rule.

A possible approach to solve this problem of "extra laziness" induced by the semantics, would be to evaluate every data constructor parameter in a strict way. This, however, would throw away most of the laziness in the language, since the evaluation would become strict on every data type.

A much more effective solution is to divide the slicing calculation into two phases. The first phase uses the semantics in Figures 6.4 and 6.5, applying it until a value, possibly containing constructors with unevaluated expressions, is retrieved. The second phase takes both the value and the heap returned by the first phase and evaluates them under a semantics which is similar to the one used in the first phase except for rule ***Con*** which is substituted by the one in Figure 6.6.

This way, strict evaluation over constructor values is introduced, though it only takes place after a resulting value has been obtained using the complete lazy semantics. Note that most of the computation is still being made in a completely lazy framework and only a final strict evaluation step is performed.

$$\textbf{\textit{Con}}$$

$$\frac{\Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{S_1} \Delta \vdash z_k}{\begin{array}{c}\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_S \\ \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'\end{array}}$$

$$\begin{array}{lrcl} \text{where} & k & \in & \{1, \ldots, a\} \\ & S & = & L_k \cup \{l'_k, l'\} \cup S_1 \end{array}$$

Figure 6.6: Con rule for strict evaluation of the result value

6.4 Adding a Slicing Criterion

Despite the relevance that *lazy print* may have in, e.g., program understanding, a further step towards effective slicing techniques for functional programs requires the explicit consideration of slicing criteria. In this section, we present an approach where a slicing criterion is specified by sets of program labels.

The slicing process proceeds as in the previous case, except that now, one is interested in collecting the program labels affected not only by a given expression, as before, but simultaneously by the expressions associated to the labels introduced by the user as a slicing criterion.

A first and straightforward approach to implement a slicer to achieve this goal takes into account the set of collected labels on both the output and the input of the evaluation function $\Downarrow$. Therefore, the semantic rule for λ-expressions changes to the one displayed in Figure 6.7. This extra rule enables the semantics to evaluate expressions taking into account a set of labels S_i supplied as a slicing criterion and its impact on the resulting slice S_f. Putting it in another way, each rule has to compute the resulting set of labels S_f considering the effect that the expressions denoted by the input labels in S_i may have in the slice being computed.

Soon, however, it becomes difficult to specify the remainder semantic rules taking into account the impact of the receiving set of labels. The problem of specifying these rules is that, in many cases, there is not enough information in the rule being specified to enable the decision of including a certain label

$$\boxed{\begin{array}{c} \textit{Lamb} \\ S_i, \Gamma \vdash (\lambda y : l_1.e) : l \Downarrow \Gamma \vdash (\lambda y : l_1.e) : l, S_f \\ \text{where } S_f = S_i \cup \bigcup\{\varphi(e, y) \mid l_1 \in S_i\} \cup \{l \mid l_1 \in S_i\} \end{array}}$$

Figure 6.7: Improved semantics

or not.

For instance, in the ***App*** rule one may not immediately decide whether to include or not label l_1 in the resulting label set. The reason for this is that one has no means of knowing in advance whether a particular expression in the heap will ever become part of the slice. If such an expression is to be included into the slice, somewhere along the remainder of the slicing process, then label l_1 will also belong to the slice as well as all the labels that l_1 affects by the evaluation of the first premise of rule ***App***.

In order to overcome this problem, one should look for some independence from the slicing process over the partial slices that are being calculated by each semantic rule. Thus, instead of calculating partial slices on the application of every rule, we compute partial dependencies between labels. This entails the need for a further modification in the rules which now have to compute maps of type $\mathbb{P}\ \mathrm{L} \leftarrow \mathrm{L}$, called *lmap*'s, rather than sets of labels. The intuition behind *lmap*'s is that all labels in their codomains depend on their corresponding labels from the *lmap* domain. The resulting semantics is presented in Figures 6.8 and 6.9 where in rule *Let* variable y_n is a fresh variable.

In the sequel the following three operations over *lmap*'s are required: an application operation, resorting to standard finite function application, defined by

$$F(x) = \begin{cases} F\ x & \text{if } x \in \mathit{dom}\ F, \\ \{\} & \text{otherwise.} \end{cases}$$

a *lmap* multiplication $\oplus$, defined as

Lamb

$$\Gamma \vdash (\lambda y : l_1.e) : l \Downarrow_F \Gamma \vdash (\lambda y : l_1.e) : l$$

$$\text{where } F = [l_1 \mapsto \varphi(e, y) \cup \{l\}]$$

Con

$$\Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \Downarrow_F \Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l$$

$$\begin{array}{lrcl} \text{where} & k & \in & \{1, \ldots, a\} \\ & F & = & [l_k \mapsto l] \end{array}$$

Figure 6.8: Higher-order slicing semantics for values

$$(F \oplus G)(x) = F(x) \cup G(x)$$

and, finally, a range union operation *urng*, defined as

$$urng\ F = \bigcup_{x \in dom\ F} F(x)$$

Again, this semantics suffers from the problem identified in the *lazy print* specification i.e., the semantics is "too lazy". Once more, to overcome such undesired effect one applies the strategy taken earlier, therefore introducing a new rule (Fig. 6.10) to replace the original *Con* rule, and the slicing process gets similarly divided into two phases.

By changing the output of the evaluation function from a set to a *lmap* of labels, we no longer have the desired slice of the program by the end of the evaluation. Instead, what is returned is a *lmap* specifying the different dependencies between all expressions that were needed to evaluate the program under analysis. Based on this *lmap* value the desired slice can then be computed as the transitive closure of the dependencies *lmap* starting by the set of labels identifying the expressions form our slicing criteria.

Furthermore, splitting the slicing process into a dependencies calculation and the computation of a slice for the set of pertinent labels (*i.e.* the slicing

App

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_G \Theta \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z}$$

$$\text{where} \quad H \quad = \quad F \oplus G \oplus [l' \mapsto \{l, l_1\}]$$

Var

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto <e, L>\ ] \vdash x : l \Downarrow_G \Delta[x \mapsto <z, \varepsilon>\ ] \vdash z}$$

$$\text{where } G = F \oplus [l \mapsto L]$$

Let

$$\frac{\begin{array}{c}\Gamma[y_n \mapsto <e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n)>\ ] \vdash \\ e[y_n/x_n] \Downarrow_F \Delta \vdash z\end{array}}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \Delta \vdash z}$$

$$\text{where } G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

Case

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (C_k\ x_1 : l_1^\star \cdots x_{a_k} : l_{a_k}^\star) : l_k^\sharp \qquad \Delta \vdash e_k[x_i/y_{ik}] \Downarrow_G \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l'_1 \cdots y_{a_j} : l'_{a_j}) : l_j^\natural\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_H \Theta \vdash z}$$

$$\begin{array}{lcl}\text{where} \quad G & = & F \oplus G \oplus [l_m^\star \mapsto \varphi(e_k, y_m) \cup \{l'_m, l_k^\natural\} | 1 \leq m \leq a_k] \oplus \\ & & [l_k^\natural \mapsto \{l\}] \oplus [l'_m \mapsto \varphi(e_k, y_m) \cup \{l_k^\natural\} | 1 \leq m \leq a_k]\end{array}$$

Figure 6.9: Higher-order slicing semantics for expressions

criterion), makes easier the calculation of slices that only differ on the slicing criterion used. For such cases, one can rely on a common dependencies *lmap* and the whole process amounts to the calculation of the transitive closure of redefined sets of labels.

6.5 Strict Evaluation

Slicing under strict evaluation is certainly easier. A possible semantics, as considered in Figures 6.11 and 6.12, can be obtained by a systematic simpli-

Con

$$\frac{\Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{F_k} \Delta \vdash z_k}{\begin{array}{c}\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_G \\ \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'\end{array}}$$

$$\begin{array}{lccl} \text{where} & k & \in & \{1, \ldots, a\} \\ & G & = & F_k \oplus [l'_k \mapsto l'] \end{array}$$

Figure 6.10: Con rule for strict evaluation of the result value

fication of the semantics used in the lazy case. Of course, this is not the only possibility. To make comparison possible between the lazy and strict case, however, we chose to keep specification frameworks as similar as possible, although we are aware that many details in the strict side could have been simplified. For example, strict semantics can always return slices in the form of sets of labels instead of calculating maps capturing dependencies between code entities.

Lamb

$$\Gamma \vdash (\lambda y : l_1.e) : l \downdownarrows_F \Gamma \vdash (\lambda y : l_1.e) : l$$

$$\text{where } F = [l_1 \mapsto \varphi(e, y) \cup \{l\}]$$

Con

$$\Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \downdownarrows_F \Gamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l$$

$$\begin{array}{lccl} \text{where} & k & \in & \{1, \ldots, a\} \\ & F & = & [l_k \mapsto l] \end{array}$$

Figure 6.11: Strict slicing semantics for values

Moreover, in the strict case there is no need to capture pending labels in the heap, since `let` expressions are evaluated as soon as they are found. This leads to a simplification of the heap from a mapping between variables and pairs of expressions and set of labels to a mapping between variables and values.

App

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1 . e') : l_2 \qquad \Delta \vdash e'[z_1/y] \Downarrow_G \Theta \vdash z}{\Gamma[x \mapsto \; z_1] \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z}$$

$$\text{where} \quad H \quad = \quad F \oplus G \oplus [l' \mapsto \{l, l_1\}]$$

Var (whnf)

$$\frac{\Gamma \vdash z \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto \; z] \vdash x : l \Downarrow_G \Delta[x \mapsto \; z\ ] \vdash z}$$

$$\text{where } G = F$$

Let

$$\frac{\Gamma \vdash e_n \Downarrow_F \Delta \vdash z_n \qquad \Gamma[y_n \mapsto z_n] \vdash e[z_n/x_n] \Downarrow_G \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_H \Delta \vdash z}\ y_n \text{ fresh}$$

$$\text{where } H = F \oplus G \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

Case

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (C_k\ x_1 : l^\star_1 \cdots x_{a_k} : l^\star_{a_k}) : l^\sharp_k \qquad \Delta \vdash e_k[x_i/y_{ik}] \Downarrow_G \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l'_1 \cdots y_{a_j} : l'_{a_j}) : l^\natural_j\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_H \Theta \vdash z}$$

$$\begin{aligned} \text{where} \quad G \quad = \quad & F \oplus G \oplus [l^\star_m \mapsto \varphi(e_k, y_m) \cup \{l'_m, l^\natural_k\} | 1 \leq m \leq a_k] \oplus \\ & [l^\natural_k \mapsto \{l\}] \oplus [l'_m \mapsto \varphi(e_k, y_m) \cup \{l^\natural_k\} | 1 \leq m \leq a_k] \end{aligned}$$

Figure 6.12: Strict slicing semantics for expressions

In what concerns the semantic rules, only the ***App*** and ***Let*** need to be changed, along with some minor adaptation of other rules that deal with the modified heap.

Another decision taken in the strict slicing semantics specification was to keep value sharing *i.e.*, sharing of values that are stored in the heap. Nevertheless, one can easily derive a slicing semantics without any sharing mechanism, for which case one could probably remove the heap from the semantics.

Finally, note that now, due to the eager strategy, there is no need to introduce a new ***Con*** rule to force the evaluation of unevaluated expressions inside result values. Therefore, unlike the two previously presented versions of lazy slicing, strict slicing is accomplished in a single evaluation phase.

6.6 Comparison

All slicing algorithms presented in this chapter were introduced as (evaluators of) a specific semantics. Such an approach provides an expressive setting on top of which one may reason formally about slices and slicers. This is illustrated in this section to confirm the intuition that, in general, under the same slicing criterion, lazy slices are always smaller than strict slices.

In the case of the *lazy print* semantics, such a proof amounts to showing that the set of labels returned by the lazy print is a subset of the set of labels yielded by an hypothetical strict print semantics.

But, since both the higher-order lazy slicing semantics and the strict one do not return sets of labels but maps of dependencies, one has to restate the proof accordingly. This can be achieved in two ways: either including the final transitive closure calculation in the slicing process, or introducing a partial order over the dependency *lmap*'s that respects subset inclusion.

We chose the latter alternative, and introduce the following partial order over *lmap*'s, which refines the standard definition order on partial functions.

$$F \preceq G \Leftrightarrow \mathit{dom}(F) \subseteq \mathit{dom}(G) \wedge (\forall x \in \mathit{dom}(F) . F(x) \subseteq G(x))$$

Now, the property that "lazy slices are smaller than strict slices" can be formulated as follows.

$$\text{If } \Gamma \vdash e \Downarrow_F \Delta \vdash z \text{ and } \Gamma \vdash e \downdownarrows_G \Theta \vdash z \text{ then } F \preceq G$$

The proof proceeds by induction over the rule-based semantics. First note that the property is trivially true for all identical rules in both semantics. Such are the cases of rules ***Lamb***, ***Con*** and ***Case*** for which the resulting *lmap*'s are equal. The proof for the remaining cases follows.

Case *App*: Evaluation of expressions under these rules takes the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1 . e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_G \Psi \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z}\ \mathit{App}$$

$$\text{where} \quad H \quad = \quad F \oplus G \oplus [l' \mapsto \{l, l_1\}]$$

$$\frac{\Gamma \vdash e \Downarrow\!\Downarrow_I \Theta \vdash (\lambda y : l_1 . e') : l_2 \qquad \Theta \vdash e'[z_1/y] \Downarrow\!\Downarrow_J \Phi \vdash z}{\Gamma[x \mapsto\ z_1] \vdash e\ (x : l') : l \Downarrow\!\Downarrow_K \Phi \vdash z}\ \mathit{App}$$

$$\text{where} \quad K \quad = \quad I \oplus J \oplus [l' \mapsto \{l, l_1\}]$$

By induction hypothesis one has that $F \preceq I$. By definition of rule ***Let***, which is the only rule that changes the heap, one has that $\mathcal{L}(\Delta) \cup \mathit{urng}\ F = \mathcal{L}(\Theta) \cup \mathit{urng}\ I$, where function $\mathcal{L}$ is overloaded to collect all the labels of the expressions in a heap. It follows that

$$\begin{array}{ll}
 & \mathcal{L}(\Delta) \cup \mathit{urng}\ F = \mathcal{L}(\Theta) \cup \mathit{urng}\ I \\
\Rightarrow & \quad \{\text{Induction Hypothesis}\} \\
 & \mathcal{L}(\Delta) \diagdown \mathcal{L}(\Theta) \subseteq \mathit{urng}\ I \\
\Rightarrow & \quad \{\text{Defintion of } \oplus\text{, noting that every possible label that } G \\
 & \quad \text{may collect from heap } \Delta \text{ is already in } I\} \\
 & G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Induction Hypothesis}\} \\
 & F \preceq I \wedge G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Definition of } \oplus\} \\
 & F \oplus G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Definition of } \oplus\} \\
 & F \oplus G \oplus [l' \mapsto \{l, l_1\}] \preceq I \oplus J \oplus [l' \mapsto \{l, l_1\}] \\
\Rightarrow & \quad \{\text{Defintion of } G \text{ and } H\} \\
 & G \preceq H
\end{array}$$

Case *Let*: Evaluation of expressions under these rules takes the following format, according to the evaluation strategy used (note that y_n is a fresh variable in both rules).

$$\textbf{\textit{Let}}$$
$$\frac{\Gamma[y_n \mapsto <e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n)>\,] \vdash e[y_n/x_n] \Downarrow_F \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \Delta \vdash z}$$
$$\text{where } G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

$$\textbf{\textit{Let}}$$
$$\frac{\Gamma \vdash e_n \downdownarrows_H \Theta \vdash z_n \qquad \Gamma[y_n \mapsto z_n] \vdash e[z_n/x_n] \downdownarrows_I \Phi \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \downdownarrows_J \Phi \vdash z}$$
$$\text{where } J = H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$$

By induction hypothesis and because $\mathcal{L}(e_n) \subseteq \mathit{urng}\ H$ one has that $F \preceq H \oplus I$. It follows that

$$\begin{array}{ll}
 & G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)] \\
\Rightarrow & \qquad \{F \preceq H \oplus I\} \\
 & G \preceq H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)] \\
\Rightarrow & \qquad \{\text{Definition of } K\} \\
 & G \preceq K
\end{array}$$

Case *Var*: Evaluation of expressions under these rules takes the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto <e, L>\,] \vdash x : l \Downarrow_G \Delta[x \mapsto <z, \varepsilon>\,] \vdash z} \quad \textbf{\textit{Var}}$$
$$\text{where } G = F \oplus [l \mapsto L]$$

$$\frac{\Gamma \vdash z \downdownarrows_H \Delta \vdash z}{\Gamma[x \mapsto\ z] \vdash x : l \downdownarrows_I \Delta[x \mapsto\ z\ ] \vdash z} \quad \textbf{\textit{Var}}$$
$$\text{where } I = H$$

By induction hypothesis one has that $F \preceq H$. Since the only way to add entries to the heap is via the rule ***Let***, and because, in strict semantics, such rule increments the dependencies *lmap* with every label from the newly introduced expressions, it follows that increments to the strict evaluation *lmap* will contain every mapping that is pending on the modified higher-order slicing heap. Thus, even though it may happen that at the time of evaluation of the ***Var*** rule, one may have $I \preceq G$, in the overall evaluation tree the dependency *lmap* for the lazy evaluation is always smaller or equal to the strict evaluation *lmap*.

Chapter 7

Contributions and Related Work

This chapter sums up the main contributions of the first part of this thesis, devoted to the development of techniques for slicing functional programs. In a final section, this research is put in perspective with respect to the literature.

7.1 Contributions and Future Work

The first part of this thesis introduced in detail three new and different techniques for slicing functional programs namely, functional slicing using Functional Dependence Graphs (FDG), a calculational approach based on algebraic program calculus, and a semantics based approach addressing high-order lazy functional programs. Besides describing the processes behind each of these strategies, we also characterized each technique in terms of its abstraction level (from "macro" entities to arbitrary expressions) as well as of the different kinds of slicing criteria employed in each case.

In what respects to the abstraction level, the first technique presented, slicing using FDG's, targets *high level functional program entities.* Here, the term high level program entities, refers to functional modules, functions, data-types, data-type constructors and data-type destructors. On the other hand the other slicing strategies presented in chapter 5 and 6 extend the previous approach by addressing low-level functional program entities, *i.e.* the functional expressions and sub-expressions upon which the functional algorithms are built upon.

Under the overall *motto* of functional slicing, the aim of the FDG based approach was twofold. On the one hand a specific dependence graph structure, the FDG, was introduced as the core infrastructure for slicing techniques and other source code analysis operations. On the other hand, slicing algorithms were expressed through simple combinators described in a language independent way and shown to provide a basis for an *algebra of generic slicing* over high level program entities.

What makes FDG a suitable structure for our purpose is parameterisation by an ontology of node types and differentiated edge semantics. This makes possible to capture in a single structure the different levels of abstraction a program may possess, although we have not explored this possibility to the limit, which would be the representation of functional expressions. This way a FDG may capture not only high level views of a software project like the ones presented here (*e.g.*, how modules or data-types are related), but also low level views (down to relations between functional statements inside function's bodies). Moreover, as different program abstraction levels are stored in a single structure, it becomes easy to jump across views according to what the analyst needs. This allows a programmer to take a high level perspective of the entire solution, by inspecting its module view, and then drilling down to a specific module in order to inspect its constituents. It would be interesting to develop this up to a point allowing the user to inspect the code behind each graph node.

This approach was supported by the development of a proof-of-concept tool – HaSlicer – described in chapter 3.

HaSlicer is a fully functional slicer targeting functional programs written in Haskell, featuring a user-friendly graph visualisation of the high level program entities contained in the analysed code, and allowing the user to easily navigate through the underlying dependencies graph in a smooth way. It should also be stressed the elliptical display of the graph, and the possibility of navigating in a scalable way through the entire graph.

Future work in HaSlicer includes the improvement of the user interface, for instance by using context menus once a node is clicked and offering to the user the possibility of performing the available operations for the node in question; the development of further navigation utilities (e.g., hiding and displaying specific parts of the graph); the implementation of more operations over the source code (such as the computation of the most used/unused program entities and the modules that are imported but not used). Moreover, given the high isolation scope of the FDG construction algorithm for each analysed functional module, we believe that a suitable parallelisation of the FDG construction exploiting this aspect, would increase the performance of HaSlicer in the analysis of bigger functional systems.

As a case-study we have shown how functional slicing techniques and tools can be used to identify and isolate software components entangled in (functional) legacy code. Both manual user oriented techniques for component discovery and a fully automatic approach based on the notions of software coupling and cohesion were discussed. The latter approach seems to have a particular interest as an *architecture understanding* technique in the earlier phases of the re-engineering process. HaSlicer has proven to be an interesting tool for isolating the discovered components which quite often appear, not as completely autonomous units, but rather with some weak dependencies to other components that have to be carefully analysed.

An interesting area for future research is the adaptation of graph clustering techniques, already available in the literature, to the discovery of components over FDG instances. Actually, we have already carried out some experiences with adjacency matrixes algorithms which pointed to a significant reduction in the time to compute component candidates. Furthermore, the automatic discovery of components could be parameterised, by allowing the user to define in which set of modules he is interested in looking for components, which limit values should cohesion and coupling have and which specific program entities should be considered for a particular component. All these parameterisations would deeply affect the performance and quality of the identified components. Finally, it would also be interesting to extend the FDG in order to capture other kinds of inter program entities dependencies, such as file, database and memory space share dependencies.

Chapter 5 introduced a completely different approach to functional slicing in which identification is formulated as an equation in an algebraic calculus of programs [BM97]. This seems a promising technique, although of limited scope.

The approach uses as slicing criteria specific functions that select part of the input or output information. Thus, although being sound by construction, this technique lacks some flexibility in the definition of slicing criteria, which in some cases can become quite non-trivial to define. Even more, the slicing criteria is also limited to operate over the input or output values of functions, thus making it difficult to define slicing criteria referring specific functional (sub-)expressions. In theory one could always divide a function definition and embed a slicing function between the resulting definitions in order to slice over specific sub-expressions of a function. Nevertheless, in practice this splitting operation often becomes too difficult to achieve.

In order to overcome the problems faced on the calculation approach, we developed a semantic based slicing process, introduced in chapter 6. This strategy applies to low level slicing of functional programs, highlighting a strong relationship between the slicing problem and the underlying evaluation strategy. In particular, this technique is able to perform slicing over higher-order lazy functional programs, and unlike the calculational approach, one may refer to particular program (sub-)expressions in order to define suitable slicing criteria. A functional implementation of this slicing process was also implemented in Haskell.

Although the whole of chapter 6 focus on forward slicing, we strongly believe that a correct inversion of the dependencies *lmap*'s, followed by a transitive closure calculation, will capture the backward cases.

The generalisation of slicing techniques to the software architecture level is the subject of the second part of this thesis. In particular we seek to make them applicable, not only to architectural specifications (as in [Zha98b]), but also, directly, to the source code level of large *heterogeneous* software systems, *i.e.*, systems implemented in multiple languages and consisting of many thousands of lines of code spread over several files.

7.2 Related Work

7.2.1 Functional Slicing

Although specific research in slicing of functional programs is sparse, the work of Reps and Turnidge [RT96] should be mentioned as sharing the same objectives. The authors perform slicing by composing projection functions with the functions being sliced. The approach they take to analyse the impact of such composition is based on regular tree grammars, which must be previously supplied. This way, slicing strictly depends on the actual program syntax given by the regular tree grammars, whereas in our approach such dependence is restricted to a pre-processing phase.

Another work slightly related to ours is [ZXG05] where a functional framework is used to formalise the slicing problem in a language independent way. Nevertheless, their primary goal is not to slice functional programs, but to use the functional *motto* to slice imperative programs given a modular monadic semantics.

The work of Vidal, et al [Vid03], where forward slicing of declarative programs is presented based on partial evaluation, shares a similar objective, but departures from a pure dynamic background, given by the partial evaluation based technique, and focus on the low level functional code entities whilst ours departures from a complete static approach with a focus in high level functional entities and their interactions.

Perhaps the work closer to ours comes from the Programmatica project tools [Hal03], which includes an HASKELL slicer based on a similar notion of slicing. However, such tool does not deal with the problem of module imports nor does it possess suitable visualisation techniques of both the intermediate structure and the results of the program transformation algorithms. Moreover, the slicing criterion is limited to some code entities, whilst in our approach it can be any node form the FDG.

7.2.2 Component Discovery

The methodology for component identification is based on the ideas first presented by Schwanke et al [SH94] [Sch91], where component design principles like coupling and cohesion are used to identify highly cohesive modules. In our own component identification and isolation strategy, documented as a case-study in chapter 4, we diverge from the existing approaches, by targeting the functional paradigm, and by making use of the lazy properties of Haskell in order to obtain answers in an acceptable time. This was never addressed before, to best of our knowledge.

A second difference between our approach to component identification and other techniques, which are usually included in the boarder discipline of software clustering [Wig97], is that we are working with functional languages with no aggregation units other then the module itself. In contrast to this, most of the software clustering algorithms are oriented to the OO paradigm, and as a consequence, they are often based on the notion of a class which is itself an aggregation construct. Thus, we had to cope with a much smaller granularity of programming units to modularise.

7.2.3 Slicing by Calculation

The requirement that programs should be first translated to a pointfree notation may seem, at first sight, a major limitation of the slicing by calculation technique described in chapter 5. However, automatic translators have been developed within the author's own research group [Cun05]. Note, however, that not only this sort of translators, but also general purpose rewriting systems able to make program calculation a semi-automatic task, are needed to scale up this approach to non academic case-studies. Fortunately this is an active area of research within the algebra of programming community.

Although specific research in slicing of functional programs is sparse, the work of Reps and Turnidge [RT96] should be mentioned as somewhat related to ours. The idea of composing projection functions to slice other functions comes from their work, but the approach they take to analyse the impact of such composition is completely different from ours. They resort to regular tree grammars, which must be previously given in order to compute the desired slices. This way, their approach strictly depends on the actual program syntax. Moreover, they limit themselves to functions dealing with lists or dotted pairs. Another work slightly related to ours is [ZXG05] where a functional framework is used to formalise the slicing problem in a language independent way. Nevertheless, their primary goal is not to slice functional programs, but to use the functional *motto* to slice imperative programs given a modular monadic semantics.

The approach outlined in this chapter is just a first attempt to perform slicing by mean of algebraic calculation. Thus there are plenty of points to proceed with this line of investigation, from which we would like to emphasise the following

- To extend the calculational process to functions defined by *hylomorphims* [BM97], with inductive types acting as virtual data structures, and

- To analyse the feasibility of this process applied to the dual picture of *coinductive* functions, *i.e.*, functions to final coalgebras.

This last extension may lead to a method for *process slicing*, with processes encoded in coinductive types (see, *e.g.*, [Sch98] or [Bar01]), with possible applications to the area of reverse engineering of software architectures (in the sense of *e.g.*, [Zha98b]).

Finally, it would be of outermost interest to take the *relational* challenge seriously and look for possible gains in calculational power by moving to a category of relations [FŠčedrov90, BH93] as a preferred semantic universe.

Being based on the solid framework of the functional calculus, the slicing process presented in this chapter is, by definition, a correct one. This is an aspect of great importance, since most of the slicing processes developed so far demand difficult, post development, verifications of its correctness. Even more, most of the evaluations are performed by means of automated testing, thus providing only partial results. Although being a sound process by construction, one should be aware that when such process is automated by means of mechanical algebraic calculational systems, the correctness of the solution relies on the correctness of the mechanical system.

An important aspect of the calculational approach presented, at least to the extent investigated here, is that one can only define slicing criteria either at the input or at the output of functions. Certainly these criteria will then influence and even modify the inner expressions of the function definition it is being applied to. But, there is no way to define slicing criteria directly involving inner function expressions, within particular occurrences inside a function definition. This is exactly the problematic addressed in the next chapter, *i.e.*, how to perform slicing on higher-order lazy functional programs having *arbitrary functional expressions* as a slicing criterion.

7.2.4 Semantic Based Slicing

While we regard our semantic based slicing work as a first incursion on higher-order lazy functional slicing, there is a number of related works that should be mentioned.

In [RT96] Reps and Turnidge provide a static functional slicing algorithm but, in contrast to our approach, theirs target first-order strict functional programs. Besides considering a different language class (first-order) and a different evaluation strategy (strict), the authors define slicing criteria by means of projection functions, a strategy similar to the one addressed in our calculational approach which we regard as a too rigid scheme when compared to our own approach resorting to a sub-expression labelling mechanism.

Reference [OSV04] presents a strategy to dynamically slice lazy functional languages. Nevertheless, they leave higher-order constructs as a topic for future work, and base their approach on redex trails. This leads to a slicing criterion definition (which consists of a tuple containing a function call with full evaluated arguments, its value in a particular computation, the occurrence of the function call and a pattern indicating the interesting part of the computed value) which is much more complex to be used in practice than our own. The latter, by pointing out a specific (sub)expression in the code, represents a more natural way for the analyst to encode the relevant aspects of the code that he/she wants isolated.

Perhaps the work most related work to our semantic based slicing is [Bis97], where the author presents an algorithm for dynamic slicing of strict higher-order functional languages followed by a brief adaptation of the algorithm to lazy evaluation. A major difference with the approach proposed by Wiggerts is that, recursive calls must be explicitly declared in the language and there is no treatment of mutual recursive functions which, as pointed out by the author, results in a considerable simplification of the slicing process. Again, we believe that our definition of the slicing criterion is more precise than the one used in [Bis97], which consists of the value computed by the program in question (even though more flexible slicing criteria are briefly discussed).

Finally, it should be emphasised that a slicing criterion, like the one proposed in our semantic approach, that permits to choose any (sub)expression of the program under analysis, deeply influences and augments the complexity of the slicing process, especially under a lazy evaluation framework like the one we have addressed. In fact, this aspect is the main reason for the evolution of the slicing algorithm from a one phase process, like the one presented in section 6.3, to a two phase process where one must first keep track of internal (sub)expression lazy dependencies before calculating the final slice with respect to the relevant (sub)expressions.

Part II

Slicing for Architectural Analysis

Chapter 8

Recovering Coordination Specifications

8.1 Introduction to Part II

As argued in the introduction, the systematic, tool-supported discovery of coordination policies from legacy code and the reconstruction of the corresponding specifications, is a main issue in program analysis. Actually, its role in software re-engineering became more and more important as software solutions evolve to a new level of reuse and dependency on foreign services and components. Faced with such a scenario, there is an urgent need of having tools and models to assist on the disentangling of the coordination structure of software systems.

The second part of this thesis is a step to tackle such a problem. Our approach is based on first building an extended system dependence graph, to provide a structural and easy-to-manipulate representation of program data, and then resorting to slicing techniques over such a graph to extract the relevant coordination policies.

Two alternatives are considered here to address the problem of discovering and extracting coordination data from legacy code, once a specific dependence graph structure, to be referred as the *coordination dependence graph* in the sequel, has been built.

- The first one, proceeds by systematically translating the data recorded in the coordination dependence graph into a specific software orchestration language. The outcome is, therefore, a high-level specification of the recovered coordination policies. We have used Orc, a recent general purpose orchestration language purposed by J. Misra and W. Cook [MC06] for this task. Orc scripts can be animated to simulate such specifications and study alternative coordination policies. Appendix B provides a brief introduction to Orc, its syntax and informal semantics.

- An alternative approach inspects the entire coordination dependence graph for the identification of graph patterns which are known to encode particular coordination schemes. For each instance of one of these such patterns, discovered in the graph, the corresponding fragment of source code is identified and returned.

This chapter introduces the construction of the coordination dependence graph and the extraction of coordination specifications according to the first of the two approaches mentioned above. Alternative generation of such specifications in WS-BPEL [JE07, Mig05] is also discussed.

The second approach to coordination recovery, based on the identification of particular graph patterns, is discussed in chapter 9. This turns out to be a more robust alternative when in presence of highly complex coordination policies.

Both approaches are *generic* in the sense that they do not depend upon the programming language or platform in which the original system was developed. Actually, they can be implemented to target any language with basic communications and multi-threading capabilities.

However, a prototype tool, intended to serve as a “proof-of-concept” for the methods proposed in the second part of this thesis, was developed to analyse *Common Intermediate Language* (CIL) source code, the language interpreted by the .Net Framework for which every Microsoft .Net compliant language compiles to.

The presentation of this tool, which we have called CoordInspector, is made on chapter 10.

Finally, chapter 11 reports on the application of these methods to a concrete project of software integration.

8.2 Recovering Coordination Specifications: An Overview

The building blocks of the (more-or-less explicit) coordination layer of a software system are the calls to *communication primitives*. These are used to invoke functionality exposed by third-party entities or simply for the exchange of information with foreign resources. Altogether, it is from them and the programming logic involved in their use, that the system coordination layer is constructed. The notion of a communication primitive, is to be understood here in the broad sense of any possible mean that a system or component can use to communicate or control another component or system. Direct foreign calls to well referenced components such as web-services calls, RMI or .Net Remoting calls to distributed objects are typically examples but, by no means, the only ones.

The specific combination of communication primitives is what allows software systems to control and interact, in complex ways, with other systems, processes, databases and services in order to achieve particular goals. Thus, it is reasonable to expect that any coordination discovery strategy should start by identifying such primitive communication statements in the source code, together with the specific program context in which they are embedded.

Depending on the type of communication primitives chosen to base the coordination discovery process upon, one will obtain different kinds of abstracted coordination layers at the end of the discovery process. I.e., if one chooses to base the discovery process over inter-thread communication primitives, than, at the end, one will obtain the orchestration specification of the system's threading layer. Similarly, *web-service* communication primitives with lead to a Service Interaction layer discovery, and interaction policies.

COM, *CORBA* and *RMI* primitives with lead to the identification of distributed objects.

In order to cope with such variety of coordination layers, our approach is *parametric* on the communication primitives as well as on the calling mode. For the latter we distinguish between *synchronous* and *asynchronous* communication calls, since this distinction plays an utmost important role in determining the coordination model. This classification of the calling mode is, of course, open and can be refined or extended to cope with other possibilities.

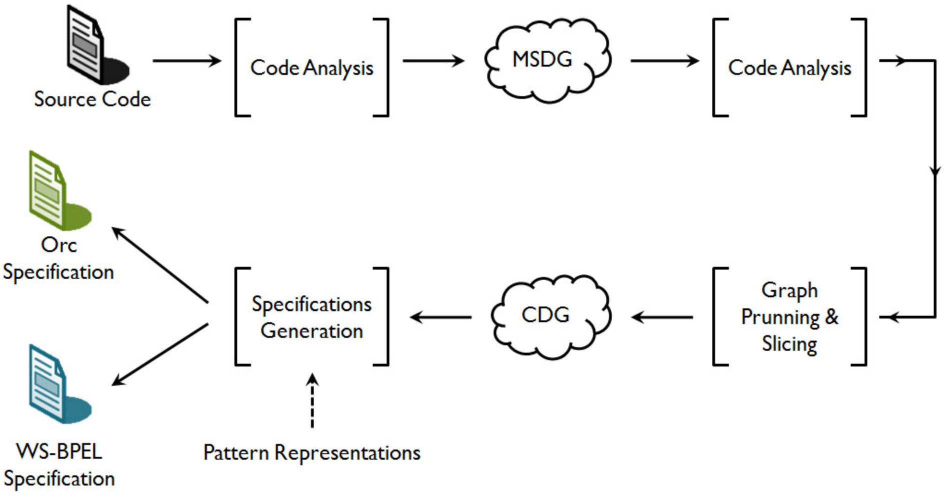


Figure 8.1: The overall strategy

The overall strategy underlying our approach to recover coordination specifications from source code is illustrated in Figure 8.1. As we have already mentioned, it is based on a notion of *coordination dependence graph*, abbreviated to CDG in the sequel, proposed here as a specialisation of standard program dependence graphs [FOW87] used in classical program analysis.

The process starts by the extraction of a comprehensive dependence graph, denoted in the sequel by the acronym MSDG (after *Managed System Dependence Graph*), from source code. Its construction, which extends an algorithm proposed in [HRB88], is detailed in section 8.3.

Once the MSDG has been built, one proceeds by identifying the vertices containing primitive communication calls in their statements. We call this operation the *labelling* phase, and as pointed out earlier, the operation is parametric on both the communication primitives as well as on the calling mode.

The result of this phase is another graph structure retaining only coordination relevant data with respect to the set of rules specifying the communication primitives to look for. This structure is called the *Coordination Dependence Graph* (to be abbreviated to CDG in the sequel). As explained, it is computed from the MSDG in a two stage process, presented in section 8.4. First nodes matching rules encoding the use of specific interaction or control primitives are suitably labelled. Then, by backwards slicing, the MSDG is pruned of all sub-graphs found irrelevant for the reconstruction of the program coordination layer. Note the first stage is parametric on the type of interaction mechanisms used in the program under analysis. Section 8.4 details the CDG construction.

Once the CDG has been generated, it is used to guide the generation of a model of the system's coordination logic. This can take the form of a formal specification in ORC, as discussed in section 8.5, or of a WS-BPEL script, as explained in section 8.6.

8.3 The Managed System Dependence Graph

The fundamental information structure underlying the coordination analysis methods proposed in the next chapter is a comprehensive dependence graph, to be referred in the sequel as the *Managed System Dependence Graph* (MSDG), which records all elementary entities and relationships that may be inferred from code by suitable program analysis techniques.

A MSDG is an extension of the concept of *System Dependence Graph* (SDG) to cope with object-oriented features, as considered in [LH96, LH98, Zha98a]. Our own contribution was the introduction of new representations for a number of program constructs not addressed before, namely, partial classes, partial methods, delegates, events and lambda expressions.

A MSDG is defined over three types of nodes, representing program entities: *spatial nodes* (subdivided into classes Cls, interfaces Intf and name spaces Nsp), *method nodes* (carrying information on method's signature MSig, statements MSta and parameters MPar) and *structural* nodes which represent implicit control structures (for example, recursive references in a class or a fork of execution threads). Formally,

$$\begin{aligned} \mathsf{Node} &= \mathsf{SNode} + \mathsf{MNode} + \mathsf{TNode} \\ \mathsf{SNode} &= \mathsf{Cls} + \mathsf{PtCls} + \mathsf{Intf} + \mathsf{Nsp} \\ \mathsf{MNode} &= \mathsf{MSig} + \mathsf{MSta} + \mathsf{MPar} \\ \mathsf{TNode} &= \{\triangle, \triangledown, \circ\} \end{aligned}$$

where + denotes set disjoint union. Nodes of type SNode contain just an identifier for the associated program entity. Other nodes, however, exhibit further structure. For example, a MSta node includes the statement code (or a pointer to it) and a label to discriminate among the possible types of statements in a method, i.e.,

$$\begin{aligned} \mathsf{MSta} &= \mathsf{SType} \times \mathsf{SCode} \\ \mathsf{Stype} &= \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}, \mathsf{assgn}, \cdots\} \end{aligned}$$

where, for instance, mcall stands for any statement containing a call to a method and cond for a conditional expression. Similarly, a MSig node, which in the graph acts as the method entry point node, records information on both the method identifier and its signature, i.e., $\mathsf{MSig} = \mathsf{Id} \times \mathsf{Sig}$.

Several kinds of program dependencies can be represented as edges in a MSDG. Formally, an edge is a tuple of type

$$\mathsf{Edge} = \mathsf{Node} \times \mathsf{DepType} \times (\mathsf{Inf} + \mathbf{1}) \times \mathsf{Node}$$

where $\mathsf{DepType}$ is the relationship type and the third component represents, optionally, additional information which may be associated to it. The main kinds of dependency relationships as follows:

The first and most basic type of dependency that a MSDG captures is control flow dependency, formally referred by $\underline{\mathsf{cf}}$. This kind of dependency represents the possibility of the program execution control of flowing from one statement to another. In well structured programming languages this type of dependency can be straightforwardly computed solely based on the semantics of the control flow language primitives (`WHILE`, `IF THEN ELSE`, `FOR`, etc). Moreover, it is based on these control flow dependencies that the other MSDG dependencies are computed upon.

Data dependencies, of type $\underline{\mathsf{dd}}$, connect statement nodes which refer to common variables in particular conditions. Formally,

$$\langle v, \underline{\mathsf{dd}}, x, v' \rangle \in \mathsf{Edge} \Leftrightarrow \ \mathsf{definedIn}(x, v) \wedge \mathsf{usedIn}(x, v')$$

where x is a statement variable and notation $\mathsf{definedIn}(x, v)$ (respectively, $\mathsf{usedIn}(x, v)$) stands for x is defined (respectively, used) in node v (respectively, v'). A variable is considered to be *defined* in a statement if its value is potentially modified in that statement. On the other hand, a variable is considered to be *used* in a statement if its value may influence the result of the statement in question.

The third type of dependencies captured are control dependencies, of type $\underline{\mathsf{ct}}$. These serve to connect control statements (e.g., loops or conditionals) or method calls to their possible continuations. Control dependencies are also used to capture dependencies between method signature nodes (which represent the entry-points on a method invocation) and each of the statement nodes, within the method, which are not under the control of another control statement. Formally, these conditions add the following assertions to the invariant of type Edge:

$$\langle v, \underline{\mathsf{ct}}, g, v' \rangle \in \mathsf{Edge} \Leftarrow \ v \in \{\mathsf{MSta}(t, -) |\ t \in \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}\}\} \wedge v' \in \mathsf{MSta}$$
$$\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} \Leftarrow \ v \in \mathsf{MSig} \wedge v' \in \mathsf{MSta}$$

where g is either undefined, in which case v represents a statement not containing a predicate (a method for instance), or the result of evaluating the statement guard contained in the control node.

8.3.1 Method Invocation

Method parameters are handled through special nodes, of type MPar, representing input (respectively, output) actual and formal parameters in a method call or declaration. The special nodes introduce auxiliary variables that mediate parameter exchange between a method call and the actual method implementation. Formally,

$$\mathsf{MPar} \;=\; \mathsf{PaIn} + \mathsf{PaOut} + \mathsf{PfIn} + \mathsf{PfOut}$$

A method call, on the other hand, is represented by a $\underline{\mathsf{mc}}$ dependence from the calling statement with respect to the method signature node. Formally,

$$\langle v, \underline{\mathsf{mc}}, vis, v' \rangle \in \mathsf{Edge} \Leftrightarrow\; v \in \mathsf{MSta} \wedge \mathsf{SType}\, v = \mathsf{mcall} \wedge v' \in \mathsf{MSig}$$

where vis stands for a visibility modifier in set $\{\mathsf{private}, \mathsf{public}, \mathsf{protected}, \mathsf{internal}\}$.

Specific dependencies are also established between nodes representing formal and actual parameters. Moreover, all of the former are connected to the corresponding method signature node, whereas actual parameter nodes are connected to the method call node via control edges.

In an object oriented framework, one has to cater for the possibility that a method, modifying not only its parameter variables, but also some class or instance variables. To deal with such situations, we follow the approach taken in [LH98], where formal vertices are introduced for each of the class or instance variables that are modified within the method, and, from the calling function side, the corresponding actual vertices are inserted. With respect to edges, a method call is represented by *method call* edges between vertices containing method calls and the method entry vertex of the called method, *parameter-in* edges between *actual-in* and *formal-in* vertices, *parameter-out* edges between *formal-out* and *actual-out* vertices. All formal vertices are connected to the method entry vertex and all actual vertices are connected to the calling vertex via control edges.

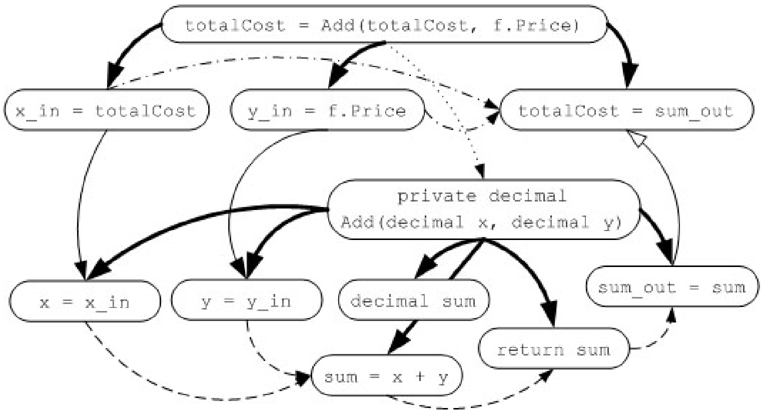


Figure 8.2: Method dependence graph

Whenever the called method introduces a direct or indirect data dependency between its formal-in and formal-out vertices, such dependencies are reflected in the calling side by introducing *transitive dependence edges* between the actual-in and actual-out vertices.

As an example of a method call consider the graph presented in Figure 8.2 representing a call to a function (`Add`) that receives two integers and returns its sum. In this graph, solid thick arrows represent control edges, solid thin arrows with filled ends define parameter-in edges, solid thin arrows with empty ends define parameter-out edges, dashed arrows represent data edges, dashed and pointed arrows define transitive edges and pointed arrows represent method call edges. Note that the two transitive dependencies, between the actual parameter vertices, bypass the need of inspecting the subgraph corresponding to function `Add` in order to access the data dependencies between the actual parameter vertices.

Summing up, these add the following assertions to the MSDG invariant:

$$\begin{aligned}
\langle v, \underline{\mathsf{pi}}, -, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\; v \in \mathsf{PaIn} \wedge v' \in \mathsf{PfIn} \\
\langle v, \underline{\mathsf{po}}, -, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\; v \in \mathsf{PaOut} \wedge v' \in \mathsf{PfOut} \\
\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\; v \in \mathsf{MSig} \wedge v' \in (\mathsf{PaIn} \cup \mathsf{PaOut}) \\
\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\; v \in \mathsf{MSta} \wedge \mathsf{SType}\, v = \mathsf{mcall} \wedge v' \in (\mathsf{PfIn} \cup \mathsf{PfOut}) \\
\langle v, \underline{\mathsf{dd}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\; v \in \mathsf{PaIn} \wedge v' \in \mathsf{PaOut} \wedge \\
&\qquad \exists_{\langle u, \underline{\mathsf{dd}}, -, u' \rangle} \,.\, (u \in \mathsf{PfIn} \wedge u' \in \mathsf{PfOut})
\end{aligned}$$

8.3.2 Properties

Properties are a special program construct found in $C^\sharp$ and other .Net languages that encapsulates the access to class variables. Given the semantic similarity to Java's *get* and *set* methods, in a MSDG properties are represented as normal *get* and *set* methods, where the methods formal-in and out vertices are inferred from the property type. A property invocation from a calling class is represented as a (already described) method call.

8.3.3 Objects and Polymorphism

As in [LH98], we represent references to objects individually i.e., each reference to an object in a statement is represented by a tree depicting all the object variables. A difference in our representation of objects from the approach taken in [LH98] concerns the representation of recursively defined classes. Instead of using a k-limiting solution (which proceeds by expanding the object tree to a level k) we use the special vertex, depicted as $\circ$, to represent recursive references in classes.

For dynamically typed references to objects, we build the object trees for every possible object type the reference may hold. Each of these trees root vertices are then connected to the corresponding object reference vertex by control edges.

8.3.4 Partial Classes and Partial Methods

Partial classes, available for instance in $C^\sharp$ 2.0, enable a class to be defined in two different partial classes (each possibly defined in a separate file) that are combined in compile time to generate a single class. The representation of such classes in a MSDG is trivially solved by using a partial class node type, formally PtCls, and a partial class dependency edge, formally $\underline{\mathsf{pd}}$, that connects a vertex representing the class and the two partial class vertices.

The graph representation of partial methods is not as straightforward as the partial classes' case, since its semantics is not the mere sequential composition, in a single method, of both partial methods definitions. In fact, partial methods work in a way similar to event subscriptions were, if a partial method has its declaration in a partial class and an implementation in the other, then everything works as if it was a normal method *i.e.*, the method is executed in all its calls. On the other hand, if a partial class declares a partial method and there is no implementation of it in any of the partial classes, then every call to the method is removed at compile time.

In order to represent this behaviour we introduce a method call dependency edge between the declaration of the partial method and its implementation. Furthermore, every node containing a call to the partial method is connected to the partial method signature and not the implementation. This assures that every call to a partial method must first pass by its declaration which assures the correctness of slicing over this kind of program constructs. Just like the compile time behaviour which removes every call to the partial method if this has no implementation available, every node containing a call to a partial method without implementation is also removed from the MSDG.

8.3.5 Delegates, Events and Lambda Expressions

Delegates, events and lambda expressions are programming constructs not available in Java nor in C++, so previous object-oriented graph representations [LH98, WR03, KMG, Zha98a] do not cover such entities.

Delegates can be seen as the $C^\sharp$ (and other .Net languages) type-safe version of C and C++ function pointers, allowing the definition of higher-order functions. A delegate defines a type of a function whose values are treated like objects, thus possibly defining class member types as well as being able to be exchanged between methods.

In what concerns to the graph representation of delegates, from the subscriber side, *i.e.*, a class that instantiates an object delegate member with a method, the object tree that it refers to is updated with a control edge between the object delegate member and the sub-graph of the method being passed. From the subscribed side, *i.e.*, the class with the delegate (type) definition that invokes the subscribed method, one adds a method vertex representing the delegate type as well as formal-in and out vertices for the arguments and return values of the delegate. Moreover, we create actual-in and out vertices connected to the method vertex which is connected to the formal-in and out vertices of the actual subscriber function. Every call to the delegate inside the subscribed class is represented by a method call edge to the method vertex introduced by the delegate type.

This way, the method vertex introduced by the delegate type definition acts like a proxy, dispatching its calls to the objects and corresponding methods that subscribed the delegate. This approach takes into consideration particular objects and, thus, it permits slicing to take place over the subscribing methods. As an example of the representation of delegates, see for instance vertex `S36` in the MSDG presented in appendix D, whose code is available in appendix C.

In what concerns its graph representation, the difference between delegates and events is that events can be subscribed by more than one method, while delegate subscriptions override each other, thus making only the last subscription to count. Given this similarity, one applies the same approach taken for delegates in the representation of events, with the detail that, in the event representation, one may have more than one method call edge between the proxy method of the subscribed and the actual method to be called in the subscriber.

Unlike pure functional lambda expressions, $C^\sharp$ lambda expressions have a state and can perform like any other method. In their graph representation, the only difference between lambda expressions and delegates is that lambda expressions, and anonymous delegates as well, do not have an identifying name. Thus, in a MSDG, lambda expressions are represented as normal delegates with the exception that their entry vertex is labelled with an automatically generated identifier.

8.3.6 Concurrency

Structural nodes TNode are introduced to cope with concurrency (case of $\bigtriangleup$ and $\bigtriangledown$) and to represent recursively defined classes (case of $\circ$).

A $\bigtriangleup$ - node captures the effect of a spawning thread: it links an incoming control flow edge, from the vertex that fired the fork, and two outgoing edges, one for the new execution flow and another for the initial one. Dually, a thread join is represented by a $\bigtriangledown$ - node with two incoming edges and an outgoing one to the singular resumed thread.

To deal with dependencies between different thread program statements, we follow the ideas in [Kri03, NR00] using interference dependence edges, formally referred by $\underline{\mathsf{id}}$.

To illustrate this strategy for representing concurrency in the graph, consider the $C^\sharp$ program in Figure 8.3. The program spawns a new thread to calculate a sum, sends a message to the user reporting the sum calculation, waits for the sum thread to finishes and presents the result to the user. Figure 8.4 represents the MSDG calculated for the concurrent code, with triangular vertices representing thread spawning and join. Note that the spawning thread triangular vertex has an outgoing edge to an object vertex. Such connections between spawning thread vertices and object vertices serve to specify the exchange of objects and variables references between threads.

8.3.7 Class and Interface Dependence

Class inheritance and the fact that a class owns a particular method is recorded as follows

$$\langle v, \underline{\mathsf{cl}}, vis, v' \rangle \in \mathsf{Edge} \Leftrightarrow \; v \in \mathsf{Cls} \wedge v' \in \mathsf{MSig}$$

```
namespace ConsoleMultithreadTest {
    class Program {
        public static void Main(string[] args) {
            Sum sum = new Sum(1, 2);
            ThreadStart ts = new ThreadStart(sum.Add);
            Thread t = new Thread(ts);
            t.Start();
            Console.WriteLine("Calculating result...");
            t.Join();
            Console.WriteLine("Result = {0}", sum.result);
        }
    }

    class Sum {
        public int x, y, result;

        public Sum(int a, int b) {
            this.x = a; this.y = b;
        }

        public void Add() {
            Thread.Sleep(3000);
            result = x + y;
        }
    }
}
```

Figure 8.3: Fragment of a concurrent program

A similar strategy is adopted for interface and namespace nodes.

8.4 The Coordination Dependence Graph

As discussed above, the purpose of a MSDG is to collect structure and represent a large amount of program data upon which more fine-grained analysis can be carried on. This is achieved by extracting from a MSDG, sub-graphs specifically focused on the collection of entities relevant to the particular sort of analysis one is interested in. The Coordination Dependence Graph (abbreviated to CDG in the sequel) is one of such structures, specifically oriented towards the recovery of coordination data. Its construction amounts basically to a selective pruning of the MSDG, to remove all the information not

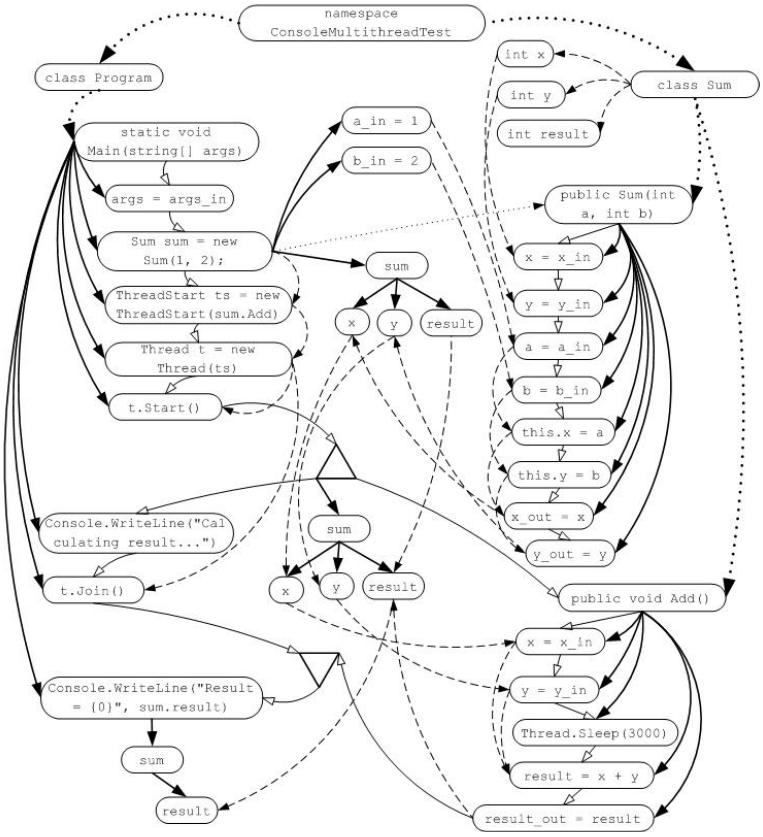


Figure 8.4: MSDG for code fragment in Figure 8.3

directly relevant for a particular set of constraints which specify the coordination relevant entities considered non relevant for coordination analysis.

This filtering is guided by the specification of a set of constraints defining all the coordination relevant primitives and entities to be looked for. By tuning such constraints appropriately, different kinds of coordination information can be captured. This gives to the analyst an additional freedom degree to focus analysis in a particular type of implementation of coordination policies (e.g. web-services calls) or in a specific set of programming language constructs known to be used, in the system under analysis, to implemented such policies.

These constraints are given by rules, specified as follows:

$$\begin{aligned}
\mathsf{CRule} &= \mathsf{RExp} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole}) \\
\mathsf{CType} &= \{\underline{\mathsf{webservice}}, \underline{\mathsf{rmi}}, \underline{\mathsf{remoting}}, \cdots\} \\
\mathsf{CDisc} &= \{\underline{\mathsf{sync}}, \underline{\mathsf{async}}\} \\
\mathsf{CRole} &= \{\underline{\mathsf{provider}}, \underline{\mathsf{consumer}}\}
\end{aligned}$$

where RExp is a regular expression [Stu07] used to retrieve a particular code fragment, CType is the type of communication primitive types (extensible to other classes of communication primitives), CDisc is the calling mode (either synchronous or asynchronous) and, finally, CRole characterises the code fragment role with respect to the direction of communication. In $C^\sharp$, for example, the identification of invocations of web-services can be captured by the following rule, which identifies the primitive synchronous framework method `SoapHttpClientProtocol.Invoke`, typically used to perform remote invocations of web-services:

$$\begin{aligned}
Regex &= \text{``}System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke\backslash(\backslash w\backslash);\text{''} \\
R &= (Regex, (\underline{\mathsf{webservice}}, \underline{\mathsf{sync}}, \underline{\mathsf{consumer}}))
\end{aligned}$$

In this particular example, our rule R identifies calls to web-services that do not take any arguments[1]. This behaviour is captured by the used regex which only matches single argument calls to function `Invoke`, whose API defines that the first argument corresponds to the web-service method to be called and the remaining, to the arguments to be passed to the remote method. Furthermore, the rule defines that the tokens matched by the regex are to be considered as *synchronous* calls to web-services from a *consumer* perspective.

However, note that, apart from this specific coordination oriented set of rules, the CDG can be used to highlight other programmatic aspects other

[1] A rule for identifying calls to web-services with a variable number of arguments is presented in chapter 11

than coordination related ones.

Given a set of rules, the CDG calculation, starts by testing all the MSDG vertices against the regular expressions in the rules. If a node of type MSta or MSig matches one of these regular expressions, it is labelled with the information in the rule's second component. The types of the resulting, labelled nodes are, therefore,

$$\begin{aligned}\mathsf{CMSta} &= \mathsf{MSta} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole}) \\ \mathsf{CMSig} &= \mathsf{MSig} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole})\end{aligned}$$

Corresponding to the annotation of statement and signature nodes selected by the specified constraints. Note that, because of this labelling process, the type of a CDG node becomes the following union

$$\mathsf{CNode} = \mathsf{Node} + \mathsf{CMSta} + \mathsf{CMSig}$$

Once completed this labelling stage, the method proceeds by abstracting away the parts of the graph which do not take part in the coordination layer. This is a major abstraction process accomplished by removing all non-labelled nodes, but for the ones verifying the following conditions:

1. method call nodes (i.e., nodes v such that $v \in \mathsf{MSta}$ with $\mathsf{SType}\, v = \mathsf{mcall}$) for which there is a control flow path (i.e., a chain of $\underline{\mathsf{cf}}$ dependence edges) to a labelled node.

2. vertices in the union of the backward slice of the program with respect to each one of the labelled nodes.

The first condition ensures that the relevant procedure call nesting structure is kept. This information will be useful to nest, in a similar way, the generated code on completion of the discovery process. The second condition keeps all the statements in the program that may potentially affect a previously labelled node. This includes, namely, MSta nodes whose statements contain predicates (e,g., loops or conditionals) which may affect the parameters for execution of the communication primitives and, therefore, play a role in the coordination layer.

This stage requires a slicing procedure over the MSDG, for which we adopt a backward slicing algorithm similar to the one presented in [HRB88]. It consists of two phases:

- The first phase marks the visited nodes by traversing the MSDG backwards, starting on the node matching the slicing criterion, and following ct, mc, pi, and dd labelled edges. The second phase consists of traversing the whole graph backwards, starting on every node marked on phase 1 and following ct, po, and dd labelled edges.
- By the end of phase 2, the program represented by the set of all marked nodes constitutes the slice with respect to the initial slicing criterion.

Except for cf labelled edges, every other edge from the original MSDG with a removed node as source or target, is also removed from the final graph. The same is done for any cf labelled edge containing a pruned node as a source or a sink. On the other hand, new ct edges are introduced to represent what were chains of such dependencies in the original MSDG, i.e. before the removal operation. This ensures that future traversals of this graph are performed with the correct control order of statements. As an example of a CDG calculation from an MSDG instance, considered the graph presented in appendix D, where the CDG corresponds to the graph obtained by the removal of the gray nodes.

The construction of the CDG follows, actually, a quite generic algorithm which prunes a MSDG according to a specific set of constraints which are validated by pattern matching against code information collected in MSDG nodes. This means, it can be easily adapted to extract, not only different sorts of coordination data, as mentioned above, but also other kind of program data for different analysis purposes.

8.5 Generation of Orc Specifications

Although the CDG already provides important coordination information about the system under consideration, the analysis will benefit if such information is presented in the form of a precise specification for the underlying coordination layer. Such is the purpose of this stage. In this section we introduce a technique for the generation of an Orc specification based on a previously constructed CDG. This Orc specification abstracts the entire coordination behaviour of the system in a rigorous specification.

We believe that a coordination specification, following closely the structure of the original system is more understandable and, moreover, easier to confront with the original system. Therefore, in order to keep the original system's procedure calls nesting structure, one generates an Orc definition for each procedure in the CDG and keeps the calls in the graph between these procedures. It is this structure preservation that justifies the first exception in the MSDG pruning phase mentioned in the previous section.

Note that we do not generate an Orc definition for every procedure in the system, since during the construction of the CDG many procedures (more specifically, the ones not contributing to the coordination layer) were dropped. Also notice that, it is quite simple to transform the nested Orc specification into a flat one, whenever this simplifies reasoning about the coordination specification at later stages.

The Orc generation process for a procedure is based on the program captured by the procedure sub-graph of the entire system CDG. The construction of the program represented by a CDG is quite straightforward and basically amounts to collecting the statements of the visited vertices by following the control flow edges.

To explain the process of generating Orc specifications from a CDG, it is probably easier to assure a particular, concrete language in which we suppose, for the sake of illustration, the CDG nodes be annotated by. Such a language is a subset[2] of $C^\sharp$ presented in Figure 8.5. The representation

[2] Actually, we address all the relevant control flow, concurrency, and communication primitives of the language.

of CDG instances in this language is a straightforward process, since of the constructs defined by the language are common to most popular language and the ones less so, like LOCALCALL and ASYNCCALL, are easily extracted from the vertices labelling information of the CDG.

$$
\begin{array}{rclcl}
z & \in & \mathit{Values} & & \\
x, x_1, x_n & \in & \mathit{Variables} & & \\
s & \in & \mathit{Sites} & & \\
e, e_1, 2_n & \in & \mathit{Expressions} & & \\
st, st_1, st_2 & \in & \mathit{Statements} & ::= & z \\
 & & & \mid & x \\
 & & & \mid & x \;\mathtt{=}\; e \\
 & & & \mid & st_1 \;\;\mathtt{;}\;\; st_2 \\
 & & & \mid & \mathtt{LOCK}\ (x)\ \{st\} \\
 & & & \mid & \mathtt{LOCALCALL}\ f\,\mathtt{(}\overline{x}\mathtt{)} \\
 & & & \mid & \mathtt{SYNCCALL}\ s\ f\,\mathtt{(}\overline{x}\mathtt{)} \\
 & & & \mid & \mathtt{ASYNCCALL}\ s\ f\,\mathtt{(}\overline{x}\mathtt{)}\ \ \prec \{st\} \succ \\
 & & & \mid & \mathtt{IF}\ p\ \mathtt{THEN}\ \{st_1\}\ \ \prec \mathtt{ELSE}\ \{st_2\} \succ \\
 & & & \mid & \mathtt{WHILE}\ p\ \mathtt{DO}\ \{st\} \\
f_1, f_n & \in & \mathit{Procedures} & ::= & f\,\mathtt{(}\overline{x}\mathtt{)}\{st\} \\
c_1, c_n & \in & \mathit{Classes} & ::= & c\ \{x_1 = e_1\ ...\ x_n = e_n\ f_1\ ...\ f_n\} \\
ns_1, ns_n & \in & \mathit{Namespaces} & ::= & ns\ \{c_1\ ...\ c_n\}
\end{array}
$$

Figure 8.5: Modified $C^\sharp$ language subset

The language is quite self explanatory and contains most of the traditional control flow constructs, thus making it relatively easy to represent programs captured by CDG instances. Nevertheless, there is a number of specific constructs which demand some further explanation.

We consider that a local procedure call is a synchronous call to a resource in the same machine not involving any communication primitive. On the other hand, every asynchronous procedure call must be performed as if being made to an external resource, in which case it must specify the resource site uniquely (internal asynchronous procedure calls may be performed using the `ASYNCCALL` construct with `localhost` as resource site). The $\prec\ \succ$ brackets used in the language definition stand for optional expressions.

As it happens in the complete $C^\sharp$ language, this subset also provides two possibilities for performing asynchronous calls. One simply launches the procedure call in a separate thread and continues execution of the rest of the program. The other executes an expression when and if the asynchronous call returns. The `LOCK` statement behaves as expected, i.e., it gives a variable access to a specific statement block execution in a single thread or process. All the remaining details concerning the syntax and the semantics of the language are borrowed from the complete $C^\sharp$ language.

It may seem strange the explicit distinction made between synchronous and asynchronous procedures calls as well as between external and internal procedure calls at the language syntax level. Note, however, that these distinctions are previously made during the construction of the CDG and are, therefore, available at this stage of the whole process. Recall that our focus in this language is only motivated by exposition purposes: in practice, one is not limited to languages containing these particular constructs in what concerns to synchronism and communication. Moreover, the generation process is entirely based on the CDG, thus removing the need to actually represent the program in a particular language. The CoordInspector tool, to be presented in chapter 10, follows this approach and bases the Orc code generation entirely on the CDG.

Assume, thus, that input of the Orc generation process is a program, represented in the language of Figure 8.5, obtained by a direct representation of the statements captured in the calculated CDG. The formalisation of this process, that follows, considers only the statements of the language. Nevertheless, the overall algorithm (as implemented in CoordInspector) recursively analyses each invoked local function.

The Orc generation is composed of two distinct phases. The first one is performed by function ψ which identifies all the variables in the language for which an access control may be required, and sets up an environment for controlling the access to such variables. The reason for having this kind of pre-processing stage is because, like in many specification languages, Orc captures coordination in a declarative way *i.e.*, without resorting to any notion of state. Thus, one simulates such a state by introducing a series of declarations of auxiliary variables, capturing pertinent initial state information and providing a basis for the diverse coordination definitions that will compose the specification.

Function ψ, presented in Figure 8.6, is responsible for the construction of our simulated state, where it basically introduces a *Lock* site for each variable in a `LOCK` statement, while keeping track of all visited variables to avoid site duplication.

The second phase of the generation process is performed by function φ, depicted in Figure 8.7, which, for every procedure body generates the corresponding Orc definition. Note that function φ assumes the previous existence of a previously created environment of sites, more specifically, an environment with a *Lock* controlling the access for each critical variable.

A brief explanation is in order. Function φ converts a value or a variable from the language to the correspondent variable or constant in Orc. A synchronous procedure invocation is also directly transformed to a site call in Orc.

The asynchronous procedure call case is not as straightforward as the previous cases. Here, one must specify in Orc the behaviour of performing a request to a site without blocking for an answer and leaving the rest of the specification to carry on executing. This behaviour can be captured in Orc by means of the *Discr* pattern and the fundamental site *Signal*, both presented in appendix B. The *Discr* pattern executes both arguments in parallel and waits for a signal from any of the sites. Since *Signal* publishes a signal immediately, the behaviour of the *Discr* with a *Signal* argument is to return immediately leaving the other argument still running. This, correctly captures the intend behaviour of an asynchronous procedure call.

Given the blocking behaviour of the fundamental site if when faced with a $false$ value, one cannot perform a direct translation of the `IF THEN` statement to the if Orc fundamental site. Such a direct translation would make the entire specification block upon a $false$ value over an if site. Thus one uses

$$\psi\ (\texttt{LOCK (}x\texttt{)}\ \{st\},\ V) \equiv$$

$$\begin{cases} (\iota_2(\mathit{Lock} > x\mathit{Lock} > \mathit{Signal}), \{x\} \cup V) & \text{if } x \notin V, \\ (\iota_1(), V) & \text{otherwise} \end{cases}$$

$$\psi\ (\texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)}\ \{st\},\ V) \equiv\ (\psi_1\ st,\ \psi_2\ st \cup V)$$

$$\psi\ (\texttt{IF}\ p\ \texttt{THEN}\ \{st\},\ V) \equiv\ (\psi_1\ st,\ \psi_2\ st \cup V)$$

$$\psi\ (\texttt{IF}\ p\ \texttt{THEN}\ \{st_1\}\ \texttt{ELSE}\ \{st_2\},\ V) \equiv$$

$$\begin{cases} (\psi_1\ st_1,\ \psi_2\ st_1 \cup V) & \text{if } \psi_1\ st_1 \neq \iota_1() \wedge \psi_1\ st_2 = \iota_1(), \\ (\psi_1\ st_2,\ \psi_2\ st_2 \cup V) & \text{if } \psi_1\ st_1 = \iota_1() \wedge \psi_1\ st_2 \neq \iota_1(), \\ (\iota_2(\psi'_1\ st_1 \gg \psi'_1\ st_2), & \\ \ \psi_1\ st_1 \cup \psi_1\ st_2 \cup V) & \text{if } \psi_1\ st_1 \neq \iota_1() \wedge \psi_1\ st_2 \neq \iota_1(), \\ (\iota_1(), V) & \text{otherwise} \end{cases}$$

$$\psi\ (\texttt{WHILE}\ p\ \texttt{DO}\ \{st\},\ V) \equiv\ (\psi_1\ st,\ \psi_2\ st \cup V)$$

$$\psi\ (st1\ \ \texttt{;}\ \ st_2,\ V) \equiv\ ((\iota_2(\psi'_1\ st_1 \gg \psi'_1\ st_2),\ \psi_1\ st_1 \cup \psi_1\ st_2 \cup V)$$

$$\psi\ (st,\ V) \equiv\ (\iota_1(),\ V)$$

where

$$\begin{aligned} \psi_1 &= \pi_1\ .\ \psi \\ \psi_2 &= \pi_2\ .\ \psi \\ \rho_1(\iota_2\ x) &= x \\ \psi'_1 &= \rho_2\ .\ \pi_2\ .\ \psi \end{aligned}$$

Figure 8.6: Function ψ

$$
\begin{array}{lcl}
\varphi\ \texttt{z} & \equiv & \mathit{let}(z) \\
\varphi\ \texttt{x} & \equiv & x \\
\varphi\ \texttt{x = } e & \equiv & \mathit{let}(e) > x > \mathit{Signal} \\
\varphi\ \texttt{x = } e\ \texttt{;}\ st_2 & \equiv & \mathit{let}(e) > x > \varphi(st) \\
\varphi\ \texttt{LOCK (}x\texttt{)}\ \{st\} & \equiv & \mathit{xLock.acquire} \gg \\
 & & \quad \varphi(st) \gg \mathit{xLock.release} \\
\varphi\ \texttt{LOCALCALL}\ f\texttt{(}\overline{x}\texttt{)} & \equiv & F(\overline{x}) \\
\varphi\ \texttt{SYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)} & \equiv & s.F(\overline{x}) \\
\varphi\ \texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)} & \equiv & \mathit{Discr}(s.F(\overline{x}), \mathit{Signal}) \\
\varphi\ \texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)}\ \{st\} & \equiv & \mathit{Discr}(s.F(\overline{x}) > \mathit{result} > \varphi(st), \\
 & & \qquad \mathit{Signal}) \\
\varphi\ \texttt{IF}\ p\ \texttt{THEN}\ \{st\} & \equiv & \mathit{IfSignal}(\mathit{let}(p), \varphi(st)) \\
\varphi\ \texttt{IF}\ p\ \texttt{THEN}\ \{st_1\}\ \texttt{ELSE}\ \{st_2\} & \equiv & \mathit{XOR}(\mathit{let}(p), \varphi(st_1), \varphi(st_2)) \\
\varphi\ \texttt{WHILE}\ p\ \texttt{DO}\ \{st\} & \equiv & \mathit{Loop}(\mathit{let}(p), \varphi(st)) \\
\varphi\ st1\ \texttt{;}\ st_2 & \equiv & \varphi(st_1) \gg \varphi(st_2)
\end{array}
$$

Figure 8.7: Function φ

the $IfSignal$ pattern which never blocks and executes the second expression in the case where the predicate evaluates to *true*.

The behaviour specification of the `IF THEN ELSE` statement is easier to capture because one of the branches of the statement is always executed. Therefore, a direct translation to the XOR pattern captures the intended behaviour. Similarly, the `WHILE DO` statement is captured by the *Loop* coordination pattern which does not block upon evaluation of false predicates.

Given functions ψ and φ, specifying the two main phases of the Orc generation process, the overall generation algorithm is obtained as follows:

$$
\beta\ (f\texttt{(}\overline{x}\texttt{)}\ \{st\}) = \begin{cases} F(\overline{x}) \triangleq \psi_1'(L, \emptyset) \gg \varphi(L) & \text{if } \psi_1(L, \emptyset) \neq \iota_1() \\ F(\overline{x}) \triangleq \varphi(L) & \text{otherwise} \end{cases}
$$

8.5.1 Example

To illustrate the generation method just introduced, consider the development of a hypothetical client application, intended to be part of a meteorological forecast network. We will manually translate the application requirements into an Orc specification and, afterwards, provide a possible $C^\sharp$ implementation. Then, we submit the code to the analysis discussed in this chapter, generating the corresponding CDG and, from it, a new Orc specification. This small example illustrates not only the specification generation method, but also a use of our approach to verify coordination specifications against actual implementations.

Suppose that instances of this weather forecast application are to be installed in a number of geographically separated stations. Each station has at its disposal a set of sensors which provide some meteorological data relative to current weather conditions. The objective of the application is, among other functionality, to communicate the data read from its sensors to a central server whose purpose is to predict the weather forecast for the next 5 days.

Since the production of weather forecasts is a demanding computational operation, the central server will be, most of the time, devoted to internal activities and only sporadically will it interact with the client stations. Therefore, such communication is required to be asynchronous, in order to free the station application so that it may perform other tasks while not interacting with the server. Another requisite of the application is that since client stations are aware of current weather conditions, they must compare the generated forecast with the weather conditions they are experiencing at the moment and, if great discrepancies are found, ask the central server to check and correct its forecast.

Although this coordination scenario is not unfeasible to be implemented directly, it has still enough details to justify the previous development of a specification of the communication protocol. The following is a specification of such a system, written in Orc.

$$
\begin{aligned}
Station()) &\triangleq Server.CalculateForecast() > fid > \\
&\quad GetResult(fid) \\
GetResult(fid) &\triangleq GetWeatherConditions() > x > \\
&\quad Server.GetForecast(x) > fc > \\
&\quad XOR(let(fc == null) \\
&\qquad , \\
&\qquad RTimer(1000) \gg \\
&\qquad GetResult(fid) \\
&\qquad , \\
&\qquad VerifyResult(fc) \\
VerifyResult(res)) &\triangleq XOR(\neg\ ConfirmForecast(res) \\
&\qquad , \\
&\qquad Server.VerifyForecast(res) > vfcid > \\
&\qquad GetVerification(vfcid)) \\
&\qquad , \\
&\qquad let(res)) \\
GetVerification(vfid)) &\triangleq Server.GetVerifiedForecast(vfid) > vf > \\
&\quad XOR(vf == null \\
&\qquad , \\
&\qquad RTimer(1000) \gg \\
&\qquad GetVerification(vfid) \\
&\qquad , \\
&\qquad let(vf))
\end{aligned}
$$

Note that in this specification $Server$ is used as the central weather forecast server. Operation $GetWeatherConditions$ is an internal operation available in each station, to inspect the current values of its weather sensors. Since it defines an internal station activity, whose details have no coordination relevance, it is intentionally left undefined. Finally, $ConfirmForecast$ denotes another internal operation which compares the generated weather forecast with the current weather conditions.

The next step in the development of the station application is to implement the above specification in a programming language. Suppose this task is given to a programmer's team which produce the following *C*♯ code:

```
class Example {
   private void GetWeatherForecast() {
      Console.WriteLine("Calculating forecast.");
      WeatherServer cs = new WeatherServer();
      int taskId = RequestServerTask(cs);
      Result res = GetResult(cs, taskId);
      if(res != null)
         Console.WriteLine("Forecast: " + res.ToString());
      else
         Console.WriteLine("Operation failed");
   }

  private int RequestServerTask(WeatherServer cs) {
      Console.WriteLine("Requesting forecast.");
      Operation op = ...current weather conditions gathering code...
      int opId = cs.CalculateForecast(op);
      return opId;
  }

   private Result GetResult(WeatherServer cs, int opId) {
      Result res = null;

      while(res == null) {
         Console.WriteLine("Querying server for forecast.");
         res = cs.GetForecast(opId);
         Thread.Sleep(1000);
      }
      // Check if the result still needs further calculation
      if(!ConfirmForecast(res)) {
           Console.WriteLine("Querying server to confirm forecast.");
           Operation op2 = ...confirm forecast parameter construction...
           int op2Id = cs.VerifyForecast(op2);
           res = GetVerification(cs, op2Id);
      }
      return res;
   }

   private Result GetVerification(WeatherServer cs, int opId) {
      Console.WriteLine("Querying server for verification result.");
      Result res = cs.GetVerifiedForecast(opId);
      if(res == null) {
         Thread.Sleep(2000);
         return GetVerification(cs, opId);
      } else {
         return res;
      }
   }
}
```

This is the point where our method may come to scene: a new Orc specification can be extracted from the source code and compared with the original one. The confrontation of the original Orc specification with the one extracted from the actual implementation offer a number of conclusions on the conformance of the system to its specification.

Figure 8.8 shows the generated MSDG. The corresponding CDG, obtained through application of rules

$$\begin{gathered}
(\text{``}\texttt{CalculateForecast(*);}\text{''}, (WebService, Sync, Consumer)) \\
(\text{``}\texttt{GetForecast(*);}\text{''}, (WebService, Sync, Consumer)) \\
(\text{``}\texttt{VerifyForecast(*);}\text{''}, (WebService, Sync, Consumer)) \\
(\text{``}\texttt{GetVerifiedForecast(*);}\text{''}, (WebService, Sync, Consumer))
\end{gathered}$$

is represented by the same graph once all dashed vertices have been removed.

From this CDG a new Orc specification is derived resorting to the Orc generation strategy presented above. The result is as follows.

$$
\begin{array}{rcl}
GetWeatherForecast() & \triangleq & new\ WeatherServer() > cs > \\
 & & RequestServerTask(cs) > taskId > \\
 & & GetResult(cs, taskId) \\
\text{RequestServerTask(cs)} & \triangleq & GetWeatherConditions() > op > \\
 & & cs.GetForecast(op) > opId \\
 & & let(opId) \\
\text{GetResult(cs, opId)} & \triangleq & Null() > res > \\
 & & Loop(let(res == null), \\
 & & \quad\quad cs.GetForecast(opId) > res > \\
 & & \quad\quad RTimer(1000)) \gg \\
 & & IfSignal(let(\neg\ ConfirmForecast(res)) \\
 & & \quad\quad\quad\quad , \\
 & & \quad\quad\quad\quad cs.VerifyForecast(op2) > op2id > \\
 & & \quad\quad\quad\quad GetVerification(cs, op2id) > res > \\
 & & \quad\quad\quad\quad Signal) \gg \\
 & & let(res) \\
\text{GetVerification(cs, opId)} & \triangleq & cs.GetVerifiedForecast(opId) > res > \\
 & & XOR(let(res == null) \\
 & & \quad\quad , \\
 & & \quad\quad RTimer(2000) \gg \\
 & & \quad\quad GetVerification(cs, opId) \\
 & & \quad\quad , \\
 & & \quad\quad res)
\end{array}
$$

Apart some minor differences concerning a few internal names, it is easy to conclude that both specifications represent the same behaviour in what respects to the invocation of the foreign services (*CalculateForecast*, *GetForecast*, *VerifyForecast*, and *GetVerifiedForecast*). This conclusion, which is quite trivial for this example, may, in practice require a bit of ORC rewriting to eventually transform both designs into a canonical form, therefore showing (or refuting) their (observational) equivalence.

8.6 Business Processes Discovery

This section introduces an algorithm for representing in Web Services - Business Process Execution Language (WS-BPEL) [JE07] the information captured by the CDG generated from a given system. This provides an alternative to Orc as a way of expressing such specifications as recovered from legacy code. WS-BPEL is an endogenous coordination language upon which one can define execution processes that are able to orchestrate the invocation and provisioning of web services resources. The language has most of the typical process algebra constructs, namely parallel and sequence composition, execution of activities (in this case, services, though it is also possible to invoke local functions) and provisioning of new process definitions (in this case, services) which can be invoke by other processes. Unlike Orc, WS-BPEL is a commercially used language that is being implemented by many software vendors who deliver coordination solutions for orchestration of web-services. Given this practical and more "realistic" aspect of WS-BPEL, the language size is dramatically greater when compared to the simple and elegant Orc syntax. References [JE07, Mig05] provide detailed introductions to both the syntax and semantics of the language, to which the interested reader is referred to.

A significant difference between WS-BEPL and Orc is that the former has an embedded notion of state, where one can define variables and resources which can be later referred to by the process orchestration. Even more, the way Orc defines structural execution of activities, completely based on (local) sites and site responses (or its absence), completely diverges from WS-BPEL where one may use imperative-like control flow constructs. Thus, the generation of WS-BPEL orchestrations quite deviates from what was previously presented for Orc.

It should be stressed that this generation algorithm is again generic (*i.e.*, "language agnostic"). To make things concrete, however, and the exposition easier to follow we present the WS-BPEL generation algorithm over the same language already used for the Orc generation, and presented in Figure 8.5. Note, again, that this is not the language in which systems to be analysed by CoordInspector are to be expressed, but rather the language used to represent CDG instances in order to facilitate the reasoning and presentation of our orchestration discovery algorithm.

The generation of the abstract WS-BPEL orchestration is accomplished by functions Φ, Φ_h, Φ_b specified in Figure 8.9, Figure 8.10 and Figure 8.11 respectively. Note that the algorithm presentation follows a functional outset, which, we believe, facilitates both its presentation and understanding.

Therefore, it resorts to some Haskell [HPW92] constructs, namely the list representation syntax, the *map* function, which applies a given function to every element of a given list, the : function which appends an element to the head of a list and the $foldr$ function which encapsulates structural recursion over lists. Furthermore, we denote the first, second and third tuple projections by functions π_1, π_2 and π_3 respectively.

To avoid declaring every string concatenation used to generate the WS-BPEL XML code, we represent constant strings values in courier font. This way, whenever there is a functional expression followed or preceded by a string constant in courier font, it should be interpreted as the concatenation of the value represented by the functional expression with the string constant. We also denote the empty string by $\perp$ and string concatenation by $+$.

Function Φ receives as input a *class* and returns the WS-BPEL orchestration capturing all service coordination contained in all class entities. This function depends upon four other auxiliary functions:

- Ψ which is responsible for converting a list of variables to their equivalent WS-BPEL forms.
- Υ which generates the WS-BPEL links declarations, to be used in the orchestration definition.
- Φ_h (shown in Figure 8.10) which derives WS-BPEL code specifying the provided services. In particular, function Φ_h receives a list of functions and, for each function with attribute `CM`, it computes a pair containing a

list of the variables found (which are converted to WS-BPEL by Φ using function Ψ) and a BPEL activity specifying the provision of a service that was performed by some specific logic in the original system.

- Φ_b (shown in Figure 8.11), which is responsible for calculating the WS-BPEL logic defined inside each function body. More specifically, function Φ_b receives a function body and returns a tuple containing a list of variables to be initialised, a list of links to be initialised and the functions body business logic translated to WS-BPEL.

Note that the generated WS-BPEL is in an abstract form as a consequence of using a number of `##opaque` attribute values, which are employed whenever there is insufficient information in the source code being evaluated. Such is the case, for instance, of the web-services url addresses that are not present in the source code and are required for some WS-BPEL constructs.

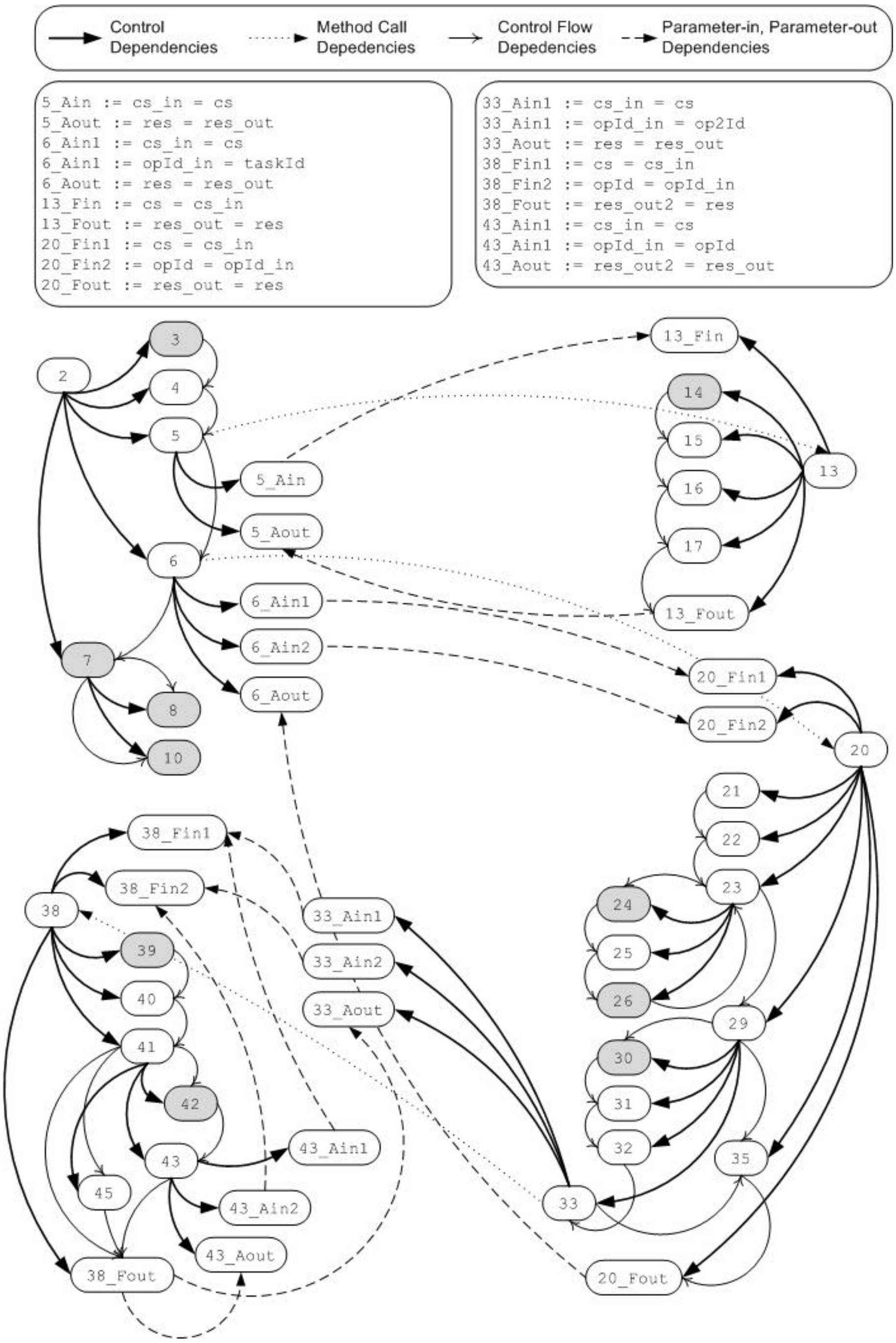
8.6.1 The Example

For a brief example of recovering coordination specifications in WS-BPEL, consider the $C^\sharp$ code from Figure 8.12, implementing a company's time sheet submission business process. The program provides a method (`SubmitTimesheet`) bound to a web-service that is responsible for receiving consultants' time sheets.

Once a time sheet arrives, its total cost is computed by the foreign web-service `GetTimesheetWithCost` according to the time sheet's consultant fees. If the total cost retrieved by `GetTimesheetWithCost` is above 2000 then the business proceeds by asynchronously invoking the `AnalyzeSheet` web-service with callback function `OnAnalyseResponse`. On completion of function `AnalyzeSheet`, the business process proceeds by evaluating function `OnAnalyseResponse` which, based on the time sheet cost approval, communicates the response to client and consultant in case of a positive approval, or requests the resubmission of the time sheet to the consultant in case of a negative response.

If the total cost of the time sheet is bellow or equal to 2000, the business process communicates the cost both to consultant and client through invocation of the web-services `NotiffyApprovedExpense` and `Communicate-ClientExpense`.

The result of applying the method introduced in this chapter to the program of Figure 8.12, is the WS-BPEL orchestration depicted in appendix E. Note that the generated WS-BPEL instance can easily be converted to an executable orchestration, by introducing some web-services url's and correct references to the local machine resources implementing the local functions being used.



```
5_Ain := cs_in = cs
5_Aout := res = res_out
6_Ain1 := cs_in = cs
6_Ain1 := opId_in = taskId
6_Aout := res = res_out
13_Fin := cs = cs_in
13_Fout := res_out = res
20_Fin1 := cs = cs_in
20_Fin2 := opId = opId_in
20_Fout := res_out = res
```

```
33_Ain1 := cs_in = cs
33_Ain1 := opId_in = op2Id
33_Aout := res = res_out
38_Fin1 := cs = cs_in
38_Fin2 := opId = opId_in
38_Fout := res_out2 = res
43_Ain1 := cs_in = cs
43_Ain1 := opId_in = opId
43_Aout := res_out2 = res_out
```

Figure 8.8: MSDG of the weather forecast example

$$\Phi\ (c\ \{x_1 = e_1\ ...\ x_n = e_n\ f_1\ ...\ f_n\}) \equiv$$
$$\quad \Psi\ ([x_1 = e_1, ..., x_n = e_n] + \pi_1\ a + \pi_1\ b)$$
$$\quad \Upsilon\ (\pi_2\ b)$$
$$\quad \texttt{<flow>}\ \pi_2 a\ \texttt{</flow>}$$
$$\quad (\pi_3\ b)$$

Where

$$a = foldr\ g\ ([], [])\ (map\ \Phi_h\ [f_1, ..., f_n])$$
$$g\ (u, v)\ (t, k) = (u : t, v : k) \qquad b = map\ \Phi_b\ [f_1, ..., f_n]$$

Figure 8.9: WS-BPEL generation

$$\Phi_h\ (\texttt{CM}\ f(\overline{x})\{st\}) \equiv$$
$$\quad (\Psi(\overline{x}),\ \texttt{<receive partnerLink=\#\#opaque>}$$
$$\qquad \texttt{operation=}\ f\ \ \texttt{variable = }(f + \texttt{Request })\ \texttt{>}$$
$$\qquad \texttt{<sources><source name=}\ f\ \texttt{/></sources>}\ )$$
$$\Phi_h\ \_ \equiv \bot$$

Figure 8.10: Function header WS-BPEL generation

$\Phi_b\ v\ l\ \texttt{z} \equiv (v,\ l,\ \texttt{<literal>}\ z\ \texttt{</literal>}\ )$

$\Phi_b\ v\ l\ \texttt{x} \equiv ((\Psi\ x) : v,\ l,\ \bot)$

$\Phi_b\ v\ l\ (\texttt{x = }\ e) \equiv ((\Psi\ (\texttt{x =}e)) : v,\ l,\ \bot)$

$\Phi_b\ v\ l\ (st_1\ \ \texttt{;}\ \ st_2) \equiv ((\pi_1\ a) + (\pi_2\ b) + v,\ (\pi_2\ a) + (\pi_1\ b) + l,\ (\pi_3\ a) + (\pi_3\ b))$

Where $a = \Phi_b\ v\ l\ st_1$

$b = \Phi_b\ v\ l\ st_2$

$\Phi_b\ v\ l\ (\texttt{LOCK}\ \{st\}) \equiv ((\pi_1\ a) + v,\ (\pi_2\ a) + l,\ \texttt{<scope isolated=yes>}(\pi_3\ a)\texttt{</scope>})$

Where $a = \Phi_b\ v\ l\ st$

$\Phi_b\ v\ l\ (\texttt{LOCALCALL}\ f\texttt{(}\overline{x}\texttt{)}) \equiv (v,\ l,\ \texttt{<invoke partnerLink=localhost operation=}\ f\texttt{/>})$

$\Phi_b\ v\ l\ (\texttt{SYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)}) \equiv (\Psi(\overline{x}) + v,\ l,\ \texttt{<invoke partnerLink=}s\ \texttt{operation=}\ f\texttt{/>})$

$\Phi_b\ v\ l\ (\texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)}) \equiv$

$(\Psi(\overline{x}) + v,\ l,\texttt{<flow><invoke partnerLink=}s\ \texttt{operation=}f\texttt{/></flow>}$

$\Phi_b\ v\ l\ (\texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)}\ \{st\}) \equiv$

$(\Psi(\overline{x}) + (\pi_1\ a) + v,\ (linkId : l) + (\pi_2\ a),$

$\texttt{<flow> <invoke partnerLink=}\ s\ \ \texttt{operation=}\ f\ \texttt{/>}$

$\texttt{<sources><source linkName =}\ linkId\ \texttt{/></sources> </flow>}$

$\texttt{<scope name=}\ f\texttt{Completed>}$

$\texttt{<targets><target linkName =}\ linkId\ \texttt{/></targets>}\ \pi_3\ a\ \texttt{</scope>}\ )$

Where $linkId = f + getUToken()$

$a = \Phi_b\ v\ l\ st$

$\Phi_b\ v\ l\ (\texttt{IF}\ p\ \texttt{THEN}\ \{st_1\}\ \prec \texttt{ELSE}\ \{st_2\} \succ) \equiv ((\pi_1\ a) + (\pi_1\ b) + v,\ (\pi_2\ a) + (\pi_2\ b) + l,$

$\texttt{<if><condition>}\ \beta(p)\ \texttt{</condition>}\ \pi_3\ b \prec \texttt{<else>}\ \pi_3\ b\ \texttt{</else>} \succ \texttt{</if>})$

Where $a = \Phi_b\ v\ l\ st_1$

$b = \Phi_b\ v\ l\ st_2$

$\Phi_b\ v\ l\ (\texttt{WHILE}\ p\ \texttt{DO}\ \{st\}) \equiv ((\pi_1\ a) + v,\ (\pi_2\ a) + l,$

$\texttt{<while><condition>}\beta(p)\texttt{</condition>}\pi_3\ a\texttt{</while>})$

Where $a = \Phi_b\ v\ l\ st_1$

Figure 8.11: Function body WS-BPEL generation

```
import System.Web.Services.Protocols.SoapHttpClientProtocol;
import System.Web.Services.Protocols;
public Class TimesheetSubmission {

    [WebMethod]
    public void SubmitTimesheet(TimeSheet t,
                                Consultant c, Client clt) {
        Decimal total = Invoke("GetTimesheetWithCost",
                           new object[] { c });
        if(total > 2000)
            this.InvokeAsync("AnalyzeSheet",
                             new object[] { t, c},
                             this.OnAnlyzeResponse, null);
        else {
            Invoke("CommunicateClientExpense",
                   new object[] { expense, total });
            Invoke("NotiffyApprovedExpense",
                   new object[] { expense, total });
        }
    }

    private void OnAnalyseResponse(object arg) {
        InvokeCompletedEventArgs invokeArgs =
            ((InvokeCompletedEventArgs)(arg));
        if (invokeArgs.Approved) {
            Invoke("CommunicateClientExpense",
                   new object[] { invokeArgs.Expense,
                                  invokeArgs.Total });
            Invoke("NotiffyApprovedExpense",
                   new object[] { invokeArgs.Expense,
                                  invokeArgs.Total });
        } else {
            Invoke("ResubmitSheet",
                   new object[] { invokeArgs.TimeSheet });
        }
}
```

Figure 8.12: The time sheet submission example

Chapter 9

Discovery of Coordination Patterns

The algorithms introduced in the previous chapter for generating coordination specifications from the CDG, either in Orc or WS-BPEL, amount basically to "complex" translation operations. Of course such translations have to deal with many details concerning, not only the CDG coordination representation format, but also the peculiarities of the languages in which such coordination specifications are to be expressed. We believe that these algorithms have a real opportunity in specific practical cases, mainly because they work in a complete automatic way and their complexity[1] is relatively low, which makes them able to retrieve answers in a reasonable time.

On the other hand, this approach has a number of problems. The first disadvantage that may arise is that, for cases where the system contains significant coordination logic, the algorithms may derive large specifications that are difficult to understand and analyse manually. Secondly, because the algorithms work in a fully automatic way. In some cases this is definitely an advantage, but in other cases, one would like to guide the coordination discovery process by suggesting some typical coordination policies to be looked for or by making the algorithm ignore some services or components of the

[1] Although we have not performed an exhaustive complexity analysis of the presented algorithms, the profiling of some practical cases indicates that the algorithms present reasonable response times.

system. In general, it would be interesting that the algorithms could take into consideration well-known coordination patterns, so that whenever instances of such patterns occur in the system under analysis, the algorithms would retrieve exactly the fragments of source code which implement them.

In order to cope with these issues, this chapter introduces a method for performing coordination discovery based on sub-graph patterns identification. The general idea is to have a knowledge base of coordination schemes described in terms of their CDG pattern representations. These coordination schemes are then passed to the coordination discovery algorithm which searches the CDG for instances of the corresponding patterns contained in the knowledge base. Whenever one of them is detected in the CDG, the algorithm returns a mapping from the coordination pattern description to the actual CDG vertices responsible for its implementation.

Unlike the algorithms from the previous chapter, this approach is not completely automatic (although one can use a constant set of patterns) because it requires to tune the algorithm by providing specific coordination patterns to be searched for. Nevertheless, once configured, this algorithm has the advantage of generating much smaller and understandable coordination specifications.

The overall discovery strategy is similar to the one presented in the previous chapter but for the last stage. This is depicted in Figure 8.1.

9.1 Describing Coordination Patterns

The first problem that has to be addressed in this approach is how to correctly describe coordination patterns so that they can be searched for on CDG instances. For this, we have chosen to keep a balance between expressiveness of the pattern description language and the feasibility of using its values to perform an automatic search in CDG instances. Therefore, we define coordination patterns as pairs formed by a *matching condition* (of type PCondition) and a graph over nodes of type NodeId as follows

$$\begin{aligned}
\mathsf{Pattern} &= \mathsf{PCondition} \times (\mathsf{NodeId} \times \mathsf{ThreadId} \times \mathsf{NodeId} \times \mathsf{PathPattern})^* \\
\mathsf{PCondition} &= \mathsf{NodeId} \rightharpoonup \mathbb{B}^{\mathsf{GNode}} \\
\mathsf{NodeId} &= \mathbb{N} \cup \{\triangle, \triangledown\} \\
\mathsf{PathPattern} &= \mathbb{N}^+ \cup \{+\}
\end{aligned}$$

A matching condition is a mapping (i.e., a partial function) which associates to each pattern node (of type NodeId) a predicate over CDG nodes (of type GNode). In practice, a simple way to define such a predicate is by a regular expression intended to be tested for matching with program data collected on CDG nodes. For simplicity, we use the symbol $*$ in place of a regular expressions, abbreviating the everywhere true predicate. Examples of pattern conditions are shown later in this section.

The second component of a pattern is a sequence of edges labelled by a thread identifier ($\mathsf{ThreadId}$), which is used to specify the intervening threads in a pattern, and a qualifier (of type $\mathsf{PathPattern}$) which specifies the number of edges in the CDG that may mediate between the node matching the source and the target node in the pattern. In particular, symbol $+$ is used to stand for one or more edges. We also assume that all nodes in the sequence of edges of a pattern which do not belong to the domain of the respective condition, are implicitly labelled by the everywhere true predicate.

Based on the data specifications above, we have defined a small language to express coordination patterns. Such notation, referred to as the *Coordination Dependence Graph Pattern Language* (CDGPL), was specifically designed to describe CDG graph patterns and to facilitate this automatic discovery.

The discovery process, in particular, is guided by what we call a *search pattern*, i.e. an expression defined simply as a pattern (of type $\mathsf{Pattern}$) or either as a conjunctive (&&) or disjunctive (||) aggregation of patterns.

For illustration purposes, however, we resort to a graphical notation to present a number of the most typically found coordination patterns. These are depicted in Figure 9.1, where notation vc_x denotes the node condition for node x. They are detailed in the sequel.

9.1.1 Synchronous Sequential Pattern

This is one of the simplest patterns in which a sequence of external services is invoked by calling one service after the other. This simple, yet often used, pattern is usually employed when there are dependencies between a number of service calls, i.e., when a service call depends on the response received to a previous one.

In our notation this pattern is specified as in Figure 9.1(a), where each node corresponds to a service call in the series of services to be invoked in sequence. If the original source code implements coordination through access to web-services, then the condition for each of these vertices can be defined by the following predicate template:

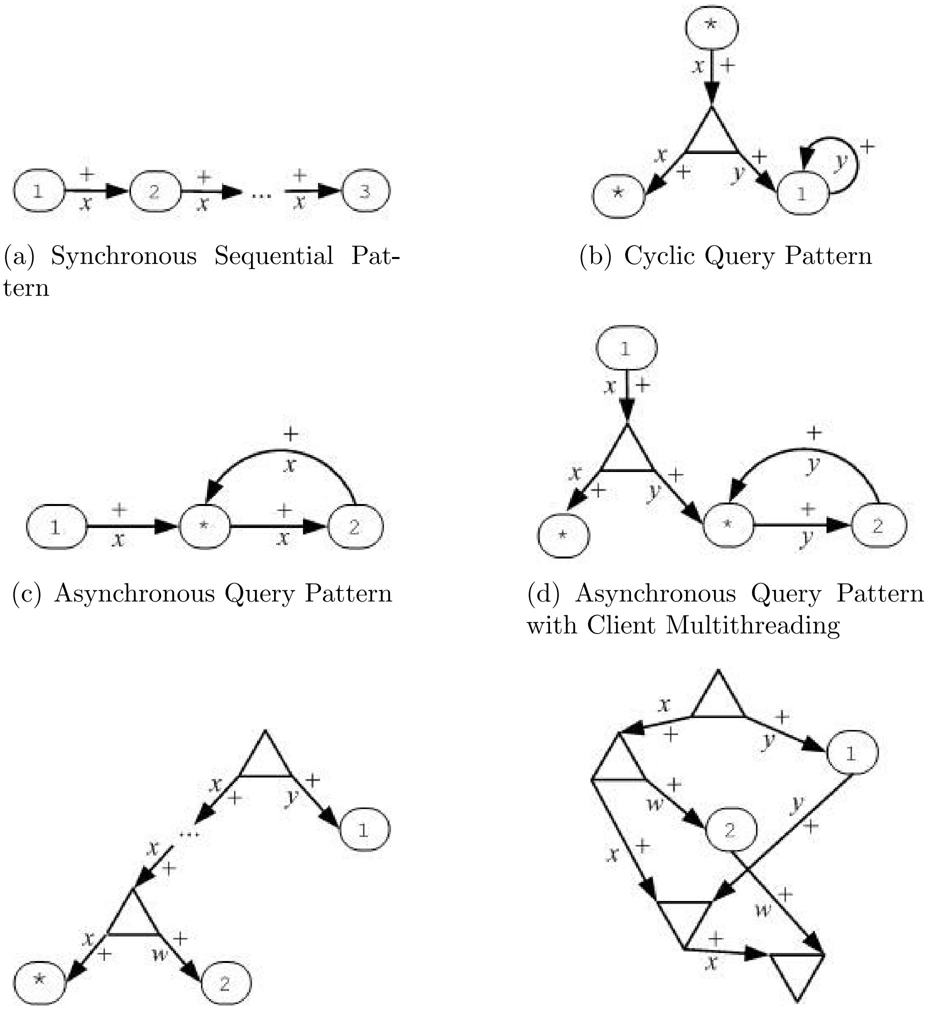
$$\begin{aligned} pc(x) = \quad & x == (\mathsf{MSta}(t,s), cp, cm, cd) \Rightarrow \\ & match(s, \text{“}\texttt{ServiceCall(*)}\text{”}) \land cp == \underline{\mathsf{webservice}} \land \\ & cm == \mathsf{sync} \land cd == \mathsf{consumer} \end{aligned}$$

where “`ServiceCall`” is to be replaced by the name of the invoked web-service method.

9.1.2 Cyclic Query Pattern

This pattern is characterized by a point in which a new thread is spawned, becoming responsible for a on-going invocation of an external service. It is often used by systems that have to monitor the state of some foreign resource or that must be constantly updating an internal resource which depends upon an external service.

In practice, this pattern appears in several variants. For instance, it may include a time delay between each cyclic service call or use different strategies to implement the service invocation cycle (e.g. resorting to a recursive function definition or to an iterative control statement).



(a) Synchronous Sequential Pattern

(b) Cyclic Query Pattern

(c) Asynchronous Query Pattern

(d) Asynchronous Query Pattern with Client Multithreading

(e) Asynchronous Sequential Query Pattern

(f) Joined Asynchronous Sequential Pattern

Figure 9.1: *CDGPL patterns*

The pattern presented in Figure 9.1(b) captures the most generic version of this pattern. It basically states that a new thread y must be spawned and that under the execution of this new thread a service must be called repeatedly. Again, vertex 1 must be instantiated with a predicate, similar to the one used in the previous pattern, limiting the scope of services to be called.

9.1.3 Asynchronous Query Pattern

.

The Asynchronous Query Pattern is usually employed whenever there is a need to invoke time consuming services and the calling threads cannot suspend until a response is returned. To overcome this situation, one of the most used solutions is one where the server component provides two methods, one for the request of an operation on the server and another for the querying of an answer (if available) from the previously posted request. Because these services are not involved in actual execution of any requested complex operations, but rather in the control of the execution of complex operations and results retrieval, both return an answer very quickly.

From the client side, this pattern is specified by the definition in Figure 9.1(c), encoding the invocation of a service to request the execution of some operation execution (node 1) and a cyclic invocation of another service (node 2) to retrieve the result. Once more, in practice, both vertices 1 and 2 may be further characterised by predicates that would clearly identify the operations for performing the request and result retrieval.

9.1.4 Asynchronous Query Pattern (with client multi-threading)

This often used pattern is actually a variation of the previous one, where the client orders the execution of an operation in one thread and then launches a second thread to retrieve the result. Note that this pattern, presented in Figure 9.1(d) is also quite similar to the cyclic pattern, except for an extra node, marked with $*$ to represent the program statement that controls the need to perform more invocations to retrieve the result.

9.1.5 Asynchronous Sequential Pattern

This pattern is similar to the *Asynchronous Sequential Pattern* except that it invokes each service in a new thread specifically created for the effect.

This pattern is often used when a system has to invoke a series of services and the order of invocation as well as the responses returned is irrelevant. Note that, under these premises, the code corresponding to this pattern is substantially faster than the one for the *Asynchronous Sequential Pattern* case, in the invocation of the series of services. This pattern is specified in Figure 9.1(e) where each of the service calling nodes (1 and 2) are invoked in different threads (y and w respectively).

9.1.6 Joined Asynchronous Sequential Pattern

The *Joined Asynchronous Sequential Pattern* is similar to the previous pattern in the sense that, in both cases, services are invoked asynchronously. The difference is that in this pattern one is interested in controlling the point where each of the called services have finished execution and, possibly, returned a value.

The specification of this pattern is presented in Figure 9.1(f) where each thread that was spawned to invoke a service, joins later in a point where the execution may proceed with the guarantee that all service calls have terminated.

9.2 The Discovery Algorithm

The algorithm presented in this section retrieves every sub-graph of a CDG that conforms to a given *graph pattern*. The notation used is self-explanatory. However, let us point out the use of dot . as a field selector in a record as well as the adoption of the HASKELL syntax for lists (including functional *map* and operators : for appending and ++ for concatenation). An assignment is denoted by the ← operator; note that it can be prefixed by an expression declaring the type of the variable being bound.

The algorithm resorts to the data types in Figure 9.2, also expressed in the Haskell syntax for data type declarations. Note, in particular, how both the CDG and the graph representing the pattern to be discovered are made available to the algorithm through embedding in *Graph* and *GraphPattern*: in both cases a node is selected as "root", i.e. as a starting point for searching.

$$\begin{array}{lcl}
\mathsf{Graph} & = & \mathsf{G}\ \{\ \mathsf{root} : \mathsf{GNode}\ \times\ \mathsf{G} : \mathsf{CDG}\ \} \\
\mathsf{GraphPattern} & = & \mathsf{GP}\ \{\ \mathsf{root} : \mathsf{NodeId}\ \times\ \mathsf{G} : \mathsf{VertexPattern}\ \} \\
\mathsf{VertexPattern} & = & \mathsf{VP}\ \{\ \mathsf{id} : \mathsf{Int}\ \times\ \mathsf{cdts} : \mathsf{GNode}\ \times\ \mathsf{visited} : \mathbb{B}\ \} \\
\mathsf{Attribution} & = & \mathsf{AT}\ \{\ \mathsf{vp} : \mathsf{VertexPattern}\ \times\ \mathsf{v} : \mathsf{GNode}\ \} \\
\mathsf{Extension} & = & \mathsf{E}\ \{\ \mathsf{g} : \mathsf{Graph}\ \times\ \mathsf{att} : [\mathsf{Attribution}]\ \}
\end{array}$$

Figure 9.2: Data types for the graph pattern discovery algorithm

The overall strategy used by the discovery algorithms 1 and 2 consists of traversing the graph pattern and incrementally constructing a list of candidate graphs with nodes of type *Attribution*. This type is used by the algorithm because it maintains a mapping between the graph pattern nodes and CDG nodes. If a pattern is found, during the traversal of the graph pattern, for which a candidate graph cannot be extend to conform with, then the graph in question is removed from the candidate's list. On the other hand, if the candidate graph can be extend with one of the several CDG candidate nodes, it originates a series of new candidate graphs (one for each CDG candidate node) and the original (incomplete) candidate is removed from the candidate's list.

Most auxiliary functions used in the algorithm presented bellow are self-explanatory[2], with the possible exception of function GetSuccCombinations. This, calculates a list of lists of *Attributions*, i.e., a list for each possible set of possible attributions for a given node pattern.

[2] The entire algorithm, expressed in $C^\sharp$, is available at `http://alfa.di.uminho.pt/~nfr/PhdThesis/SubGraphIsomorphismAlgorithm.zip`

Algorithm 1 Pattern Discovery - Part I

```
function DiscoverPattern(Graph cdg, GraphPattern cdgp)
    cdgp ← FillCandidateVertices(cdg, cdgp)
    cdgp ← FillCandidateEdges(cdg, cdgp)
    Graph bg ← emptyGraph()
    [Extension] gel ← [(bg, map $(\lambda x \rightarrow (cdgp.root, x))$ cdgp.root.cdts]
    repeat
        $\mathbb{B}$ b ← False
        for all Extension ge in gel do
            for all Attribution datt in ge.att do
                datt.vp.visited ← True
                $c_1$ ← HasSuccessors(cdgp, datt.v)
                $c_2$ ← ! HasSuccessors(ge.g, datt.vp)
                if $c_1 \wedge c_2$ then
                    [Extension] dgel ← ExtendBaseGraph(ge.g, datt)
                    [Extension] r ← ge : r
                    [Extension] a ← dgel : a
                    b ← $b \vee$ Length(dgel) $> 0$
                end if
            end for
        end for
        gel ← Remove(gel, r)              ▷ Remove all r elements from gel
        gel ← gel ++ r                    ▷ Add all a elements to gel
        r ← []
        a ← []
        nv ← NotVisited(cdgp)   ▷ Get first not visited Vertex Pattern
        if $b \wedge nv \neq null$ then
            b ← True
            vpa ← map $(\lambda x \rightarrow (nv, x))$ nv.cdts
            map $(\lambda x \rightarrow (x.g, vpa))$ gel
        end if
    until b == True
    return gel
end function
```

Algorithm 2 Pattern Discovery - Part II

```
function ExtendBaseGraph($Graph\ bg, Attribution\ att$)
    $tcs \leftarrow$ GetSuccCombinations($cdgp, vp$)
    for all $tc$ in $tcs$ do
        $ng \leftarrow bg$
        $gel \leftarrow (ge,\ [])$
        for all $cv$ in $tc$ do
            if $b \wedge nv \neq null$ then
                AddEdge($ng,\ att,\ cv$)
                $ge.DiscoveredAttributions.Add(cv)$
            else
                $gel.Remove(ge)$
                break
            end if
        end for
    end for
    return $gel$
end function
```

This algorithm makes it possible to identify coordination schemes as graph patterns in a CDG. For each pattern identified, the corresponding code fragment in the source can be recovered — a strategy implemented in our "proof-of-concept" tool described in next chapter. Another use of such patterns would be to generate coordination specifications based on their translations to a suitable coordination language (such as Orc) and composition.

Chapter 10

CoordInspector

10.1 Motivation

The second part of this thesis addressed the problem of extracting the coordination logic entangled in legacy software. It was also claimed that, for most real-world cases, such extraction is not a trivial task to accomplish, mainly because it needs to cope, simultaneously, with the extension of the source code to be analysed, the heterogeneity of languages and technologies employed and the specific level of coordination (inter-thread coordination, component coordination, services coordination, etc) that one is looking for in each particular case. Moreover, one wants the result of such a recovery process to deliver, as much as possible, well-known coordination patterns, and clear specifications of the coordination policies, facilitating their re-use and re-engineering.

This entails the need for suitable tool support to the effective application of the techniques discussed in the previous chapters. In particular we seek for the possibility of automatically extracting the (often implicit) coordination layer of a system and representing it in suitable visual ways.

The whole, constantly expanding, family of *service-oriented* applications is certainly an interesting target for such tools. A recovered coordination model, exposing services calls and the programming logic that directly (or indirectly) influences (or is influenced by) such calls, would facilitate the evolution of legacy systems towards the service oriented paradigm, as well as the development of new service oriented systems and also their maintenance.

Furthermore, this tool should be able to capture multithreaded information and to confront it with the services calling model. It would then be able to assist the developer in answering questions like: What services are actually being invoked in the implementation of a particular functionality? How are these services combined to achieve the desired functionality? If one of these services fails, how does the system behaves? What is the logic, in terms of internal and external services invocations, behind the system provision of services?

Such was the motivation behind the development of CoordInspector, a tool for coordination analysis, targeting the family of Microsoft .Net languages, and partially implementing many of the ideas presented in the previous chapters.

10.2 Implementation

The tool, a snapshot of which is presented in Figure 10.4, is available from

`http://alfa.di.uminho.pt/~nfr/Tools/CoordInspector.zip`.

A basic choice in CoordInspector design, was to make it as generic as possible. Therefore, the actual prototype version currently available, targets the Common Intermediate Language (CIL)[MR03] code, the native language of the Microsoft .Net Framework, to which every .Net compilable language ultimately gets translated to before being executed in the framework. The decision to target CIL code was not arbitrary. Indeed we intended the tool to be able to cope with as many programming languages as possible, because most real world software systems are developed in more than one language. Moreover, given the potential of the tool to assist legacy systems evolution, this sort of "language agnosticism" becomes even more important. Thus, by choosing CIL, the tool is presently able to analyse more than 40 programming languages, and this number has only but potential to increase.

In order to take advantage of existing CIL analysis tools, CoordInspec-

TOR is developed as a plug-in for the CIL decompiler .NET REFLECTOR[1]. The only two components COORDINSPECTOR borrows from .NET REFLECTOR are the parser for CIL code, which delivers an object tree representation of the CIL abstract syntax tree, and the code representation plug-ins, which transform CIL code into higher-level languages, like $C^\sharp$ and C++.

Such tree is then processed to build the corresponding MSDG instance. Given the intrinsic modularity of this process, it is executed by different components that are responsible for the calculation of each of the MSDG constituents, *i.e.*, the nodes representing statements and every kind of dependency between such nodes, as detailed in section 8.3. Each component traverses the concrete syntax tree and collects the relevant information for the construction of a particular graph.

When applied to real world systems, and if executed sequentially, the MSDG calculation process can be a time consuming task because of the size and computational complexity involved in its construction. In order to cope with this situation we improved the MSDG calculation performance by multi-threading the independent tasks which build each MSDG set of dependencies. This improvement reduced significantly the MSDG calculation time.

The CDG calculation implemented by COORDINSPECTOR closely follows the approach presented in section 8.4, starting by labelling the vertices based on rules identifying communication primitives and, then, pruning the vertices according to the strategy presented in the same section. At the moment of writing, COORDINSPECTOR is only instantiated with rules identifying web-services communications, distinguishing between synchronous and asynchronous calls as well as between invocation and provisioning of functionality using web-services.

The graph pruning and slicing operations were implemented according to the specifications presented in the section 8.4 and by a series of graph traversal algorithms and transformation functions.

The tool is currently able to generate ORC specifications, corresponding to the discovered coordination logic. For this, it closely follows the algorithm presented in section 8.5, with the exception that the algorithm was slightly

[1] http://www.aisto.com/roeder/dotnet

adapted to meet the object oriented paradigm and instead of immediately generating syntactic Orc expressions the implemented version generates an object tree version of Orc which is then traversed to originate the final Orc scripts.

The tool is also able to re-construct the analysed code, *i.e.* the code represented by the calculated CDG instance, which focus the specific aspects determined by the set of rules used. For this feature CoordInspector uses the specific code representation plug-ins, available for .Net Rotor. Because the tool depends on the available code representation plug-ins, for now it is only capable of representing code in *C♯*, Visual Basic, MC++, Chrome, Delphi and, of course, CIL itself.

CoordInspector is also able to depict and navigate through both the calculated MSDG and CDG graphs. For this, the tool resorts to the *Microsoft Research Graph Layout Execution Engine* (MSR GLEE) graph library. The generated graphs provide different colours for the vertices, based on the labels the vertices hold, which facilitates direct manual reasoning over such structures.

The graphical presentation of the graphs is also able to supply the user with specific vertex information, like labelling and the CIL code captured, by double clicking on a particular vertex of the graph.

10.2.1 Architecture

The architecture of CoordInspector is depicted in Figure 10.1 by a typical box component diagram, representing the main components upon which the implementation was divided. Note that, some of them contain themselves other minor components, but, for the sake of understandability, we will abstract our presentation from these.

Reading the diagram from top to bottom, the first component block represents .Net Reflector, which contains many more components than the represented ones, but, from the CoordInspector architecture perspective, this component only exposes the *CIL parser* and the *Multilanguage Generator* sub-components. The *MSR GLEE* component is used for the

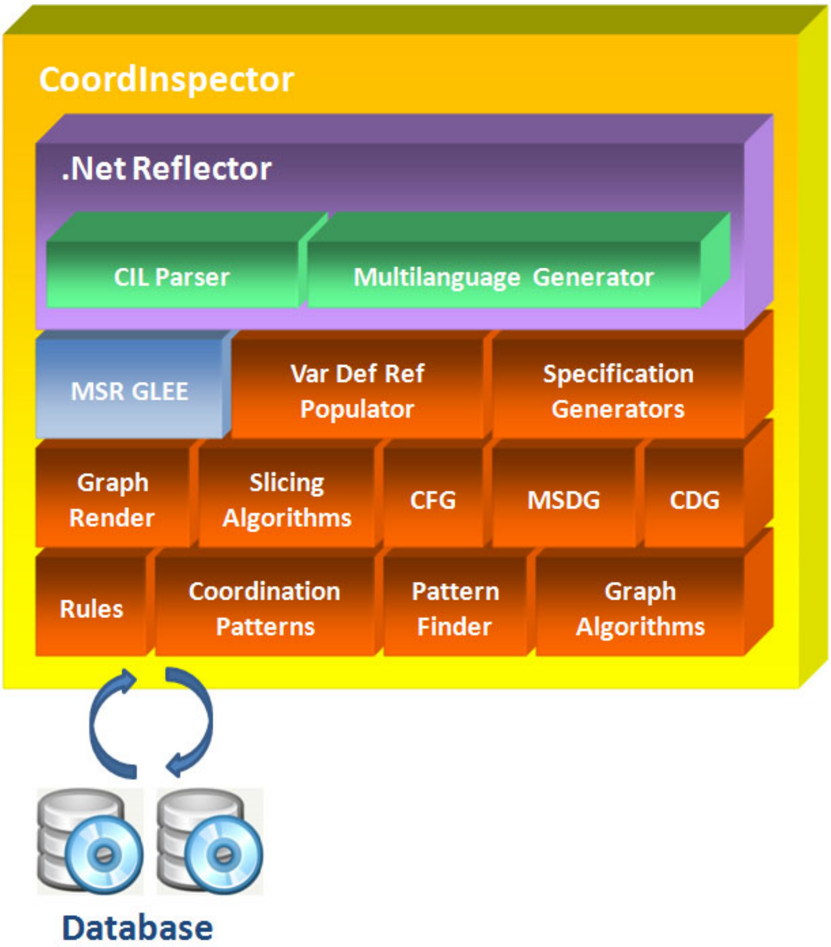


Figure 10.1: CoordInspector architecture

graphic layout of all graphs calculated during the analysis process. This component is completely isolated from the remaining components and uses a graph representation that is different from the ones used internally by CoordInspector for instance, to capture MSDG's and CDG's. Therefore, whenever a component has to display graphically a CDG or MSDG, it resorts to the *Graph Render* component, which is responsible for translating the CoordInspector internal graph representations to the representation used by the *MSR GLEE* component.

Apart from .Net Reflector and *MSR GLEE*, all the remaining components were developed specifically for CoordInspector. The *CFG* component interprets the abstract syntax tree retrieved by the *.Net Rotor CIL Parser* and extracts the control flow graph by translating the base language control flow statement semantics into a graph representation. *Var Def Ref Populator* component, navigates back and forth along the CFG (resorting to the *Graph Algorithms component*) in order to calculate, for each CFG vertex, the set of variables defined and used in each programming construct contained in the vertex. This information is then vital for the *MSDG* component which is responsible for the calculation of the MSDG, following closely the process explained in chapter 8.

The calculation of the CDG is, of course, at the responsibility of component *CDG* with resorts to the rules captured by the rule management component *Rules* and to the *Slicing Algorithms* component in order to reduce the MSDG according to the strategy defined in chapter 8. As expected, the *Rules* component is responsible for create, read, update and delete (CRUD) operations for rules, using a XML database for this matter.

The *Specification Generators* component contains a set of sub-components for the code generation of the different coordination specifications. Each of these code generation sub-components contains abstract representations of the targeted specification language and often resort to the *Graph Algorithms* component for traversing and consuming the CDG. For now, the *Specification Generators* component is populated by a single sub-component, responsible for the generation of Orc specifications.

Besides consuming the CDG in order to generate coordination specifications, the tool is also able to discover previously defined coordination patterns. For this matter, CoordInspector uses the *Pattern Finder* component, which implements the coordination pattern discovery algorithm presented in chapter 9. The coordination patterns used for this task are managed by the *Coordination Patterns* components, that, like the *Rules* component, implements CRUD operations and uses an XML database for the permanent storage of patterns.

Figure 10.2 presents the main interactions between the CoordInspector components, which correspond to the different phases of our analysis process. Note that the interaction model follows a typical pipeline architec-

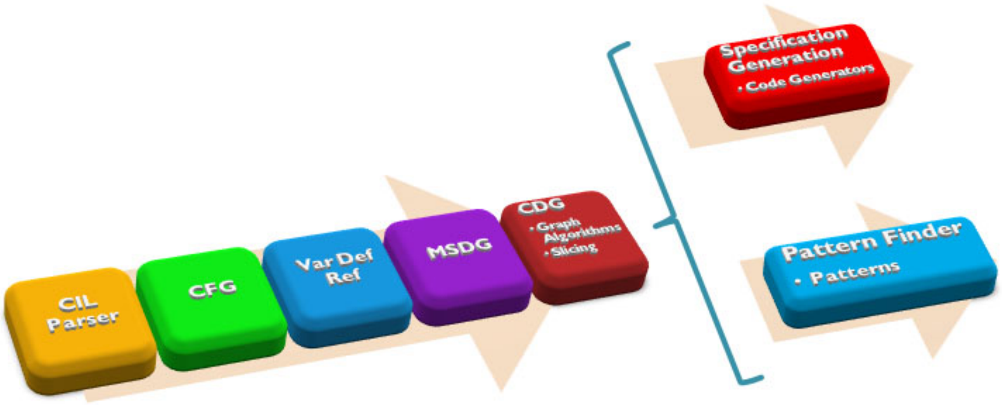


Figure 10.2: Simplified CoordInspector analysis implementation

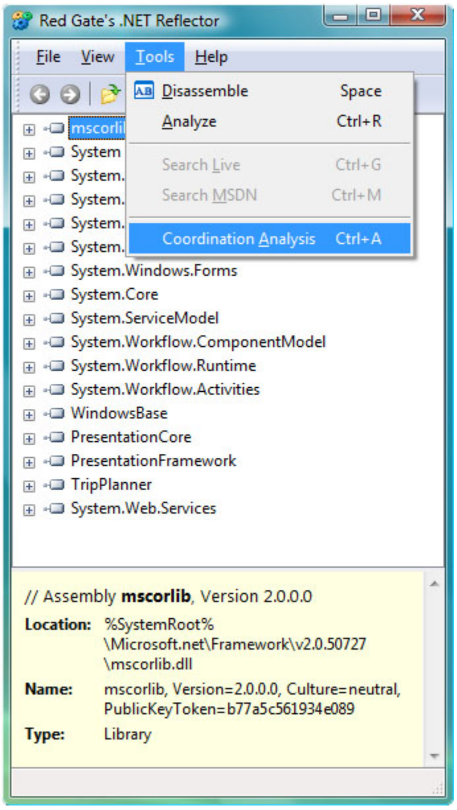
ture, ending in two different ends, one for each of the coordination analysis approaches introduced in the previous chapters. Figure 10.2 focus on the main components involved in the analysis process, so, it excludes all the components in Figure 10.1 which play an auxiliary role.

10.3 Using CoordInspector

Once CoordInspector is launched, the user is provided with a form similar to the one presented in Figure 10.3. This form, displays a tree, of which the first level is expanded and each node (in the first level) corresponds to the assemblies loaded in the tool, usually the base assemblies that compose the .Net Framework.

To analyse a program, the user must use the CoordInspector main menu option *File* → *Open...*, followed by the selection of the file containing the main method of the application to be analysed. This will load the assembly of the program to be analysed into the tool, corresponding to a new node being added to the tree displayed in Figure 10.3.

The next step consists of launching the coordination analysis user interface, which can be done by clicking the main menu option *Tools* →



// Assembly **mscorlib**, Version 2.0.0.0

Location: %SystemRoot%\Microsoft.net\Framework\v2.0.50727\mscorlib.dll

Name: mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

Type: Library

Figure 10.3: CoordInspector initial form

Coordination Analysis. This will make CoordInspector to take an aspect similar to the one presented in Figure 10.4.

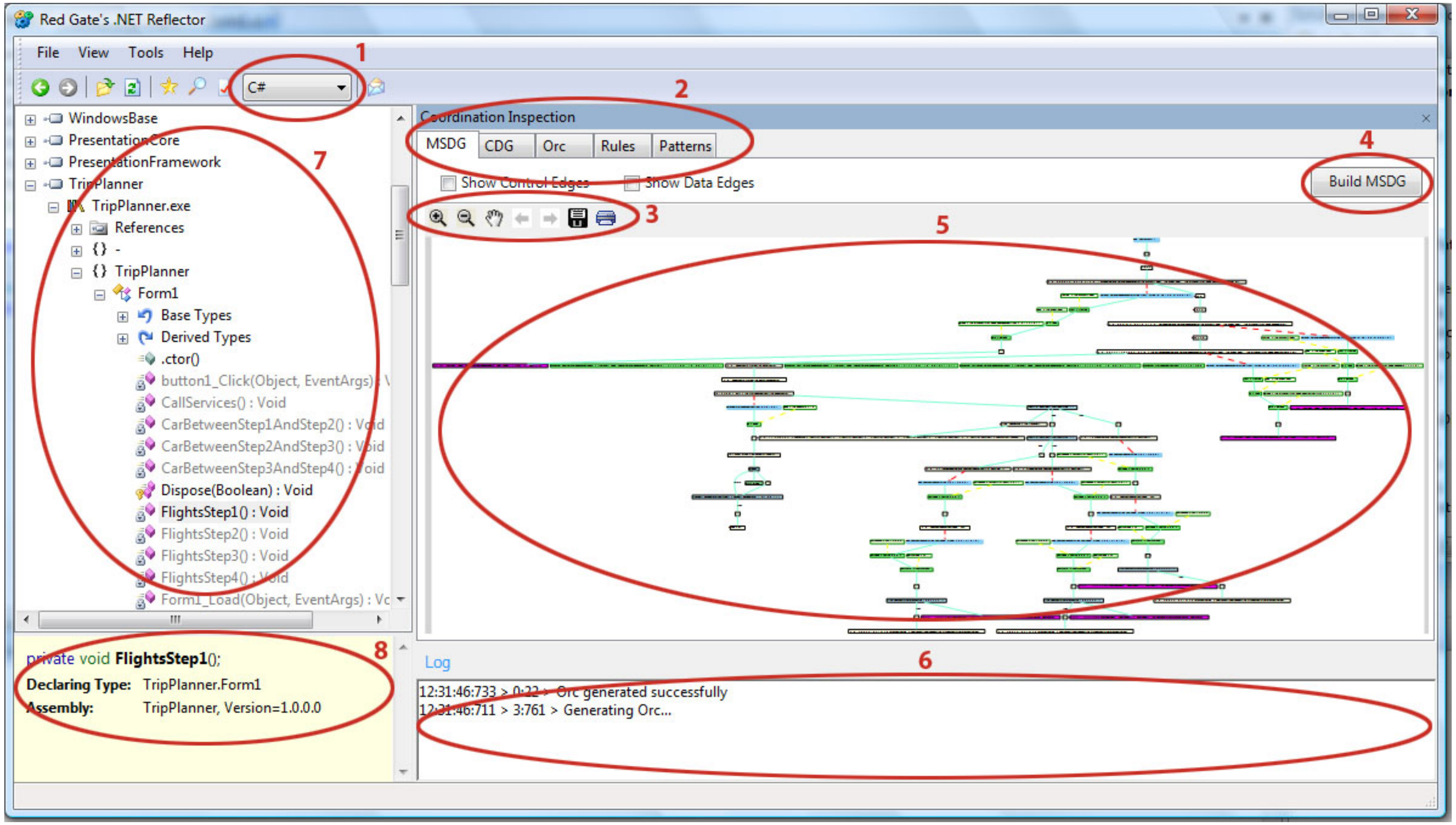
Now, real coordination analysis can start. For this, one has to choose the programming entity upon which the analysis process will begin, by choosing a node in the programming entities tree, displayed in Figure 10.4 by area 7. Once the programming entity is selected, its details appear in area 8, and the user may click the button in area 4 to start the MSDG calculation. During the MSDG calculation, area 6 will provide information about progress and details of the calculation process. Once CoordInspector finishes calculating and rendering the MSDG, the graph is displayed in area 5, which can be inspected by the graphical operations provided in area 3. The user may perform this same operation over other program entities displayed in the tree, which allow him to inspect the different MSDG's of the application to be analysed.

Once the MSDG has been calculated, the user may proceed to the CDG calculation, by clicking on a button similar to the one presented in area 4, but this time in the tab *CDG* displayed in area 2. Again, area 6 will provide the user with information and progress about the calculation process.

Finally, the user may generate an Orc specification based on the calculated CDG, and by accessing the *Orc* tab in area 2 followed by a click on the *Generate Orc* button. The Orc specification is provided in central area of the *Orc* tab.

At any time during the analysis process, the user may change the rules upon which the CDG is calculated, by using the rules management interface provided in *Rules* tab.

The coordination pattern discovery follows a similar interface to the Orc specifications generation, and can be accessed in the *Patterns* tab. This tab also provides an interface for the management of the coordination patterns, to be used in the pattern discovery algorithm.



private void FlightsStep1();
Declaring Type: TripPlanner.Form1
Assembly: TripPlanner, Version=1.0.0.0

```
12:31:46:733 > 0:22 > Orc generated successfully
12:31:46:711 > 3:761 > Generating Orc...
```

Figure 10.4: CoordInspector analysing a software system

Chapter 11

Case Study: Coordination Analysis in Software Systems Integration

11.1 Introduction

In this chapter we apply the previously presented coordination discovery techniques to a project of software systems integration. The problem of integrating software systems, usually referred in the literature by *Enterprise Application Integration* (EAI) [Lin00, SS99, GBR04, HW03], constitutes one of today's most resource consuming tasks in enterprise software systems management. In fact, according to Forrester Research, more than thirty percent of all the investments made in information technologies are spent in the linkage of software systems in order to accomplish global coherent enterprise software solutions.

There are many reasons justifying the need for integrating enterprise software applications. Among them the fusion or acquisition of companies, the necessity companies have to explore new and different markets (e.g. the internet or mobile software), the physical distribution of companies (internationalization), the evolution of internal software solutions (which often overlap each other both on data and functionality) and the introduction of new applications or new versions which often demand an integration with the already existing systems.

The objective of EAI consists of the union of services, data and functionality from different software systems, with the objective of achieving a single, integrated and coherent enterprise software solution. The systems integration should be performed at both the data and process automation levels, and in such a way that a user of the integrated system (called the *enterprise software system*) should not have to worry about the synchronization of both data and processes between applications.

A great deal of work behind most EAI projects concerns the definition and implementation of a specific coordination model between the systems being integrated. This model should aid the software architect answering questions like, which software systems are connected, how do they communicate (messages, distributed objects, Web-Service), what type of communication is being held (asynchronous or synchronous) and what are the dependencies between the connections. All these and other questions can be answered, and specific policies, proved correct, by coordination models. That is why EAI provides an interesting case study for the techniques discussed in the second part of this thesis.

There are several technologies employed in the implementation of integrated software systems solutions, and some of them have even been specifically developed for addressing this particular problem. Popular examples include, XML [BPSM97], SOAP [Jep01], CORBA [BVD01], middleware message systems [HW03] and Web-Services [Che01, Gai04], just to mention but a few. However, for the specific case addressed in this chapter, we will focus on Web-Service communications, because such was the technology used by the company from which the concrete case study discussed here was chosen. Another reason for focusing on Web-Services is that CoordInspector has a specific instantiation with rules for analysing this particular kind of communication primitive.

However, note that this narrowing of the analysis scope over EAI projects using Web-Services is by no means a limitation of the entire process analysis presented in previous chapters. Moreover, the techniques presented can also be of use in forward engineering projects as well as with different types of communication primitives.

This chapter discusses a case study on the use of a coordination analysis process for the verification of design time integration strategies. Thus, we start from a scenario were the integration has already been implemented, and our aim being to validate if it respects the systems integration model that had been defined (most of the times informally) at the integration design time. Therefore, we have first to specify such model. This formalisation will disambiguate many of the integration strategies adopted and eventually lead to changes in the integration implementation.

The case study concerns a Portuguese training company with facilities in six different cities spread over the north of Portugal. Because the company has a great degree of specialisation in delivering computer network courses, it developed an internal department for reselling specific networking communication device products. Given these two major activities for the company, the main information automation needs are the management of training courses (trainers, trainees, summaries, training modules, etc) and the management of networking device sales (stocks, pricing, campaigns, discounts, suppliers, etc).

The remainder of this chapter is divided in three sections, the first of which presents the motivation for the integration and also the initial disconnected scenario in which the company system's where running before the integration project. The following section briefly presents the integration process as well as the final collaboration layout between the systems. Finally, section 11.4 introduces the CDGPL representation of some integration operations. For each case the actual coordination pattern was first extracted from the implementation code, a process assisted by CoordInspector. Then, it was analysed and re-engineered. The new coordination solution was finally suggested to the implementation team.

11.2 Disconnected Software Systems

Before having decided to embark on the software integration project, the training company was working with four main software systems. These systems, which we shall call *base components*, were composed by an *Enterprise Resource Planning* (ERP) system, a *Customer Relationship Management* (CRM) system, a *Training Server* (TS) system, and a *Document Management System* (DMS). The decision to integrate all these systems was primarily pushed by the necessity of introducing a Web Portal, with the main objective of selling both training courses and networking devices online. Thus, the final integrated software solution is composed of these four existing components, plus the Web Portal to be developed during the integration project.

The ERP solution was mainly used for controlling monetary expenses and profits *i.e.*, for billing, managing invoices, calculating balances, recording expenses from suppliers, management of customers credit, management of bank accounts, salary processing and the calculation of some periodical financial reports. The ERP solution presents a typical n-tier architecture from which one can clearly identify a database management layer, an application layer and a plug-in layer, upon which it is possible to develop new functionality. Although the source code of the ERP application is not available, the database is open and can be augmented with both new tables and procedures. The development of new functionality over the ERP is done via the plug-in layer which provides a fairly complete API for the main ERP entities and relations.

The CRM application was primarily used by the commercial and marketing departments for managing product and course campaigns (general and focused customers' campaigns), managing customer contacts, scheduling sales calls and mailings, tracking customers' responses to campaigns, perform budgets and client proposals. The CRM application in question is a software solution from one of the major players in the international CRM market. Because the CRM product intends to be as easy to integrate with other applications as possible, it provides a complete and well documented

API, which can be used to integrate all the existing CRM functionalities with third party solutions. Moreover, because in most cases the generic CRM solution as to be customised to the specific customer reality, the product provides a graphical user interface administration tool which aids the user in the development of new data structures and functionalities. Thus, the CRM application provides fairly good mechanisms for both the integration and extension of this solution with the rest of the systems.

The DMS system was used for the automation of some internal processes that are not handled by any of the other base components. This system is also used as an archiving solution for the different types of documents generated manually or by the other base components. In what respects to the manageability of this application, it provides a reasonably documented Standard Development Kit (SDK) which can be used to both interface and develop new functionality on this application.

The TS solution consists of a Web Application specifically developed to address the management of training courses. The main functionality provided by TS is the generation and management of the set of training sessions for each course. Each of these sessions record contains several different kinds of important information, like trainees and trainers absences, the type of training involved in training sessions, the time and duration of training sessions, the room it takes place, etc. This information is then used by TS to generate salary values for the trainers, fees for the trainees, exclusion of trainees under specific circumstances, to generate customised diplomas based on the actual hours attended and lectured, etc. Besides controlling all the details of a running course, the solution also provides mechanisms for trainees to communicate with each other and with the administration, as well as the possibility to implement politics for the selection of both trainers and trainees.

The TS solution was developed by a local software company which is willing to develop the new functionality required for the integration with the rest of the base components. Thus, in theory, there are no bounds for the adaptation of this system with the remainder software solutions.

The only existing connection between these base components is a one-way Web-Service invocation from the CRM to the ERP, which serves to send billing information from the CRM to the accounting department each time a proposal or budget has been accepted by a client. Apart from this creation of an invoice in the ERP from the CRM, every other base component lives in complete isolation.

The situation faced by the training company, having all the base components acting in isolation, led to numerous information synchronisation problems which had to be dealt with manually, at a daily basis. If this was a situation which could be manually treated in the past, with the recent growth of the company it is no longer feasible to maintain all the information synchronised manually. Thus, several incoherencies in critical company data inevitably started to emerge. A selection of the most important ones is presented bellow, to give a more concrete idea of what the integration project will have to deal with.

Base Comp. : CRM and ERP.

Entities : Products, courses and clients.

Issue : The insertion of a course, product or client in the CRM with no correspondence in the ERP and vice-versa.

Consequences : When a client approves a budget or proposal, some of the products or training courses may not be included in the invoice because the ERP ignores their existence. Products and courses may not be included in budgets and proposals because they only exist in the ERP and not on the CRM.

Base Comp. : CRM and ERP.

Entities : Product and courses

Issue : The courses and products details may not be updated consistently (or at all) in both the CRM and the ERP.

Consequences : A proposal or budget performed on the CRM which has been delivered and accepted by the client leads to the creation of an invoice from the ERP with different price values.

Base Comp. : CRM and ERP.

Entities : Pricing logic

Issue : The pricing logic may not be updated consistently (or at all) in both the CRM and the ERP.

Consequences : A proposal or budget performed on the CRM with a specific pricing logic (*i.e.*, taking into consideration the type of client, the sales volume, the quantity, the current client and product campaigns, etc) originates an invoice on ERP with different price values.

Base Comp. : CRM, ERP, TS, DMS.

Entities : Authentication and Authorisation

Issue : The authentication and authorisation policies can be introduced incoherently in each of the base components.

Consequences : A trainer in the TS may not have access to different documents regulating the training activities that are available in the DMS, because he does not have correct credentials to access the DMS. A CRM user who realises that some products or pricing logic is wrongly stored in the ERP, does not have access or permission to apply the necessary changes to the ERP.

Base Comp. : ERP, TS.

Entities : Salaries and Grants

Issue : The actual number of hours lectured by the trainers and received by the trainees may not be the same in the TS and the ERP.

Consequences : The TS solution calculates the actual hours lectured by a trainee (considering the absences, training places and other aspects) but, because the TS is not connected to the ERP, the employee who launches the salaries in the ERP may introduce errors in the salary calculation. Moreover, the TS solution performs a periodically runtime calculation of the hours to be payed to each trainee and trainer, based on the absences, justification of absences, co-lecturing of training modules, etc.

Base Comp. : CRM, ERP, TS.

Entities : Training and Courses

Issue : The training courses may not be inserted and updated correctly between the CRM, ERP and TS.

Consequences : A training course is being offered by some CRM campaign with some specific details (number of hours, training place, trainers, etc) and that same course is registered with different information in the TS, which is the system that actually determines the training course details.

This set of synchronisation problems is by no means a systematic description of all the functional operations that have to be taken care in the integration phase. In fact, many of these problems will have to be mitigated with several atomic integration operations (in this case using Web-Services), and some may even raise new integration problems that also have to be addressed in order to achieve the desired level of integration.

With all these synchronisation problems to fix between the different base components, and with the further need to include an E-Commerce solution, demanding the implementation of even more complex integration requisites, the company decided to embark on a EAI project.

11.3 Integrating Base Components

Like in any other software integration project, the first issue that has to be addressed is the definition of the integration architecture to be followed during the EAI implementation. A good software integration architecture should clearly identify the base components involved, a description of each base component in terms of the functionalities provided to the enterprise system (every internal detail of each base component should be omitted), the connections between the base components (where there may exist more than one connection between the same base components) and the specific details of each connection, in terms of the information flowing over the connection, the type of communication to be used (synchronous or asynchronous) and the properties that the connection must provide (atomicity, integrity, fault tolerance, etc).

There are several patterns and best practices [Lin00, Lin03, HW03] which facilitate the design and implementation of software integration architectures. Nevertheless every EAI project is different from each other, mainly because there are too many variables in stake that may dramatically influence the definition of the integration architecture. Among other aspects one may find that there are some details that differentiate EAI projects from regular software development project, namely, the level of customisation of each base component (potentially making each base component significantly different from every other integration project), the usually great size of base components and the different and often peculiar interfaces that have to employed to connect base components.

Given the objectives and definition of a good software integration architecture, it is clear to us that the use of coordination models and formalisms for their specification are essential to the validation of the desired properties in the final enterprise system. But, such is not the case in most EAI projects, and nor was it in our training company integration project. Nevertheless, even with an informal description of the integration architecture, it is possible to take advantage of the techniques presented in this thesis, as will be made clear in the remainder of this chapter.

The details and the process that led to the adoption of the integration architecture implemented in the training company, constitutes by itself an interesting discussion topic. Nevertheless, such issues are of no relevance for our objective of validating if the proposed integration architecture is correct with respect to what was actually implemented.

Before presenting the informal integration architecture used in the AEI project, we will first focus on some of the integration issues that arise from the insertion of the E-Commerce System (*ECS*) to the enterprise software system. Many of the integration problems concerning the ECS depend on the strategy used to solve the integration issues presented in the previous section, special the ones between the CRM and the ERP. Therefore, every time an ECS operation depends on the CRM and ERP integration, one will refer to these two base components as a single integrated one, referred to as CRM&ERP.

Base Comp. : ECS, ERP.

Entities : Invoices and Receipts

Issue : The invoices and receipts of every sale performed at the ECS must be in complete accordance with the invoices and receipts stored in the ERP.

Consequences : The company accounting may not reflect correctly the actual sales made by the ECS, registered on the receipts and invoices stored in the ERP.

Base Comp. : ECS, CRM&ERP.

Entities : Products and Courses

Issue : The products and courses together with the details of each may not be synchronised between ECS and CRM&ERP.

Consequences : The ECS may be offering courses and products that are no longer offered by the company nor register on the CRM&ERP. The details of courses and products, like price, stock and scheduling may be publicised incorrectly in the ECS.

Base Comp. : ECS, CRM&ERP.

Entities : Pricing and Campaigns

Issue : The pricing and campaigns logic must be the synchronised between the ECS, CRM&ERP.

Consequences : The ECS may be calculating course and product prices that do not reflect the current company pricing and campaign logic.

Base Comp. : ECS, CRM.

Entities : Users (Clients) and Commercial Activities

Issue : The portal users that have performed some commercial operation must be registered in the CRM.

Consequences : If a ECS user that has performed some commercial activity within the Web Portal is not registered in the CRM, than all the marketing and commercial analysis performed may be erroneous.

Base Comp. : ECS, CRM&ERP.

Entities : Discount Vouchers

Issue : The discount vouchers and current state must be kept coherent between the ECS and CRM&ERP.

Consequences : A discount voucher may be used twice or used beyond the validity date. The preconditions imposed on discount vouchers may not be enforced by the ECS.

Base Comp. : ECS, CRM.

Entities : Client Card

Issue : The client card and all the associated information must be kept synchronised between the ECS and the CRM.

Consequences : A newly issued card at the CRM is not accepted by the ECS impeding the registration of the products and courses bought as well as the calculation of prices and campaigns based on sales quantity and volume.

Note that, when not properly implemented, even a single of these integration issues may introduce great problems to the normal work of the company. Thus, all these issues must be taken into consideration by the integration architecture, and even more, their correctness should be properly verified.

To verify the different functional properties the enterprise system should expose, the only pieces of information available were the informal integration architecture presented in Figure 11.1 and the source code of the ECS and the TS system. Of course, the collaboration of the EAI implementation team was also available in order to clarify any details about the implementation.

An important detail was no access was given to the source code of the CRM, ERP, nor the DMS. We could inspect what services these system provided to the Web Portal, but we could not inspect the logic behind each of the services in question.

The first observation that emerges from the EAI architecture in Figure

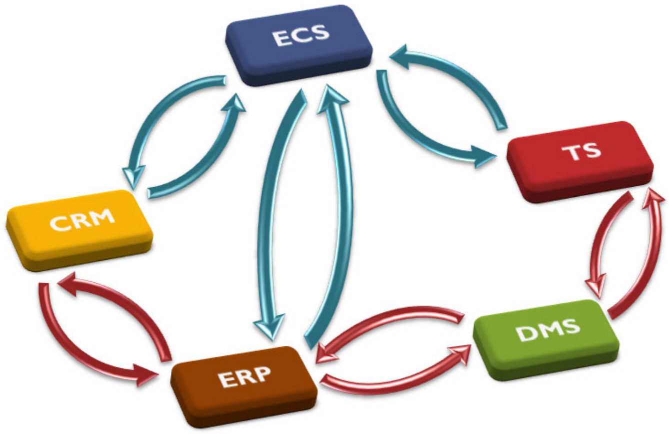


Figure 11.1: EAI architecture

11.1, is that it uses a *point-to-point* integration style [Lin00], instead of a centralised style, like a *message broker* or a *database oriented middleware* [Lin00], as one might have expected. The problems associated with a point-to-point integration are various, and range from the lower level of scalability, given the potential great number of connections to be implemented between systems[1], the difficulty in coordinating connections involving different base components and the difficulty to maintain the system secure under so many connections. Therefore, for an EAI project starting with five base components and with potential to include more of them in the future, it seemed a rather strange decision that the development team opted for the *point-to-point* integration style.

When questioned about this choice, the development team argued that, although they were aware of its limitations, the *point-to-point* style gave them more control during the project development, since once completed, each integration connection could be tested and deployed to the working environment. This way, the team could not only control the quality and effectiveness of each milestone in the AEI project, but also to change the users' habits in a smooth way. Another important argument for choosing the *point-to-point* style was that the company administration realised that such

[1] In the worst case, for the integration of n base components, this figure may rise to $\frac{n(n-1)}{2}$ connections.

an integration architecture would reduce dramatically the risk of success of the entire project. In practice, the administration could assess the success of the project regularly by inspecting the connections already achieved and the impact they have in the overall company activity.

If a centralised integration style were chosen, the *middleware* system would have to be completely specified and implemented (with all the intermediate data structures, and integration operations), before one could be able to assess the correctness and effectiveness of the entire enterprise system. Furthermore, although a centralised style would have many functional and non-functional advantages over the *point-to-point* one, it also introduces a single point of failure in the entire enterprise system, a risk that in this case was found to prevail over the drawbacks of the point-to-point style.

The EAI architecture defines that the ECS component should communicate and guaranteed the synchronisation of information with the CRM, ERP and TS systems. In such a model, the ECS component does also have to take into consideration the integration of both operations and data between the components it interacts with. Therefore, although the red arrows in Figure 11.1 may lead one to think that the integration between the CRM, ERP and TS components is already being taken care of, in fact they are just resolved for the user interface operations and not for the supplied service operations, like the ones invoked by the ECS. Thus, giving this lack of integration between the ERP, CRM and TS systems at the service level, the ECS integration logic has to guarantee, for instance, that whenever a product is inserted in the ECS it has also to be inserted in both the CRM and ERP, and in the case of a training course, in the CRM, ERP and TS.

Given the architecture and material provided by the company, we will concentrate on the blue arrows in Figure 11.1 *i.e.*, in the connections between ECS and the CRM, ERP and TS components. In particular we will focus on the outgoing arrows from ECS to the other components, given that we have full access to the ECS source code and to the informal specification of how these connections should behave. In what respects to the incoming arrows to the ECS component, we assume they were provided for each operation and that the programming logic associated to each invocation correctly implements the behaviour expected by the base component responsible for the invocation.

11.4 Coordination Patterns

In order to verify the coordination of each operation implemented in the ECS that involves some, or all, of the remainder base components, one must first clearly identify the operations to be addressed and derive its representation in the CDGPL language. Many of the operations to be verified have already been informally presented in the previous sections. Nevertheless, now, one needs to formalise such operations in CDGPL in order to achieve a rigorous definition of the integration operations, as well as to permit the use of CoordInspector for the verification of the implemented operations.

The remainder of this section presents each coordination operation both in CDGPL and informally. Note that, the synchronisation strategies to be described are not always the best solutions for the problems in hands. Rather, they are the solution described in the integration architecture (sometimes, they were just orally transmitted by the development team) which one will later try to verify.

In the sequel the acronym *CUD* will be used, referring to the topical *CRUD* acronym (*Create*, *Read*, *Update* and *Delete*) without the *Read* operation. The reason to discharge the Read operation is that almost every base components has a local version of the information it needs to operate upon, making Read operations rarely invoked between base components. This architectural decision intends to reduce significantly the time to perform data read operations, which are thought to be the vast majority of operations performed by every base component.

The notation $n_1 \rightarrow n_2$ will be used to denote an edge from vertex n_1 to vertex n_2. The following definitions are also used to ease the description of the regular expressions associated to each vertex.

$R =$ "System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke"
$arg =$ "$, \backslash s * \backslash w \backslash s *$"

$$\begin{aligned} \mathit{CallWs}\ (ws, n) = {} & \mathit{PrimMeth} + \text{``}\backslash s * \backslash(\backslash s * \backslash''\text{''} + ws + \text{``}\backslash''\backslash s *\text{''} + \\ & (\mathit{concat}\ .\ \mathit{replicate}\ n\ \$\ \mathit{arg}) + \text{``}\backslash)\backslash s*;\text{''} \end{aligned}$$

The regular expression syntax [Stu07] used is a rather universal one, that can be found in many languages, such as Perl, JavaScript, $C^\sharp$ and Java. In the definition of the macros we also use the HASKELL list function *concat*, which concatenates a list of elements, and *replicate*, which receives an integer n and an element e and returns a list containing n repetitions of the element e.

The subsequent calls to web-services used in each coordination pattern are always performed with two arguments. This is not a simplification of the real calls used in the implementation, but reflects the design decision which defines that every web-service call should use only two arguments. The first argument is an XML document and serves to pass any number of arguments to the service being called, while the second argument serves to pass a security token which is analysed by the receiving component in order to verify the integrity and sometimes the confidentiality of the first argument. The main advantage of this remote call policy is that every component uses the same security and integrity mechanism to verify the integrity of the received calls. The main drawback of such a decision comes from the impossibility of performing a static type check on the arguments of each remote call.

11.4.1 Op1 – Profile CRU

A profile defines a set of authorisation policies that can be associated to users in order to grant them operational privileges in the CRM, ERP and ECS. The TS system profiles are bound to its business logic and cannot be changed, so there is no need to synchronise any profiles with the TS component. The CDGPL description of the profile creation operation is presented in Figure 11.2.

The creation operation is performed by spawning two different threads (y and z) each of which is responsible for invoking a web-service to create the profile on the CRM and ERP respectively. The loop in vertices 1 and

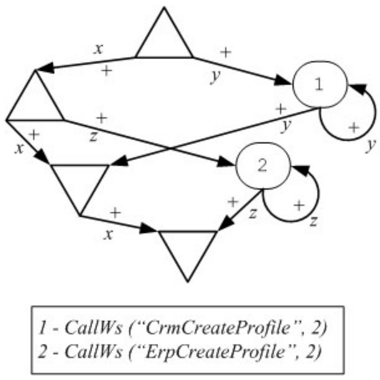

x
+
+
y
1
+
y
+
y
+
z
x
+
2
+
z
+
z
+
x
1 - CallWs ("CrmCreateProfile", 2)
2 - CallWs ("ErpCreateProfile", 2)

Figure 11.2: Profile creation operation

2 refers to a recurring implementation technique used by the development team, which serves to re-execute the creation operation in case an error or exception occurs in the previous try.

One of the first observations made once this pattern was discovered in the ECS implementation, was that such a coordination operation could lead to multiple creations of the same profile in both the ERP and CRM. More specifically, the replication of profiles can occur when a response from the invocation of the methods in vertices 1 or 2 is not received. In such a case, the pattern re-executes the insertion operation over the base components. The problem with this coordination behaviour is that a non received response from a remote operation does not necessarily mean that the operation has not been performed remotely, it only means that the response did not reach the ECS component. Therefore, in a situation where only the response had been lost, the coordination pattern leads to repetitive insertions of the same profile.

The possible unintended behaviour in the creation of a profile was communicated to the development team, who, even though had agreed with our interpretation, did not made any changes to the implementation of the discovered coordination pattern. The explanation for not changing the implementation is twofold. First, the insertion of repetitive profiles, in both the ERP and CRM, does not introduce any direct problems to the behaviour of these systems, since the profiles are only used to attribute authorisation

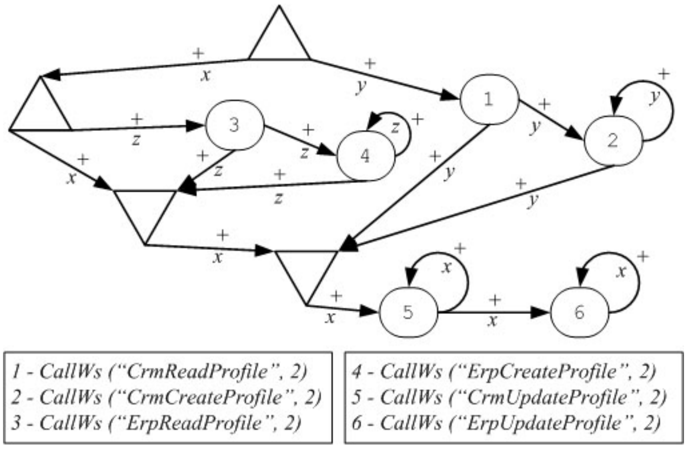

+
x
+
y
+
z
3
+
z
4
z
+
1
+
y
2
y
+
+
x
+
z
+
z
+
y
+
y
+
x
+
x
+
x
5
+
x
+
x
6
x
+
1 - CallWs (“CrmReadProfile”, 2)
2 - CallWs (“CrmCreateProfile”, 2)
3 - CallWs (“ErpReadProfile”, 2)
4 - CallWs (“ErpCreateProfile”, 2)
5 - CallWs (“CrmUpdateProfile”, 2)
6 - CallWs (“ErpUpdateProfile”, 2)

Figure 11.3: Profile update operation

credentials to users. Secondly, the mitigation of the problem would led to a decrease in performance of the profile insertion operations, given that one would have to use transactional web-services calls or enhance the coordination with more Web-Service calls.

The profile update operation, presented in Figure 11.3 is somewhat more complex than the previously presented creation operation. The reason for having a more complicated logic derives from two special situations that have to be handled properly. First, one must cope with the situation where the profile to be updated does not exist in the CRM or ERP. In this case, one must first create (vertices 2 and 4) the profile, before updating it (vertices 5 and 6). The second situation occurs when the information associated to the same profile does not coincide in both the CRM and ERP systems. To overcome this problem, one has to merge the information from the CRM and ERP[2], a procedure that is performed between the second join and vertex 5.

From the CDGPL definition and the discovered implementation instance, one can make several observations about the details of the profile update operation. First, the actual update procedures (vertices 5 and 6) could be performed in parallel, reducing the amount of time to perform the update operations. Even more, based on the observations made from the profile creation, the update operation could lead to the replication of profiles. Another

[2] In case of conflict it was decided that the CRM information prevails over the ERP.

aspect of the pattern, is that it does not update the profile in the CRM nor in the ERP, in the case where one of these systems did not answered (or the answer was lost) upon the invocation of the *read* or *create* operations (vertices 1, 2, 3 and 4). This changes the accomplishment of this synchronisation operation from a single point of failure to two points of failure.

From these observations, only the first issue, concerning the performance of the pattern, led to a modification of the implementation. Again, the remainder changes were not implemented due to the nonexistent impact of the change in the functional properties of the system and because the complexity and potential decrease in performance associated to them could influence the overall system performance.

Note that, the profile removal operation follows a strategy very similar to the creation of profiles, with the exception of resorting to different Web-Services in vertices 1 and 2.

11.4.2 Op2 – User CRU

In theory, the registration of a user in the ECS should always lead to its registration in the CRM, ERP and TS, so that these systems may store commercial, accounting, and training information associated to the created user. However, because the ECS user registration is open to the public, everyone can register himself at the ECS, leading to a potential great number of new user registrations per day.

Under a complete synchronisation of users, the CRM, ERP and TS systems would be constantly flooded with requests concerning user creations and updates. Moreover, for the vast majority of the users being introduced the information associated with them would be of no relevance for the CRM, ERP or TS. So, instead of keeping user data synchronised between systems, the integration architecture defines that only the ECS users with any relevance towards some other base component *i.e.*, that are actually involved in some logic concerning one of the other base components, should actually be synchronised with the base component in question.

In order to analyse the coordination involved in a user creation operation,

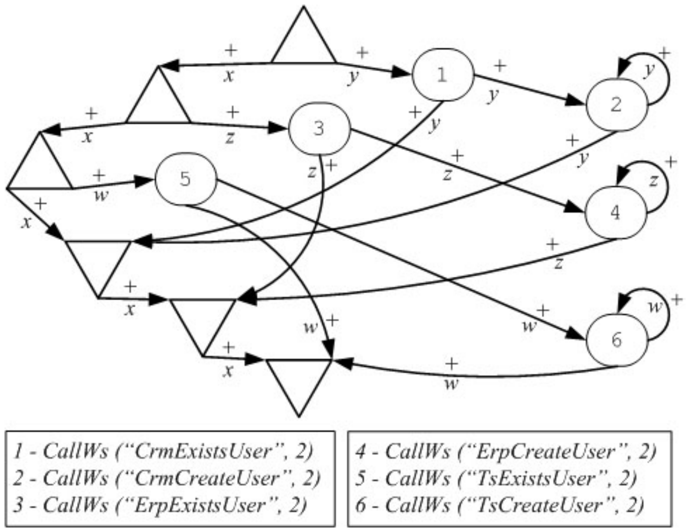

1
2
3
4
5
6
x
y
z
w
+
1 - CallWs ("CrmExistsUser", 2)
2 - CallWs ("CrmCreateUser", 2)
3 - CallWs ("ErpExistsUser", 2)
4 - CallWs ("ErpCreateUser", 2)
5 - CallWs ("TsExistsUser", 2)
6 - CallWs ("TsCreateUser", 2)

Figure 11.4: User create operation

we shall focus on the most complex case of this procedure, *i.e.*, the case where the user plays a role and has to be created in all three base components (the CRM, the ERP and the TS). The other cases are easily captured by a simplification of the presented pattern.

Figure 11.4 presents the implemented coordination logic for this operation. The behaviour is similar to the one found in the creation of a profile, but this time, instead of dealing with only two base components, the behaviour has been extended to three. Another important difference between this create operation and the previous one, is that each call to a remote creation operation (vertices 2, 4, 6) is always preceded by a remote call to check if the user already exists in the base component in question. If the user is found in the base component, the remote creation operation is not carried out (edges between vertices 1, 3, 5 and the respective join vertices), otherwise the pattern performs a remote call to the user create operation.

Given that this coordination pattern was implemented as a reuse of the previous case concerning profiles, the strategy to cope with failures in the creation operation amounts to the re-invocation of the remote create call (loop edges in vertices 2, 4 and 6). By using this same resilience strategy, one also inherits the replication problem of the entities being created. But,

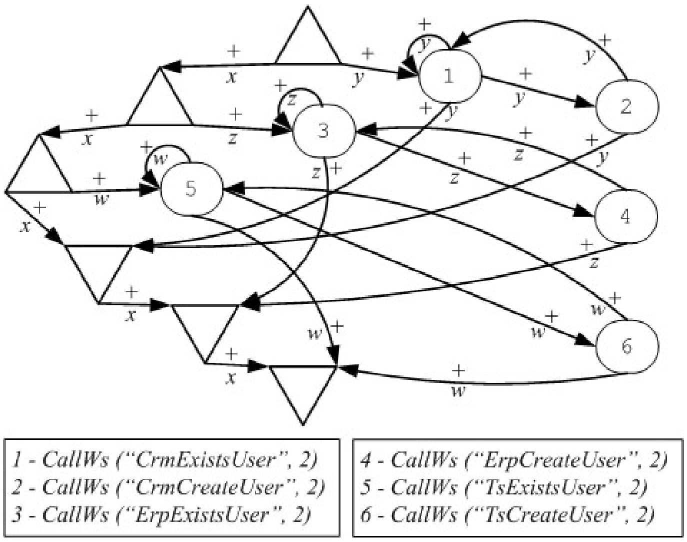

1
2
3
4
5
6
+ x
+ y
+ z
+ w
1 - CallWs ("CrmExistsUser", 2)
2 - CallWs ("CrmCreateUser", 2)
3 - CallWs ("ErpExistsUser", 2)
4 - CallWs ("ErpCreateUser", 2)
5 - CallWs ("TsExistsUser", 2)
6 - CallWs ("TsCreateUser", 2)

Figure 11.5: Corrected user create operation

unlike the previous case, it can be very problematic to have replicated users inside the enterprise system. If such a replication occurs, the entire enterprise system integrity could be at stake, given that, for instance, relevant information about a single user information could be scattered around its replicates.

To fix this problem one has changed the coordination model, transforming the loop edges 2 → 2 , 4 → 4 and 6 → 6 to edges 2 → 1, 4 → 3 and 6 → 5. This way, before re-invoking the create operation, the pattern always performs a check to verify if the user already exists in the relevant base component.

The resilience of the model can also be improved by introducing loop edges in the existence check operations (edges 1 → 1, 3 → 3 and 5 → 5). The advantages of the transformed model, presented in Figure 11.5, were clearly understood and accepted by the development team, who changed the implementation according to the new design.

In what respects to the update of users, the development team chose once more to reuse the coordination pattern of the profile update case. Again, in the more complex case of this operation, the pattern has to interact with all three base components. Figure 11.6 presents the coordination model

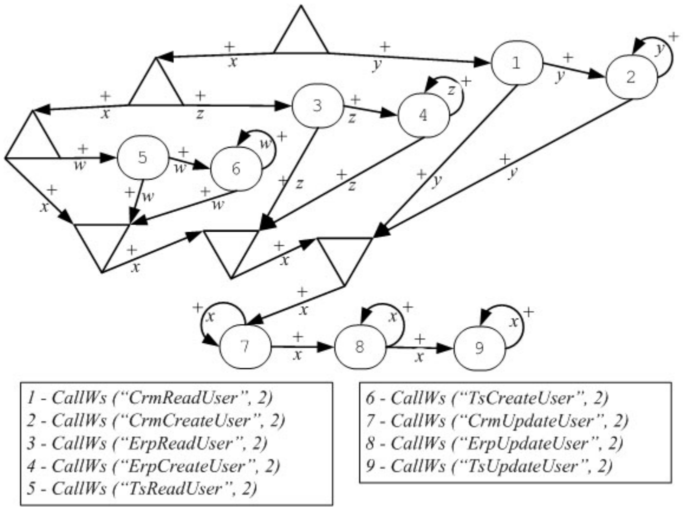

1
2
3
4
5
6
7
8
9
+ x
+ y
+ z
+ w
1 - CallWs ("CrmReadUser", 2)
2 - CallWs ("CrmCreateUser", 2)
3 - CallWs ("ErpReadUser", 2)
4 - CallWs ("ErpCreateUser", 2)
5 - CallWs ("TsReadUser", 2)
6 - CallWs ("TsCreateUser", 2)
7 - CallWs ("CrmUpdateUser", 2)
8 - CallWs ("ErpUpdateUser", 2)
9 - CallWs ("TsUpdateUser", 2)

Figure 11.6: User update operation

implemented in the integration project implementation. Note that, this is a straightforward extension to three base components, of the update operation presented above. The discovery and specification of the model in terms of the pattern presented, clearly evidences the possibility of repetitive creation of users as well as some room for performance improvements.

In what concerns to the problem of repetitive creation of users, one can apply the previous solution and insert extra vertices for checking if a user already exists in the relevant base components. However, in this case, the calls to the remote creation operations (vertices 2, 4 and 5) are always carried after a read operation (vertices 1, 3 and 5), which makes the first remote call to the creation operation to be aware of the existence, or not, of the user. Thus, only the subsequent remote creation operations (executed by following the loop edges 2 → 2, 4 → 4 and 6 → 6) suffer from the problem of inserting duplicate users. Therefore, one has made a small change to the previous solution to this problem, and opted to insert the extra user existence check vertices (vertices 3, 6 and 9 in Figure 11.7) after the remote creation calls.

The actual remote update of a user is only performed at the very end of this coordination pattern, in vertices 7, 8 and 9. Moreover, the updates are performed in a single thread, making each previous call to possibly introduce delays in all subsequent remote calls, eventually resulting in significant delays

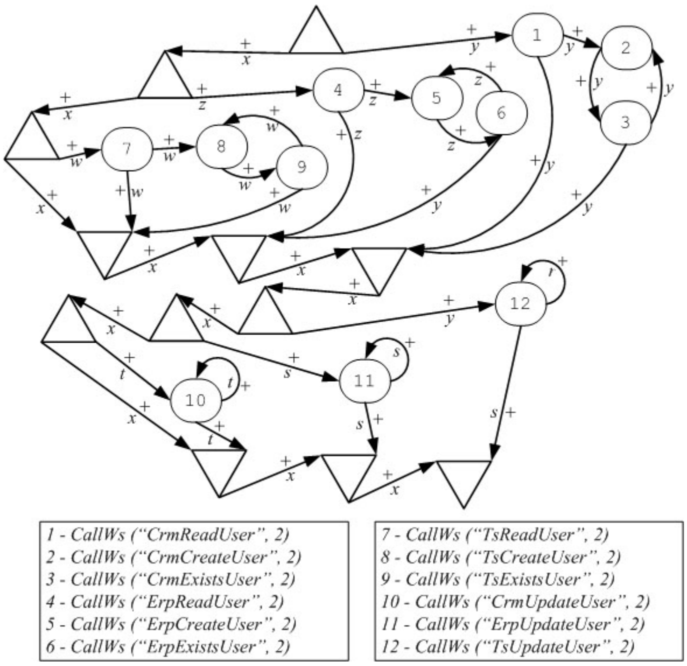

1
2
3
4
5
6
7
8
9
10
11
12
1 - CallWs ("CrmReadUser", 2)
2 - CallWs ("CrmCreateUser", 2)
3 - CallWs ("CrmExistsUser", 2)
4 - CallWs ("ErpReadUser", 2)
5 - CallWs ("ErpCreateUser", 2)
6 - CallWs ("ErpExistsUser", 2)
7 - CallWs ("TsReadUser", 2)
8 - CallWs ("TsCreateUser", 2)
9 - CallWs ("TsExistsUser", 2)
10 - CallWs ("CrmUpdateUser", 2)
11 - CallWs ("ErpUpdateUser", 2)
12 - CallWs ("TsUpdateUser", 2)

Figure 11.7: Corrected user update operation

to the overall remote operation. This single thread sequence of remote calls also demands for a rigorous exception and error handling, given that each call may influence the subsequent calls and consequently the entire operation. In this case-study, once the pattern was discovered, one had to manually inspect for the proper exception and error handling, since CDGPL does not have yet the capability of representing this kind of programming logic.

In the corrected coordination model, presented in Figure 11.7, the sequence of update calls are transformed in parallel calls, so that each call cannot introduce delays nor influence the others. Both transformations proposed in this section were accepted and used by the development team to modify the implementation, whose performance revealed to be improved significantly in every user update operation.

Although one has achieved the improvement and verification of the implemented coordination strategies, there is still the possibility for a remote call to continually fail. In such a case, each of the previous models would not only fail, but, what is worse, enter in a deadlock situation, which could ultimately lead to a complete halt of the entire ECS solution. Note that the discovery of such deadlock situations can be performed by a graph loop discovery algorithm.

To overcome deadlock situations present in every studied coordination pattern, one has introduced a counter for each of the discovered loops and include a guard (or extend one, if a guard was already there) in the loop that inspects the number of cycles performed. In case one of these loops has overcome the maximum number of cycles allowed, the guard not only guarantees that the program control leaves the loop, but also that the operation not carried out is written to an *error log* table.

The deadlock removal strategy introduces a mechanism for error recovery as well as it enables the introduction of different amounts of tries for each remote call. Furthermore, the *error log* table can be used, for instance, to run a periodically *batch* job responsible for the re-invocation of failed operations.

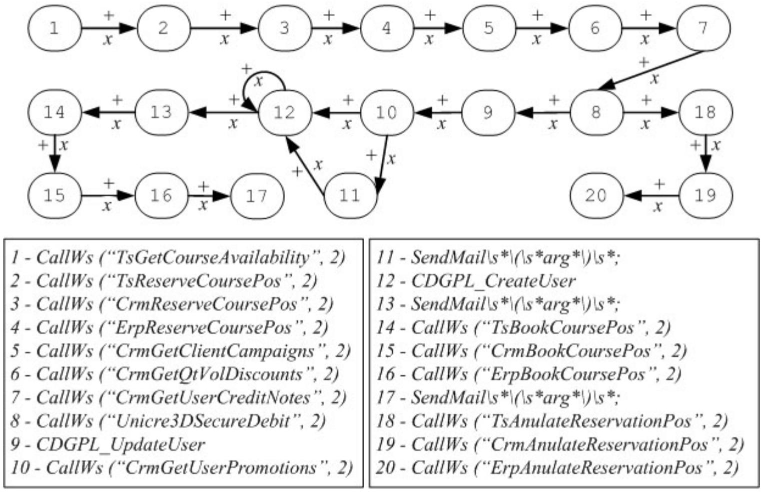
The remainder entities CRU integration operations present in the enterprise system were found to be implemented using coordination patterns quite similar to the ones discovered for the user and profile cases discussed here. In each case, one has identified each pattern and suggested similar modifications for the improvement of the overall integration project.

11.4.3 Op3 – Multiple Sale of Training Courses

To close the discussion of this case-study, we shall now consider a more complex coordination operation: the online sale of a set of training courses to be performed by the ECS.

It might seem strange that someone would ever buy a set of training courses online, however, it often happens when, for instance, enterprises are purchasing training for their employees or associates. Actually, most of the training course sales made by the training company consist of multiple course purchases. Thus, this operation should not just be verified but also improved if possible.

The set of courses referred in this operation, is actually composed by a set of possibly different training course with possibly different quantities for



1 - CallWs (“TsGetCourseAvailability”, 2)
2 - CallWs (“TsReserveCoursePos”, 2)
3 - CallWs (“CrmReserveCoursePos”, 2)
4 - CallWs (“ErpReserveCoursePos”, 2)
5 - CallWs (“CrmGetClientCampaigns”, 2)
6 - CallWs (“CrmGetQtVolDiscounts”, 2)
7 - CallWs (“CrmGetUserCreditNotes”, 2)
8 - CallWs (“Unicre3DSecureDebit”, 2)
9 - CDGPL_UpdateUser
10 - CallWs (“CrmGetUserPromotions”, 2)

11 - SendMail\s*\(\s*arg*\)\s*;
12 - CDGPL_CreateUser
13 - SendMail\s*\(\s*arg*\)\s*;
14 - CallWs (“TsBookCoursePos”, 2)
15 - CallWs (“CrmBookCoursePos”, 2)
16 - CallWs (“ErpBookCoursePos”, 2)
17 - SendMail\s*\(\s*arg*\)\s*;
18 - CallWs (“TsAnulateReservationPos”, 2)
19 - CallWs (“CrmAnulateReservationPos”, 2)
20 - CallWs (“ErpAnulateReservationPos”, 2)

Figure 11.8: Training courses sale operation

each one. The online sale of a set of courses follows a similar process to the one employed in the traditional (presential) case. The main activities to be performed on such operation can be resumed by the following steps.

1. Check availability for every course;

2. Reserve training course positions to ensure that the courses will be available by the end of the selling process;

3. Check current course and client campaigns. If any current campaign applies to one of the courses or to the client, then it should be applied;

4. Check quantity and volume discounts. Again, if any of these apply to this sale, the value should be deducted in the final price;

5. Check client credit notes in the ERP. If there are any, they should also be applied;

6. Calculate the total price based on the previous numbers;

7. Collect payment from the client;

8. If payment succeeded;

   (a) Issue and invoice and a receipt from the ERP;

   (b) Update purchasing user (client) information in the CRM, ERP and TS;

   (c) If user achieved any new promotional benefit with the current sale;

      i. Send an email to the user reporting the promotional benefit achieved;

   (d) Create an user entry in the TS system for every training course position purchased;

   (e) Collect the TS users' login information and email it to the purchasing user (client);

   (f) Transform the training course reservations in the CRM and ERP to actual training bookings;

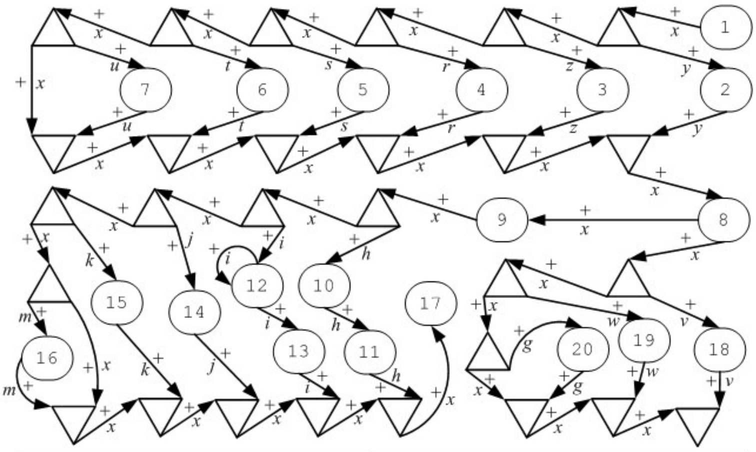
   (g) Send a notification (by mail, if possible) to the client with the trainees' login and purchase details (invoice, receipt and other relevant information);

9. If payment did not succeed;

   (a) Remove training courses reservations.

This workflow concerning the sale of training courses is full of integration issues between all four base components, which have to be properly taken care of in order to guarantee the correctness of the entire operation. Figure 11.8 presents the pattern that was actually found in the integration implementation code. Note that, although the pattern may seem simplistic at first sight, vertices 9 and 12 are actually reuses of the previously presented patterns for users update and creation respectively. Even more, one of this pattern reuse vertices, vertex 12, contains a loop capturing the iterative creation of users (the trainees of the purchased courses) in the ERP, CRM and TS.

The first observation one can make about the coordination pattern in Figure 11.8 is that, even though the user update and create operations are multithreaded, the actual sale operation is entirely performed on a single thread. This, of course, facilitates considerably the correctness verification of the implementation but, it introduces a great performance penalty in the overall sale operation. Moreover, the training course sale operation contains several activities that do not depend on each other and, if handled properly, can be run in parallel.



1 - CallWs ("TsGetCourseAvailability", 2)
2 - CallWs ("TsReserveCoursePos", 2)
3 - CallWs ("CrmReserveCoursePos", 2)
4 - CallWs ("ErpReserveCoursePos", 2)
5 - CallWs ("CrmGetClientCampaigns", 2)
6 - CallWs ("CrmGetQtVolDiscounts", 2)
7 - CallWs ("CrmGetUserCreditNotes", 2)
8 - CallWs ("Unicre3DSecureDebit", 2)
9 - CDGPL_UpdateUser
10 - CallWs ("CrmGetUserPromotions", 2)
11 - SendMail\s*\(\s*arg*\)\s*;
12 - CDGPL_CreateUser
13 - SendMail\s*\(\s*arg*\)\s*;
14 - CallWs ("TsBookCoursePos", 2)
15 - CallWs ("CrmBookCoursePos", 2)
16 - CallWs ("ErpBookCoursePos", 2)
17 - SendMail\s*\(\s*arg*\)\s*;
18 - CallWs ("TsAnulateReservationPos", 2)
19 - CallWs ("CrmAnulateReservationPos", 2)
20 - CallWs ("ErpAnulateReservationPos", 2)

Figure 11.9: Improved training courses sale operation

The time penalty introduced by the lack of concurrency not only diminishes the user satisfaction when purchasing training courses, which by itself constitutes a strong motive for changing this operation, but it also increases the chances of incurring in an error or exception. In a situation where the time to finalise the sequential sale operation is n times slower than its concurrent equivalent, it is more likely that a communication error will occur, or that the session storage becomes corrupted or even times-out[3]. Thus, al-

[3] Note that, since this integration project is using Web-Services, the communication is performed via HTTP, a stateless and disconnected protocol that demands the intervening systems to maintain sessions and time-outs to regulate all communications.

though the discovered coordination pattern was not found to bare any *functional* errors, the performance *non-functional* property may induce future functional errors.

In order to overcome the performance penalty detected in this operation, and at the same time to mitigate future functional errors, one has proposed the modification of the previous coordination pattern to the one presented in Figure 11.9. The main modification concerns the introduction of concurrency between every independent activity. However, some activities still have to be performed in sequence, for instance, vertex 9 is still sequentially executed after vertex 8, because the update activity associated to the former depends on the success of the payment operation contained in the latter.

The examples discussed here of discovered and re-engineered coordination patterns in the integration code for this EAI case study, illustrate how the methods and tool proposed in this thesis may help the working software engineer in practice.

The focus of our work, as described in the last 3 chapters, was on the identification and extraction from source code of coordination policies and their abstractions on a graph structure — the CDG. From there they could be rendered as coordination specifications in Orc or WS-BPEL, or else as particular graph patterns with direct correspondence to the source code. An obvious second stage in this process would be the development of calculi and tools to *transform* such coordination specifications.

Some coordination models have already such calculi available (for example Orc and Reo) and some form of tool support. Although such second stage is clearly out of the scope of our thesis, this case study actually makes the case for further research on the integration of both phases of

- discovery/analysis
- and re-engineering by model transformation

whose relevance for the engineering of legacy software cannot be understated.

Chapter 12

Conclusions and Future Work

12.1 Discussion of Contributions

The second part of this thesis addresses the identification, extraction and recovery of the coordination model entangled inside legacy software systems. Such layer is often spread among various parts of a system and, even more problematic, it is usually mixed up with code devoted to implement internal computations.

The clear understanding of how a software system makes use and manipulates third-party entities is of extreme importance to the (re-)construction of the system software architecture. The importance of this coordination understanding is getting increasingly relevant as software systems are being, more and more, built on top of external services and components.

We have introduced a method, based on slicing and graph analysis, for reverse engineering of software systems' coordination layers. The method is based on the notions of Managed System Dependence Graph (MSDG) and Coordination Dependence Graph (CDG), two program representation structures which characterises the different program entities used in the code and captures several types of dependencies between them.

An important aspect of our reverse engineering process is that it is parametric on the type of coordination it abstracts. This feature enables the process, when parameterized accordingly, to extract, for instance, the web-service coordination layer of a system or its distributed object calling model, or even its multithread coordination layer. Even more, it is possible to analyse more than one of these types of coordination layers, given that the appropriate parameterisation of the communication primitives is passed to the labelling phase. Due to the language heterogeneity that most real world systems present, the "language agnosticism" of the technique stands as another very important feature. However, it should be pointed out that we have not obtained a complete language independence, since that would imply the technique to cope with unstructured[1] languages as well as with languages not providing a precise definition of statements. Nevertheless, most used (commercial) languages do not hold such characteristics, which make them possible targets of our analysis process.

The analysis process is divided into two parts. The first one consists of a generic processing phase which delivers a graph representation of the system (the CDG), focusing on the specific coordination aspects identified by the rules which parameterize the process. The second part deals with the generation of concrete coordination specifications expressed in specific coordination modelling languages. Depending on the language or formalism chosen to represent the discovered coordination, the first part of the process is maintained unchanged whilst the second part needs to be rebuilt in order to generate the desired format.

The coordination languages ORC and WS-BPEL where chosen to express the recovered coordination policies. Nevertheless, the whole method can be adapted to target other coordination specification languages.

In what respects to the second part of the method, the generation of ORC or WS-BPEL specifications is quite straightforward. Actually, this phase resorts to a translation of the extracted information into a small set of ORC and WS-BPEL behavioural patterns. This, can sometimes lead to big and repetitive specifications that demand further simplifications to facilitate understanding and re-engineering. In the ORC case, the language is accompanied by a well defined formal semantics and a calculational framework, which makes the manipulation and transformation of specifications easy. Even in

[1] Languages containing arbitrary jump statements, like the `GO TO` expression.

the WS-BPEL case, where a formal semantics is still lacking, there is a number of analysis tools which may also play a significant role in the understanding and transformation of such coordination specifications. Nevertheless, it should be possible to tune the whole generation process in order to make it driven by more complex and well-known coordination patterns. Moreover, such an improvement would much facilitate the coordination analysis of systems, since the coordination specification would become less verbose and its analysis could be based on previous knowledge about such coordination patterns.

That was the reason why we developed a second method for the discovery of coordination policies based on coordination patterns. Unlike the first two approaches, more syntactically oriented, this one is based on well-known coordination patterns that must be previously encoded in a special language developed for the effect. The approach then identifies instances of such coordination patterns, resorting to a sub-graph pattern detection algorithm specifically developed for this purpose.

Although the most direct application of our algorithms and tool serve to assist on the coordination analysis of legacy systems, they can also be used to assess the correctness of systems implementations with respect to their design specifications or even with respect to the growing software quality regulations. Even more, with the provision of rules for COM or RMI communication discovery, it can be used to assist the conversion of legacy distributed object systems to web-service oriented systems (or vice versa).

Many of the ideas presented in the second part of the thesis are implemented in CoordInspector, a tool targeting Microsoft .Net Framework systems. The tool is available from `http://alfa.di.uminho.pt/~nfr/Tools/CoordInspector.zip`, and besides providing many of the operations presented it is also capable of displaying and navigating through the graph structures computed along the analysis. Given that the tool analyses Common Intermediate Language (CIL) code, it is potentially capable of performing coordination analysis in every programming languages compilable to the .Net framework.

In order to validate the applicability of the techniques proposed, we have applied our pattern based coordination analysis to a real project of software integration. The project amounted to the integration of five different types of applications, where web-services were used as the primary communication primitive. There were two primary outcomes of this experience. First, it was possible to confirm the implementation was respecting many of the assumptions made in the (informal) design of the project. Moreover, this confirmation was made with strong evidences collected from the actual implementation code, which, of course, lead to a greater degree of confidence in the integration process. Secondly, it was possible to identify specific problems that in some cases arose from the design of the operations, where in other cases, the implementation was found to be the main responsible for the possible for the erroneous behaviour. Again, every possible behavioural deviation report was supported by strong evidence collected from the implementation code.

Overall, we regard this work as part of the broad area of software architectural analysis, where the ultimate goal is the discovery of the business process orchestration logic laying beneath a software system implementation. We strongly believe that the techniques, like the ones presented in this thesis, contribute to the correct discovery of business processes (or even to perform it automatically), and to the evolution of such systems towards the (web) service oriented world or to future coordination paradigms.

12.2 Future Work

An interesting topic for future work is the classification of orchestration patterns, as in [AHKB03], and their representation in the *Coordination Dependence Graph Pattern Language*. Such categorisation of coordination patterns would facilitate not only the development of new software systems but also provide a basis for CoordInspector and similar tools.

Another interesting improvement would be to allow changes to be made on the generated specifications (in this case, in Orc or WS-BPEL) and, based on such changes, regenerate equivalent transformations to be applied to the original source code. Such a development would permit to implement a *round-trip* behaviour in the coordination analysis tool, which would help to assess the impact of certain coordination design decisions.

Although the presentation of our analysis process bifurcates into a more traditional code generation technique and a pattern based one, we do not regard these two approaches as incompatible. In fact, even though we have not tried to combine them, we believe that the use of the pattern based technique, when applicable, together with the more straightforward code generation for the remaining code, could lead to an even greater quality degree of the discovered coordination logic. This raises, of course, many questions for future work, such as, how to perform code generation without considering the code fragments discovered by the pattern base approach? Which would then be a suitable set of coordination patterns to be used in such combination? Should the code generation process be aware of the pattern based approach or could we compose this two approaches sequentially.

CoordInspector was our laboratory for most of the ideas and techniques discussed in this thesis. Thus, it is of utmost importance that this laboratory evolves in parallel with all other future work lines to validate their applicability and potential for analysis.

12.3 Related Work

To the best of our knowledge, there is no previous research directly related to our work on the specific problem of discovering and extracting coordination logic from a system's source code. However, there is a number of works addressing architectural recovery from legacy software that should be mentioned at this stage.

The Alborz system [SYS06, SDS06] centres its software architectural recovery strategy in both dynamic and static analysis techniques. While the former strategy completely diverges from our work, the latter has similarities to our pattern-based coordination discovery technique. In particular it resorts to architectural patterns (which seem also to be defined as graphs) to guide the discovery process. However, little detailed is given about neither, the internal representation of the system to be analysed nor about the description or expressiveness of the architectural patterns. Moreover, the Alborz system is unable to analyse, nor exploit, object-oriented and concurrent concepts.

In [Bou99], Boucetta et al. propose a method for recovery of software architectures based on both bottom-up and top-down analysis techniques. This approach may be considered somehow related to ours, in the sense that we also use bottom-up techniques, by calculating special purposed graph structures directly from source code, and top-down ones, by defining high-level coordination patterns to be discovered over the code graphs representation. However, unlike [Bou99], our focus is centred on the recovery of coordination logic, an aspect which dramatically influences and drifts our definitions of architectural patterns and code representation structures from the ones presented in [Bou99]. Moreover, no details are presented in [Bou99] about intermediate source code representations, architectural pattern languages or the algorithms to perform a match between architectural patterns and the actual system under analysis.

Reference [MMCG02], presents a series of improvements to the Bunch software architecture system. The tool produces *Module Dependency Graphs* (MDG), which are computed with automatic module clustering techniques. The improvements presented are mainly concerned with the possibility of manually introducing clustering information about the system, with the objective of retrieving more accurate structures. The points of contact with our own work are limited, mainly concerned with the manual tuning of the architectural discovery process. Nevertheless, the techniques used in Bunch could also be used in our component discovery case study, presented in chapter 4 from the first part of this thesis.

12.4 Epilogue

We have now reached the end of this thesis. As the title indicates, the main objective of our work was to investigate the application of slicing techniques to the extraction of high-level models describing the underlying architecture of legacy software systems. This led us to a journey that started on the study of the available slicing theories, algorithms and implementations. Our own interest in Functional programming and the fact that this paradigm is often neglected in mainstream slicing research, justifies our initial work on slicing techniques for functional programs which is reported in the first part of this thesis. This experience also served as a training bed for grasping the main concepts and difficulties associated to program slicing.

The continuation of our journey, lead us towards our main objective: using slicing as a basis for the recovery of high-level software architectural models, with a specific emphasis in tracing back the pathways of interaction, their structures and nature, in legacy source code. We believe this may turn out to be an important component in real program understanding and re-engineering projects. Given the massive *service oriented* trend that software engineering witnesses, this was an additional source of motivation for our work.

Reaching the end of this thesis, we would like to use the following few pages, summing up, not the thesis conclusions, which were already presented before, but a few considerations on the definition and role of software architecture for the working systems developer. The relevance of architectural issues, as perceived in our own practice, was part of our motivation for this work. In retrospect, its results may contribute to improve methods and practices in this domain.

Thus, as an epilogue, we would like to comment on what we understand by *software architecture* and the roles it may play in the practice of Software Engineering. Our starting point is the following definition in [BCK98]:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them"

We would like to dissect this definition to fully understand how it characterises the notion of a software architecture. First, the definition states that a software architecture is the structure or structures of a software system. In opposition to some definitions that elect a single and complete master model of the system as being the software architecture itself, we share this notion that a software architecture may be composed by more than one structure or model of the system. Furthermore, these models should complete themselves towards providing an overall view of the system and highlighting different aspects of the project. This entails another question, *i.e.*, how many and what kind of models should be used to provide a good and complete architecture? Unfortunately, the answer is not unique: it depends on the specific details of the software system under analysis. So, for a simple object oriented library it may be sufficient to have a system architecture just composed of a single class diagram module. In this case, the single class diagram is itself the software architecture of the system since it exposes a high-level view of the main aspects of the library. Such cases, where the software architecture is correctly expressed by a single model, may be responsible for the generalisation of the idea (in our view erroneous) that there is a single special model that correctly describes a system's software architecture in every situation. On the other hand, for large service oriented architectures, which handle different workflow scenarios simultaneously at runtime, one certainly needs a set of modules emphasising different aspects of the solutions. In this case, one would probably need a module diagram to divide the project into isolated work modules to be assign to different development teams, several class diagram modules for the parts of the system that were developed using an object oriented paradigm, and a coordination mode in order to suitably control the coordination of the different services provided and potentially consumed by the system.

The definition continues stating that a software architecture comprises software components. However, it does not define what a component is, which we believe to be a wise approach to take given that the concept of a component bares little definition consensus among the Software Engineering community. Again, this is in accordance to our interpretation of a software architecture composed by different and complementary models where each model may have a different notion about the units or entities it reasons about. Our interpretation of the definition is that these entities, generally referred to as components, must be clearly identified in each model so that there remains no doubt about what are the target entities the model intends to capture. Just to put things a bit more concrete, in the specific case of a coordination model, a software component would typically be an autonomous process or service running in its own thread.

Another important issue in the definition is the statement that a software architecture is concerned with the *externally visible properties* of the addressed components. This clearly means that a software architecture should abstract some of the details of the components it deals with and only expose specific aspects (in this case referred to as visible properties) of the component being described. Otherwise, the software architecture of a system would be the system itself, possibly in another representation form, but still with little use an abstraction.

With respect to the properties which should be abstracted, we diverge a bit from the classification in [BCK98] who defined these as the visible component properties. Actually, we prefer a more general definition, restating the previous definition to the component's properties relevant for the model under consideration. So, if for instance a particular property of a module is a private property of the module but it is fundamental to the definition of the work module that must be given to a developer team, one would include such a property in the system module diagram, even though it would not be a typically *visible* property of the component. Of course, it will be omitted from a coordination model, as other processes could not rely, or even be aware of it.

Finally, let us concentrate on the final part of the definition stating that a software architecture is also concerned about capturing the *relationships* between the system components. Here one dares to take a step further, and state that for the great majority of the systems being developed, these relationships between components are even more important than the components themselves. The reason being that modern software systems are becoming more and more based on externally developed components, leaving the majority of the actual system development work to the task of correctly connecting and synchronising different components and services, *i.e.*, instantiating the interaction paths.

But why does one need a software architecture, after all? We believe that there are three mains reasons justifying the need for comprehensive and verifiable software architectures.

The first, and often neglected, advantage of an architectural description is as an effective means of project documentation, which can be shared with every stakeholder in the project. This way software architects and system developers can start discussing the project with every interested part, based on a precise, though high-level, description of the system. Formal specifications may be, at present, too hard to fill this goal. But, models like the workflow description of the processes being implemented and graphical user interface prototypes, would undoubtedly be of utmost importance for the developer team to discuss and precisely understand what problems should the system really solve. In practice, this advantage of having a software architecture comes with at least two important positive outcomes. The first is that the architect and the development team can understand more precisely what every stakeholder is expecting from the system and use this information to guide the development and structural construction of the system. The second outcome is that one may correctly assess what functionalities the system does not need to provide and this way removing work that would be needed to develop the useless functionalities and also reducing the entire complexity of the system.

The second, and most important, advantage of documenting a software architecture before the actual system starts being developed is the ability that, by reasoning upon the architectural models, one is able to take early design decisions that would be catastrophic to take at later development stages. As an example of this advantage in practice, take for instance the case of developing a software solution where some agents have to consume services from a particular provider. At a first glance, such a scenario fits perfectly in the well-known client-server model, possibly even using web services to implement the communication between the parties. Now, imagine that the development of the solution continues under such model and somewhere, at an intermediate stage, it is found that some clients have to be actively called from the server. This situation is clearly in complete discordance with the client-server model adopted before, where only clients were able to call the server and not the opposite. Thus, for the system to be correctly developed one would need to adopt a different architecture, say a *blackboard* or an *event based* architecture. This change would transform most of the developed parts of the system into completely useless artefacts, and thus force the development team to re-start the project from scratch.

On the other hand, if one would have taken some time to develop an architecture of the system, which in this case would most certainly demand for a coordination model of the entities involved, one would certainly come to the conclusion that there were some situations where clients had to be actively called by the service provider entity. Such a conclusion would have guided the development of the system to the adoption a different architecture from the outset.

The third advantage in having a software architecture of a system being developed takes place during the actual development phase. In the absence of a software architecture and during the entire development phase, the project can easily drift away from the initial plan and end up being something completely different from the initial requirements or not fulfilling some of the non-functional requisites, a situation which is often explained by the attempt to adapt legacy sub-systems to fulfil some initial functional requisites. Again, in the presence of a software architecture, developers can continuously base their decisions on the architecture and check regularly to what extent the implementation is sticking to the architectural model.

This also explains the relevance of architectural models for re-engineering projects. The challenge has, in our opinion, two complementary sides. On the one hand, we need rigorous specification notations to describe architectures and, inside them, what we have called in this thesis the *coordination-driven* view of architectures. ORC or Reo are promising frameworks for expressing and transforming coordination policies as well as for reasoning about them. On the other hand, there is a need for techniques, methods and tools to extract, identify, represent and analyse such policies from running systems, preferably acting at the source code level.

This thesis intended to contribute to this second direction. No doubt, a lot of work remains to be done to combine the level of (forward) *specification* and the one of (reverse) *understanding* of coordination policies and frame them in mainstream architectural research. But probably any thesis ends raising more questions than the ones it tried to solve.

Appendix A

HASKELL Bank Account System Program

```
module Slicing where

import Mpi

data System = Sys { clients  :: [Client],
                    accounts :: [Account] } deriving Show

data Client = Clt { cltid :: CltId,
                    name  :: CltName } deriving Show

data Account = Acc { accid  :: AccId,
                     amount :: Amount } deriving Show

type CltId   = Int
type CltName = String
type AccId   = Int
type Amount  = Double

initClts :: [((CltId, CltName), (AccId, Amount))] -> System
initClts = (uncurry Sys) . split (map ((uncurry Clt) . fst))
                                 (map ((uncurry Acc) . snd))

findClt :: CltId -> System -> Maybe Client
findClt cid sys =
    if (existsClt cid sys)
       then Just . head . filter ((cid ==) . cltid) .
            clients $ sys
       else Nothing

findAcc :: AccId -> System -> Maybe Account
findAcc acid sys =
    if (existsAcc acid sys)
       then Just . head . filter ((acid ==) . accid) .
            accounts $ sys
       else Nothing

existsClt :: CltId -> System -> Bool
existsClt cid = elem cid . map cltid . clients

existsAcc :: AccId -> System -> Bool
existsAcc acid = elem acid . map accid . accounts

insertClt :: (CltId, CltName) -> System -> System
insertClt (cid, cname) (Sys clts accs) =
    if (existsClt cid (Sys clts accs))
       then error "Client ID already exists!"
       else Sys ((Clt cid cname) : clts) accs

insertAcc :: (AccId, Amount) -> System -> System
insertAcc (acid, amount) (Sys clts accs) =
    if (existsAcc acid (Sys clts accs))
       then error "Account ID already exists!"
       else Sys clts ((Acc acid amount) : accs)

removeClt :: CltId -> System -> System
removeClt cid (Sys clts accs) =
    if (existsClt cid (Sys clts accs))
        then Sys (filter ((cid /=) . cltid) clts) accs
        else Sys clts accs

removeAcc :: AccId -> System -> System
removeAcc acid (Sys clts accs) =
    if (existsAcc acid (Sys clts accs))
```

```
        then Sys clts (filter ((acid /=) . accid) accs)
        else Sys clts accs

updateClt :: (CltId, CltName) -> System -> System
updateClt (cid, cname) sys =
    if (existsClt cid sys)
       then insertClt (cid, cname) . removeClt cid $ sys
       else insertClt (cid, cname) sys

updateAcc :: (AccId, Amount) -> System -> System
updateAcc (acid, amount) sys =
    if (existsAcc acid sys)
       then insertAcc (acid, amount) . removeAcc acid $ sys
       else insertAcc (acid, amount) sys

getCltName :: CltId -> System -> Maybe CltName
getCltName cid sys = case findClt cid sys of
                        Just clt -> Just . name $ clt
                        Nothing  -> Nothing

getAccAmount :: AccId -> System -> Maybe Amount
getAccAmount acid sys = case findAcc acid sys of
                           Just acc -> Just . amount $ acc
                           Nothing  -> Nothing
```

Appendix B

A Brief Introduction to Orc

B.1 Purpose and syntax

Many traditional concurrency problems, like the integration of business workflows, resource sharing or composition of web-services, can be regarded as orchestrations of third party resources, encompassing a general-purpose, exogenous, coordination model, for which a number of formal semantics have already been proposed [HMM04, KCM06, AM07]. This provides a solid theoretical background upon which a calculus to reason and transform coordination specifications can be based.

This appendix provides a brief introduction to Orc its syntax and informal semantics, to the extent required for understanding the coordination specification recovered from legacy code as presented in chapter 8. The reader is referred to [MC06, HMM04, KCM06, AM07] for detailed presentations of the language, its formal semantics and applications.

Unlike other coordination models, Orc regards the orchestration of different activities and participants in a centralised way. Thus, in Orc, external services never take the initiative of initiating communications; there is a central entity to control the invocation of foreign operations.

In Orc, third party services are abstracted as sites which can be called. Included in this notion of site, are user interaction activities and third party data manipulation.

The language builds upon few simple basic constructs to build orchestrations. An orchestration consists, therefore, of a set of auxiliary definitions and a main goal. In summary, the language provides a medium to evaluate such expressions. It can be regarded as a platform for simple specification of third-party resources invocations with a specific goal to accomplish, while managing concurrency, failure, time-outs, priorities and other constrains.

$$
\begin{array}{rcllll}
e, f, g, h & \in & \textit{Expression} & ::= & M(\overline{p}) \;\|\; E(\overline{p}) \;\|\; f > x > g \;\|\; f \mid g \;\| \\
 & & & & f \textbf{ where } x :\in g \;\|\; x \\
p & \in & \textit{Actual} & ::= & x \;\|\; M \;\|\; c \;\|\; f \\
q & \in & \textit{Formal} & ::= & x \;\|\; M \\
 & & \textit{Definition} & ::= & E(\overline{q}) \triangleq f
\end{array}
$$

Figure B.1: Orc syntax

The syntax of the language is presented in Figure B.1, where definitions for Orc *Expressions*, *Actual* parameters $\overline{p}$, *Formal* parameters $\overline{q}$ and *Definitions* are given.

An Orc expression can be composed of a site call $M(\overline{p})$, an expression call $E(\overline{p})$, a sequential execution of expressions $f > x > g$, a parallel execution of expressions $f|g$, an asymmetric parallel composition of expressions f **where** $x :\in g$, or a variable x.

There are a few fundamental sites in Orc which are essential for effective programming of real world examples. Such sites along with its informal semantics are described in Table B.1.

$let(x, y, ...)$	Returns argument values as a tuple.
$if(b)$	Returns a signal if b is true, and it does not respond if b is false.
$Signal$	Returns a signal. It is same as $if(true)$
$RTimer(t)$	Returns a signal after exactly t time units

Table B.1: Fundamental sites in Orc

ORC also provides means for creating dynamic orchestrations, i.e., orchestrations that are able to create local sites at runtime. This feature is provided by special sites, called *Factory Sites*, which return a local site when invoked [CPM06]. Table B.2 describes some useful factory sites together with an informal description of its semantics. These factory sites, are used in chapter 8 for capturing specific coordination schemas.

Site	**Operations**	**Description**
Buffer	*put*, *get*	The *Buffer* factory site returns a $n{-}buffer$ local site with two operations, *put* and *get*. The *put* operation stores its argument value in the buffer and sends a signal after the storage. The *get* operation removes an item from the buffer and returns it. In case the buffer is empty the *get* operation suspends until a value is putted in the buffer.
Lock	*acquire*, *release*	The *Lock* factory site returns a *lock* local site which provides two operations, *acquire* and *release*. When an expression invokes the *acquire* operation on a *lock*, that expression becomes its owner and subsequent calls to *acquire* from other expressions will block. Once the *lock* owner expression releases it, ownership of the lock will be given to one of the *acquire* waiting operations, if any.

Table B.2: Factory sites in ORC

B.2 Informal semantics

A site in ORC is an independent entity with the capacity of publishing values to the calling expressions. The evaluation of a site call holds indefinitely (possibly forever, if the site never publishes a value) until the called site publishes a value.

An expression call, simply transfers the control from the expression under evaluation to the called expression with the associated parameters.

A sequential execution of expressions $f > x > g$ proceeds by evaluating expression f, binding the value published by f to x and then evaluating expression g which may contain references to x. In cases where x isn't used by g, the sequential expression is abbreviated to $f >> g$.

Parallel composition of expressions is carried out as in most concurrent process algebras *i.e.*, by the concurrent execution of the intervening expressions.

Finally, asymmetric parallel composition f **where** $x :\in g$ proceeds by evaluating f and g in parallel, suspending the evaluation of f whenever it depends on variable x and g has not published any value to this variable. Once g publishes a value, its evaluation is halted and the value produced is stored in x, enabling expression f evaluation to continue. For a formal semantics of the language see [HMM04, KCM06, AM07].

Table B.3 presents a number of typical Orc definitions, which encapsulate useful coordination used in the body of this thesis. For now, they serve to provide a few examples and some intuition on the execution of Orc expressions.

$$\begin{array}{lcl}
\mathit{XOR}(p, f, g) & \triangleq & if(p) \gg f \mid if(\neg p) \gg g \\
\mathit{IfSignal}(p, f) & \triangleq & if(p) \gg f \mid if(\neg p) \\
\mathit{Loop}(p, f) & \triangleq & p > b > \mathit{IfSignal}(b, f \gg \mathit{Loop}(g, f)) \\
\mathit{Discr}(f, g) & \triangleq & \mathit{Buffer} > B > ((f \mid g) > x > B.put(x) \mid B.get)
\end{array}$$

Table B.3: Some Orc definitions

The *XOR* definition takes as arguments a predicate expression p, and two orchestrations f and g. If p evaluates to *true* then orchestration f is executed, otherwise g is chosen. Regard that, in spite of the parallel operator the definition only executes one of the expressions, f or g, and that one of them is always executed.

The $IfSignal$ definition receives a predicate and an orchestration and executes the orchestration if the predicate evaluates to $true$. Again, notice that irrespective p evaluates to $true$ or $false$ the definition never blocks; it always publishes a value, thus permitting the calling orchestration to proceed.

The *Loop* expression receives a predicate p and an orchestration f. This definition executes f continuously until predicate p evaluates to $false$. If predicate p evaluates to $false$ the definition does not block and returns a signal in order for the calling orchestration to proceed.

Finally, definition $Discr$ makes use of the factory site $Buffer$ in order to capture the signal of the first of its two parameter orchestrations to respond. Once one of the orchestrations returns, the signal is forwarded to the calling orchestration, but leaving the other orchestration running until, eventually, it reaches termination.

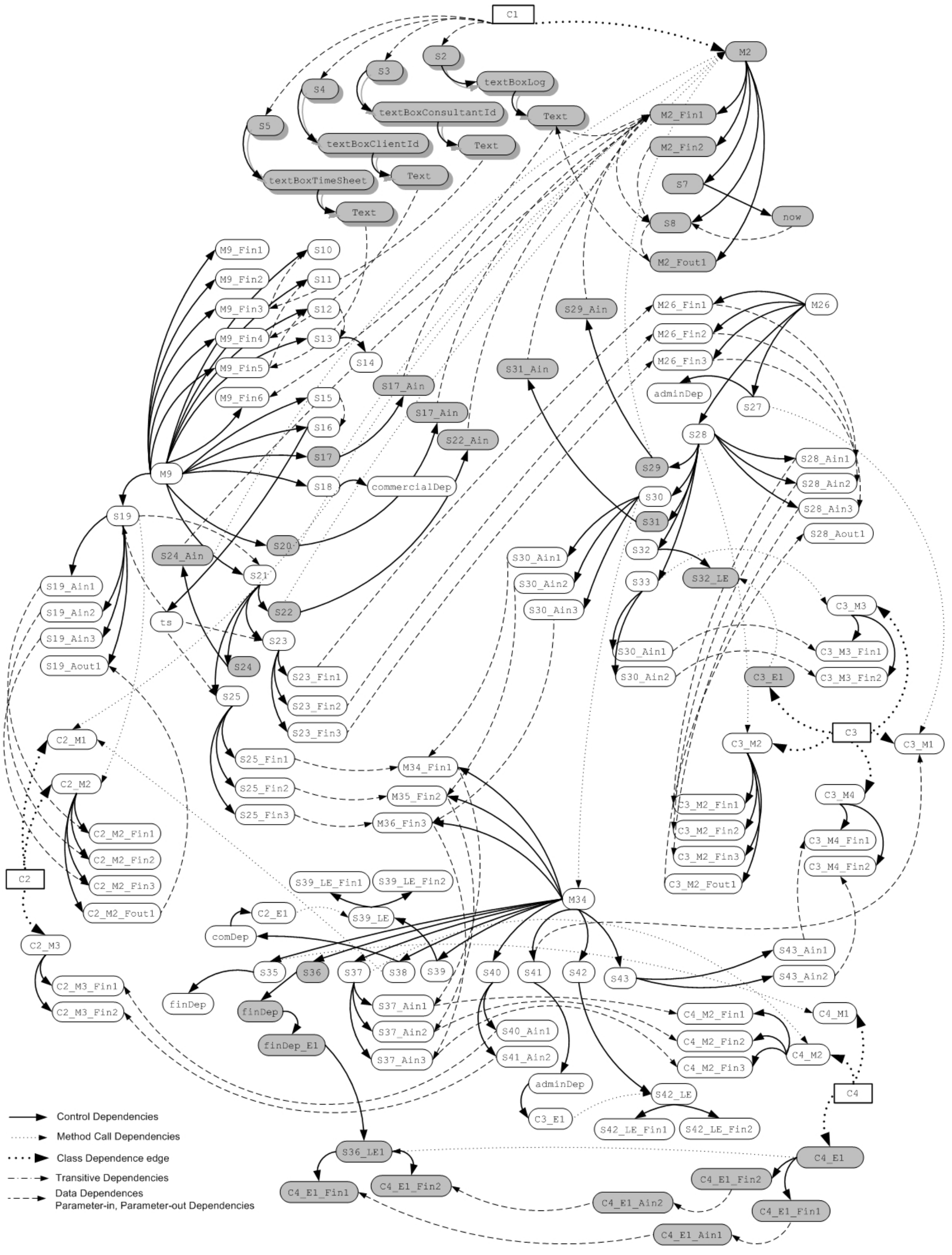
Appendix C

Consultant Time Sheet Example Code

```
C1:  public class ConsultantSubmitTimeSheetApp {
S2:    textBoxLog = new TextBox();
S3:    textBoxConsultantId = new TextBox();
S4:    textBoxClientId = new TextBox();
S5:    textBoxTimeSheet = new TextBox();

M6:    private void UpdateLog(string message) {
S7:      DateTime now = DateTime.Now;
S8:      textBoxLog.Text += now.ToShortDateString() + " " +
           now.ToShortTimeString() + ": " + message +
           Environment.NewLine;
       }

M9:    private void button2_Click(object sender,
                                  EventArgs e) {
S10:     int cId = Convert.ToInt32(textBoxConsultantId.Text);
S11:     int cltId = Convert.ToInt32(textBoxClientId.Text);

S12:     FileInfo fi = new FileInfo(textBoxTimeSheet.Text);
S13:     if (!fi.Exists)
S14:       MessageBox.Show(
             "No timesheet valid file available.");

S15:     StreamReader sr =
           new StreamReader(textBoxTimeSheet.Text);
S16:     TimeSheet ts = new TimeSheet(sr.ReadToEnd());

S17:     UpdateLog("Calculating time sheet total cost...");
S18:     CommercialDepService.CommercialDep commercialDep =
           new CommercialDepService.CommercialDep();
S19:     decimal totalCost =
           commercialDep.CalculateTimeSheetTotalCost(ts,
                                                     cId,
                                                     cltId);
S20:     UpdateLog("Total cost = " + totalCost.ToString());

S21:     if (totalCost > 1000) {
S22:       UpdateLog(
             "Requesting administration validation...");
S23:       RequestAdministrationValidation(ts, cId, cltId);
         } else {
S24:       UpdateLog("Time sheet expense approved...");
S25:       ProceedWithTimeSheetExpenseProcessing(ts,
                                                 cId,
                                                 cltId);
         }
       }

M26:   private void RequestAdministrationValidation(
       TimeSheet ts, int cId, int cltId) {
S27:     AdministrationDepService.AdministrationDep adminDep =
           new AdministrationDepService.AdministrationDep();
S28:     if (adminDep.ValidateTimeSheet(ts, cId, cltId)) {
S29:       UpdateLog(
             "Administration accepted time sheet expense...");
S30:       ProceedWithTimeSheetExpenseProcessing(ts,
                                                 cId,
                                                 cltId);
         } else {
S31:       UpdateLog(
             "Administration refused time sheet expense...");
S32:       adminDep.UpdateConsultantScoreNegativelyCompleted+=
             ((o, ea) =>
                UpdateLog(
                  "Consultant score updated negatively"));
S33:       adminDep.UpdateConsultantScoreNegativelyAsync(ts,
                                                         cId);
         }
       }

M34:   private void ProceedWithTimeSheetExpenseProcessing(
       TimeSheet ts, int cId, int cltId) {
S35:     FinancialDepService.FinancialDep finDep =
           new TimeSheetConsultantClient.FinancialDepService.
               FinancialDep();
S36:       finDep.SubmitExpenseCompleted +=
             ((o, ea) =>
                UpdateLog(
                  "Time sheet expense received by " +
                  "financial dep"));
S37:       finDep.SubmitExpenseAsync(ts, cId, cltId);

S38:       CommercialDepService.CommercialDep comDep =
             new TimeSheetConsultantClient.CommercialDepService.
                 CommercialDep();
S39:       comDep.CommunicateExpenseToClientCompleted +=
             ((o, ea) =>
                UpdateLog(
                  "Expense communication received by " +
                  "commercial dep"));
S40:       comDep.CommunicateExpenseToClientAsync(ts, cltId);

S41:       AdministrationDepService.AdministrationDep adminDep =
             new AdministrationDepService.AdministrationDep();
S42:       adminDep.UpdateConsultantScorePositivelyCompleted+=
             ((o, ea) =>
                UpdateLog(
                  "Consultant score updated positively"));
S43:       adminDep.UpdateConsultantScorePositivelyAsync(ts,
                                                         cId);
       }
     }
```

Appendix D

Appendix C Example Code MSDG



```
S17_Ain1 := message_in = “Calculating time sheet total cost...”
C2_M1 := public CommercialDep()
C2_M2 := public decimal CalculateTimeSheetTotalCost(TimeSheet t,
                int consultantId, int clientId)
C2 := class CommercialDep
C2_M2_Fin1 := t = t_in
C2_M2_Fin2 := consultantId = consultantId_in
C2_M2_Fin3 := clientId = clientId_in
C2_M2_Fout1 := result_out = ((decimal) results[0])
S19_Ain1 := t_in = ts
S19_Ain2 := consultantId_in = cId
S19_Ain3 := clientId = cltId
S19_Aout1 := totalCost := result_out
```

```
S22_Ain := message_in = “Requesting administration validation...”
C3 := public class AdministrationDep
C3_M1 := public AdministrationDep
C3_M2 := public bool ValidateTimeSheet(TimeSheet t, int consultantId, int clientId)
C3_M3 := public void UpdateConsultantScoreNegativelyAsync(TimeSheet t, int consultantId)
C3_M4 := public void UpdateConsultantScorePositivelyAsync(TimeSheet t, int consultantId)
M26_Fin1 := ts (and its defintion tree which is abstracted)
C3_E1 := pubic event UpdateConsultantScoreNegativelyCompleted
C4 := class FinancialDep
S32_LE := (o, ea) => UpdateLog("Consultant score updated negatively")
C4_M1 := public class FinancialDep
C4_M2 := public void SubmitExpenseAsync(TimeSheet t, int consultantId, int clientId)
C2_M3 := public void CommunicateExpenseToClientAsync(TimeSheet t, int clientId)
```

Figure D.1: Example program

Appendix E

Abstract WS-BPEL of the Business Process in Figure 8.12

```
<process>
  <variables>
    <variable name="SubmitTimeSheetRequest" />
    <variable name="GetTimesheetWithCostResponse" />
    <variable name="AnalyzeSheetResponse" />
  </variables>
  <flow>
    <receive partnerLink="##opaque"
             operation="SubmitTimesheet"
             variable="GetTimesheetWithCostResponse">
      <sources><source linkName="SubmitTimesheet" />
      </sources>
    </receive>
  </flow>
  <scope name="SubmitTimesheet">
    <targets><target linkName="SubmitTimesheet" />
    </targets>
    <sequence>
      <invoke partnerLink="##opaque"
              operation="GetTimesheetWithCost"
              input="SubmitTimeSheetRequest"
              output="GetTimesheetWithCostResponse" />
      <if>
        <condition>
```

```
 getVariableProperty(GetTotalCostResponse,
           total) > 2000
</condition>
<flow>
  <invoke partnerLink="##opaque"
   operation="AnalyzeSheet"
   input="SubmitTimeSheetRequest"
   output="AnalyzeSheetResponse" >
   <sources>
    <source linkName="OnAnalyseResponse" />
   </sources>
  </invoke>
</flow>
<scope name="OnAnalyseResponse">
 <sequence>
  <targets>
   <target linkName="OnAnalyseResponse" />
  </targets>
  <if>
   <condition>
    getVariableProperty(AnalyzeSheetResponse,
              Approved)
   </condition>
   <invoke partnerLink="##opaque"
       operation="CommunicateClientExpense"
       input="GetTimesheetWithCostResponse" />
   <invoke partnerLink="##opaque"
       operation="NotiffyApprovedExpense"
       input="GetTimesheetWithCostResponse" />
   <else>
    <invoke partnerLink="##opaque"
       operation="ResubmitSheet"
       input="GetTimesheetWithCostResponse" />
   </else>
  </if>
 <sequence>
</scope>
<else>
 <invoke partnerLink="##opaque"
```

```
            operation="CommunicateClientExpense"
            input="GetTotalCostResponse" />
    <invoke partnerLink="##opaque"
            operation="NotiffyApprovedExpense"
            input="GetTotalCostResponse" />
  </else>
</if></sequence></scope></process>
```

Bibliography

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[ADS93] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[AF04] L. F. Andrade and J. L. Fiadeiro. Composition contracts for service interaction. *Journal of Universal Computer Science*, 10(4):751–761, 2004.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[AHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[AHS93] Farhad Arbab, Ivan Herman, and Pål Spilling. An overview of manifold and its implementation. *Concurrency - Practice and Experience*, 5(1):23–70, 1993.

[All97] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997.

[AM07] Musab AlTurki and José Meseguer. Real-time rewriting semantics of orc. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of* *declarative programming*, pages 131–142, New York, NY, USA, 2007. ACM.

[Arb96] Farhad Arbab. The iwim model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Proc. Coordination Languages and Models, First Inter. Conf., COORDINATION '96, Cesena, Italy, April 15-17*, volume 1061, pages 34–56. Springer Lect. Notes Comp. Sci. (1061), 1996.

[Arb98] Farhad Arbab. What do you mean, coordination. In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, 1998.

[Arb03] F. Arbab. Abstract behaviour types: a foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.

[Arb04] F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.

[Bac78] J. Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[Bac02] R. Backhouse. Fixed point calculus. In R. Crole, R. Backhouse, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Constuction*, pages 89–148. Springer Lect. Notes Comp. Sci. (2297), 2002.

[Bac03] R. Backhouse. *Program Construction.* John Wiley and Sons, Inc., 2003.

[Bar01] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In *CMCS'01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.

[BCG97] Robert Bjornson, Nicholas Carriero, and David Gelernter. From weaving threads to untangling the web: A view of coordination from linda's perspective. In David Garlan and Daniel Le Metayer, editors, *Proc. of Second Inter. Conf. on Coordination Languages and Models, COORDINATION '97, Berlin, Germany*, pages 1–17. Springer Lect. Notes Comp. Sci. (1282), 1997.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd ed.)*. Addison-Wesley, 2003.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In *Proc. First Inter. Workshop on Web Services and Formal Methods*, volume 105, pages 73–94, Pisa, Italy, 2004.

[BDG$^+$06] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program.*, 62(3):228–252, 2006.

[BG96] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BGG$^+$05] N. Busi, R. Gorrieri, C. Guidi, R. Luchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for systems design. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proc. ICSOC 2005 Thrid Inter. Conf. on Service-Oriented Computing*, pages 228–240, 2005.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.

[BHR95] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[Bir87] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.

[Bir98] R. Bird. *Functional Programming Using Haskell.* Series in Computer Science. Prentice-Hall International, 1998.

[Bis97] Sandip Kumar Biswas. *Dynamic slicing in higher-order programming languages.* PhD thesis, Philadelphia, PA, USA, 1997. Supervisor-Carl A. Gunter.

[BM97] R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.

[Bou99] S. et al Boucetta. Architectural recover and evolution of large legacy systems. In *Proc. Int. Work. on Principles of Software Evolution IWPSE*, 1999.

[BPSM97] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language. *World Wide Web J.*, 2(4):29–66, 1997.

[BSAR05] L. S. Barbosa, M. Sun, B. K. Aichernig, and N. Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Development. World Scientific, 2005.

[BSR06] Silvia Breu, Marc Schlickling, and Nuno Miguel Feixa Rodrigues. 05451 group 5 – bananas, dark worlds, and aspecth. In David W. Binkley, Mark Harman, and Jens Krinke, editors, *Beyond Program Slicing*, number 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/491> [date of citation: 2006-01-01].

[BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA, Third Edition.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998. Graduate Texts in Computer Science.

[CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Program Slicing, Information and Software Technology*, 40:595–607, 1998. (special issue).

[CCLL94] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 424–433, Washington, DC, USA, 1994. IEEE Computer Society.

[CCM94] G. Canfora, A. Cimitile, and M. Munro. Re^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, 6(2):53–72, 1994.

[Che01] Timothy M. Chester. Cross-platform integration with xml and soap. *IT Professional*, 3(5):26–34, 2001.

[CLM96] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8(3):145–178, 1996.

[CPM06] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

[Cun05] Alcino Cunha. *Point-Free Program Calculation.* PhD thesis, Dep. Informática, Universidade do Minho, 2005.

[dLFM96] Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.

[Fia04] J. L. Fiadeiro. Software services: scientific challenge or industrial hype? In K. Araki and Z. Liu, editors, *Proc. First International Colloquim on Theoretical Aspects of Computing (ICTAC'04), Guiyang, China*, pages 1–13. Springer Lect. Notes Comp. Sci. (3407), 2004.

[FL97] J. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proc. of TAPSOFT'97*, pages 505–519. Springer Lect. Notes Comp. Sci. (1214), 1997.

[FL98] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Pratical Tools and Techniques in Software Development.* Cambridge University Press, 1998.

[FLM$^+$05] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems.* Springer Verlag, 2005.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[FP97] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach.* PWS Publishing Co., Boston, MA, USA, 1997.

[FŠčedrov90] P. J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library.* North-Holland, 1990.

[Gai04] Jeannine Gailey. *Understanding web services specifications and the WSE.* Microsoft Press, Redmond, WA, USA, 2004.

[Gar03] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors and events. In M. Bernardo and P. Inverardi, editors, *Third International Summer School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003).* Springer Lect. Notes Comp. Sci, Tutorial, (2804), Bertinoro, Italy, September 2003.

[GB03] Keith Gallagher and David Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada*, pages 316–322, 2003.

[GBR04] Beth Gold-Bernstein and William Ruh. *Enterprise Integration: The Essential Guide to Integration Solutions.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[GC92] D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.

[Gib97] J. Gibbons. Conditionals in distributive categories. CMS-TR-97-01, School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[GMW97] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *CASCON'97*, 1997.

[GP94] David Garlan and Dewayne E. Perry. Software architecture: Practice, pitfalls and potential. In *16th International Conference on Software Engineering*, pages 3–17, 1994.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering (volume I)*. World Scientific Publishing Co., 1993.

[GS04] V. Gruhn and C. Schäfer. An architecture description language for mobile distributed systems. In Ron Morrison Flavio Oquendo, Brian Warboys, editor, *Software Architecture - Proceedings of the First European Workshop, EWSA 2004*, pages 212–218. Springer-Verlag, 2004.

[GS06] David Garlan and Bradley Schmerl. Architecture-driven modelling and analysis. In *SCS '06: Proc. of the 11th Australian Workshop on Safety Critical Systems and Software*, pages 3–17. Australian Computer Society, Inc., 2006.

[Hag87] T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283), 1987.

[Hal03] Thomas Hallgren. Haskell tools from the programatica project. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 103–106, New York, NY, USA, 2003. ACM Press.

[HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, 1995.

[HG98] Mark Harman and Keith Brian Gallagher. Program slicing. *Information & Software Technology*, 40(11-12):577–581, 1998.

[HH99] Rob Hierons and Mark Harman. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9:233–262, 1999.

[HHD$^+$01] M. Harman, R.M. Hierons, S. Danicic, J. Howroyd, and C. Fox. Pre/post conditioned slicing. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 138, Washington, DC, USA, 2001. IEEE Computer Society.

[HLS05] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 25–34, Washington, DC, USA, 2005. IEEE Computer Society.

[HM86] R. Harper and K. Mitchell. Introduction to standard ML. Technical Report, University of Edimburgh, 1986.

[HMM04] T. Hoare, G. Menzel, and J. Misra. A tree semantics of an orchestration language, August 2004.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[HPW92] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.

[HRY95] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. In *ICSE-17: Proc. of the 17th Int. Conf. on Software Engineering*, pages 186–195. Association for Computing Machinery, Inc., 1995.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[JE07] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. OASIS Standard 2.0, OASIS, Post Office Box 455, Billerica, MA 01821, USA, April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[Jep01] Tom Jepsen. Soap cleans up interoperability problems on the web. *IT Professional*, 3(1):52–55, 2001.

[JMA96] Daniel Le Metayer Jean-Marc Andreoli, Chris Hankin. *Coordination Programming: Mechanisms, Models, and Semantics*. Imperial College Press, 1996.

[Jon86] Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.

[JR94] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical report, Pittsburgh, PA, USA, 1994.

[KC98] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[KMG] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. Static slicing of java programs.

[Kri03] Jens Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, 2003.

[LAK$^+$95] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann. Specifications and analysis of system architecture using Rapide. *IEEE Tran. on Software Engineering (special issue in Software Architecture)*, 21(4):336–355, April 1995.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[LH96] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[LH98] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 358, Washington, DC, USA, 1998. IEEE Computer Society.

[Lin00] David S. Linthicum. *Enterprise application integration.* Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[Lin03] David S. Linthicum. *Next Generation Application Integration: From Simple Information to Web Services.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LV03] Ralf Lämmel and Joost Visser. A strafunski application letter. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 357–375, London, UK, 2003. Springer-Verlag.

[MA86] E. Manes and A. Arbib. *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science. Springer Verlag, 1986.

[Mal90] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[MC06] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.

[MC07] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Software and System Modeling*, 6(1):83–110, 2007.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conference*, 1995.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.

[Mig05] Antony Miguel. Ws-bpel 2.0 tutorial. Tutorial 1, Scapatech, October 2005. http://www.eclipse.org/tptp/platform/documents/design /choreography_html/tutorials/wsbpel_tut.html.

[MKMG97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, January 1997.

[MMCG02] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software systems structures. In *Proc. AWASA 2002*, 2002.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *4th ACM Symp. on Foundations of Software Engineering SIGSOFT'96*, 1996.

[MR03] James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development. Addison-Wesley Professional, 1 edition, November 2003.

[NA03] O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.

[NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 180–190, New York, NY, USA, 2000. ACM.

[Oli01a] J.N. Oliveira. Bagatelle in c arranged for vdm solo. *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering*, Colloquium in Honor of Peter Lucas, Institute for Software Technology, Graz University of Technology, May 18-19, 2001).

[Oli01b] José Nuno Oliveira. "bagatelle in c arranged for vdm solo". *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering*, Colloquium in Honor of Peter Lucas, Institute for Software Technology, Graz University of Technology, May 18-19, 2001).

[Oli08] José N. Oliveira. Transforming data by calculation. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, Lecture Notes in Computer Science, pages 134–195. Springer-Verlag, Berlin, Heidelberg, October 2008.

[Oqu04] F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14, 2004.

[OSV04] Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the*

2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 123–134, New York, NY, USA, 2004. ACM Press.

[PA98] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RB03] N. Rodrigues and L. S. Barbosa. On the specification of a component repository. In Hung Dang Van and Zhiming Liu, editors, *Proc. of FACS'03, (*Formal Approaches to Component Software*)*, pages 47–62, Pisa, September 2003.

[RB06a] N. Rodrigues and L. S. Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.

[RB06b] N.F. Rodrigues and L.S. Barbosa. Program slicing by calculation. *Journal of Universal Computer Science*, 12(7):828–848, 2006. `http://www.jucs.org/jucs_12_7/program_slicing_by_calculation`.

[RB07] Nuno F. Rodrigues and Luís S. Barbosa. Higher-order lazy functional slicing. *Journal of Universal Computer Science*, 13(6):854–873, jun 2007. `http://www.jucs.org/jucs_13_6/higher_order_lazy_functional`.

[RB08a] N. F. Rodrigues and L. S. Barbosa. Extracting and verifying coordination models from source code. In *Proc. of the Joint FLOSS-FM / OpenCert Worshops, at Int. Conf. on Open*

Source Software, IFIP WCC, Milan, September, 2008, pages 64–78. UNU-IIST, Macau, 2008.

[RB08b] Nuno F. Rodrigues and Luís S. Barbosa. Coordinspector a tool for extracting coordination data from legacy code. In *SCAM '08: Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Washington, DC, USA, 2008. IEEE Computer Society. (To appear).

[RB08c] Nuno F. Rodrigues and Luís S. Barbosa. On the discovery of business processes orchestration patterns. In *2008 IEEE Congress on Services*, pages 391–398, Washington, DC, USA, July 2008. IEEE Computer Society, IEEE Computer Society Press.

[RH07] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007.

[Rod08] Nuno F. Rodrigues. Discovering coordination patterns. In *Proc. of FACS 2008: 5th International Workshop on Formal Aspects of Component Software, Malaga, SP*, 10–12 September 2008. (To appear).

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, New York, NY, USA, 1995. ACM.

[RT96] Thomas W. Reps and Todd Turnidge. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *ICSE '91: Proceedings of the 13th in-*

ternational conference on Software engineering, pages 83–92, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Sch98] D. Schamschurko. Modeling process calculi with PVS. In *CMCS'98, Elect. Notes in Theor. Comp. Sci.*, volume 11. Elsevier, 1998.

[Sch06] D.C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SDS06] Kamran Sartipi, Nima Dezhkam, and Hossein Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 61–70, 2006.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SG98] B. Spitznagel and Garlan. Architecture-based perfomance analysis. In *10th Int. Conf. on Software Engineering and Knowledge Engineering SEKE'98*, 1998.

[SH94] Robert W. Schwanke and Stephen Jos&#233; Hanson. Using neural networks to modularize software. *Mach. Learn.*, 15(2):137–168, 1994.

[SN99] J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[SS99] Kent Sandoe and Aditya Saharia. *Enterprise Integration.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Stu07] Tony Stubblebine. *Regular expression pocket reference, 2nd edition.* O'Reilly, 2007.

[SvdMK$^+$04] R. Seker, A. J. van der Merwe, P. Kotze, M. M. Tanik, and R. Paul. Assessment of coupling and cohesion for component-based software by using shannon languages. *J. Integr. Des. Process Sci.*, 8(4):33–43, 2004.

[SVM$^+$93] Dan Simpson, Sam Valentine, Richard Mitchell, Lulu Liu, and Rod Ellis. Recoup—maintaining fortran. *SIGPLAN Fortran Forum*, 12(3):26–32, 1993.

[SYS06] Kamran Sartipi, Lingdong Ye, and Hossein Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 256–259. IEEE Computer Society, 2006.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Vid03] Germán Vidal. Forward slicing of multi-paradigm declarative programs based on partial evaluation. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR 2002)*, pages 219–237. Springer LNCS 2664, 2003.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362, London, UK, 2001. Springer-Verlag.

[VO01] G. Villavicencio and J.N. Oliveira. Formal reverse calculation supported by code slicing. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE 2001, 2-5 October 2001, Stuttgart, Germany*, pages 35–46. IEEE Computer Society, 2001.

[Wad92] P. Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2:461–493, 1992. (Special issue of selected papers from 5'th Conference on Lisp and Functional Programming.).

[War02] Martin P. Ward. Program slicing via fermat transformations. In *Proc. of 26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, 26-29 August 2002, Oxford, UK*, pages 357–362. IEEE Computer Society, 2002.

[War03] Martin P. Ward. Slicing the scam mug: A case study in semantic slicing. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*, pages 88–97. IEEE Computer Society, 2003.

[Wei79] M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Methods.* PhD thesis, University of Michigan, An Arbor, 1979.

[Wei82] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[Wig97] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[WL86] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[WLF01] M. Wermelinger, A. Lopes, and J. Fiadeiro. A graph based architectural (re)configuration language. In V. Gruhn, editor, *Proc. of ESEC/FSE'01.* ACM Press, 2001.

[WR03] M. Walkinshaw and M. Roper. The java system dependence graph, 2003.

[WZ07] Martin P. Ward and Hussein Zedan. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2):1–52, 2007.

[WZH05] Martin P. Ward, Hussein Zedan, and T. Hardcastle. Conditioned semantic slicing via abstraction and refinement in fermat. In *Proc. of 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK*, pages 178–187, 2005.

[YC79] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1979.

[Zha98a] Jianjun Zhao. Applying program dependence analysis to java software. In *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, December 1998.

[Zha98b] Jianjun Zhao. Applying slicing technique to software architectures. In *Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems*, pages 87–98, August 1998.

[ZKG02] W. Zhao, D. Kearney, and G. Gioiosa. Architectures for web based applications. In *Proc. AWASA 2002*, 2002.

[ZXCH07] Qui Zongyan, Zhao Xiangpeng, Cai Chao, and Yang Hongli. Towards the theoretical foundation of choreography. In P. Patel-Schneider and P. Shenoy, editors, *Proceedings of the*

16th Int Conf. on World Wide Web, pages 973–982. ACM, 2007.

[ZXG05] Yingzhou Zhang, Baowen Xu, and Jose Emilio Labra Gayo. A formal method for program slicing. In *2005 Australian Software Engineering Conference (ASWEC'05)*, pages 140–148. IEEE Computer Society, 2005.

[ZXS+04] Yingzhou Zhang, Baowen Xu, Liang Shi, Bixin Li, and Hongji Yang. Modular monadic program slicing. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 66–71, Washington, DC, USA, 2004. IEEE Computer Society.