View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositório da Universidade Nova de Lisboa

Francisco José Marcelino e Cunha

Bachelor in Computer Science

Optimizing Service Orchestration in OutSystems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Advisers: Carla Ferreira, Associate Professor,
NOVA University of Lisbon
Paulo Ferreira, Software Engineer, OutSystems

Examination Committee

Chairperson: José Legatheaux Martins, Professor, NOVA University of Lisbon
Raporteur: António Menezes Leitão, Auxiliar Professor, IST
Members: Carla Ferreira, Associate Professor, NOVA University of Lisbon
Paulo Ferreira, Software Engineer, OutSystems

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Optimizing Service Orchestration in OutSystems

Copyright © Francisco José Marcelino e Cunha, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculty of Sciences and Technology e a NOVA University of Lisbon têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To Zero, may he always play around in dog heaven.

Acknowledgements

First off, I would like to thank Faculdade de Ciências e Tecnlogia da Universidade NOVA de Lisboa (FCT-UNL), in particular the Informatics Department, for equipping me with the essential skill set that will allow me to start off my professional career. Thank you to OutSystems for providing a scholarship for this dissertation.

A big thank you to my advisers, Carla Ferreira and Paulo Ferreira for the mentoring you granted me throughout this dissertation. You gave me the freedom to progress my own way, but also total support whenever I needed. I still smile as I recall some of the best moments in our regular meetings – those were great!

Thank you to my friends and colleagues at OutSystems, with whom I shared so many laughs and snooker games, amusing lunchtime conversations and trips to the local supermarket. Those were António Ferreira, David Mendes, Francisco Magalhães, Joana Tavares, João Gonçalves, Lara Borisoglebski, Mariana Cabeda, Michael Silva, Miguel Loureiro, Miguel Madeira, Nuno Calejo and Nuno Pulido. You all made going in to work infinitely better, and I am thankful for having had the amazing company of every single one of you every day throughout this journey.

A huge thank you to my family. To my parents for their absolute, unconditional, wholehearted support. To my brother for the amusing banters that never failed to brighten up my mood. To my sister for being so understanding and for the great conversations that always helped me clear my mind. Also thank you to Twix, our four-legged, energy-filled family member who makes it an absolute joy to come home to.

Finally, a special thank you to my dearest, closest friends who mean the world and beyond to me. You know who you are. Without you at my side I would have never been the person I am today. The merit of this work – and of everything I have achieved thus far – is also yours, and for that I am forever grateful.

You have to always think one step ahead.
Like a... carpenter... that makes stairs...

Andy Bernard, The Office

Abstract

The growing demand for continuous development and deployment is causing many to steer away from the traditional monolithic architectural style and opt instead for Service-Oriented Architectures (SOAs). Adopting an architecture that is based on loosely-coupled services leads to enhanced modularity and flexibility, further translated into a philosophy of iterative, evolutionary development. The benefits of this development pattern were also made available in the *OutSystems* low-code platform, with the introduction of services as a new development building-block.

Moreover, the *independence* innate to the multiple services that comprise a SOA hints at possible opportunities for task parallelism: as long as different remote calls to services don't interfere with each other, they could be performed *in parallel*. As an immediate result, there could be speedups in multiple parts of an application's layers. Idle time waiting for data could be reduced, along with internal business logic that could be carried out faster, factors that would positively impact the overall flow of *any* application.

In this thesis we propose the design and implementation of an optimization process that targets the *heart* of the SOA: the *orchestrator* itself, the conductor of service interaction that enables the different business processes involved in a software system. The work produced comprises a set of data analysis and representation techniques that work together with the goal of detecting and informing opportunities for safe parallelism in the interaction and composition of the services that make up software factories. The formal definition of the algorithm is accompanied with a prototype that targets the OutSystems platform, with the achievement of considerable speedups in common scenarios. The results obtained suggest the viability of such a mechanism in the world of SOAs.

Keywords: OutSystems, OutSystems Platform, Data-Flow Analysis, Control-Flow Analysis, Parallelization Techniques, Dependence Analysis, Dependence Representation

Resumo

A popularidade crescente de técnicas de *continuous development* e *continuous integration* está a levar muitos a deixar o tradicional estilo de arquitetura monolítica e optar, em vez disso, por arquiteturas orientadas a serviços (SOAs). A adoção de uma arquitetura baseada em serviços mais desacoplados resulta em melhorias de modularidade e flexibilidade, traduzindo-se numa filosofia de desenvolvimento iterativo e evolutivo. Os benefícios deste padrão de desenvolvimento foram disponibilizados na plataforma *low-code OutSystems*, com a introdução de serviços como uma nova abstração de desenvolvimento.

Para além disso, a *independência* inata aos múltiplos serviços que compõem uma SOA sugere possíveis oportunidades para o paralelismo de tarefas: desde que diferentes chamadas remotas a serviços não interfiram umas com as outras, podem ser realizadas em paralelo. Como resultado imediato, poderia haver melhorias de performance em várias partes da aplicação. O tempo de espera por dados poderia ser reduzido, juntamente com a lógica interna da aplicação, que poderia ser executada mais rapidamente, fatores que impactariam positivamente o fluxo geral de qualquer aplicação.

Nesta tese propomos o desenho e implementação de um processo de otimização cujo alvo é o *coração* da SOA: o próprio orquestrador, o condutor das interações entre serviços que permite a execução dos diferentes processos envolvidos num sistema de software. O trabalho realizado foca a composição de um conjunto de técnicas de análise e representação de dependências numa cooperação que visa identificar oportunidades para a paralelização das chamadas aos diferentes serviços que compõem uma fábrica de *software*.

A definição formal do algoritmo desenvolvido é acompanhada por um protótipo desenvolvido para a plataforma *OutSystems*, no qual foram verificados *speedups* consideráveis em cenários de operações comuns. Os resultados obtidos sugerem a viabilidade de uma solução desta natureza no mundo das SOAs.

Palavras-chave: OutSystems, Plataforma OutSystems, Análise de Data-Flow, Análise de Control-Flow, Técnicas de Parallelização, Análise de Dependências

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Solution Overview . . . 3
1.4 Contributions . . . 4
1.5 Outline . . . 5

2 Background **7**
2.1 The OutSystems Platform . . . 7
2.1.1 Service Studio . . . 7
2.1.2 Exposing and Reusing Functionality . . . 9
2.1.3 Server Actions and Service Actions . . . 9
2.1.4 Database Manipulation Primitives . . . 10
2.2 Key Concepts . . . 12
2.2.1 Dependence Analysis . . . 12
2.2.2 Data Dependence . . . 13
2.2.3 Control Dependence . . . 16
2.2.4 Representing Dependences . . . 19
2.3 The Three Rules for Safe Parallelism . . . 21

3 Related Work **23**
3.1 Static Analysis Techniques . . . 23
3.1.1 The Limitation of Static Analysis . . . 24
3.2 Dynamic Analysis Techniques . . . 25
3.3 Hybrid Analysis Techniques . . . 27
3.4 Discussion . . . 29

4 Implementation **31**
4.1 Defining a Grammar . . . 31
4.2 Solution . . . 33
4.2.1 Overview . . . 35
4.2.2 Read-Write Set Extraction . . . 35
4.2.3 Data-Flow Analysis . . . 39

4.2.4 Control-Flow Analysis 44
4.2.5 Program Dependence Graph 49
4.2.6 Analysis of Parallelism 50
4.3 Prototype in OutSystems 54

5 Evaluation **59**
5.1 Common Patterns 59
5.2 On the Development Guidelines Proposed 68

6 Conclusions **71**
6.1 Contributions 72
6.2 Future Work 72

Bibliography **75**

Chapter 1

Introduction

This chapter introduces the thesis, contextualizing the problem and discussing the motivation behind it. After an overview of the proposed solution and expected contributions, the outline of the remaining document is reviewed.

1.1 Context

The paradigm of Service-Oriented Architecture (SOA) has been gaining significant traction in the IT world. Microservices, in particular, have become a dominant architectural style choice in the service-oriented software scene, with many large-scale Web companies thoroughly using them [4]. The adoption of SOA raises the degree of independence in both development and deployment life-cycles. The increased reuse of functionality, coupled with a higher flexibility to evolve the software system make up some of the top drivers of SOA, according to a survey conducted by Baroudi et al. [6].

A fundamental aspect of any SOA is the behind-the-scenes interaction and composition of independent services that takes place when higher-level endpoints are consumed, in order to carry on specific business processes. This coordination can be done in two main ways, orchestration and choreography, with our focus being on the former. The *orchestrator* works as a centralized composite service that invokes and combines the different services in a coordinated fashion; this makes sensible orchestration essential for the success of SOA systems.

An orchestrator has to coordinate several requests, respecting the dependencies among them and any protocols they might follow. Naive orchestration strategies perform the requests sequentially, spending time just waiting for the results. Moreover, the orchestrated services themselves may be orchestrators as well, meaning that this idle time waiting for results can grow surprisingly fast.

A possible strategy to improve the efficiency of the orchestration process is to apply *task parallelism* when performing requests that are independent of each other. There is plenty of investigation and relevant work around the identification of parallelism, alongside the verification of its correctness. Techniques range from static, to dynamic, to hybrid. Static techniques are typically based on dependence analysis algorithms [1, 18, 37], whilst dynamic ones tend to focus on speculative execution [12, 15, 45] or (dynamic) transactional memory [23, 41]. There is also a body of hybrid techniques [16, 40] that combine elements from both approaches.

1.2 Motivation

The OutSystems Platform is a low-code development platform that enables rapid development of applications in a mostly-visual way. Furthermore, the introduction of services in OutSystems 11 made it possible to opt for SOA as the structural backbone of one's application. The highly independent nature of SOAs fuels an idea of potential task parallelism between service calls that do not depend on each other. When developing an application in the OutSystems platform, we have knowledge of consumed APIs and their dependencies within a software factory. We can thus leverage that knowledge in an attempt to inform parallelization of the orchestration that guides the communication between the different services in runtime. In other words, remote calls to independent services could be performed in parallel, bringing significant speedups in many scenarios.

To exemplify, consider a hypothetical online platform focused around books, readers and book reviews, architected under a service-oriented model, with independent services for books details, readers and reviews. Using a search feature, the visitor can get the basic details for a particular book, given that book's title. In the meantime, the readers and reviews for the particular book are also fetched and then displayed.

Behind the scenes, this interaction flow involves *three* API calls to *three* different services. First, one to the `books` service, in order to fetch the details of the book whose title matches the name passed by the user. Among these details is the book's identifier, which is then used as input to the service calls to the `readers` and `reviews` services.

Figure 1.1 illustrates how that task is currently done in OutSystems. Each node represents the respective service call. The figurative time elapsed between the moment a call is started and the moment it returns is labeled on top of each node.

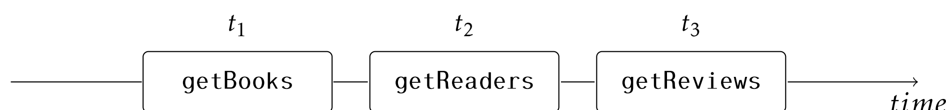

Figure 1.1: Performing remote service calls sequentially.

Unless the developer resorts to custom code, the three calls are handled synchronously and one after the other. On this sequential process, as portrayed in Figure 1.1, the total

time elapsed for all the information to be ready for the visitor would be given by $t_1 + t_2 + t_3$.

Now consider Figure 1.2. What if we knew that, after the call to the `books` service, the calls to the `readers` and `reviews` services could be carried in parallel?

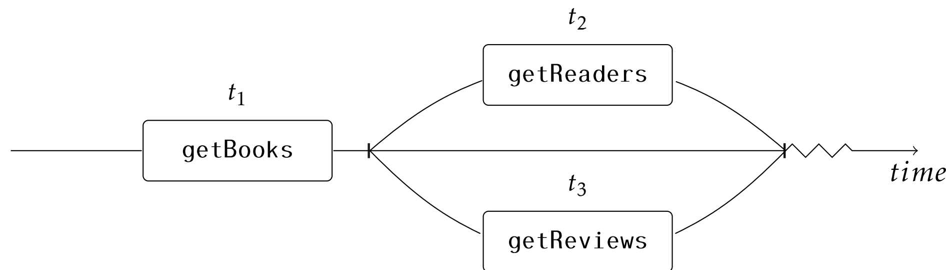

Figure 1.2: Performing independent remote service calls in parallel.

By parallelizing the independent remote calls, we would cut the total time needed from $t_1 + t_2 + t_3$ to $t_1 + max\{t_2, t_3\}$. Assuming that the three calls take similar time to complete, this would mean a speedup of roughly 33% regarding the sequential version.

1.3 Solution Overview

There are a few key principles that guide the design and behaviour of our proposed solution. First and foremost, the solution must always guarantee *correctness* of the optimized program. Not finding opportunities for safe parallelism could be an acceptable worst case scenario; marking as *parallelizable* tasks that are effectively *not* safe to parallelize, however, is never an acceptable output.

Moreover, the solution should introduce little-to-no runtime overhead; the number of remote API calls inside OutSystems software factories could be arbitrarily large, and so even a small amount of runtime overhead could cause some performance hindrance.

Finally, to implement a solution that is (mostly) invisible to the OutSystems developer is another goal we consider rather important. Our solution should never impose an obstacle nor require extra effort from the developer. Whether or not it finds opportunities to optimize the program, that shouldn't be noticeable on the development process.

With this in mind, the solution we propose has *five* fundamental high-level steps. To start, a static analysis will scan the APIs of any remote services used in the application, extracting information regarding which database entities are accessed by them. For illustration purposes, we are here going to instantiate each step in the book-related scenario presented in the previous section. For the three service calls in Figure 1.1, this scan would yield the following results: *a*) none of the actions *modify* any database record, and *b*) all of them read resources from different database tables.

That information is then integrated in the next two steps, where data and control dependence analyses are run on the actions that *perform* the aforementioned remote calls. Figure 1.1 is, in fact, one such action; an encapsulation of business logic that happens to perform three remote API calls, one after the other. The dependence analyses will

conclude that there are no control dependences in our action – from Figure 1.1 this is easily seen, as there is no branching of control-flow. There will, however, be a data dependence detected: the calls to the `readers` and `reviews` services take as input a value that is only known after `getBooks` returns – the book's *identifier*, as mentioned in the previous section – and so that relative order must be preserved. Yet, no constraints between the `readers` and `reviews` calls are detected - no data nor control dependence, and no clashing database accesses - leading to the conclusion that the two can be parallelized.

After a fourth step, where the dependence information gathered is summarized into a single, easily-parsable structure, this information is used to separate the operations of the program into what would be dependence-independent sections. The entirety of this information would then be consumed on the fifth and last step of our solution, where a (safe) parallel version of the original program is generated. Though this last step is not implemented in the work, the intuition behind it is thoroughly discussed.

1.4 Contributions

We wrap up this dissertation with the study, documentation and running prototype – in the context of the OutSystems platform – of an optimization process for service orchestration. This optimization targets requests that require several remote API calls, with the aim of finding and applying opportunities for informed parallel execution. This work produced some results that stand out as individual contributions themselves:

- A static analysis algorithm that extracts information regarding database *reads* and *writes* performed by remote API calls. This can be mapped to different platforms and data sources;

- A dependence analysis algorithm, capable of summarizing both the control and data dependences of different operations in a program during compile time. The algorithm generates a Program Dependence Graph, and is capable of distinguishing between the different types of data and control dependences. Despite our target being the OutSystems platform, the algorithm is adaptable to other scenarios where a graph representation can be constructed;

- An algorithm capable of partitioning the operations of a program based on the dependences between the operations. This information can then be used to aid in the construction of the optimized program;

- A working prototype targeting (a sub-set of) the OutSystems platform that employs the techniques described above. Though not extensively tested, the prototype yielded favorable results, hinting at the viability and applicability of our solution.

1.5 Outline

The rest of this report is organized as follows:

- Chapter 2 - Background: this chapter presents background information on the OutSystems platform, followed by concepts important for this thesis, with an emphasis on dependence analysis calculation and representation;
- Chapter 3 - Related Work: this chapter elaborates on different techniques for parallelization, wrapping up with a discussion of their viability for our use case;
- Chapter 4 - Implementation: here our solution is covered in detail, with every step escorted with a practical example. Its application to the OutSystems model is also discussed, as we elaborate on the prototype developed;
- Chapter 5 - Evaluation: presents the results obtained for this work;
- Chapter 6 - Conclusions: this chapter concludes the dissertation. After high-level reflection on the solution developed and difficulties faced, a quick overview of the work produced takes place; to close off, we discuss what we identified to be worthwhile future work.

Chapter 2

Background

This chapter starts out by presenting some background for the OutSystems platform, the setting we are targeting. Alongside, relevant notions in the realm of program dependence analysis will be presented, due to their usefulness in understanding if parts of a program can be ran in parallel.

2.1 The OutSystems Platform

The OutSystems Platform is a low-code *aPaaS*[1] that enables rapid development, agile deployment and continuous management of cross-platform, enterprise-grade applications. Inside Service Studio, the OutSystems development environment, developers can build entire applications (from the user interface to the business logic) all in a visual, drag-and-drop fashion. Furthermore, integration with external services and custom code is made easy, meaning scalability and architecture are never compromised [30].

2.1.1 Service Studio

The OutSystems Service Studio is the low-code visual development environment for OutSystems applications. Inside the IDE, one can develop the different parts of the stack that make up a fully-fledged application, ranging from the user interface all the way to the database model. This is explained in more detail below.

Interface The elementary building block of User Interface (UI) development in OutSystems is the *Web Screen*, a component that will translate into one of the screens in the deployed application. Web Screens can be populated by composing visual widgets – such as buttons, check-boxes and lists – in order to achieve all sorts of layouts. Web Screens

[1] Application Platform as a Service: https://www.outsystems.com/p/platform-as-a-service/

may have input variables and local variables. More so, they can hold one *preparation action*, a place where the developer can define custom logic to be executed prior to the screen's rendering. The way screens flow onto one another can also be visualized, as illustrated in Figure 2.1.

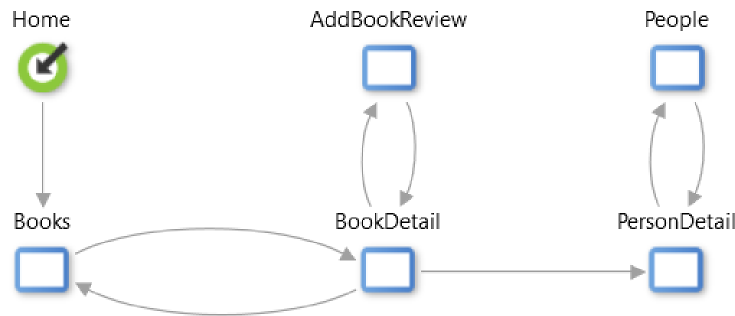

Figure 2.1: A simple UI flow of an OutSystems application.

Logic Pieces of an application's business logic can be encapsulated inside *actions*, which are developed in a visual, graph-like fashion using constructs such as conditions and assignments, database queries and even other actions. Actions can have input parameters, output parameters and local variables. It is also possible to integrate actions with external web services.

Figure 2.2 illustrates a simple Server Action, a type of action that runs logic on the server side. The orange node labeled `GetBooksByRating` is also a Server Action itself. The output of `GetBooksByRating` is assigned to some variable that is local to the illustrated action; this assignment is represented by the blue node labeled `Books`.

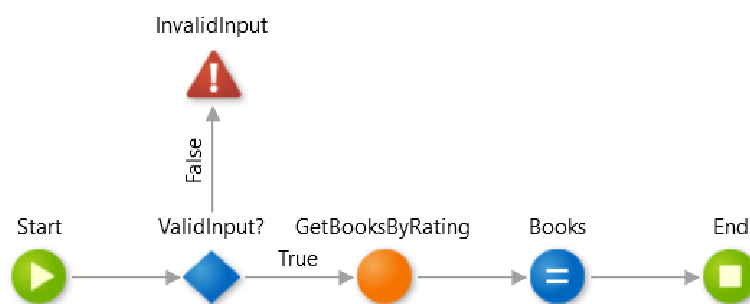

Figure 2.2: A simple Server Action.

Data Database modelling, including the management of *Entities* and their *Attributes*, is also done through Service Studio. An application's database schema (i.e., all the entities and their interactions with each other) can also be visualized (Figure 2.3). For the cases

when one is dealing with static data, the development experience can be further enriched with strongly-typed, enumerate-like entities called *Static Entities*.

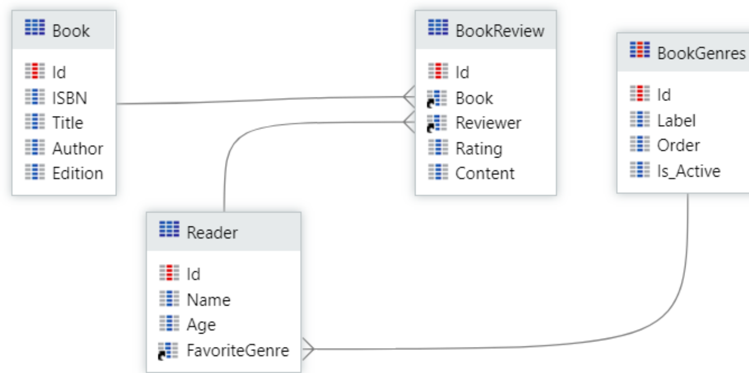

Figure 2.3: A simple database schema of an OutSystems application.

2.1.2 Exposing and Reusing Functionality

A typical OutSystems application is made out of multiple modules – named *eSpaces* – that can be used across different applications. This modular fashion highlights the importance of *reusability* all across the development process. The exposure and reuse of elements (e.g., actions, UI widgets or data models) creates a *producer-consumer* relationship: the module that implements and exposes functionality is the producer module, whereas the module that reuses the exposed functionality is the consumer module. Elements exposed by a producer module are considered as dependencies in the consumers [34].

When some exposed functionality is updated and the respective producer module publishes the changes, an automatic *impact analysis* step is performed. As we will discuss below, the impact of those changes in the running consumer modules will be vary according to the type of dependency between the producer module and its consumers.

2.1.3 Server Actions and Service Actions

In OutSystems, Server Actions (Figure 2.2) are said to run in-process, that is, they are executed in the same process as the consumer module, as if they were defined there. For this to be possible, the consumer module needs to know, in runtime, both the signature *and* the implementation of the element exposed by the producer. In other words, exposing a Server Action generates a *strong dependence* (Figure 2.4) from the consumer to the producer module, in a tightly-coupled way. Each time the implementation of an exposed Server Action changes, all the consumer modules must be refreshed and republished to start using the latest version.

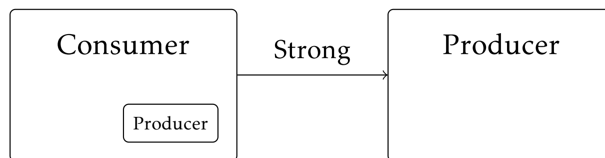

Figure 2.4: A *strong* dependence between producer and consumer modules.

With OutSystems' version 11, Services and Service Actions were introduced [35]. From the standpoints of development and usage, Service Actions are similar to Server Actions. The big difference, however, lies in the execution model and in the strength of the producer-consumer relationship that is created. Service Actions are REST-based remote calls; they are executed in a remote process, conversely to Server Actions. Exposing a Service Action generates a *weak dependence* (Figure 2.5) from the consumer to the producer module: each time the implementation of an exposed Service Action changes, that change takes immediate effect in the consumer modules.

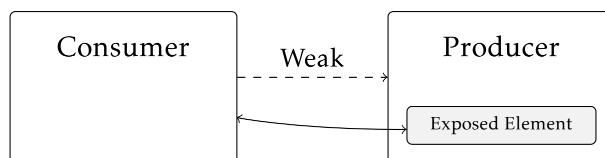

Figure 2.5: A *weak* dependence between producer and consumer modules.

In this work, we want to explore the potential for parallelization of Service Actions and similar abstractions to remote calls, where we have knowledge and control over both the producer and the consumer modules.

2.1.4 Database Manipulation Primitives

Of particular relevance to us is information regarding data accesses done by the producer and the consumer modules, so that possible interference can be identified. Thus, some insights about the existing database manipulation primitives are worth discussing here.

Foundational to database modeling in OutSystems is the concept of *Entity*, an element which allows the application developer to manage and persist business information. The information held by each Entity is stored in *attributes* – much like columns in a relational database table – and each instance of an Entity is called a record. For every Entity that is created, the OutSystems Platform generates a set of *Entity Actions*, SQL abstractions for simple CRUD[2] functionality to act upon single records of the Entity. Entity Actions make up one of the simplest primitives of database manipulation, and an example of such an action can be seen in Figure 2.6 on the left.

Data fetching with the aim of retrieving multiple records from an Entity – or information from multiple entities – is made possible with the use of *Aggregates* (Figure 2.6 in the

[2]Create, Read, Update and Delete, the four elementary functions of persistent storage.

middle). Despite how expressive the OutSystems Domain-Specific Language is for handling data (with these Entity Actions and Aggregates), there are times where the intricate nature of some business logic demands the need to write more complex queries. To account for those situations, the OutSystems platform enables custom SQL to be developed and encapsulated inside a *SQL* element (Figure 2.6 on the right).

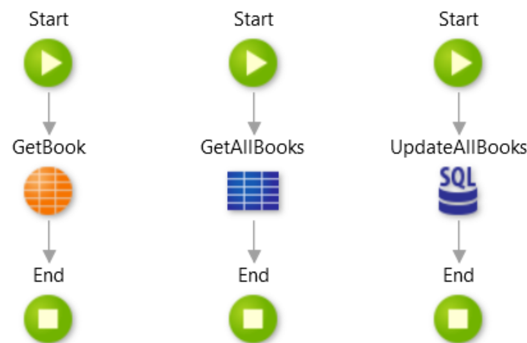

Figure 2.6: Different ways to manipulate a database.

Entity Actions are automatically named in a predictable way (e.g., `Get<Entity>`, `CreateOrUpdate<Entity>`), making it easy to identify which database table they read or write. When dealing with Aggregates, we can access their sources to understand what entities are being read – this can be seen in Figure 2.7, where an inner join between the `Book` and `BookReview` entities is performed.

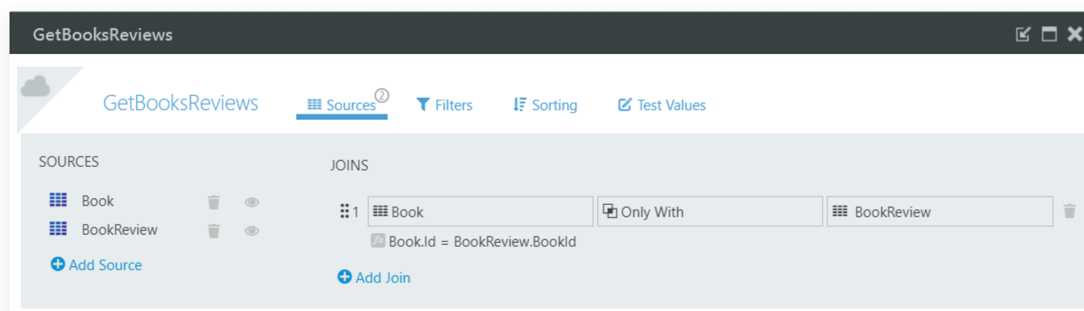

Figure 2.7: Sources view of an Aggregate in OutSystems.

Even when database manipulation is done with custom SQL, we can access the entities used by knowing that their names are always inside curly brackets, as seen in Figure 2.8. We could thus parse the queries in order to detect which entities are *read from* and which ones are *written to*.

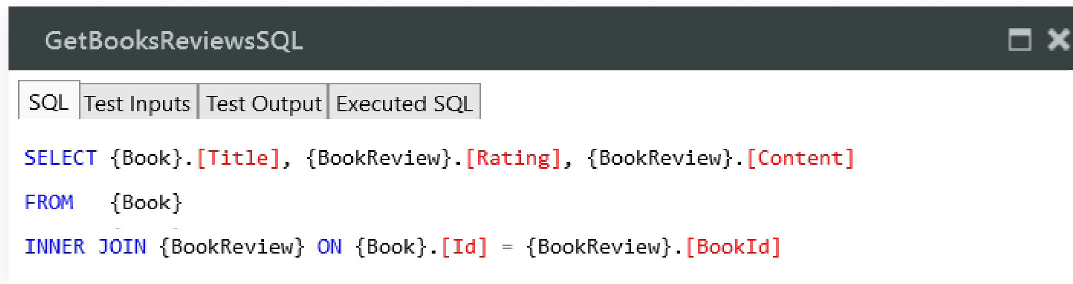

Figure 2.8: A snippet of custom SQL in OutSystems.

2.2 Key Concepts

This section provides some key concepts for a proper understanding of this thesis. To allow for a better flow, a simple code example, shown in Listing 2.1, will be used to instantiate the various concepts presented throughout this chapter.

```
x = 25;
y = x * 2;
if (x > 0)
  y = 0;
else
  y = 1;
x = 5;
```

Listing 2.1: A simple program.

2.2.1 Dependence Analysis

Dependence analysis is the process of studying the relationship between program elements (e.g., instructions or statements) in terms of the data they access and the execution paths they may follow.

In the context of program optimization, the set of all dependences for a program may be viewed as inducing a partial order that must be followed in runtime to preserve the semantics of the original program [18]. In compiler theory, *dependence analysis* commonly used to determine whether it is safe to reorder or parallelize statements [1].

Dependence analysis is versatile when it comes to the granularity of the operations to inspect. Besides individual statements, the analysis may be abstracted to higher levels; as an example, sequences of statements can be considered instead. Extending the analysis across multiple procedures (inter-procedural [25]) can also be done, though it comes with considerable overheads in run time and memory consumption [21]. Dependence analysis is twofold, consisting of both *data* and *control* dependence analysis, described in more detail in the following sections.

2.2.2 Data Dependence

A data dependence arises between two statements whenever they access the same variable and at least one of the accesses is a write (as opposed to a read). A statement S_2 is *data dependent* on statement S_1 if a variable appearing in S_2 may have an incorrect value if the two statements are reversed. This means that the execution order of the two statements is not interchangeable, and so they're not safely parallelizable.

Three types of data dependences can hinder opportunities for parallelism. A statement S_2 is flow dependent on S_1 if S_1 modifies a resource that S_2 reads and S_1 precedes S_2 in execution (represented as $S_1 \rightarrow S_2$). Notationally we write $S_1\ \delta^{\mathrm{f}}\ S_2$, where f stands for flow, the type of dependence. Furthermore, flow dependence may also be called Read-After-Write dependence, for which an example can be seen in Listing 2.2. Notice how, in the example, variable x is read in statement S_2 after being written to in statement S_1. If we were to reverse the order of execution of the two statements (i.e., if S_2 were to execute *before* S_1), then the value of variable y would be different than the one obtained with the original order.

```
Statement S1: x = 25;
Statement S2: y = x * 2;
```

Listing 2.2: Flow dependence example.

The opposite situation generates an anti-dependence (Write-After-Read): a statement S_2 is anti-dependent on S_1 (written $S_1\ \delta^{\mathrm{a}}\ S_2$) if S_2 modifies a resource that S_1 reads and $S_1 \rightarrow S_2$. An example of an anti-dependence can be seen in Listing 2.3. In the example, swapping the execution orders of the two statements would cause variable y to read a different value of variable x.

```
Statement S1: y = x * 2;
Statement S2: x = 25;
```

Listing 2.3: Anti-dependence example.

An output dependence between two statements occurs when the two modify the same resource. Thus, a statement S_2 is said to be output dependent on S_1 (written $S_1\ \delta^{\mathrm{o}}\ S_2$) if S_1 and S_2 modify the same resource, with $S_1 \rightarrow S_2$. Looking at Listing 2.4, we can identify an output dependence between statements S_1 and S_3; if we were to execute $S_3 \rightarrow S_2 \rightarrow S_1$ instead of $S_1 \rightarrow S_2 \rightarrow S_3$, then $S2$ would read different values of variable x.

```
Statement S1: x = 25;
Statement S2: y = x * 2;
Statement S3: x = 5;
```

Listing 2.4: Output dependence example.

Control Flow Graph

The base for many static techniques of dependence analysis is the Control Flow Graph (CFG) [3], a directed graph that represents all the paths that might be traversed through a program during its execution. The CFG is made out of a set of nodes $\mathcal{N}$ and a set of directed edges $\mathcal{E}$. Each node in the CFG is a basic block, i.e., a linear sequence of code without jumps or branching, and the edges guide the transfer of control flow between basic blocks. Many representations of the CFG have two special nodes to ease notation. Conventionally labeled as *Entry* and *Exit* [18], those nodes represent the single entry and exit points of the program, respectively. Every node is reachable from *Entry*, and *Exit* is reachable from every node. A *path* in the CFG from nodes X to Y (where $X, Y \in \mathcal{N}$) is denoted as $X \rightarrow^{+} Y$. Figure 2.9 shows a sample program and its respective CFG.

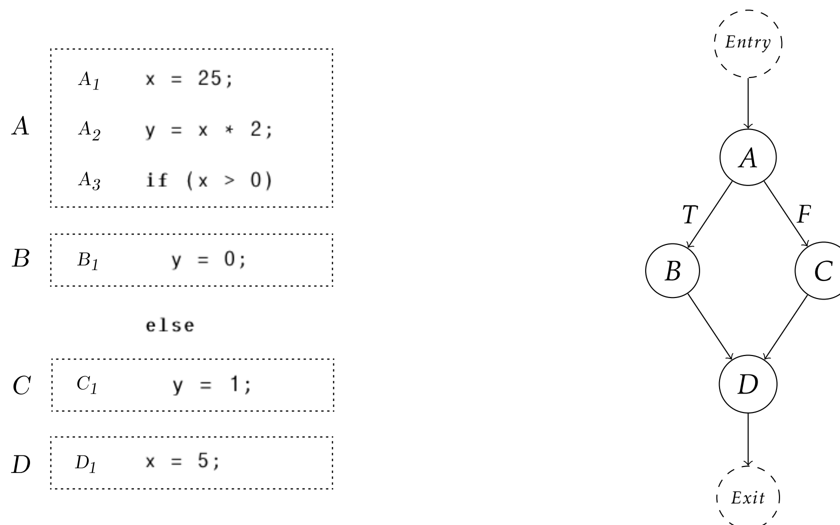

Figure 2.9: The CFG for the program of Listing 2.1.

Data-Flow Analysis

Data-Flow Analysis [1] refers to the set of compile-time analysis techniques that derive specific information about how data flows along a program by observing such flow at specific *program points*.

To perform any type of data-flow analysis we use the program's CFG as a starting point, considering the boundaries of each basic block as the program points of interest to consider – this is enough to summarize the flow of data inside the entire block. For each of these program points, a *data-flow value* is computed, an abstraction of the set of all possible *program states* that can be observed for that point. Each program state will hold just the information that is relevant to the particular analysis being applied.

In any kind of data-flow analysis, the goal is to gather knowledge at both the entrance and the exit states of each basic block. For some basic block b, we represent these knowledge points with IN_b and OUT_b, respectively. The way this information is gathered depends on whether the specific analysis is a forward analysis or a backwards one. In forward data-flow analysis, the exit state of a block is a function of the block's entry state; the other way around is true for backwards analysis.

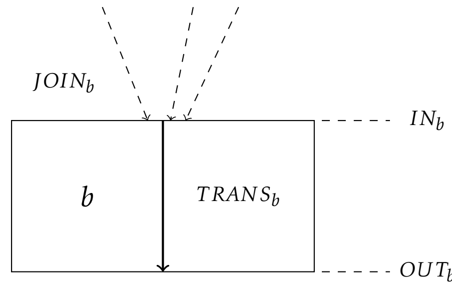

Figure 2.10: Forward data-flow analysis.

Figure 2.10 depicts the key principles of forward data-flow analysis [1]. In forward data-flow analysis, the knowledge at the entrance of a basic block b (IN_b) is a function of the exit states of b's predecessors. This function (the join function, here represented $JOIN_b$) merges all OUT_{b_i} for all basic blocks b_i that flow into b - in other words, the predecessors of b. The knowledge at the exit state of b, OUT_b, is a function of both IN_b and of what is inside b itself. This function that maps IN_b to OUT_b - called the transfer function - is here represented as $TRANS_b$. Mathematically, for some basic block b, IN_b and OUT_b can be defined as follows:

$$\begin{aligned} OUT_b &\triangleq TRANS_b\,(IN_b) \\ IN_b &\triangleq JOIN_{p \,\in\, predecessors(b)}\,(OUT_p) \end{aligned} \tag{2.1}$$

These equations hold for every data-flow problem; data-flow problems consist of solving these equations repeatedly until the the IN and OUT sets stabilize, i.e., reach what is called a fixpoint. Of course, the transfer and the join functions will vary, depending on the particular data-flow problem that is to be solved.

The computation of data-flow values is always an estimate. Knowing that, in general, the exact path that an executing program will effectively follow through cannot be known during compile time, the analysis assumes any path in the CFG can be taken, in a safe, conservative way. If that weren't the case, relying on its results for further computations could lead to errors.

Not only does data-flow analysis serve as a framework for numerous compiler optimizations [1, 42], but also for debugging tools [33], program verification [17] and parallelization [18]. Of particular relevance to us is the usage of these techniques to compute Read-Write Sets, the sets of variables read and written by a statement or program.

Read-Write Sets

For conciseness, let $\mathtt{R}(S)$ and $\mathtt{W}(S)$ be the sets of variables read and written by, respectively, statement S, and let S_1 and S_2 be two statements where $S_1 \rightarrow S_2$. Under a scenario where a variable is first written in S_1 and then read in S_2 we know for sure that $\mathtt{W}(S_1) \cap \mathtt{R}(S_2) \neq \varnothing$. Notice how this is the exact definition of flow dependence, and so we get to the following equivalence: $\mathtt{W}(S_1) \cap \mathtt{R}(S_2) \neq \varnothing \iff S_1\ \delta^{\mathrm{f}}\ S_2$. Similar equivalences can be achieved for anti and output dependences, if we consider $\mathtt{R}(S_1) \cap \mathtt{W}(S_2)$ and $\mathtt{W}(S_1) \cap \mathtt{W}(S_2)$, respectively. The absence of data dependence between any two statements S_1 and S_2 can thus be guaranteed when the following predicate holds:

$$[\mathtt{W}(S_1) \cap \mathtt{R}(S_2)] \ \cup\ [\mathtt{R}(S_1) \cap \mathtt{W}(S_2)] \ \cup\ [\mathtt{W}(S_1) \cap \mathtt{W}(S_2)] = \varnothing \tag{2.2}$$

This is called Bernstein's Condition [9], and is fundamental in the detection of opportunities for parallelism, as will be discussed in Section 2.3.

2.2.3 Control Dependence

A statement S_2 is *control dependent* on S_1 (notationally $S_1\ \delta^{\mathrm{c}}\ S_2$) if the execution of S_2 is conditionally guarded by S_1. Consider the example shown in Listing 2.5, where S_2 may or may not execute, depending on the outcome of S_1. In this situation, we say that S_2 is control dependent on S_1.

```
Statement S1: if (x > 0)
Statement S2:   y = 0;
```

Listing 2.5: Control dependence example.

Dominance and Postdominance

In [18], Ferrante et al. introduce a formal definition of control dependence in terms of a CFG and *postdominators*. In graph theory, the concept of dominance [39] is defined as follows: A CFG node X *dominates* a node Y if every path from *Entry* to Y has to pass through node X. By definition, every node dominates itself, and thus, if X dominates Y and $X \neq Y$ then X is said to *strictly* dominate Y.

It is not uncommon for a CFG node to have more than one dominator. The *immediate dominator* of a node X is the strict dominator of X that is closest to X on any path from *Entry* to X in the CFG. Using this notion of immediate dominance, a *dominance tree* can be constructed as a compact representation of the dominance relationship between nodes. The dominator tree has the same nodes as the CFG, albeit in a different order. A node in the dominator tree dominates all its descendants and immediately dominates all its (direct) children.

Here's once again the Control-Flow Graph for our example program:

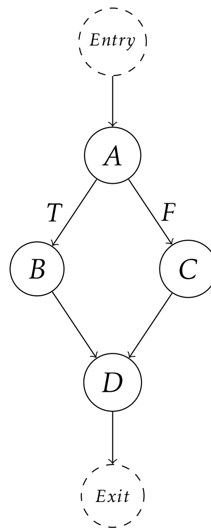

Figure 2.11: The Control-Flow graph for the program of Listing 2.1.

The dominator and postdominator trees for our example program would be as depicted in Figure 2.12 below:

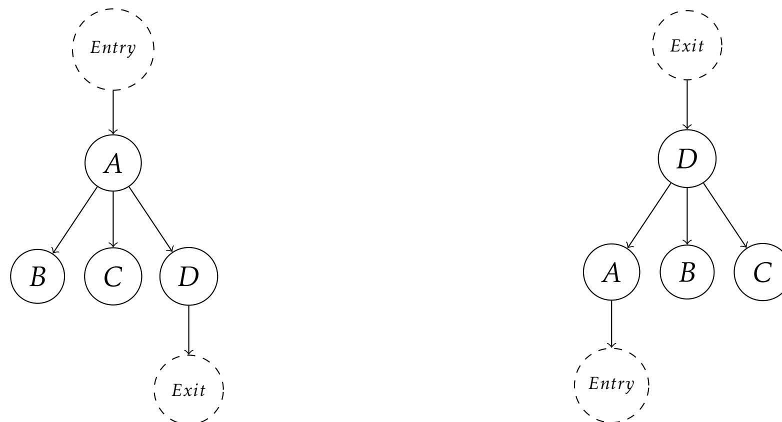

Figure 2.12: Dominator (left) and postdominator trees for the program of Listing 2.1.

The dominator tree and the CFG can be used together to construct a *dominance frontier*. A node Y belongs to the dominance frontier of a node X (written $Y \in \mathtt{DF}(X)$) if and only if X dominates some predecessor of Y but does not strictly dominate Y.

Conversely, if a node Y appears in every path from node X to *Exit*, then Y is said to *postdominate* X - or *strictly* postdominate X, if $X \neq Y$. Postdominance plays a fundamental role in control dependence analysis. As per the original definition from [18], two conditions must hold for there to exist control dependence between two CFG nodes. A node Y is *control dependent* on another node X if and only if:

1. There is a path $X \rightarrow^+ Y$ such that Y postdominates every node between X and Y;

2. X is not postdominated by Y.

This relationship can be represented directly with a *postdominance frontier* (PDF). Intuitively, PDF(N) is the set of blocks closest to N where a choice was made of whether to reach N or not. A node X is thus control dependent on another node Y if and only if $Y \in$ PDF(X). Figure 2.13 shows the DF and the PDF for every node of the original program's CFG, which can be constructed with the aid of both the CFG and the respective dominance/postdominance tree.

Node (N)	A	B	C	D
DF(N)	$\varnothing$	$\{D\}$	$\{D\}$	$\varnothing$

Node (N)	A	B	C	D
PDF(N)	$\varnothing$	$\{A\}$	$\{A\}$	$\varnothing$

Figure 2.13: Dominance (left) and postdominance frontiers for the program of Listing 2.1.

Calculating the postdominance frontier in the CFG is shown equivalent [18] to calculating the dominance frontier in the *reverse control flow graph*[3], and, in practice, it's easier and more efficient to opt for the latter.

The state-of-the-art is extensive when it comes to finding dominators [11, 14, 20, 29, 36]. Among the most widely adopted is the algorithm proposed by Tarjan and Lengauer [29], based on depth-first search and union-find. With E and N representing the number of edges and nodes of a graph, respectively, the simplified version of their algorithm achieves an asymptotic complexity of $O(E * log(N))$, with the more sophisticated version granting an almost-linear $O(E * \alpha(E,N))$, where $\alpha(E,N)$ is a very slow-growing function. Other interesting approaches include that of Cooper et al. [14], who tackle the problem of finding dominators with a data-flow oriented approach, ending with a much simpler – though asymptotically bigger ($O(N^2)$) – algorithm than Tarjan and Lengauer's.

On Termination Sensitivity

The definition of control dependence presented by Ferrante et al. [18] is often referred to as *standard* (or *classical* [10]). However, other definitions have proved useful under certain scenarios, and so they're worth considering. One of them is the notion of *weak* control dependence, as introduced by Podgurski and Clarke [38].

By definition, the dominator-based approach to find standard control dependences only works under the assumption that the program terminates, i.e., reaches $Exit$. Weak control dependence analysis, on the other hand, is termination-sensitive, meaning it also captures the possibility of non-termination. In [21], the authors implement and evaluate termination-sensitive support for weak control dependences in the context of information flow, reporting a considerable hit in the precision of their analysis, due to the increased conservatism associated with this type of dependences. Unless we find

[3]The *reverse control flow graph* (reverse CFG) can be obtained by reversing the direction of every edge in the original CFG. The reverse CFG will have the same number of nodes and edges as the regular CFG. For the dominance and postdominance computations, the roles of the $Entry$ and $Exit$ nodes are reversed as well.

relevant scenarios that indicate otherwise, assuming termination should be reasonable for our work. Therefore, standard control dependences should be sufficient.

2.2.4 Representing Dependences

Graphs are a flexible, widely used way of representing programs and their dependences. With the program CFG as starting point, one can work their way to build graph representations that showcase data dependences and control dependences individually or collectively.

Data Dependence Graph and Control Dependence Graph

DFA techniques can be applied to gather the information needed to build a Data Dependence Graph [27]. There is an edge $X \rightarrow Y$ in the Data Dependence Graph if and only if Y is data dependent on X. Flow, anti and output dependences are differentiated and marked accordingly in each edge.

Control dependences can be represented with the Control Dependence Graph [18], which can be derived directly from the postdominance frontier of the CFG - or, equivalently, the dominance frontier on the reverse CFG. Let $\mathtt{DF}_\mathtt{R}(Y)$ denote the dominance frontier of some node Y in the reverse CFG. Node Y is said to be control dependent on another node X if and only if $X \in \mathtt{DF}_\mathtt{R}(Y)$. Such dependence is represented with an edge $X \rightarrow Y$ in the CDG. The Control Dependence Graph and the Data Dependence Graph for the program of Listing 2.1 can be seen in Figure 2.14. Notice how the nodes of the Control Dependence Graph are the program's basic blocks, while the nodes of the Data Dependence Graph are the program's statements. This is due to the fact that control flow does not change inside a basic block, but data flow does.

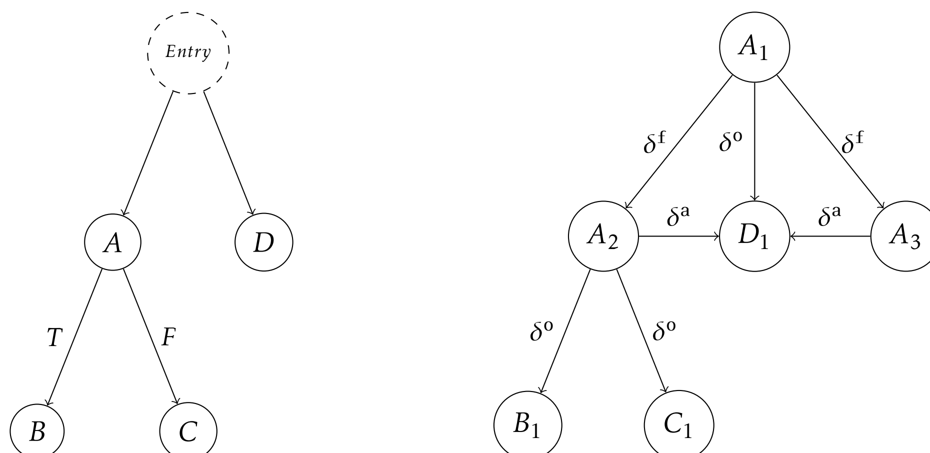

Figure 2.14: The Control Dependence Graph (left) and the Data Dependence Graph for the program of Listing 2.1.

Space efficiency and reasonable scalability are among the main benefits of these graphs that isolate the representation of dependences. However, having separate graphs

for the different types of dependences may not be practical for all cases. Certain computations or optimizations that require interaction of both data and control dependences, such as *code motion*[4], would require one to create, manage and traverse the two graphs, which could be inefficient.

Hybrid Dependence Graphs

Approaches that represent the two types of dependences in a single graph are known as *hybrid* [37]. The Program Dependence Graph [18] is an example of such a graph, consisting of a Data Dependence Graph augmented with control dependence arcs. This can be an improvement over maintaining and traversing two different structures, assuring the Program Dependence Graph many applications in the realm of compiler optimization, namely in the detection of parallelism. The original algorithm behind the creation of this graph involves many of the analysis techniques described in Section 2.2.2 and 2.2.3. Figure 2.15 shows the Program Dependence Graph for the program of Listing 2.1.

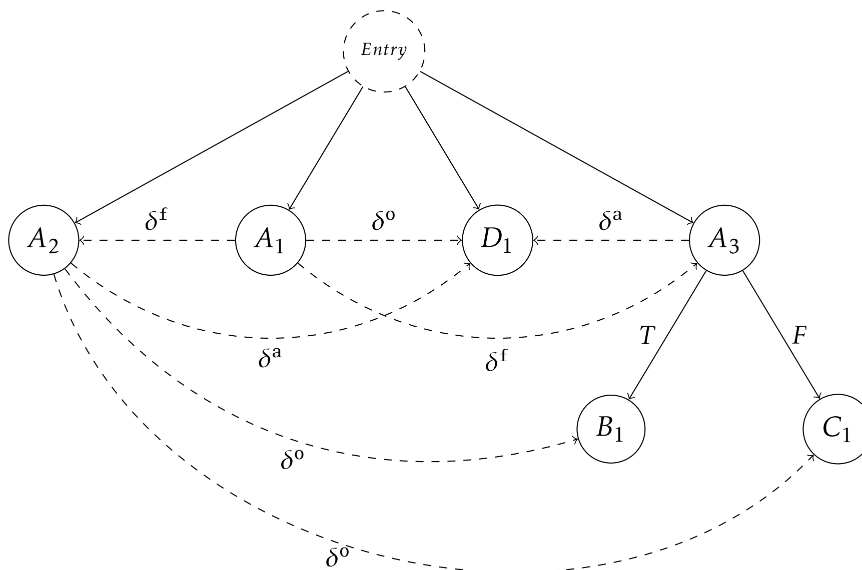

Figure 2.15: The Program Dependence Graph for the program of Listing 2.1.

In an attempt to address some of the drawbacks of the Program Dependence Graph – namely the spatial complexity involved and the difficulties in formal program correctness verification [26] – Pingali et al. introduced the Dependence Flow Graph [26, 37].

While the Program Dependence Graph merely represents the two types of dependences, the Dependence Flow Graph *combines* them. The nodes in a DFG communicate with each other by exchanging information along the arcs, a task made possible by the

[4] Code motion is a compiler optimization technique that consists of moving operations around in a way that improves performance without compromising the semantics of the original program [1].

introduction of special operator nodes. To handle data flow, a global store model is implemented with the aid of special `load` and `store` operators. Additional `switch` and `merge` operator nodes are used to effectively combine control information with data dependence information. This results in an executable hybrid graph, which the authors argue facilitates both progressive development and proofs of correctness.

2.3 The Three Rules for Safe Parallelism

We can now start expanding on a concept of *safe parallelism*. The absence of *both* data and control dependences is a requirement for running operations in parallel in a way that does not alter the behavior of the sequential program. Moreover, the presence of *either* one of those dependences is enough to invalidate opportunities for safe parallelism. This knowledge might prove useful to short-circuit analysis algorithms.

Nevertheless, two otherwise parallelizable operations, i.e., operations that share no data nor control dependences, might not be safe to execute in parallel if at least one of them causes side-effects that might harm the behavior of the program. The relevance of this characteristic is more evident when we enter the realm of speculative execution, as will be discussed in the upcoming chapter.

We can thus say that two operations A and B are *safely parallelizable* if and only if all the following conditions hold:

1. A and B are not data dependent on each other;
2. A and B are not control dependent on each other;
3. No ordering of their execution ($A \rightarrow B$ or $B \rightarrow A$) could potentially cause unwanted side-effects.

Finding opportunities for *safe parallelism* will be the focus of this work; this term will be used throughout the next chapters.

Chapter 3

Related Work

This chapter starts with an overview of some key static analysis techniques to find opportunities for task parallelism. As we come across the inherent conservatism of purely static approaches, the idea of speculation is brought into the spotlight, opening up the door for a whole body of dynamic and hybrid approaches. To finish off the chapter, we discuss the techniques presented and review their suitableness for our problem.

Many of these techniques are oriented towards low-level code and/or heavily focused on loops. Still, there are interesting ideas on these approaches, which makes their study and abstraction to a higher level an interesting challenge by itself.

3.1 Static Analysis Techniques

Purely static analysis techniques are heavily based on the dependence analysis techniques described in sections 2.2.2 and 2.2.3. The field of automatic optimization done by compilers is where the bulk of related work on the subject can be found.

The absence of runtime overhead makes static approaches enticing. Implicit to this body of techniques, however, is an inherent *conservatism* that can be seen as a double-edged sword. On the one hand, it assures the non-existence of false positives; the conservative analysis will never mark as parallelizable tasks that are effectively *not* safe to parallelize. On the other hand, conservatism can lead to false negatives, i.e., situations where opportunities for safe parallelism are discarded due to the uncertainty of the analysis that ends up opting for the safe way. This aspect of static analyses will be discussed throughout the chapter.

Global Code Scheduling

Global Code Scheduling (GCS) [1] is a compiler optimization technique that works by reordering operations across basic blocks, in ways that improve runtime efficiency. Based on the concept of *code motion*, GCS is made possible after a static dependence analysis step extracts the dependences that exist between operations. Once said extraction is complete, operations can be moved across basic blocks and reordered, as long as:

- There are no data dependence nor control dependence violations;
- All operations in the original program are executed in the optimized program;
- The optimized program does not cause unwanted side-effects.

Though the reordering itself does not necessarily imply parallelization, the constraints involved in the process of reordering make it similar to that of finding situations for safe parallelism (as described in Section 2.3).

3.1.1 The Limitation of Static Analysis

There are cases where performing a conclusive dependence analysis during compile time is simply not possible, e.g., when some information is only known in runtime, such as user input. This leads to static approaches that are highly conservative, which therefore fail to exploit certain opportunities for parallelism.

This property of static approaches has two sides to it. On the one hand, the high degree of conservatism guarantees correctness of the optimized program, considering that, under scenarios of uncertainty, a static analysis algorithm would opt for the safer route. On the other hand, programs structured in certain (common) patterns will tend to not benefit from added parallelism.

```
ActionA() {
  ...
  readFromBooksTable(...);
  ...
  ...
}
```

```
ActionB(write: Boolean) {
  ...
  if (write)
    writeToBooksTable(...);
  ...
}
```

Figure 3.1: Two actions that may or may not access the same data.

Consider the pseudocode example of Figure 3.1. On the left, `ActionA` reads from a `Books` database table. On the right, `ActionB` may or may not write to that same table, depending on the value of the user input variable `write`.

Assuming this eventual interference to be the only obstacle to the safe parallelization of `ActionA` and `ActionB`, could the two be safely parallelized? The impossibility of knowing the value of argument `write` during compile time would lead a static analysis

to conservatively opt for the safe route and declare unsafe the parallelization of `ActionA` and `ActionB`. This prospect hints at the existence of possible trade-offs and alternative solutions, and motivates the upcoming sections of this chapter.

3.2 Dynamic Analysis Techniques

Despite lacking the scalability of static techniques, purely dynamic approaches are plenty and bring interesting ideas to the table.

Software Thread-Level Speculation

Thread-Level Speculation (TLS) [43] is a dynamic parallelization technique that uncovers parallelism that static approaches fail to exploit, by launching multiple threads to perform different tasks, optimistically, only ensuring absence of dependence violations during runtime. Considering there is no actual reordering of operations, TLS ensures that the resulting parallel program's semantics remains true to that of the original program.

In a standard TLS implementation, the sequential program is first broken down into separate units of work (or tasks), each one to be executed on a different thread, according to a *thread spawning* strategy. Considering the efficiency of this entire execution model is highly impacted by the partitioning technique chosen, thread spawning should not be purely random, and different techniques exist to fit different needs [13, 31].

As the application executes, each thread running in parallel collects and maintains information regarding all of its accesses to shared memory in a *speculative buffer* [45]. The speculative buffer, local to each thread, keeps thread-specific changes until it is proven that no conflicts between threads occurred. Due to this uncertainty of success during execution, threads whose tasks access shared memory are said to be *speculative*.

During – or after[1] – speculative execution of each thread, a *conflict detection* phase takes place to ensure that there were no dependence violations between any threads running speculatively. This inspection will dictate whether it is safe to propagate the local changes to the shared memory – i.e., *commit*, if there were no conflicts – or if the buffered changes should be erased (or *rolled-back*), in case some conflict was detected. In case of a rollback, the conflicting thread is to execute again until success. The life-cycle of a speculative thread is summarized in Figure 3.2.

As opposed to hardware TLS [2, 19], software TLS [12, 15] is not restricted to specialized processors and/or modified memory systems. However, the need for software TLS to buffer speculative updates for an indefinite time period, only committing them to shared memory when the speculation is guaranteed to succeed, can lead to significant storage overheads. Introduced by Oancea et al., SpLIP [32] is an implementation of software TLS that addresses this issue by performing shared memory updates *in-place*, keeping an *undo*

[1] The conflict detection phase may take place on every speculative access (Eager Conflict Detection) *or* after a thread has finished its whole execution (Lazy Conflict Detection) [45].

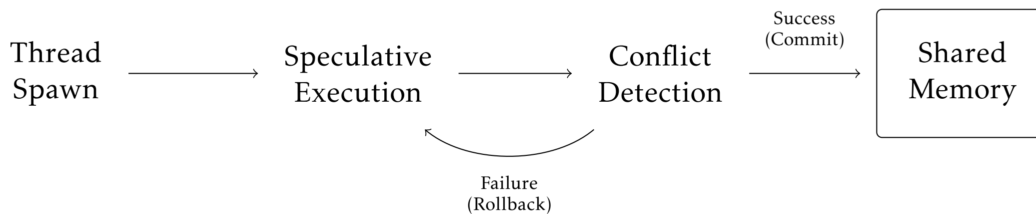

Figure 3.2: Life-cycle of a speculative thread.

log in case conflicts occur. This optimistic approach requires much lower storage needs and leads to faster commits. Potential rollbacks, however, are more expensive to perform than in common software TLS, making this technique rather weak for situations where the conflict rate is not minimal.

Software Transactional Memory

Software Transaction Memory (STM) is a concurrency control mechanism for shared memory systems – as opposed to a parallelization technique – pioneered by Shavit and Touitou [41]. Under the STM model, a shared memory is solely mutated by means of *transactions*, blocks of code that perform shared memory accesses and are meant to execute atomically. Blocks that are to run as transactions are manually annotated by the programmer, a task that is usually carried out in a declarative fashion, using high-level abstractions such as the one proposed by Harris and Fraser in [23]. Constructs like this make STM a rather concise and readable option, as can be seen in Listing 3.1.

```
atomic (i != 0) {
  shared_memory->read(x)
  shared_memory->write(y)
}
```

Listing 3.1: Example of a transaction, following a syntax similar to the one proposed in [23]. The use of a condition ($i \neq 0$, in this case) is optional. The operations on shared memory are wrapped inside an annotated block. Implementations of STM following this same style can be found for various languages, e.g., [5, 44].

While executing concurrently, transactions access and update the shared resources directly, without ensuring mutual exclusion – a behavior that makes STM a type of Optimistic Concurrency Control [28]. All the read/write accesses to shared memory – known as the *data set* of a transaction – are recorded in a log. As the transaction finishes its tasks, it checks for possible conflicts that may have happened in the meantime. If no conflicts were found, then the work performed by the transaction is effectively committed to shared memory. If, on the other hand, a conflict is found, one of the conflicting transactions is automatically aborted; its changes to shared memory are rolled-back and the transaction is rerun. This process is repeated until all transactions succeed.

TLSTM: Unifying TLS and STM

In [7], Barreto et al. propose TLSTM, a hybrid approach that complements Software Transactional Memory (STM) with Thread-Level Speculation (TLS). The way TLSTM operates is twofold. On a first step, the programmer manually annotates sections of code as transactions, as they would under standard STM. This results in a coarse-grained parallelization, where each transaction is to run on a different thread.

Right before the execution of each user-defined transaction, TLS is applied to speculatively break down the transaction into multiple, finer-grained tasks. For this second step, the spawning strategy can be backed up by compile-time code inspections, or it can be done purely in runtime.

Dependence-Aware Scheduling

Dependence-Aware Scheduling (DAS) [46], as introduced by Zhuang et al., is a dynamic parallelization technique that strives to reduce worst-case scenario slowdown. On its core, DAS works by simultaneously launching, on program start-up, a *main* thread that keeps running the program sequentially and a pool of *worker* threads.

While the main thread is running, the parallel worker threads calculate dependences for different, independent *slices* of the program not yet executed by the main thread. In case the dependences calculated do not constraint execution, i.e., slices do not depend on each other, then the worker threads execute the slices in advance.

It's worth noticing that the original focus of DAS is on loops. As a consequence, the slices analysed – and eventually executed – by the worker threads correspond to specific loop iterations. Before executing an iteration, the main thread checks if the work has already been done; if that's the case, then it skips the iteration and proceeds execution.

The effectiveness of this technique lies in both the slice calculation function and the orchestration work of a *scheduler* thread, that assigns different slices to the different worker threads. Nevertheless, on a worst-case scenario, i.e., a scenario where no opportunities for parallelism are found, the main thread runs the program sequentially, with the only slowdown-inducing overhead coming from the "completeness" checks.

Though originally focused on loop iterations, the nature and granularity of the slices to be examined by the worker threads could be adapted to one's needs. As an example, one could opt to steer the dependence analyses away from loop iterations and towards remote calls or specific procedures.

3.3 Hybrid Analysis Techniques

Hybrid techniques attempt to overcome the limitations of static approaches by extending the analysis to runtime whenever needed, i.e., when the information available at compile time is not enough to prove the presence – nor the absence – of opportunities for

parallelism. This extension tends to be done without resorting to speculative, conflict-management-inducing techniques so common in purely dynamic techniques.

Sensitivity Analysis for Automatic Parallelization

Rus et al. introduced Sensitivity Analysis (SA) [40], a hybrid technique for compiler optimization where low-overhead runtime evaluations complement an initial static analysis. This strategy is focused on the finding and parallelization of DO-ALL[2] loops, though the core idea should be applicable to other scenarios as well.

The SA algorithm works by statically creating a Dependence Set for every loop. A Dependence Set is defined as the intersection of the Read-Write Sets (discussed in Section 2.2.2) of the entire loop, across all iterations.

If enough information is available at compile time, then the Dependence Set will be conclusively proved either *empty* or *non-empty*, i.e., the loop will be proved parallel or not. The generated code for the loop will be either the parallel version or the sequential one, respectively. In case the problem cannot be solved in compile time, the Dependence Set is turned into a boolean predicate and inserted into the generated code, guarding a parallel version of the loop. The runtime evaluation of the predicate will then dictate which version of the code should be executed; the parallel one, in case the predicate evaluates to true, or the sequential one otherwise.

Optimistic Hybrid Analysis

More than strictly a parallelization technique, the authors of Optimistic Hybrid Analysis (OHA) [16] argue that their approach can be seen as a general-purpose framework for different types of analyses. Foundational to OHA is the idea that an unsound static analysis can be augmented with a sound dynamic analysis that executes speculatively, resulting in a fast and precise hybrid analysis.

An initial runtime profiling step takes place with the aim of gathering assumptions about the way the program executes. Since these dynamically-gathered assumptions are not guaranteed to be true for all execution scenarios, they are referred to as *likely invariants* of execution. Leveraging the likely invariants found, a subsequent static analysis step takes place. The likely invariants relax the analysis, making it less conservative – and thus more precise – than standard static analyses, at the cost of possibly lost soundness.

The soundness is thus guaranteed by a final dynamic analysis. This dynamic analysis is constructed optimistically, with just low-overhead checks to verify if the likely invariants assumed *do* hold for the analyzed execution. If all the invariants hold during runtime, the program is executed normally. Otherwise, execution is rolled-back and the program is executed again after a traditional, non-optimistic hybrid analysis takes place.

[2] A loop presents DO-ALL parallelism when no iterations depend on any other iteration [1].

3.4 Discussion

In this section we will weigh the positives and negatives of the examined strategies, and discuss their suitableness for our scenario.

The arbitrariness of program and action sizes to examine makes static analysis an all-around viable pick. A purely static analysis should be more scalable than any other type of analysis, due to not bringing runtime overhead. The fact that it is a conservative approach implies that any opportunities for parallelism found are guaranteed to be safe.

Software TLS, the first dynamic technique we examined, requires no extra programmer effort, as it is completely automatic. Yet, it comes with two big difficulties that escalate the runtime overhead: the thread spawning strategy and the presence of side-effects. The success of TLS is strongly dependent on the thread spawning strategy used. Without a very specific strategy, non-trivial non-numerical applications would end up having many conflicts, causing rollbacks that will significantly degrade performance [32]. Even with an optimal spawning strategy, speculative execution – and eventual re-execution – is dangerous because the absence of unwanted side effects is never guaranteed. This would, of course, be a violation of the definition of safe parallelism from Section 2.3.

With STM similar problems are encountered. STM is not fully automatic, requiring the programmer to manually annotate transactions. Despite being low effort and an overall elegant, composable technique, the manual process of specifying which operations are to execute atomically (as transactions) is error-prone. The programmer must have a deep understanding of the program semantics and make sure that the annotated blocks are commutative for every possible execution order. They need to be aware of side-effects, otherwise conflicts will happen, which may cause a large number of rollbacks/retries and/or undefined behavior.

Ideally, every transaction should be referentially transparent[3], i.e., every execution should produce the same result. Adding a verification step to check the correctness of the annotations could be a viable complement to this approach. Enforcement mechanisms for transactions to be side-effect free – so that they can be safely rolled-back – have been done before, namely in software TLS models implemented under the functional realm [22, 24], where it is possible to isolate side-effect-free computation from side effects.

TLSTM naturally inherits the drawbacks of the previous approaches. DAS, on the other hand, can exploit more opportunities for parallelism than purely static analysis *without* resorting to speculation or execution rollbacks. Despite being the only dynamic technique studied that can guarantee safe parallelization, we see two shortcomings of this approach. First, any dependence analysis further than the loop-level analysis that the authors work with might prove expensive to perform in runtime. Second, in the worst-case scenario, an arbitrarily heavy analysis will be performed, ending up finding no opportunities for safe parallelization. This worst-case scenario would be acceptable for

[3] Referential Transparency: https://wiki.haskell.org/Referential_Transparency

a static approach, but not so much for a dynamic one. The idea of performing dependence analysis in parallel is something to consider, though.

The hybrid solutions we studied are fully automatic, in that they require no additional intervention from the programmer. Furthermore, the fact that there is a static analysis phase *at all* translates into a runtime overhead that will typically be smaller than that of purely dynamic techniques. Yet, the two techniques are polarizing in the way the combine compile time and runtime computation. While SA can be seen as a mostly-static analysis technique that extends to runtime if needed, OHA is more of a runtime-heavy approach backed up by a prior compile time analysis.

OHA suffers from drawbacks similar to those of purely dynamic techniques, namely when it comes to speculative execution and rollbacks. SA, however, is an interesting technique for us to consider; it could be worth evaluating whether the benefits of extending the static analysis surpass the runtime overhead introduced.

Based on this study, we made some decisions regarding the development of our solution. Dynamic analyses are going to be avoided; though they *might*, in some cases, be more accurate than purely static analyses, their potentially unsafe behavior combined with the added runtime overhead go against two of the three design goals we described in Chapter 1 (those were guaranteeing program correctness, focusing on low overhead in runtime and keeping it all effortless and invisible to the developer). In fact, those goals are so important to us that our solution will focus on purely static analyses. SA, the mostly-static hybrid analysis technique, could be interesting to (potentially) enhance the final code generation phase. It would require a considerable amount of additional work, though, pushing it out of the time scope of this dissertation. Nevertheless, it could be worth considering for future work.

Chapter 4

Implementation

This chapter thoroughly covers the solution implemented during this dissertation. Every algorithm used and/or developed will be explained, discussed and exemplified in a more abstract, general-purpose way. Later on the chapter, we will focus on how these techniques were applied to the prototype developed for the OutSystems platform. This will come accompanied with relevant implementation details and technical considerations.

The chapter is organized in three sections. The first section will present a grammar we defined with the aim of formalizing the scenarios that our solution can cover. Next, the overall solution is discussed, with a focus on the design decisions that motivated each individual step and always escorted with an illustrative example. To finalize, on the third and last section of this chapter an exhibition and discussion of the prototype developed for the OutSystems platform will take place.

4.1 Defining a Grammar

Despite the focus on the OutSystems platform, it is important to understand that the algorithms that make up the solution that follows are, on a high level, applicable to both different programming languages and different scenarios. To better convey this notion, we defined a grammar (Figure 4.1) that represents the space in which our solution is applicable. The grammar will be used throughout the upcoming sections, abstracting away platform-specific notation and details whilst still being easily mappable *to* and *from* the OutSystems platform.

Regarding this grammar – and therefore the scenarios covered by our solution – there are a few important things to keep in mind. First and foremost, it is *module-oriented*; the system we represent can be seen as a set of modules. Besides having their own local definitions ($\vec{D}$), each module can import/reference functionality ($\vec{R}$) from other modules

$U ::= \vec{M}$	(system)
$M ::= \texttt{module}(m, \vec{R}, \vec{D})$	(module)
$R ::= \texttt{ref}(m, f, r, w)$	(value reference)
$D ::= n = \tau$	(type definition)
$\mid \texttt{def}\ f(n, \ldots n)\ s$	(function definition)
$s ::= s\ \texttt{;}\ s$	(sequence)
$\mid \texttt{while}\ (b)\ \texttt{do}\ s$	(while loop)
$\mid \texttt{if}\ (b)\ \texttt{then}\ s\ \texttt{else}\ s$	(conditional)
$\mid \texttt{write}(n, e)$	(write)
$\mid \texttt{return}\ e$	(return)
$\mid \texttt{skip}$	(skip)
$e ::= \mathit{num}$	(numeric literal)
$\mid \mathit{string}$	(string literal)
$\mid e \oplus e$	(binary operator)
$\mid x$	(identifier)
$\mid f(e)$	(call)
$\mid e.a$	(field selection)
$\mid \{\overrightarrow{a_k = e}\}$	(record literal)
$\mid \texttt{read}(n)$	(read)
$b ::= \texttt{true}$	(true)
$\mid \texttt{false}$	(false)
$\mid e == e$	(equality)
$\mid b \wedge b$	(conjunction)
$\mid \neg b$	(negation)
$\tau ::= \texttt{int}$	(number type)
$\mid \texttt{string}$	(string type)
$\mid \{\overrightarrow{a_k : \tau}\}$	(record type)
$\mid n_k$	(named type)

Figure 4.1: Syntax of the language.

as well. Each reference R is uniquely identified by the name m of the module it belongs to, and contains a function definition f. Moreover, each reference has their own Read-Write Sets, here denoted r and w. Calls between modules are *remote* calls, and thus the main target our analysis in the upcoming sections. For clarity, code samples will represent these remote calls prefixed by the name of the respective module/service. We assume that the *only* side-effects present in the language are solely performed by the *read* and *write* operations, which resemble the basic data access operators of a database or some other data source. This is a simplification from our side, considering that accurately identifying different kinds of side-effects in an automatic fashion would be complex and outside the time frame assigned to this work.

Common programming language constructs (e.g. conditions, loops) and primitive types (e.g. numbers and strings) are also supported. There are also more complex data types, such as *record types*, which describe records, collections of fields.

Finally, the language does not support exceptions, leaving that for future work. This was a simplification, as our target - the OutSystems platform – *does* support them. Exceptions are troublesome when we are dealing with parallelism. Interruptions caused by an exception in some thread make it difficult to resume the program's flow; catching them generates intricate logic, error-prone on a parallel setting. Moreover, adding support for exceptions would translate into more intricate logic, introducing a considerable amount of noise into the flow of our solution.

4.2 Solution

In this section we will present and discuss our solution in detail. To aid comprehension, we will illustrate every single step of our solution with operations from the example scenario presented in Chapter 1. To recall, we are talking about the book-centered platform architected under a service-oriented model, with independent services and database tables for operations such as fetching book details, readers and reviews. This section first introduces an overview of how our solution works on a high level, and further expands on how every piece works individually, focusing on the motivation and design decisions behind them. The next subsection focuses on Read-Write Sets Extraction, followed by Data-Flow and Control-Flow analyses. Program Dependence Graphs are subsequently explored, and the section closes with a discussion on the methodology employed to study the dependence graph and give insights on opportunities for operation parallelism.

Before diving into the solution itself, we will now present an example operation to be used throughout the next sections of this chapter, instantiated in the different phases of our solution. This example is an extension of the example operation presented in Section 1.2 under our hypothetical Service-Oriented Architecture. In this operation we call `SearchBook`, where we first attempt to fetch a book's details (ISBN, ID, author...), given a title. If the book cannot be found in the system, the operation terminates. Otherwise, we use the book's ID (which we get from `GetBookDetails`) to increment the book's view

count (i.e., the number of times the book is viewed in the platform) and to fetch the book's readers and reviews. This pictures a simple yet credible example operation; written in our grammar, this operation would resemble the code shown in Listing 4.1.

```
// BooksService
def SearchBook(title)
  details = BooksService.GetBookDetails(title);          // "Details"
  if (details.id != NOT_FOUND) then                      // "Found?"
    BooksService.IncrementBookViewCount(details.id);     // "Views"
    ReadersService.GetBookReaders(details.id);           // "Readers"
    ReviewsService.GetBookReviews(details.id);           // "Reviews"
```

Listing 4.1: Pseudocode of `SearchBook`.

The different functions called in `SearchBook` encapsulate data access operations. It is important to know *which* operations are performed and *which* tables are accessed; *how* they're performed behind the scenes is not relevant here. Listing 4.2 shows the definitions of these functions. Notice how `GetBookDetails`, for instance, performs a `read` operation on a `Books` database table, and how `IncrementBookViewCount` performs both a `read` and a `write` on that same table.

```
// BooksService
def GetBookDetails(title)
  details = read(BooksTable B where B.title == $title);
  return details;

// BooksService
def IncrementBookViewCount(id)
  var oldCount = read(BooksTable B where B.id == $id).ViewCount;
  write(BooksTable B where B.id == $id, oldCount + 1);

// ReadersService
def GetBookReaders(id)
  readers = read(ReadersTable R where R.IdsOfBooksRead Contains $id);
  return readers;

// ReviewsService
def GetBookReviews(id)
  reviews = read(ReviewsTable R where R.BookId == $id);
  return reviews;
```

Listing 4.2: Pseudocode of the operations called inside `SearchBook`.

Figure 4.2 illustrates the Control-Flow Graph for `SearchBook`. For brevity, the nodes are named in accordance to the comments that escort each line-of-code in Listing 4.1.

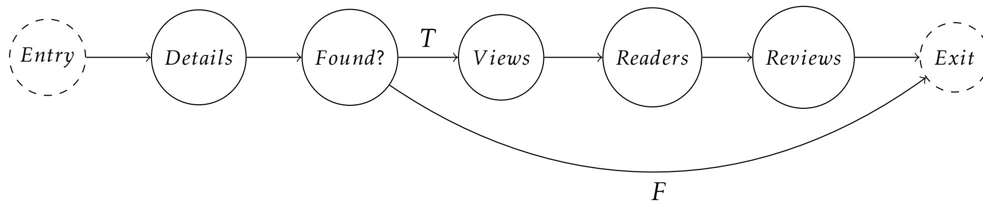

Figure 4.2: Control-Flow Graph for SearchBook.

This graph will serve as the starting point for many of the steps explored in the upcoming subsections. As we will see, this is a good example to illustrate all the steps that will take place throughout our solution.

4.2.1 Overview

Looking back at our solution overview in Chapter 1, there were a few initial directives established to guide our solution. Those were guarantees of correctness, introduction of little-to-no overhead in runtime, and that it should be invisible to the developer.

Furthermore, a new guideline organically emerged from the aforementioned ones, and that is *compositionality*. Having a solution that is composable would lead to a naturally loosely-coupled architecture, making it easier to develop, iterate and especially *test* functionality in isolation. Another important point is that proving each of the components' correctness individually would grant us some confidence in assuring that their *composition* would also be correct.

The solution we developed is a combination of purely-static techniques that are followed through in a pipeline-like fashion, none of them requiring any intervention of the developer. The first three analyses (Read-Write Set Extraction, Data-Flow Analysis and Control-Flow Analysis) gather dependence information. Read-Write Set Extraction must always precede Data-Dependence Analysis, but apart from that, the order of these three is interchangeable.

The dependence information is then summarized into a single graph (Dependence Graph), which is further used to analyse the original program in order to find opportunities for parallelism (Parallelism Analysis).

4.2.2 Read-Write Set Extraction

As concluded in Section 2.3, one of the conditions needed for safe parallelism is the absence of unwanted side-effects. Furthermore, the grammar we defined in Section 4.1 dictates that the *only* side-effects we are considering are database reads and writes. The first step of our solution is thus to extract information regarding database accesses performed by the functions in our system; this information is what we call Read-Write Sets. Granularity-wise, we'll stop the analysis at the point where we know *which* tables are accessed. A finer-grained exploration (e.g. table-row level) could allow for parallelism in

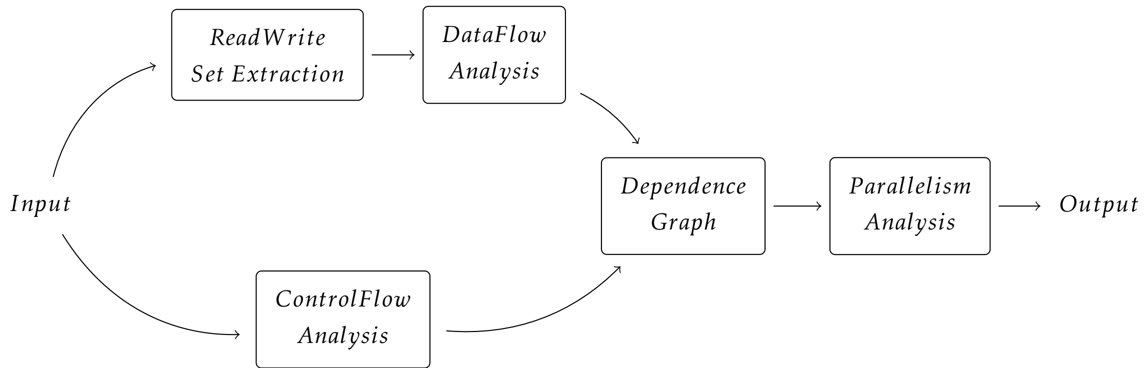

Figure 4.3: Overview of our solution.

a broader range of scenarios. Consider as an example a loop where each iteration writes to a different row of some database table `T`; whilst our algorithm would detect dependences between every iteration, a row-level analysis would recognize the iterations as independent, allowing them to be executed in parallel. We leave that for future work.

Information about Read-Write Sets is *not* enough to assure opportunities for safe parallelism by itself, but it *is* enough to invalidate them: if we have two functions f_A and f_B performing database accesses to the same same table, and at least one of those accesses is a *write* (as opposed to a *read*), then their execution order is not interchangeable – we do not need further analysis to prove that f_A and f_B are not safe to parallelize.

Data Access Pairs and Triples

A Read-Write Set is nothing more than a set of database access records. These records can be seen as pairs, *Data Access Pairs* as we will call them from now on. Each Data Access Pair identifies a database access that targets some table (`DatabaseTableAccessed`) and executes either a read or a write operation (`AccessType`).

$$\langle\ \mathit{DatabaseTableAccessed},\ \mathit{AccessType}\ \rangle \tag{4.1}$$

Now, as with most of the upcoming steps in our solution, the starting point for Read-Write Set Extraction is the Control-Flow Graph, which we traverse in order to extract Read-Write Sets from *every* node. In practice, we will need to keep track of this information for the different nodes. To do so, we assign a unique identifier to every node; the Data Access Pairs are thus augmented into triples:

$$\langle\ \boldsymbol{NodeId},\ \mathit{DatabaseTableAccessed},\ \mathit{AccessType}\ \rangle \tag{4.2}$$

To exemplify, a read access on database table `BooksTable` performed in some node N would be represented by the following triple:

$$\langle\ \mathit{RealIdOfNodeN},\ \mathit{BooksTable},\ \mathit{READ}\ \rangle \tag{4.3}$$

Algorithm

The algorithm to extract Read-Write Sets is shown in Listing 4.3. It takes a CFG as input and creates Data Access Triples as it goes through every node extracting Read-Write Sets.

```
def ExtractReadWriteSets(cfg)
  dataAccesses = [];
  cfg.Nodes.ForEach { node:
    nodeReadsAndWrites = ReadsWrites(node);
    nodeDataAccesses = nodeReadsAndWrites.Map { rw:
      ⟨ node.Id, rw.DatabaseTableAccessed, rw.AccessType ⟩
    });
    dataAccesses.AddAll(nodeDataAccesses);
  }
  return dataAccesses;
```

Listing 4.3: Pseudocode of `ExtractReadWriteSets`.

Inspecting the code, it is noticeable that the bulk of the work in the extraction of Read-Write Sets is made by the function called `ReadsWrites`. We defined this function for every element of our grammar, as can be seen in equations 4.4 - 4.16 below:

$$ReadsWrites(num \mid string) \triangleq \{\} \tag{4.4}$$

$$ReadsWrites(x) \triangleq \{\} \tag{4.5}$$

$$ReadsWrites(\texttt{skip}) \triangleq \{\} \tag{4.6}$$

$$ReadsWrites(\texttt{read}(n)) \triangleq \{\langle\, n, \textsf{READ} \,\rangle\} \tag{4.7}$$

$$ReadsWrites(\texttt{write}(n,e)) \triangleq \{\langle\, n, \textsf{WRITE} \,\rangle\} \cup ReadsWrites(e) \tag{4.8}$$

$$ReadsWrites(e.a) \triangleq ReadsWrites(e) \tag{4.9}$$

$$ReadsWrites(f(e)) \triangleq ReadsWrites(f) \cup ReadsWrites(e) \tag{4.10}$$

$$ReadsWrites(e_1 \oplus e_2) \triangleq ReadsWrites(e_1) \cup ReadsWrites(e_2) \tag{4.11}$$

$$ReadsWrites(s_1 \,;\, s_2) \triangleq ReadsWrites(s_1) \cup ReadsWrites(s_2) \tag{4.12}$$

$$ReadsWrites(\texttt{return}\ e) \triangleq ReadsWrites(e) \tag{4.13}$$

$$ReadsWrites(\texttt{while}\ (b)\ \texttt{do}\ e) \triangleq ReadsWrites(b) \cup ReadsWrites(e) \tag{4.14}$$

$$ReadsWrites(\texttt{if}\ (b)\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \triangleq \bigcup_{k=b,e_1,e_2} ReadsWrites(k) \tag{4.15}$$

$$ReadsWrites(a_k \overset{n}{\overrightarrow{=}} e_k) \triangleq \bigcup_{k=1}^{n} ReadsWrites(e_k) \tag{4.16}$$

The `ReadsWrites` function is based on Data Access Pairs and set operations involving those. Primitive data types, variables and the `skip` operation (4.4 through 4.6) do not perform database accesses, and thus the `ReadsWrites` function returns the empty set.

For the primitive data reading operator (4.7), `ReadsWrites` directly maps `read(n)` to a set with a single element, the Data Access Pair ⟨ n, READ ⟩. The data writing operator (4.8) is not just a direct mapping, considering that a `write(n,e)` operation must have e evaluated for Read-Write Sets as well.

For all other elements of the grammar (4.9 through 4.16), `ReadsWrites` is defined recursively. Furthermore, it is defined in a *conservative* manner, as guided by the static analysis approach we're taking. This is made clear on cases such as 4.15, where the *union* operator elucidates our intentions to accommodate for all possible scenarios.

Applying To Our Example

With the algorithm defined, we can now illustrate how it would be applied to our example scenario. The `ExtractReadWriteSets` function is called, taking the Control-Flow Graph as input:

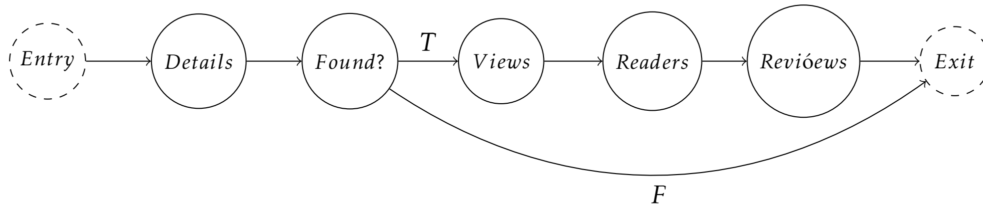

Figure 4.4: Control-Flow Graph for `SearchBook`.

As each of the five nodes is traversed[1], the `ReadsWrites` function is called. For the *Details* node, it would unroll like this (the original code is shortened for clarity):

$$
\begin{aligned}
& ReadsWrites(details = \texttt{read}(BooksTable)\ ;\ \texttt{return}\ details) && (4.12)\\
\equiv\ & ReadsWrites(details = \texttt{read}(BooksTable)) \cup ReadsWrites(\texttt{return}\ details) && (4.12)\\
\equiv\ & \{\langle\ BooksTable,\ \text{READ}\ \rangle\} \cup ReadsWrites(\texttt{return}\ details) && (4.7)\\
\equiv\ & \{\langle\ BooksTable,\ \text{READ}\ \rangle\} \cup ReadsWrites(details) && (4.13)\\
\equiv\ & \{\langle\ BooksTable,\ \text{READ}\ \rangle\} \cup \{\ \} && (4.5)\\
\equiv\ & \{\langle\ BooksTable,\ \text{READ}\ \rangle\} && (\text{-})
\end{aligned}
$$

Assuming each node has an *id* field, we then use this information to instantiate a single Data Access Triple:

$$\langle\ Details.id,\ BooksTable,\ \text{READ}\ \rangle$$

This procedure would be very similar for the *Readers* and *Reviews* nodes, as their functionality is identical to that of *Details*, the only difference being the tables accessed. This would yield the following triples:

$$\langle\ Readers.id,\ ReadersTable,\ \text{READ}\ \rangle$$
$$\langle\ Reviews.id,\ ReviewsTable,\ \text{READ}\ \rangle$$

[1] The *Entry* and *Exit* nodes are ignored; refer to Chapter 2.

The Found? node does not perform database accesses, and would thus not yield any triples. The branches are omitted, as they are analysed next.

$$\begin{aligned} & ReadsWrites(\texttt{if}\ (details.id\ != NOT_FOUND)\ \texttt{then}\ _\ \texttt{else}\ _) && (4.15)\\ \equiv\ & ReadsWrites(details.id\ != null) && (4.15)\\ \equiv\ & ReadsWrites(details.id) \cup ReadsWrites(null) && (4.11)\\ \equiv\ & ReadsWrites(details) \cup ReadsWrites(null) && (4.9)\\ \equiv\ & \{\,\} \cup ReadsWrites(null) && (4.5)\\ \equiv\ & \{\,\} && (\text{-}) \end{aligned}$$

Finally, the *Views* node is an interesting one, as it contains both read and write accesses:

$$\begin{aligned} & ReadsWrites(readers = \texttt{read}(BooksTable)\ ;\ \texttt{write}(BooksTable, oldCount + 1)) && (4.12)\\ \equiv\ & ReadsWrites(\texttt{read}(BooksTable)) \cup ReadsWrites(\texttt{write}(BooksTable, oldCount + 1)) && (4.12)\\ \equiv\ & \{\langle\ BooksTable,\ \mathsf{READ}\ \rangle\} \cup ReadsWrites(\texttt{write}(BooksTable, oldCount + 1)) && (4.7)\\ \equiv\ & \{\langle\ BooksTable,\ \mathsf{READ}\ \rangle\} \cup \{\langle\ BooksTable,\ \mathsf{WRITE}\ \rangle\} \cup ReadsWrites(oldCount + 1) && (4.8)\\ \equiv\ & \{\langle\ BooksTable,\ \mathsf{READ}\ \rangle\} \cup \{\langle\ BooksTable,\ \mathsf{WRITE}\ \rangle\} \cup \{\,\} && (4.5)\\ \equiv\ & \{\langle\ BooksTable,\ \mathsf{READ}\ \rangle, \langle\ BooksTable,\ \mathsf{WRITE}\ \rangle\} && (\text{-}) \end{aligned}$$

This would generate two Data Access Triples:

$$\langle\ Views.id,\ BooksTable,\ \mathsf{READ}\ \rangle$$
$$\langle\ Views.id,\ BooksTable,\ \mathsf{WRITE}\ \rangle$$

The triples generated are enough to give us some useful insights. Notice how, in our example, there is a clash in Read-Write Sets found in two of the generated triples, as they access the same data source and at least one of the accesses is a write:

$$\langle\ Details.id,\ BooksTable,\ \mathsf{READ}\ \rangle$$
$$\langle\ Views.id,\ BooksTable,\ \mathsf{WRITE}\ \rangle$$

This means that, without needing any more information, we know for sure that Details and Views cannot be safely executed in parallel. As we will see, this is a recurring pattern in the upcoming steps of our solution: individual phases can flag impossibilities for parallelism, but not *uncover* opportunities for safe parallelism. The *union* of the information gathered by all the phases, though, is.

4.2.3 Data-Flow Analysis

Consider Figure 4.5, the Control-Flow Graph for a simplified version of our example.

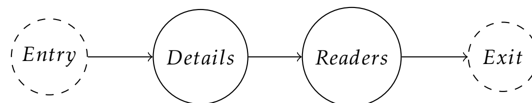

Figure 4.5: Control-Flow Graph of a simple program.

Recall that in the previous step of Read-Write Sets Extraction, we concluded that `Details` and `Readers` both perform database accesses to different tables (`BooksTable` and `ReadersTable`, respectively). This means that, based solely on this information, `Details` and `Readers` are safe to be executed in parallel. However, looking back at Listing 4.1, we notice something important: `GetBookReaders` takes as input `details.id`, where the `details` object is the output of `GetBookDetails`. This implies a data dependency *Details* $\longrightarrow$ *Readers* which requires a specific execution order and means the two nodes are effectively *not* safely parallelizable. Finding these dependences is where data-flow analysis comes into play as the next step in our pipeline.

Our strategy of data-flow analysis is twofold. A first step of Reaching Definitions Analysis gives us insights on which variable definitions are active at each node. This is the starting point we need to start checking for clashes (dependences) caused by variable uses and definitions across nodes, a static analysis step we call Data Dependence Analysis.

Incorporating Read-Write Sets

Despite being gathered separately, in different phases, Read-Write Sets are treated in a similar fashion to regular data dependences. Ultimately, Read-Write Sets represent data accesses just as much as input/output scenarios that happen with regular variables, and we should treat them that way. Thus, the output of the Read-Write Set Analysis phase is injected into the entirety of the Data-Flow Analysis step.

Reaching Definitions

Reaching Definitions is a classic data-flow analysis algorithm that determines which variable definitions (writes) may reach a given point in the code. The set of data-flow equations (Chapter 2) for Reaching Definitions are as presented below, for some generic node b:

$$\begin{aligned} OUT_b &\triangleq DEF_b \cup (IN_b - KILLS_b) \\ IN_b &\triangleq \bigcup_{p \in preds(b)} (OUT_p) \end{aligned} \tag{4.20}$$

DEF_b holds the set of variable definitions made in node b. For implementation purposes, and throughout the remaining of this document, we will represent variable uses (reads) and definitions (writes) as pairs $\langle$ `Variable Used/Defined`, `BasicBlock` $\rangle$. The IN, OUT, DEF and $KILLS$ structures are all sets of $VariableDefinition$. $KILLS_b$ holds the set of all definitions that were killed (overwritten) in node b; as an example, if node b (only) defines some variable v (a definition we would represent as $\langle\, v,\ b\,\rangle$), and $ALLDEFS(v)$ holds all definitions of v, then $KILLS_b \triangleq ALLDEFS(v) - \langle\, v,\ b\,\rangle$. As we will see, the computation of variable definitions (DEF) and definitions overwritten ($KILLS$) makes reaching definitions an important first step for the calculation of data dependences.

Listing 4.4 shows the algorithm used to compute reaching definitions.

```
def ComputeReachingDefinitions(cfg)
  cfg.Nodes.ForEach { N: OUT_N = {}};
  ChangedSet = CFG.Nodes;
  while (ChangedSet != {}) do
    N = ChangedSet.RemoveAny();
    // Populate IN
    IN_N = ⋃_{p ∈ Predecessors(N)} OUT_p;
    OldOut = OUT_N;
    // Populate OUT
    OUT_N = DEF_N ∪ ( IN_N − KILLS_N );
    if (OUT_N != OldOut) then
      ChangedSet = ChangedSet ∪ N.Successors();
```

Listing 4.4: Algorithm to compute reaching definitions.

Data Dependence Analysis

Data dependences are very similar to clashes in Read-Write Sets: if two operations OpA and OpB both access some variable v and at least one of the accesses is an assignment (i.e., a definition, as opposed to an use), then there is a data dependence between OpA and OpB. This means that, in the same way we need DEF to hold information regarding variable definitions, it is also crucial to know where are variables *used*, so that we can find these *clashes*. Here we assume that this knowledge of variable uses and definitions is available for every node.

Recalling from Chapter 2, there are three types of data dependences: Read-After-Write (or flow dependence, δ^{f}), Write-After-Read (or anti dependence, δ^{a}) and Write-After-Write (or output dependence, δ^{o}). Though the actual *types* of dependences are not relevant for our solution, it is crucial for our analysis to target each one of them individually in order to uncover *all* existent data dependences. Listings 4.5, 4.6 and 4.7 demonstrate the procedures we created to detect each type of dependence for node b. In practice, these procedures will be applied to every node of the program we are analysing.

```
def ComputeReadAfterWriteDependences(n)
  for each node b in the graph
    for each VariableUse in node b, ⟨ v,b ⟩ do
      for each VariableDefinition in IN_b where v is defined, ⟨ v,n ⟩ do
        dependence detected: n --δf<v>--> b
```

Listing 4.5: Algorithm to find Read-After-Write data dependences for some node n.

The algorithm of Listing 4.5 uncovers Read-After-Write dependences for scenarios such as the one depicted in Figure 4.6.

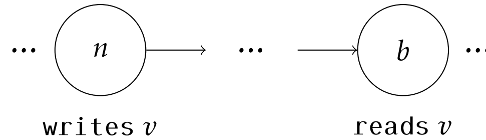

Figure 4.6: Example graph scenario with a Read-After-Write data dependence.

Here we represent a data dependence between two nodes n and b as $n \rightarrow b$. The type of dependence and the variable involved are written on top of the arrow. The Read-After-Write dependence $n \xrightarrow{\delta^{f}<v>} b$ is read as *node b reads a variable v that was previously written in node n*. The *direction* of the dependence establishes the relative execution order between the two involved nodes. As we will see, this direction is relevant for the upcoming steps.

```
def ComputeWriteAfterReadDependences(n)
  for each node b in the graph
    for each VariableUse in b, ⟨ v,b ⟩ do
      for each VariableDefinition in KILLS_n where v is killed, ⟨ v,n ⟩ do
        if (n.PostOrderIndex > b.PostOrderIndex) then
          dependence detected: b --δa<v>--> n
```

Listing 4.6: Algorithm to find Write-After-Read data dependences for some node n.

In Listing 4.6, the `PostOrderIndex` comparison is used to assure that the dependence found is valid in terms of the program's flow of execution. Post-order indexing is achieved with a depth-first search where the successors of a node are visited (and indexed) *before* the node itself is (post-order will be further discussed in the next subsection).

A Write-After-Read dependence is valid if and only if the write operation happens effectively *after* the read one. Figure 4.7 below exemplifies the type of Write-After-Read scenarios that would be discovered by the procedure of Listing 4.6.

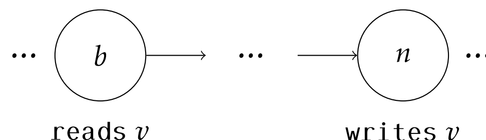

Figure 4.7: Example graph scenario with a Write-After-Read data dependence.

As can be seen in Listing 4.7, Write-After-Write data dependences are detected by looking at pairs of kills and definitions of the same variable across different nodes. Figure 4.8 below depicts the type of scenarios that would be signaled as Write-After-Write dependences by our algorithm.

```
def ComputeWriteAfterWriteDependences(n)
  for each node b in the graph
    for each VariableDefinition in KILLS_b where v is killed, ⟨ v,n ⟩ do
      if (n.postOrderIndex < b.PostOrderIndex) then
        dependence detected: n --δ°<v>--> b
```

Listing 4.7: Algorithm to find Write-After-Write data dependences for some node n.

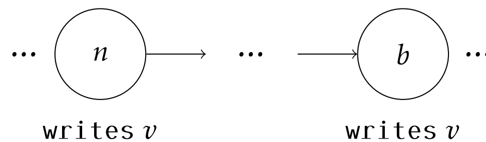

Figure 4.8: Example graph scenario with a Write-After-Write data dependence.

Notice how the procedures to compute Write-After-Read and Write-After-Write (Listings 4.6 and 4.7) data dependences make use of the *PostOrderIndex*; this is necessary in order to make sure that the dependences found are valid in regards to the natural flow of the analysed program. For Read-After-Write dependences this is not necessary; for some basic block b, every variable definition on IN_b happened on some node that necessarily precedes b in execution.

Applying To Our Example

The first step in applying data-flow analysis to our example is to summarize the variable uses and definitions for each node. Table 4.1 displays such information. As an example, both `name` and `BooksTable` are used (read) inside the `Details` node, whereas `details` is defined (written). We're identifying every definition with a shorthand, for brevity (in this case `#d` and `#B` for the `details` and `BooksTable` definitions, respectively).

	Details	Found?	Views	Readers	Reviews
Uses	name, BooksTable	details	name, BooksTable	name, ReadersTable	name, ReviewsTable
Defs	details (#d)	-	BooksTable (#B)	-	-

Table 4.1: Variable uses and definitions for every node of our example.

The next step is to apply the algorithm of Reaching Definitions (Listing 4.4) to our example, which would yield the results presented in Table 4.2.

This information is enough for us to uncover the data dependences existent in our example. Applying the procedures examined in Listings 4.5, 4.6 and 4.7, three flow dependences and one anti-dependence are uncovered, as shown in Figure 4.9.

Node (N)	IN_N	DEF_N	$KILL_N$	OUT_N
Details	∅	{#d}	∅	{#d}
Found?	{#d}	∅	∅	{#d}
Views	{#d}	{#B}	∅	{#d,#B}
Readers	{#d,#B}	∅	∅	{#d,#B}
Reviews	{#d,#B}	∅	∅	{#d,#B}

Table 4.2: Reaching definitions for our example.

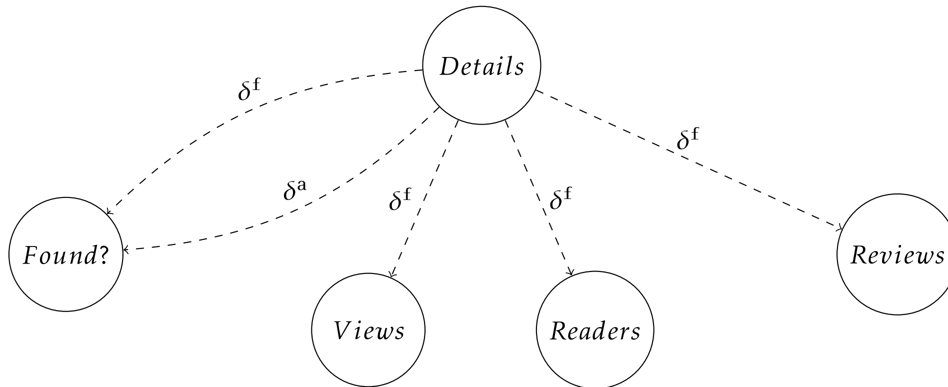

Figure 4.9: Data dependences found for our example program.

4.2.4 Control-Flow Analysis

Analysis of control flow comes into play to detect situations of *conditional execution*. The concept of conditional execution is closely related to that of control dependence, as discussed in Chapter 2. Consider Figure 4.10 below, a stripped-down version of our original example.

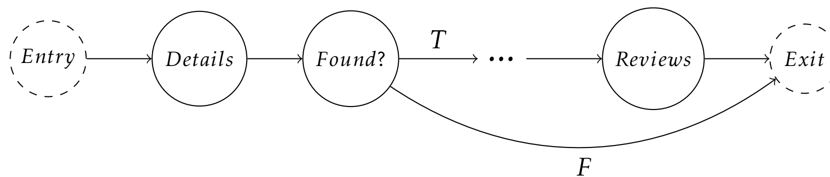

Figure 4.10: Control-Flow Graph of a simple program.

To detect control dependences between nodes (and therefore, scenarios of conditional execution), we followed the techniques studied during the preparation phase, presented in Chapter 2: creating a (post-) dominator tree and building a (post-) dominance frontier.

Algorithm

The topic of finding dominators in control-flow graphs has been thoroughly researched [11, 14, 20, 29, 36]. Two of the most well-known algorithms are Tarjan and Lengauer's [29]

and the one by Cooper et al. [14]. Though the former theoretically outperforms all other algorithms when it comes to time complexity, its rather intricate implementation led us to opt for the latter, which was the clear winner in the trade-off between performance and implementation effort. Moreover, efficiency was not a critical criterion for us, as we were more interested in studying the *integration* and *pipelining* of a set of techniques, and not specific implementations.

The complete algorithm to detect control dependences is made up of three computation moments: post-order, post-dominator tree and post-dominance frontier. The original algorithms by Cooper et al. operate over the nodes of the CFG indexed by post-order value in order to compute the dominator tree and the dominance frontier. Post-order is a depth-first-search traversal that consists of visiting all the successors of a node and *only then* visit the node itself. The algorithm to compute the post-order indexes of nodes in a graph is shown in Listing 4.8.

```
visitedNodes = {};
postOrderIndexes = [];
currentPostOrderIndex = 0;
...
def ComputePostOrder(rootNode)
  visitedNodes.Add(rootNode);
  rootNode.Successors().ForEach { succ:
    if (!visitedNodes.Contains(succ)) then
      ComputePostOrder(succ);
  }
  postOrderIndexes[rootNode] = currentPostOrderIndex++;
```

Listing 4.8: Algorithm to calculate the post-order indexes of nodes in a graph.

The function takes a root node as input, adds it to the set of nodes visited and recursively calls `ComputePostOrder` for every successor of that root node. The code snippet assumes the existence of three data structures: the aforementioned set of visited nodes, a counter to keep track of the current post-order index and an array to hold the indexes calculated.

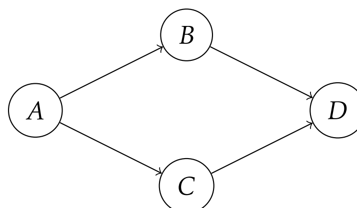

Figure 4.11: A simple graph.

For a given graph, there are multiple possible correct post-orders. Take a look at Figure 4.11: for this graph, calling `ComputePostOrder` with `A` as the root node could either yield `D,C,B,A` or `D,B,C,A`, depending on the inner workings of the data structures/iterators

used and on the implementation itself. Though both post-orders are valid, this can be a problem, something we will discuss at the end of the section. Now, because it is *post*-dominance we are interested in, for the uncovering of control dependences, we will follow the strategy discussed in Chapter 2: computing (regular) dominance relationships on the *reverse* control-flow graph. This means that, before computing post-order and applying the original algorithms proposed by Cooper et al. [14] (Listings 4.9, 4.10, 4.11), we will reverse our original control-flow graph.

```
def ComputeImmediateDominators(cfg)
  nodes = cfg.Nodes;
  nodes.ForEach { n: doms[n] = Undefined };
  doms[startNode] = startNode;
  changed = true;
  while (changed) do
    changed = false;
    for all nodes, b, in reverse postorder (except startNode)
      newIdom = b.Successors().Where { c: doms[c] != Undefined }.First();
      for all other predecessors, p, of b
        if doms[p] != Undefined then
          newIdom = Intersect(p, newIdom);
      if doms[b] != newIdom then
        doms[b] = newIdom;
        Changed = true;
```

Listing 4.9: Cooper et al.'s algorithm to compute the dominator tree [14].

```
def Intersect(node1, node2)
  finger1 = node1;
  finger2 = node2;
  while (finger1 != finger2) do
     if (finger2 < finger1) finger2 = doms[finger2];
     if (finger1 < finger2) finger1 = doms[finger1];
  return finger1
```

Listing 4.10: Auxiliary Intersect function (Cooper et al. [14]).

```
def ComputeDominanceFrontier()
  CFG.Nodes.ForEach { n:
    if (n.Predecessors() >= 2) then
      n.Predecessors().ForEach { p:
        runner = p;
        while (runner != doms[b]) do
          domFrontier[runner].Add(b);
          runner = doms[runner];
      }
  }
```

Listing 4.11: Cooper et al.'s algorithm to compute the dominance frontier [14].

Applying To Our Example

Right after an initial step where we create the reverse control-flow graph of our original graph, the post-order algorithm is applied on this reverse graph, indexing the nodes as shown in Figure 4.12:

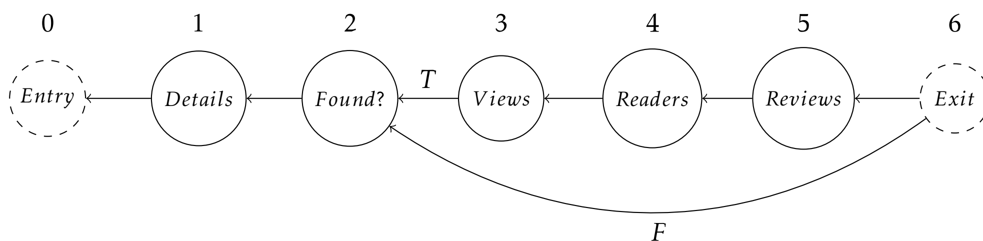

Figure 4.12: Reverse Control-Flow Graph of `SearchBook`, with each node indexed by post-order.

Next, we compute the post-dominator tree for our example; we do so by computing dominators on the reverse control-flow graph. Considering we are looking for *post*-dominators, we apply the dominator tree algorithm (Listing 4.9) to our reverse control-flow graph and reverse the roles of the *Entry* and *Exit* nodes in the computation of those post-dominators. Recalling the theory from Chapter 2, a node X is said to dominate a node Y if X appears in every path from *Entry* to Y. To compute our post-dominator tree, we just swap the *Entry* node for the *Exit* node in this formulae. The result is the post-dominator tree presented in Figure 4.13.

The final step is to compute the post-dominance frontier, given the immediate post-dominators and the (reverse) control-flow graph of our program. Applying the algorithm of Listing 4.11 we get the post-dominance frontier shown in Table 4.3.

By definition, and as explained in Chapter 2, a node X is control-dependent on a node Y if and only if Y is present on the post-dominance frontier of X. Following this notion and looking at our results in Table 4.3, we can see that `Views`, `Readers` and `Reviews` are control-dependent on `Found?`, as can be seen in Figure 4.14.

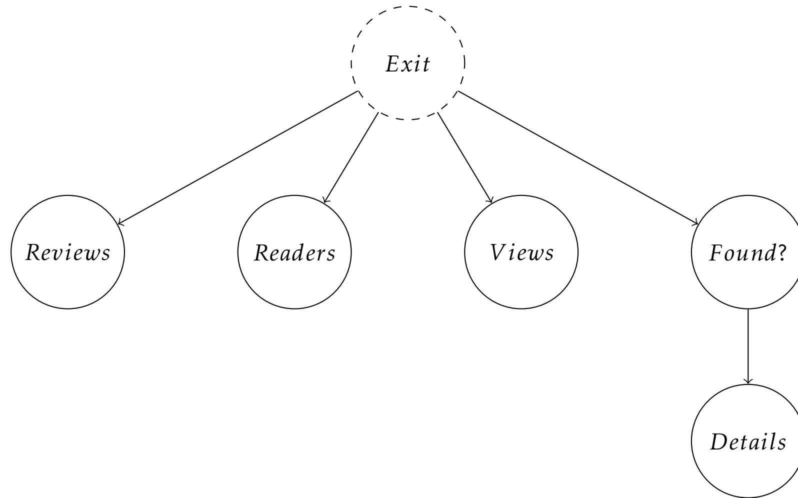

Figure 4.13: Post-dominator tree for our example.

	Entry	Details	Found?	Views	Readers	Reviews	Exit
Post-Dom. Frontier	∅	∅	∅	{Found?}	{Found?}	{Found?}	∅

Table 4.3: Post-dominance frontier for every node of our example.

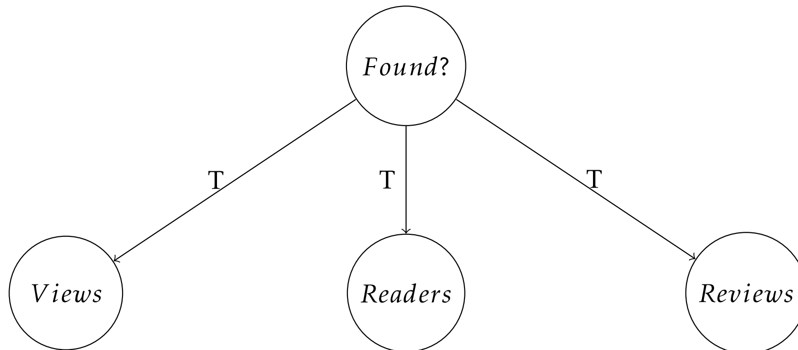

Figure 4.14: Control dependences found for our example program.

One thing to take into account is that *cycles* in directed graphs impose a problem for the computation of post-order indexes. Consider an example graph where we have edges $X \longrightarrow Y$ and $Y \longrightarrow X$. When faced with such a scenario, the predecessor/successor relationship between the two nodes is *undefined*. Whilst the post-order traversal algorithm *does* terminate when faced with this type of graphs, the resulting indexing is not as clearly defined. Keeping in mind that post-order indexing is the starting point for the control-dependence algorithms, we can see that this particularity can translate into unreliable results when we're faced with cyclic graphs.

From what we researched, there is not a lot of literature on solving this type of scenarios for graphs. The only possible solution we found that could prove viable to handle these cases could be to compute our graph's Strongly Connected Components [8] (i.e. grouping every loop into a single component) and apply post-order traversals on those

components. The analysis and implementation of this technique is left for future work.

4.2.5 Program Dependence Graph

With both data and control dependences calculated, the next step is to summarize them all into a single data structure, one that clearly displays *all* the dependences of *every* node and is easy to parse and analyse. This data structure is the Program Dependence Graph[2], and the one for our example can be seen in Figure 4.15 below. The control and data dependences found in the previous steps are represented with solid lines and dashed lines, respectively.

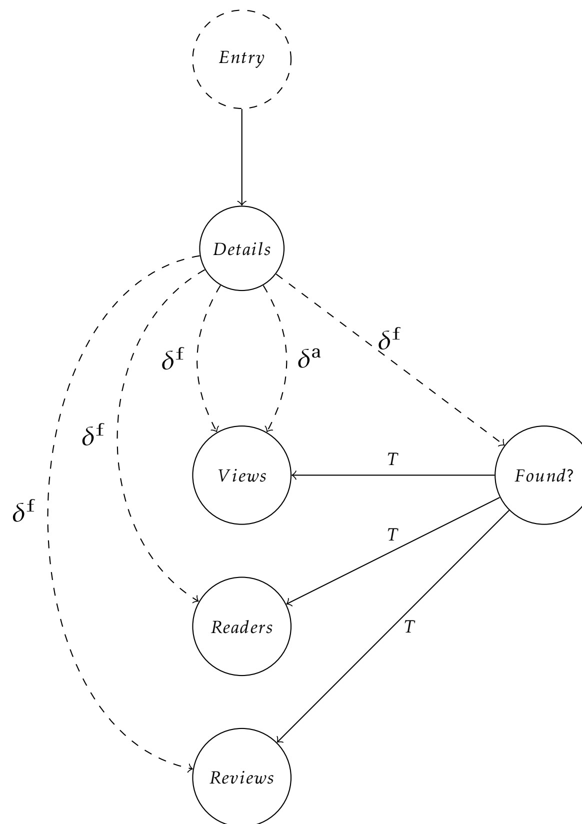

Figure 4.15: Program Dependence Graph for our example program.

Notice how, in the graph, there is an additional dependence between the `Entry` and `Details` nodes. For every node N in the program dependence graph that has *no* incoming edges, we create a dependence $Entry \longrightarrow N$; this way we have a well-defined traversing order and analyse the graph starting from the *Entry* node.

Moreover, because the *types* of data dependences are irrelevant to our analysis, we restrict the number of data dependence edges from one node to another to one. For representation purposes, our graph depicts both a flow and an anti dependence from nodes `Entry` to `Details`. In practice, however, we only create an edge for the first of the two dependences that is uncovered; having multiple edges conveys no extra information.

[2] Program Dependence Graphs were discussed in Chapter 2.

One important attribute of our Program Dependence Graph is that there will never be circular dependences between nodes, e.g., for two nodes M and N, both dependences $M \longrightarrow N$ and $N \longrightarrow M$. The presence of circular dependences implies that we are ignoring the program's natural flow during the dependence calculation steps; not taking into account this flow makes it rather difficult to parallelize blocks. This absence of circular dependences is a result of the way data dependences are calculated, as the post-order indexing on procedures 4.5, 4.6 and 4.7 is used to define a relative ordering between the nodes, and thus, between their dependences. This will prove useful for the next step.

4.2.6 Analysis of Parallelism

The final step in our pipeline is to analyse the program dependence graph, looking at the strictly-necessary dependences between the nodes and establishing a relative execution order between them. The way we institute this order is by assigning every node to what we call a *section*. A section holds a set of nodes, and is tagged with a number; no two sections will ever have the same number, and we represent a section tagged i as S_i. Consider two sections, S_i and S_j, where $i < j$. The tag numbers inform us that *every* node in S_j is dependent on one or more nodes assigned to S_i. In other words, the nodes in S_j should only get to execute after *all* the nodes assigned to S_i have finished their execution. If a section has more than one nodes assigned to it, they can all, in theory, be safely executed in parallel (although special attention is needed for a certain scenario, something we will discuss shortly). This behavior is illustrated in Figure 4.16 below.

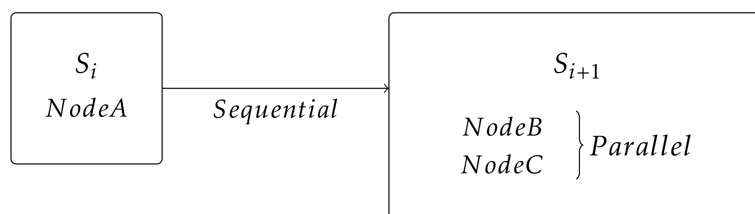

Figure 4.16: Flow of node assignments to sections, and how it can be interpreted.

We determine which section a node gets assigned to by looking at the section tag numbers of all the *predecessors* of that node in the graph. Consider an example where our dependence graph contains two nodes, M and N. If there is a dependence edge $M \longrightarrow N$, and M is assigned to section S_m, then N will necessarily be assigned to some section S_n where $m < n$. If $M \longrightarrow N$ is the *only* incident edge for N, then N gets assigned to S_{m+1}. Otherwise, we check for the *maximum* tag number in all of N's predecessors and increment it, a behaviour exemplified in Figure 4.17 below.
The algorithm that traverses the dependence graph and assigns nodes to sections is shown in Listing 4.12.

```
def AssignSectionsToNodes(pdg)
  sections = empty map<Node, Section>;
```

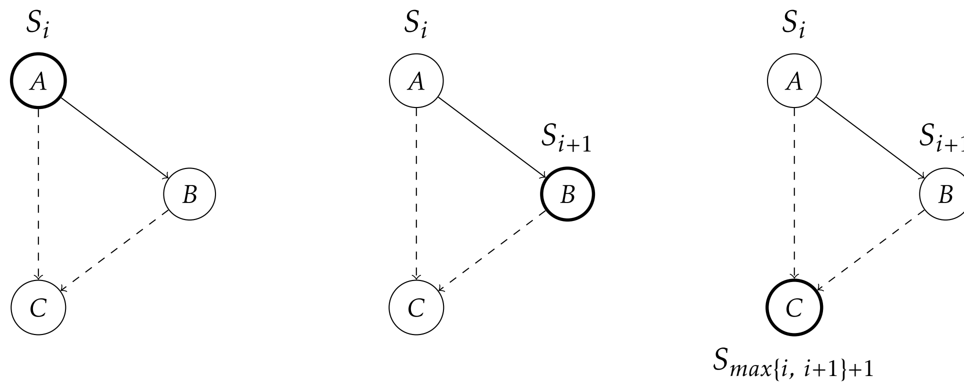

Figure 4.17: A simple Program Dependence Graph showcasing the assignments nodes to sections. The dependence information on the edges is omitted for simplicity.

```
currentNode = pdg.Nodes.Entry;
toExplore = empty queue;
successors = currentNode.Successors();
toExplore.EnqueueAll(successors.SortedByIncomingEdges());
while (toExplore not empty) do
  currentNode = toExplore.Dequeue();
  for each incoming edge E into currentNode {
    currentNode.AddDependenceFromSection(sections[E.nodeFrom]);
  }
  sections[currentNode] = S_{max of DependencesFrom[currentNode] + 1};
  for each node N in currentNode.Successors().SortedByIncomingEdges() {
    if (N not in toExplore) then
      toExplore.Enqueue(N);
  }
```

Listing 4.12: Algorithm that assigns the Program Dependence Graph's nodes to sections.

At its core, the algorithm works by looking at the incoming edges (the dependences) of every node and tagging it accordingly. By default, the `Entry` node is always assigned to S_0, and the `Exit` node is omitted, as it is always the last one to execute. In our example, being the only successor of `Entry`, the `Details` node gets assigned to S_1, and so we proceed to its successors. It is important that we hold the nodes yet to explore in a queue, and that the successors of a node are added to that queue in ascending order of incident edges. Figure 4.18 depicts why this is important.

The simplified versions of our example dependence graph represent different traversal orders to explore the successors of the `Details` node. On the left graph, the `Readers` node is explored *before* the `Found?` node. The problem with this approach is that at the point where `Readers` is being analysed, the section of one of its dependences, `Found?`, is still undefined, making it impossible to determine the section of `Readers` itself. If, on the other hand, we start our traversal from the successor with the *least* amount of dependences, then

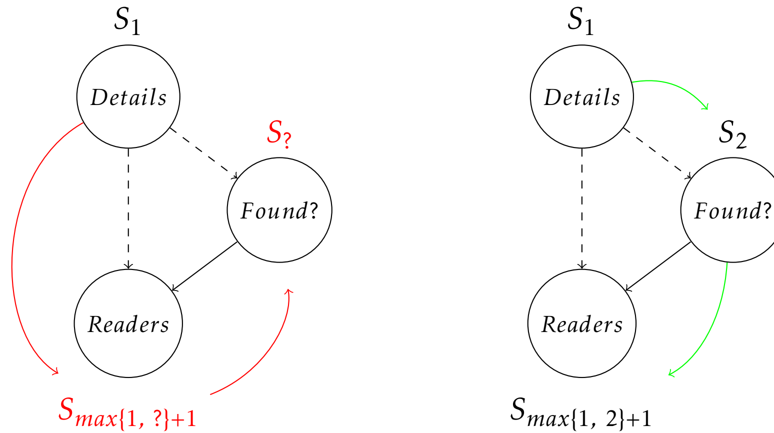

Figure 4.18: Simplified Program Dependence Graphs for our example, showcasing the importance of the traversal order.

these cases of dependences between sibling nodes are handled correctly (graph on the right). Applying the algorithm to our example scenario would yield the results displayed in Table 4.4 below.

Node	Dependences From	Section
Entry	–	S_0
Details	Entry (S_0)	S_1
Found?	Details (S_1)	S_2
Views	Details (S_1), Found? (S_2)	S_3
Readers	Details (S_1), Found? (S_2)	S_3
Reviews	Details (S_1), Found? (S_2)	S_3

Table 4.4: Section assignments for the nodes in our example.

The information gathered regarding section assignments lets us reorganize the code in a way that makes use of parallelism where possible. Following the rationale illustrated in Figure 4.16, different sections are executed sequentially, following the order of the tag numbers, whilst the nodes *inside* each section execute in parallel.

There are, however, a couple of special cases that do not fully comply with this rule, and thus should be handled with care.

First, if we have any two nodes that have been assigned to the same section *but* are mutually exclusive in the flow of the original program, then these two nodes should *not*, of course, be executed in parallel; that would translate into a violation of the semantics of the original program. These are the scenarios where the Program Dependence Graph would come in handy in this process of parallel code generation: when dealing with nodes that have control dependences, inspecting the *condition* on the predecessor and the *label* on the control dependence arc can grant a guard condition that the code generation algorithm would use to escort the parallel execution of those same nodes.

Second, loops in the program present yet another interesting setting, as there can be both parallelism *across* iterations and parallelism on the operations inside the body of the loop. The operations that comprise the body of a loop are uncovered by inspecting the Program Dependence Graph and checking if the branching of flow comes from a loop node (i.e., not a conditional node). We assume parallelism across iterations on a for loop can exist as long as *no* operation inside the body of the loop performs any operation that mutates state somehow, such as database writes or variable definitions. Parallelism for the operations inside the body of each iteration follows the aforementioned rules: if the operations are assigned to the same section, then they can be safely executed in parallel.

Even though, due to time constraints, the step of parallel code generation was not formalized - nor did it make it to the prototype – we believe that having this intuition is quite a relevant stepping stone to achieve this in a future time. For clarity, Chapter 5 will further exemplify these scenarios.

Below, Listing 4.13 recalls what the code for our original example looked like:

```
// BooksService
def SearchBook(title)
  details = BooksService.GetBookDetails(title)      // "Details",  S1
  if (details.id != NOT_FOUND) then                 // "Found?",   S2
    BooksService.IncrementBookViewCount(details.id) // "Views",    S3
    ReadersService.GetBookReaders(details.id)       // "Readers",  S3
    ReviewsService.GetBookReviews(details.id)       // "Reviews",  S3
```

Listing 4.13: Code for our example scenario.

With the intuitions described above, the parallel code generation step would transform the original code into something like what is shown in Listing 4.14. For illustration purposes, assume the existence of a `parallel` construct that wraps some operations and performs them in parallel.

```
// BooksService
def SearchBook(title)
  details = BooksService.GetBookDetails(title);         // "Details",  S1
  if (details.id != NOT_FOUND) then                     // "Found?",   S2
```

```
parallel {
  BooksService.IncrementBookViewCount(details.id);  // "Views",    S3
  ReadersService.GetBookReaders(details.id);        // "Readers",  S3
  ReviewsService.GetBookReviews(details.id);        // "Reviews",  S3
}
```

Listing 4.14: Parallel version of our original example.

What we gain by doing this is a significant cut in runtime. Let's say that the runtime for our original version of SearchBook, represented $t_{SearchBook}$, is approximately given by the sum of the runtimes of its operations:

$$t_{SearchBook} = t_{Details} + t_{Found?} + t_{Views} + t_{Readers} + t_{Reviews}$$

With the optimized version shown in Listing 4.14, where operations Views, Readers and Reviews are all executed in parallel, the runtime changes to the following:

$$t_{SearchBook} = t_{Details} + t_{Found?} + max\{t_{Views}, t_{Readers}, t_{Reviews}\}$$

Considering how the remote calls make up the bulk of runtime, we can consider the execution time of Found? to be negligible. Moreover, if we surmise that each service action call takes approximately the same time to execute from start to finish - which is a rough, nevertheless reasonable assumption, considering there is always some latency involved - then, with the optimized version, we would achieve a relative speedup of around 50% in this scenario. Naturally, different programs will yield different patterns, and therefore different levels of operation parallelism exploitation. This is something we will address as we discuss evaluation on Chapter 5.

4.3 Prototype in OutSystems

We developed a prototype in the context of the OutSystems platform, employing the strategies and processes discussed in the previous chapter, and here we will discuss it very briefly. Figure 4.19 shows the SearchBook operation implemented in OutSystems.

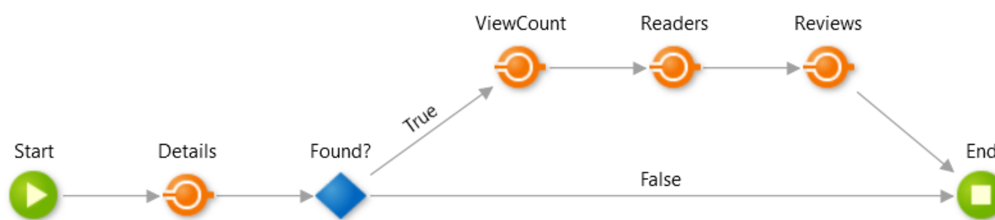

Figure 4.19: Our example scenario modeled in OutSystems.

As Read-Write Sets get extracted from our actions, they are stored as Data Access Triples in that action's *signature*, a collection of metadata relevant to the action. A simplified snippet of the signature for BookProcedure can be seen in Figure 4.20. When omitted, the AccessType is a read access.

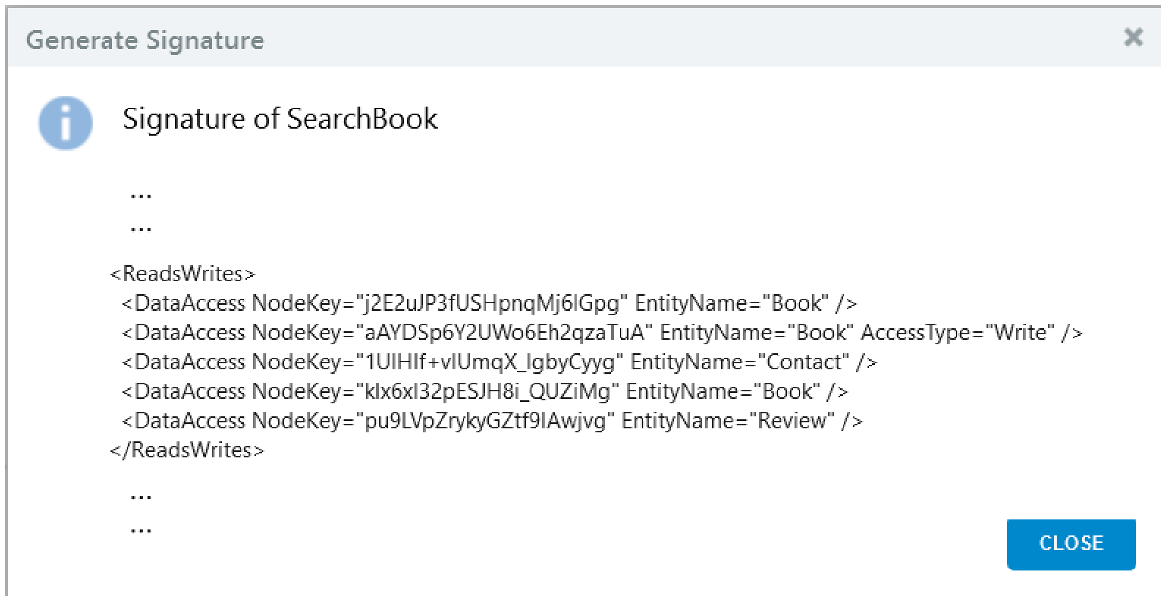

Figure 4.20: Action Signature of SearchBook displaying its Read-Write Sets.

As with software application development in general, the logic of an OutSystems application is in constant, rapid change during development. Thus, we found it important to make our analysis *reactive* in the prototype; when nodes get added, updated or removed from an action, a new Read-Write Sets extraction is triggered and the new values are updated on the action signature. This triggering is optimized to only happen for nodes that can effectively perform database accesses, and therefore impact the Read-Write Sets. Figure 4.21 presents those nodes, the database access primitives of OutSystems, which can be roughly modeled by the read and write primitives from the language presented earlier. To extract which database tables are accessed in an Advanced Query (the rightmost node on the figure), we developed a simple SQL parser capable of distinguishing read from write accesses and identifying the names of the tables accessed. Considering how Server Actions and Service Actions[3] can – and often do – encapsulate those same database access primitives in their logic, they too are triggering nodes themselves. Due to the complex life-cycle and internal behavior of these nodes, keeping the firing of events to a minimum was also something we had to optimize.

The most dynamism on the prototype, however, is achieved with the *propagation* of Read-Write Set changes to all the *consumers* of the action that suffered change, i.e., to all the other actions that call it. Moreover, these consumers can themselves be called by

[3]Server and Service Actions in OutSystems were discussed in Subsection 2.1.3.

Figure 4.21: Database access primitives in OutSystems.

other actions (which can *also* be called by other actions...), and so those too need to have their own Read-Write Sets refreshed. This behavior is depicted in Figure 4.22 below:

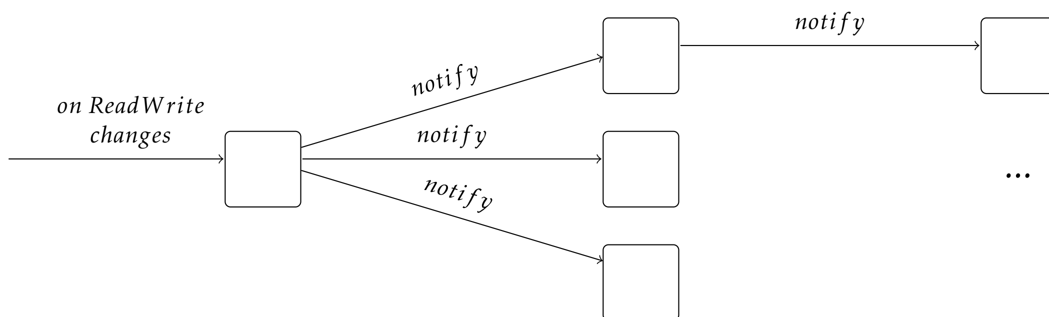

Figure 4.22: Propagation of Read-Write Set changes on our prototype.

Changes are propagated not only to actions from the same module but also across modules, once the module is published and the consumers of that module fetch the lasted updated version of it. Testing this behavior was the main reason why this step of Read-Write Set Analysis was developed on the Service Studio (the OutSystems IDE) codebase. All other analyses were developed on the compiler side.

Both the extractions of Read-Write Sets and the propagation of their changes support recursive (*Action A* calls *Action A*) and mutually-recursive (*Action A* calls *Action B* and *Action B* calls *Action A*) scenarios. Mutually-recursive scenarios proved particularly challenging to handle, as Read-Write Set changes would cause the flow of notifications (Figure 4.22) to create an infinite loop, terminating in a stack overflow. To handle these situations, we augmented the events with information regarding *which* action caused the triggering in the first place, so that it could be skipped by all the observers that also call that same action in their own flow.

The remaining analyses that comprise our solution took place in the OutSystems compiler, which came with the inevitable overhead associated with learning and navigating yet another large codebase.

The first thing that happens in the compiler side is the translation of the original program graph modeled in OutSystems to a simpler version of it, mapping the original nodes and edges to lighter structures that hold *just* the information we need. All the steps

that come next are thus applied on top of this simplified version.

For Data-Flow Analysis, we built on an existing data-flow algorithm that already existed in the compiler – though it was not complete for all nodes, and was only available during a specific compilation moment, details that arose some challenge – enriching it with more information. Normalizing the representation of Read-Write Sets so that they could fit this analysis (i.e., be treated similarly to regular data dependences) was also an important detail.

Throughout the implementation phase, care was taken to make sure that the code for had a clean API and an organized, expandable and loosely-coupled structure; this made it easier not only to develop and test each component in isolation, but also to accommodate for future work changes.

Chapter 5

Evaluation

One of our main goals for development was to evolve the prototype to a point where identifying the next step of future work would be a reasonably straightforward task. This checkpoint, as we call it, was successfully achieved with the sectioning algorithm showcased in 4.2.6. Within the time frame of this dissertation, however, we were not able to bring the OutSystems prototype to a code-generation phase, meaning we cannot see the full result of our solution *in practice*. All the things discussed in Chapter 4 made their way into the OutSystems prototype; this was, of course, the concrete target of our work, and it is in that context in which we will evaluate our solution.

In this chapter, we will expand on the ways we evaluated our solution. The evaluation was mostly based on empirical testing, once again with the help of the book-centered, service-oriented platform example we have been using along the way. Different scenarios will be instantiated in OutSystems, going from recurring patterns in user-facing software applications to more specific edge cases, but always highlighting *some* aspect of our solution. Whether we managed to respect the development guidelines we imposed in Chapter 1 will also be a topic of discussion.

5.1 Common Patterns

Some UI patterns are very common in modern applications, and so is the *logic* that handles the data fetching/management needed to back them up. In this section we will present some of these well-known patterns in a simplified fashion, in the context of our book-centered example. We will discuss how the underlying logic would be carried out currently in OutSystems, and the applicability of our work.

Fetching Data from Multiple Sources

Displaying a handful of information from different sources in some kind of list is a common occurrence in user-facing applications. Let's assume that, as they enter our application, the user is presented with a dashboard containing information about the most popular books, the most popular reviews and the most prolific readers of the month (i.e. the ones who have read the most books in that month). In OutSystems, we could model this scenario with three Service Action calls, as shown in Figure 5.1.

Figure 5.1: OutSystems logic for fetching data from multiple sources.

Behind the scenes, each of these Service Actions will perform a remote call to the respective service, which in turn will make the query to the appropriate database table. From left to right, the Service Action calls will target the `Books`, `Reviews` and `Readers` database tables, respectively, and these accesses are detected by our Read-Write Sets Extraction algorithm. There is no branching of control-flow, nor there are any input-output data dependences, which leads to a simple Program Dependence Graph (node names are shortened for display reasons):

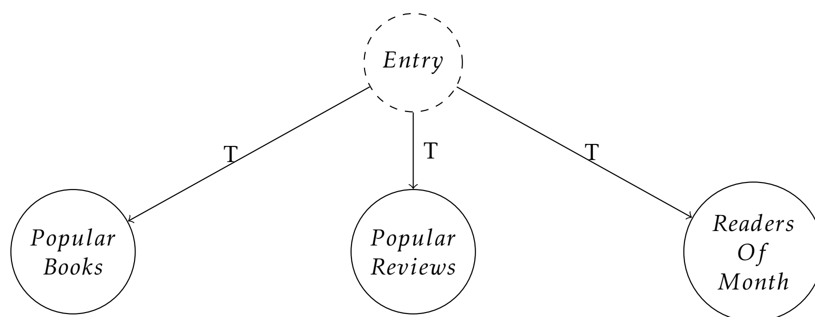

Figure 5.2: Program Dependence Graph for our example of Figure 5.1.

The sectioning algorithm is then run, yielding the results displayed in Table 5.1.

The information in Table 5.1 tells us that, to the extent of assumptions of our algorithm, the three service calls can all be executed in parallel. Due to the intrinsic simplicity of this scenario – where we do not need any information that is only available in runtime – injecting this information in the code generation step would signal the compiler to wrap the three calls in a parallel block.

Node	Dependences From	Section
Entry	–	S_0
GetPopularBooks	Entry (S_0)	S_1
GetPopularReviews	Entry (S_0)	S_1
GetReadersOfTheMonth	Entry (S_0)	S_1

Table 5.1: Section assignments for the nodes in our example of Figure 5.1.

Let's say that the runtime for our original version of this data-fetching logic is approximately given by the sum of the runtimes of its operations:

$$t_{total} = t_{GetPopularBooks} + t_{GetPopularReviews} + t_{GetReadersOfTheMonth}$$

With the optimized version where operations `GetPopularBooks`, `GetPopularReviews` and `GetReadersOfTheMonth` are executed in parallel, the runtime changes:

$$t'_{total} = max\ \{ \ t_{GetPopularBooks},\ t_{GetPopularReviews},\ t_{GetReadersOfTheMonth} \ \}$$

If we once again suppose that each service action call takes approximately the same time to execute from start to finish – which is a rough, nevertheless reasonable *assumption*, then, with our optimized version, we would achieve a relative speedup of around 66% in this scenario.

Conditional Data Fetching

Conditional data fetching represents a common pattern where it may or may not be feasible to execute actions in parallel. For instance, in Figure 5.3 we can see the OutSystems logic necessary to look up a user in the database, given their name, and in case the user is found, gather information regarding the books they've read.

In this example, the `GetUserInfoByName` and `GetBooksReadByUser` target two different tables (`Users` and `Books`, respectively), and thus our algorithm detects no Read-Write Set clashes between the two. Implementation details aside, it does, however, detect a data dependence (input-output) between `GetUserInfoByName` and `UserFound?`, and a control dependence between `UserFound?` and `GetBooksReadByUser`. No opportunities for safe parallelization are found by our algorithm; in other words, it flags the program as bound to execute in a sequential fashion - and rightfully so.

Nevertheless, there are scenarios of conditional data fetching where there is room for some parallelism. Take a look at Figure 5.4, an extension of the previous example:

The difference from this example to the one of Figure 5.3 is the introduction of the service call to `GetTopReviewsByUser`. Now, after the `UserFound?` verification we have

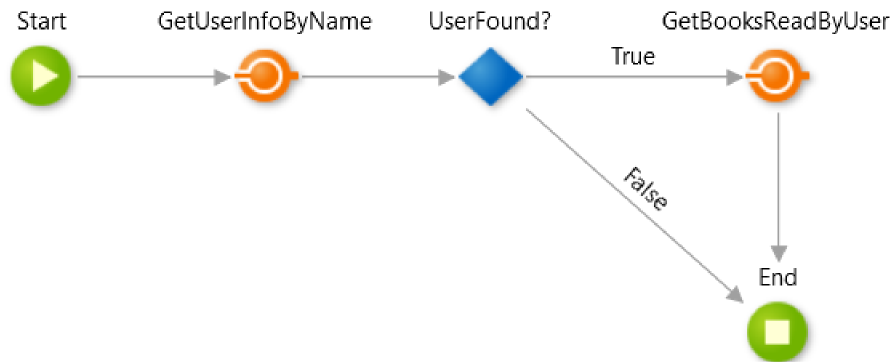

Figure 5.3: OutSystems logic for a simple example of conditional data fetching.

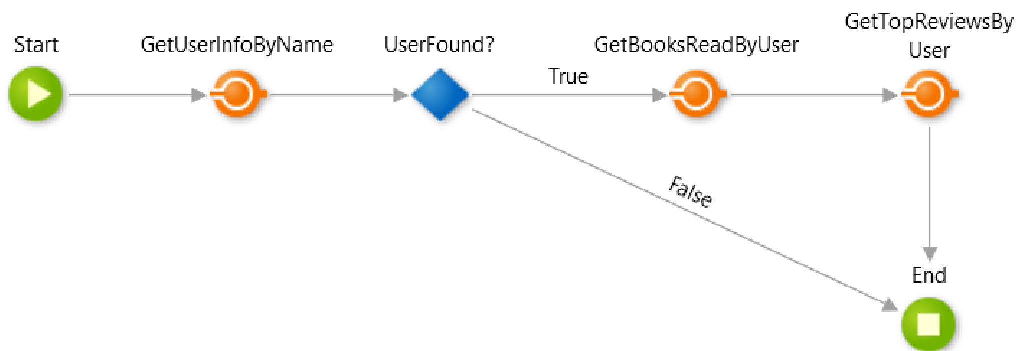

Figure 5.4: OutSystems logic for a simple example of conditional data fetching.

not one but two service calls. Given that only data *reads* are performed, there are no Read-Write Set clashes between the two actions. This leaves us with a scenario similar to that of Figure 5.1, and a dependence graph as shown in Figure 5.5. The dependence graph would then allow us to section our nodes as displayed below in Table 5.2.

Taking into account that it is the service action calls that make up the bulk of runtime in these pieces of logic, let's say that the runtime for our original version is, once again, approximately given by the sum of the runtimes of its service calls:

$$t_{total} = t_{GetUserInfoByName} + t_{GetBooksReadByUser} + t_{GetTopReviewsByUser}$$

With the optimized version where operations `GetPopularBooks`, `GetPopularReviews` and `GetReadersOfTheMonth` are all executed in parallel, the runtime changes, grant a relative speedup of roughly 33%:

$$t'_{total} = t_{GetUserInfoByName} + max\left\{ t_{GetBooksReadByUser}, t_{GetTopReviewsByUser} \right\}$$

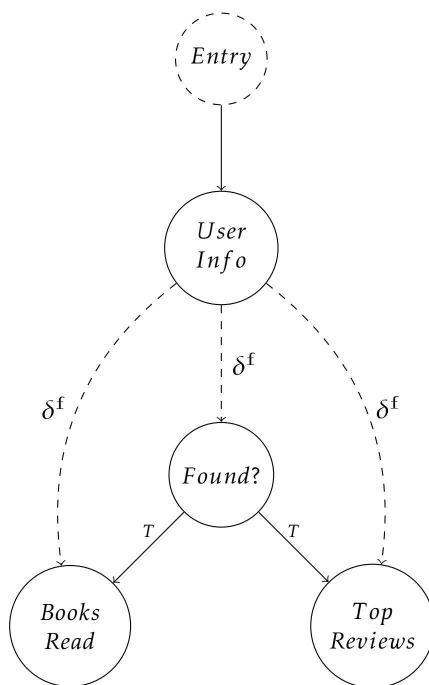

Figure 5.5: Program Dependence Graph for our example of Figure 5.4.

Node	Dependences From	Section
Entry	–	S_0
GetUserInfoByName	Entry (S_0)	S_1
UserFound?	GetUserInfoByName (S_1)	S_2
GetBooksReadByUser	GetUserInfoByName (S_1), UserFound? (S_2)	S_3
GetTopReviewsByUser	GetUserInfoByName (S_1), UserFound? (S_2)	S_3

Table 5.2: Section assignments for the nodes in our example of Figure 5.4.

Dependence-Independent, Yet Mutually Exclusive Operations

There are scenarios where sectioning information by itself is not enough to make decisions on operation parallelism. As discussed near the end of Section 4.2.6, special care should be taken when dealing with nodes that have control dependences in order to make sure that the original semantics of the program are not violated. Figure 5.6 portrays such a scenario: when a user wants to check some book, a Service Action call is made to fetch the books's details. If the book has any reviews – information that we can assume to be available on the details fetched beforehand - those reviews are fetched; otherwise a collection of similar books is requested and displayed instead.

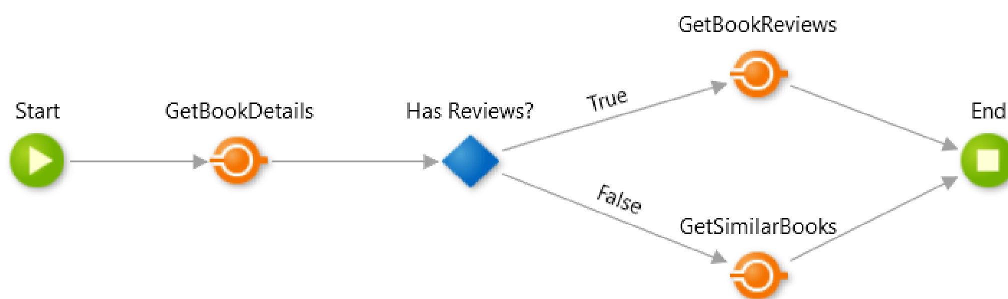

Figure 5.6: Logic for an OutSystems operation with mutually exclusive calls, `GetBookReviews` and `GetSimilarBooks`.

Behind the scenes, all three Service Action calls in this operation perform database read accesses; the absence of database writes means no Read-Write Set dependences exist. The condition node is data-dependent on the call to `GetBookDetails` (input-output dependence), and the fetching of reviews and similar books are both control-dependent on the `HasReviews?` condition and data-dependent on the information given by `GetBookDetails`. The Program Dependence Graph produced by our algorithm is displayed on Figure 5.7.

The sectioning algorithm is then executed over our the Program Dependence Graph of Figure 5.7, and the results are as displayed in Table 5.3.

There is a clear resemblance between the Program Dependence Graph and section assignments of this example and the previous one, *Conditional Data Fetching*; this is because in, practice, our novel example *does* perform conditional data fetching, and its operations are of similar nature. There is, however, a key difference that makes this example stand out as a special case, and that is the mutual exclusivity of `GetBookReviews` and `GetSimilarBooks`. On the original program flow, there is never a scenario where both operations get executed, meaning that, despite both being assigned to S_3, their execution in parallel would imply generating a program with semantics different from the original. Thus, our algorithm ends up finding no opportunities for safe parallelism here.

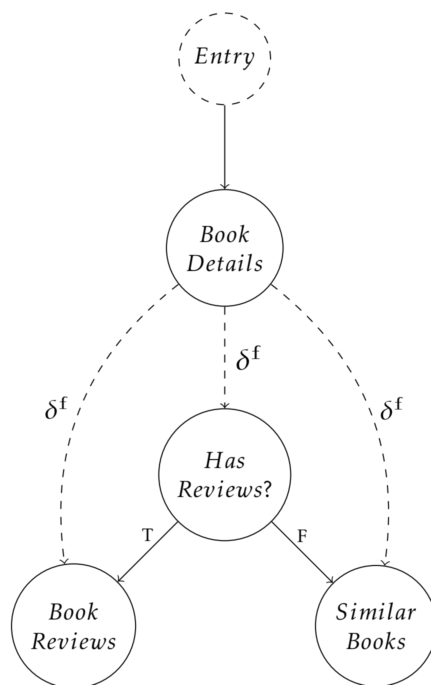

Figure 5.7: Program Dependence Graph for our example of Figure 5.4.

Node	Dependences From	Section
Entry	–	S_0
GetBookDetails	Entry (S_0)	S_1
HasReviews?	GetBookDetails (S_1)	S_2
GetBookReviews	GetBookDetails (S_1), HasReviews? (S_2)	S_3
GetSimilarBooks	GetBookDetails (S_1), HasReviews? (S_2)	S_3

Table 5.3: Section assignments for the nodes in our example of Figure 5.6.

For-Each Loops

Operating over each item in a collection of data is a rather common pattern in application development. The most common way to do this in OutSystems is to use the `ForEach` node, as illustrated in Figure 5.8.

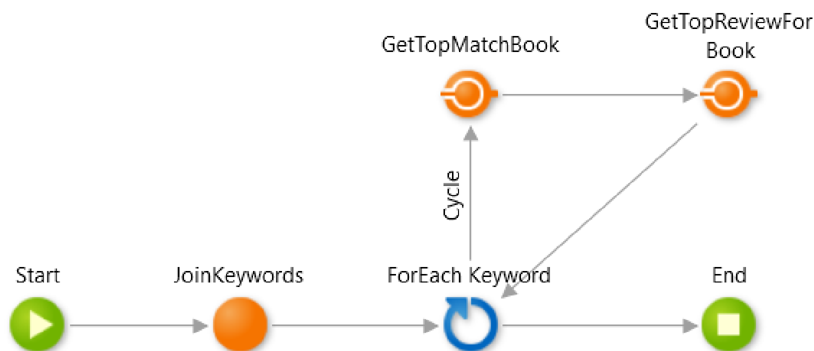

Figure 5.8: OutSystems logic for an example with a for-each loop.

This example showcases what could be a fuzzy search operation based on keywords. In this operation, a string with comma-separated keywords – which we can assume to be given by user input – is first joined into a list; then, for each keyword, we fetch whatever single book (for simplicity) matches the keyword the best. We can assume the matching to be based on title similarity, though the inner logic is not relevant here. We can also suppose that the operation always returns *some* book. Alongside the book itself, we then fetch that book's most popular review to then display this information to the user.

The list returned by `JoinKeywords` is fed not only to the `ForEach` node, but also to the `GetTopMatchBook` operation; though `GetTopMatchBook` does only handle one item at a time, our analysis is not fine-grained enough uncover this, and so a data dependence is identified. The two nodes that comprise the body of the loop perform database read accesses to the Books and Reviews tables; moreover, they are, control-dependent on the `ForEach` node. The Program Dependence Graph for this example is displayed in Figure 5.9.

Running the sectioning algorithm, the results are as shown in Table 5.4.

Under this scenario, safe parallelism inside the body of the loop is not possible, considering the data dependence between `GetTopReviewForBook` and `GetTopMatchBook`. Parallelism *across* iterations, however, could be suitable here, as the operations inside the loop simply perform database reads – there are no mutating operations.

Of course, how analysis of loop scenarios is admittedly rough, due to scope and time constraints. Nevertheless, it is still applicable at least to simpler scenarios.

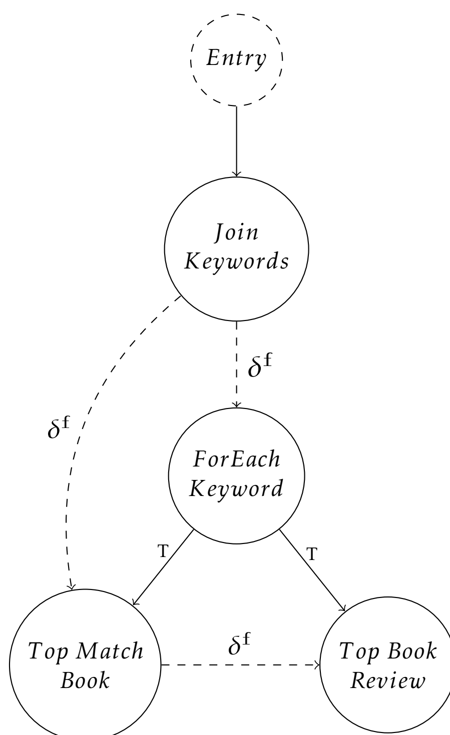

Figure 5.9: Program Dependence Graph for our example of Figure 5.8.

Node	Dependences From	Section
Entry	–	S_0
JoinKeywords	Entry (S_0)	S_1
ForEachKeyword	JoinKeywords (S_1)	S_2
TopMatchBook	JoinKeywords (S_1), ForEachKeyword (S_2)	S_3
TopBookReview	ForEachKeyword (S_2), TopMatchBook (S_3)	S_4

Table 5.4: Section assignments for the nodes in our example of Figure 5.8.

5.2 On the Development Guidelines Proposed

Back in Chapter 1, whilst giving an overview of our solution, we introduced what we called *solution design guidelines*, a set of goals whose achievement would guide the decisions made during the development of this work. In this section we will reflect on how these guidelines were (or not) followed, and what this meant for the prototype.

"First and foremost, the solution must always guarantee correctness of the optimized program. Not finding opportunities for safe parallelism could be an acceptable worst case scenario; marking as parallelizable tasks that are effectively not safe to parallelize, however, is never an acceptable output."

Guaranteeing the correctness of a program is challenging, even more so as we enter the realm of parallelism and asynchronicity. In the context of our problem, the conservative behavior of the different pieces that make up our solution is somewhat of an assurance of a minimum degree of correctness for each one of them. In practice, the multiple parts that comprise our solution were unit tested, though some with more coverage than others. The testing of the work developed *as a whole* was done empirically, based on an admittedly short set of examples – and though the results obtained were mostly satisfying, it is fair to say that this does not bring a whole lot of confidence in the correctness of our solution.

"The solution should introduce little-to-no runtime overhead; the number of remote API calls inside OutSystems software factories could be arbitrarily large, and so even a small amount of runtime overhead could cause some performance hindrance."

The entirety of our solution can happen strictly in compile time, and so we can mark this goal as completed. Nevertheless, part of our Read-Write Set analysis is done live during development time, triggered by the changes the developer does, which is also something we have to take into account. Though this dynamism is important to make our solution be truly relevant *in practice*, we did not have the time to evaluate the true impact of these computations on large OutSystems modules. In case this would prove too heavy, a possible way to reduce the overhead could to change the trigger/invalidation strategy to specific moments, such as to when an application is published. Though unrelated to runtime overhead *per se*, the *storage* overhead of storing Read-Write Sets for every action in a cumulative fashion should also be considered. On a large scale, minifying the representation of the Data Access Triples or storing pointers to other nodes (and not those nodes' Read-Write Sets) could be viable options to reduce storage size.

"The solution should be mostly invisible to the OutSystems developer. Our solution should never impose an obstacle nor require extra effort from the developer."

We developed a solution that is fully automatic and also non-intrusive to the developer,

checking this goal as met. The fully automatic facet reflects the fact that *no* intervention at all is required from the developer. Furthermore, because everything happens behind the scenes, there is no increase in development overhead. Now, to cover the full spectrum, we expect the need for at least *some* developer interaction and awareness – there is definitely a trade-off between developer work and range of opportunities for parallelism uncovered – though the cases we handle with our solution can still be mostly invisible.

Chapter 6

Conclusions

When dealing with remote action calls, an orchestrator has to coordinate several requests, respecting the dependencies among them and any protocols they might follow. Naive orchestration strategies perform the requests sequentially, spending time just waiting for the results. Moreover, the orchestrated services themselves may be orchestrators as well, meaning that this idle time waiting for results can grow surprisingly fast. A possible strategy to improve the efficiency of the orchestration process, though, is to employ *task parallelism* when performing requests that are independent of each other.

With this work we aimed to develop an algorithm that would analyse programs looking for opportunities for safe parallelism between operations. A lot of the literature on the uncovering of parallelism emphasizes scientific computation/loop-level parallelism [40, 46] or speculative execution [12, 13, 15, 32], which, though not directly viable for our use case, hinted at some ideas and paths to follow.

We ended up achieving this by studying and developing a pipeline of dependence analyses and dependence representation techniques. Design and development guidelines were defined early on, highlighting our goal for a solution to be safe, unobtrusive to the developer and to mostly take place in compile time. Though designed from the ground up with general-purpose applicability in mind, these guidelines were *also* designed to ensure our solution would be applicable to the low-code philosophy of OutSystems, the platform for which we developed a working prototype.

The results we obtained via empirical testing were promising, and we are satisfied with the work produced and with the individual contributions that came along with it. Nonetheless, we would like to point out that, inherent to the problem we had in hands, there is a whole range of promising future work waiting to be explored, something we discuss in Section 6.2.

6.1 Contributions

This dissertation produced the following contributions:

- A static analysis algorithm that extracts information regarding database *reads* and *writes* performed by remote API calls. This can be mapped to different platforms and data sources;
- A dependence analysis algorithm, capable of summarizing not only control and data dependences, but also database-access dependences of different operations in a program during compile time. The algorithm generates a Program Dependence Graph, and is capable of distinguishing between the different types of data and control dependences. Despite our target being the OutSystems platform, the algorithm is adaptable to other scenarios where a graph representation can be constructed;
- An algorithm capable of partitioning the operations of a program into different sections, based on the dependences between those operations. This information can then be used to aid in the construction of the optimized program;
- A working prototype targeting (a sub-set of) the OutSystems platform that employs the techniques described above. A simple yet effective SQL parser was also developed under this context. Though not extensively tested, the prototype yielded favorable results, hinting at the viability and applicability of our solution.

6.2 Future Work

As made clear in the previous chapter, the objectives for this work were fulfilled in accordance with our initial objective. Even though it would have been ideal to have achieved parallel code generation in the prototype, it ended up not being feasible within the time allocated to this dissertation (at least without compromising other parts of the solution). Nevertheless, the nature of the problem we had in hands opens up doors for a whole range of future work that could be considered:

- *Graphs with cycles*: The lack of proper support for *cyclic* graphs is arguably the biggest handicap in the solution we developed. Because they are so common in programs, dealing with cyclic graphs correctly would be a major stepping stone to a more complete solution. As discussed in Subsection 4.2.4, implementing Strongly Connected Components [8] could be a good starting point;
- *Handling exceptions*: Adding support for programs with exception-handling logic would further wide the applicability of our solution;
- *Read-Write Set Analysis granularity*: Implementing a more fine-grained Read-Write Set Extraction analysis could translate into opportunities for parallelism that are

not uncovered by our solution as it is right now; instead of inspecting database accesses at a table level, diving deeper into table *row* level could leverage more scenarios of parallelism. This was also mentioned in Section 4.2.2;

- *Performance evaluation*: A performance study on the impact of Read-Write Set recalculations and invalidations during design time would be valuable to assess whether or not we should make changes to our technique. Evaluating the storage overhead introduced as we store Read-Write Set triples in the way we currently do could also be the target of some analysis;

- *Implement code generation*: Implementing parallel code generation in the prototype would be the immediate next step to make it fully functional;

- *SQL parser robustness*: In our OutSystems prototype, the SQL parser mentioned in Section 4.3 is only capable of distinguishing between reads and writes when it comes to simple SQL queries. Though it was, by no means, the focus of this work, further strengthening the parser to handle more complex queries could be valuable.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[2] H. Akkary and M. A. Driscoll. "A Dynamic Multithreading Processor." In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 226–236. ISBN: 1-58113-016-3. URL: http://dl.acm.org/citation.cfm?id=290940.290988.

[3] F. E. Allen. "Control Flow Analysis." In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.acm.org/10.1145/390013.808479.

[4] N. Alshuqayran, N. Ali, and R. Evans. "A systematic mapping study in microservice architecture." In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51.

[5] J. Bakić. *Shielder, a STM implementation for .NET*. Accessed: 2018-12-22. 2013. URL: https://github.com/jbakic/Shielded.

[6] C Baroudi and F. Halper. "Executive Survey: SOA Implementation Satisfaction." In: *Hurwitz and Associates* (Jan. 2006).

[7] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. "Unifying Thread-level Speculation and Transactional Memory." In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 187–207. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442639.

[8] E. Bendersky. *Directed graph traversal, orderings and applications to data-flow analysis*. 2015. URL: https://eli.thegreenplace.net/2015/directed-graph-traversal-orderings-and-applications-to-data-flow-analysis/ (visited on 08/12/2019).

[9] A. J. Bernstein. "Analysis of programs for parallel processing." In: *IEEE Transactions on Electronic Computers* 5 (1966), pp. 757–763.

[10] G. Bilardi and K. Pingali. "A Framework for Generalized Control Dependence." In: *SIGPLAN Not.* 31.5 (May 1996), pp. 291–300. ISSN: 0362-1340. DOI: 10.1145/249069.231435. URL: http://doi.acm.org/10.1145/249069.231435.

[11] H. Christensen. "Algorithms for Finding Dominators in Directed Graphs." Doctoral dissertation. Aarhus Universitet, Datalogisk Institut, 2016.

[12] M. Cintra and D. R. Llanos. "Design Space Exploration of a Software Speculative Parallelization Scheme." In: *IEEE Trans. Parallel Distrib. Syst.* 16.6 (June 2005), pp. 562–576. ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.69. URL: https://doi.org/10.1109/TPDS.2005.69.

[13] L. Codrescu and D. S. Wills. "On dynamic speculative thread partitioning and the MEM-slicing algorithm." In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. Oct. 1999, pp. 40–46. DOI: 10.1109/PACT.1999.807404.

[14] K. D. Cooper, T. J. Harvey, and K. Kennedy. "A Simple , Fast Dominance Algorithm." In: *Rice University, CS Technical Report 06-33870* (Jan. 2001).

[15] F. H. Dang and L. Rauchwerger. "Speculative Parallelization of Partially Parallel Loops." In: *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. LCR '00. London, UK, UK: Springer-Verlag, 2000, pp. 285–299. ISBN: 3-540-41185-2. URL: http://dl.acm.org/citation.cfm?id=648049.746025.

[16] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. "Optimistic Hybrid Analysis: Accelerating Dynamic Analysis Through Predicated Static Analysis." In: *SIGPLAN Not.* 53.2 (Mar. 2018), pp. 348–362. ISSN: 0362-1340. DOI: 10.1145/3296957.3177153. URL: http://doi.acm.org/10.1145/3296957.3177153.

[17] M. B. Dwyer and L. A. Clarke. "Data Flow Analysis for Verifying Properties of Concurrent Programs." In: *SIGSOFT Softw. Eng. Notes* 19.5 (Dec. 1994), pp. 62–75. ISSN: 0163-5948. DOI: 10.1145/195274.195295. URL: http://doi.acm.org/10.1145/195274.195295.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization." In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: http://doi.acm.org/10.1145/24039.24041.

[19] R. J. Figueiredo and J. A. B. Fortes. "Hardware Support for Extracting Coarse-Grain Speculative Parallelism in Distributed Shared-Memory Multiprocesors." In: *Proceedings of the 2001 International Conference on Parallel Processing*. ICPP '02. Washington, DC, USA: IEEE Computer Society, 2001, pp. 214–226. ISBN: 0-7695-1257-7. URL: http://dl.acm.org/citation.cfm?id=645535.657150.

[20] L. Georgiadis, R. E. Tarjan, and R. F. F. Werneck. "Finding dominators in practice." In: *J. Graph Algorithms Appl.* 10.1 (2006), pp. 69–94.

[21] J. Graf. "Information Flow Control with System Dependence Graphs - Improving Modularity, Scalability and Precision for Object Oriented Languages." Doctoral dissertation. 2016.

[22] R. H. Halstead Jr. "MULTILISP: A Language for Concurrent Symbolic Computation." In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538. ISSN: 0164-0925. DOI: 10.1145/4472.4478. URL: http://doi.acm.org/10.1145/4472.4478.

[23] T. Harris and K. Fraser. "Language Support for Lightweight Transactions." In: *SIGPLAN Not.* 38.11 (Oct. 2003), pp. 388–402. ISSN: 0362-1340. DOI: 10.1145/949343.949340. URL: http://doi.acm.org/10.1145/949343.949340.

[24] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. "Composable Memory Transactions." In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: http://doi.acm.org/10.1145/1065944.1065952.

[25] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pp. 26–60.

[26] R. Johnson and K. Pingali. "Dependence-based Program Analysis." In: *SIGPLAN Not.* 28.6 (June 1993), pp. 78–89. ISSN: 0362-1340. DOI: 10.1145/173262.155098. URL: http://doi.acm.org/10.1145/173262.155098.

[27] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. "Dependence graphs and compiler optimizations." In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 1981, pp. 207–218.

[28] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control." In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567. URL: http://doi.acm.org/10.1145/319566.319567.

[29] T. Lengauer and R. E. Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph." In: *ACM Trans. Program. Lang. Syst.* 1.1 (Jan. 1979), pp. 121–141. ISSN: 0164-0925. DOI: 10.1145/357062.357071. URL: http://doi.acm.org/10.1145/357062.357071.

[30] M. LLC. *The OutSystems Platform.* 2019. URL: https://www.outsystems.com/platform/ (visited on 01/11/2019).

[31] P. Marcuello, A. González, and J. Tubella. "Thread Partitioning and Value Prediction for Exploiting Speculative Thread-Level Parallelism." In: *IEEE Trans. Comput.* 53.2 (Feb. 2004), pp. 114–125. ISSN: 0018-9340. DOI: 10.1109/TC.2004.1261823. URL: https://doi.org/10.1109/TC.2004.1261823.

[32] C. E. Oancea, A. Mycroft, and T. Harris. "A Lightweight In-place Implementation for Software Thread-level Speculation." In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. New York, NY, USA: ACM, 2009, pp. 223–232. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584050. URL: http://doi.acm.org/10.1145/1583991.1584050.

[33] R. A. Olsson, R. H. Crawford, and W. W. Ho. "A Dataflow Approach to Event-based Debugging." In: *Softw. Pract. Exper.* 21.2 (Feb. 1991), pp. 209–229. ISSN: 0038-0644. DOI: 10.1002/spe.4380210207. URL: http://dx.doi.org/10.1002/spe.4380210207.

[34] OutSystems. *Expose and Reuse Functionality in OutSystems 11*. Accessed: 2019-01-15. 2018. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor/Expose_and_Reuse_Functionality_Between_Modules.

[35] OutSystems. *Use Services to Expose Functionality in OutSystems 11*. Accessed: 2019-01-16. 2018. URL: https://success.outsystems.com/Documentation/11/New_in_OutSystems_11.

[36] K. Pingali and G. Bilardi. "Optimal Control Dependence Computation and the Roman Chariots Problem." In: *ACM Trans. Program. Lang. Syst.* 19.3 (May 1997), pp. 462–491. ISSN: 0164-0925. DOI: 10.1145/256167.256217. URL: http://doi.acm.org/10.1145/256167.256217.

[37] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies." In: (July 1999). DOI: 10.1145/99583.99595.

[38] A. Podgurski and L. A. Clarke. "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance." In: *IEEE Trans. Softw. Eng.* 16.9 (Sept. 1990), pp. 965–979. ISSN: 0098-5589. DOI: 10.1109/32.58784. URL: https://doi.org/10.1109/32.58784.

[39] R. T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams." In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). New York, NY, USA: ACM, 1959, pp. 133–138. DOI: 10.1145/1460299.1460314. URL: http://doi.acm.org/10.1145/1460299.1460314.

[40] S. Rus, M. Pennings, and L. Rauchwerger. "Sensitivity Analysis for Automatic Parallelization on Multi-cores." In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS '07. New York, NY, USA: ACM, 2007, pp. 263–273. ISBN: 978-1-59593-768-1. DOI: 10.1145/1274971.1275008. URL: http://doi.acm.org/10.1145/1274971.1275008.

[41] N. Shavit and D. Touitou. "Software Transactional Memory." In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: 10.1145/224964.224987. URL: http://doi.acm.org/10.1145/224964.224987.

[42] A. Thakur and R. Govindarajan. "Comprehensive Path-sensitive Data-flow Analysis." In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '08. New York, NY, USA: ACM, 2008, pp. 55–63. ISBN: 978-1-59593-978-4. DOI: 10.1145/1356058.1356066. URL: http://doi.acm.org/10.1145/1356058.1356066.

[43] J. Torrellas. "Thread-Level Speculation." In: *Encyclopedia of Parallel Computing, Springer Science+Business Media LLC* (May 2011).

[44] P. Veentjer. *Multiverse, a STM implementation in Java*. Accessed: 2018-12-22. 2012. URL: https://github.com/pveentjer/Multiverse.

[45] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján. "Optimizing Software Runtime Systems for Speculative Parallelization." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 39:1–39:27. ISSN: 1544-3566. DOI: 10.1145/2400682.2400698. URL: http://doi.acm.org/10.1145/2400682.2400698.

[46] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien. "Exploiting Parallelism with Dependence-Aware Scheduling." In: *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 193–202. ISBN: 978-0-7695-3771-9. DOI: 10.1109/PACT.2009.10. URL: https://doi.org/10.1109/PACT.2009.10.