Impact Analysis of Database Schema Changes

Andy Maule

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of the
University of London.

Department of Computer Science
University College London

2009

I, Andrew Maule, confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the dissertation.

Abstract

When database schemas require change, it is typical to predict the effects of the change, first to gauge if the change is worth the expense, and second, to determine what must be reconciled once the change has taken place. Current techniques to predict the effects of schema changes upon applications that use the database can be expensive and error-prone, making the change process expensive and difficult. Our thesis is that an automated approach for predicting these effects, known as an *impact analysis*, can create a more informed schema change process, allowing stakeholders to obtain beneficial information, at lower costs than currently used industrial practice. This is an interesting research problem because modern data-access practices make it difficult to create an automated analysis that can identify the dependencies between applications and the database schema. In this dissertation we describe a novel analysis that overcomes these difficulties.

We present a novel analysis for extracting potential database queries from a program, called *query analysis*. This query analysis builds upon related work, satisfying the additional requirements that we identify for impact analysis.

The impacts of a schema change can be predicted by analysing the results of query analysis, using a process we call *impact calculation*. We describe impact calculation in detail, and show how it can be practically and efficiently implemented.

Due to the level of accuracy required by our query analysis, the analysis can become expensive, so we describe existing and novel approaches for maintaining an efficient and computational tractable analysis.

We describe a practical and efficient prototype implementation of our schema change impact analysis, called SUITE. We describe how SUITE was used to evaluate our thesis, using a historical case study of a large commercial software project. The results of this case study show that our impact analysis is feasible for large commercial software applications, and likely to be useful in real-world software development.

Acknowledgements

I would like to thank my supervisor, Wolfgang Emmerich, first, for deciding to take me on as a PhD student, and second for allowing me the freedom to take this research in directions that I found interesting, whilst still giving me the support, guidance and advice that turned an interesting research project into a PhD dissertation.

Very important to this research, was the funding from Microsoft Research, which ran out quicker than I might have liked, but was exceedingly generous. I would like to thank Stephen Emmott, Luca Cardelli, Fabien Petitcolas and Gavin Bierman at Microsoft Research Cambridge for all your help during the PhD process. I hope you feel that it was a worthwhile investment.

I would like to thank Anthony Finkelstein and David Rosenblum for their roles as assessor and second supervisor respectively, and for all the advice and help.

I've had the good fortune to be able to discuss my work with many academics that have visited UCL, or that I've met at conferences and workshops. I'd like to thank Lori Clarke for discussions on dataflow analysis, Barbara Ryder for discussion of context sensitive analysis, Kyung-Goo Doh and Oukseh Lee for discussion of their string analysis, Anders Moeller for the discussion of JSA and related work, Chris Hankin for discussion on k-CFA dataflow analysis, Mark Harman and David Binkley for discussions on program slicing.

I'd like to thank James Higgs and the Interesource team for the provision of our case study.

I'd like to thank James Skene and Franco Raimondi for proof reading this dissertation, and the very helpful advice and comments.

I've made many good friends at UCL, and to all of you, thanks for the Friday nights spent playing table football, and lunch times spent searching for cheap and plentiful food around Bloomsbury. There are too many of you to list here, but you know who you are.

Finally I'd like to say thank you to my family, my parents and my sister. I know you have no idea what I'm talking about in this dissertation, yet you've supported me through it anyway, even when I've been very stressed or going through difficult times. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In the simplest definition, a *database* is a collection of data. Software applications that are designed to help create, maintain and use databases are called *database management systems* (DBMS). DBMSs provide abstractions by which the logical model of the data can be separated from the way it is physically stored. An application developer, or a database administrator (DBA), can manipulate the data in a database using a high level logical representation, whilst the DBMS manages the persistence of the data to the physical storage on the available hardware.

DBMSs must address the complex issues that arise when managing data, such as concurrent access to data and efficient execution of complex queries. Addressing these problems requires a significant engineering effort, which often makes the use of commercial DBMSs more cost-effective than creating application specific solutions for persistent data management. For these reasons, databases are commonly used, as shown by a report by Gartner Inc. on the DBMS software market in 2005 [Fabrizio Biscotti, 2006], which shows that the total revenue allocated to the market in Europe, the Middle East and Africa, totalled 4.7 billion Euros, and is continuing to grow.

A *database schema* defines the type and structure of the high-level logical model of the database; the schema typically specifies the types of the entities being modelled and the relationships between the entities. The DBMS has the task of managing the data, making sure it conforms to the database schema, and managing the conversion of an instance of this logical model, to and from a physical model of the data that is stored on the system hardware. Each database typically has a schema which can be altered and changed as required during the lifetime of the database. Applications which use a database may be dependent on the schema of any database that they use, expecting the data to be structured in a specific way. For example, an inventory application may expect a database to contain information about products, including product names, and without this information the application will not function correctly. We refer to a software application which uses a database managed by a DBMS, as a *database application*.

We say that an application with a large amount of dependency upon a database is *tightly coupled* to the database. Coupling is a measure of the degree of dependency that exists between two components, and in software engineering, coupling is usually minimised to help create maintainable applications [Pfleeger, 1998], allowing one component to be changed without requiring change in

another. However, in many modern applications the coupling between applications and databases, remains high [Fowler, 2003], despite attempts to minimise the coupling between applications and databases [Atkinson et al., 1983, 1996].

The need to change software systems can arise for many different reasons, and changes to database systems can be common [Sjoberg, 1993]. Good software engineering practice requires that we estimate the cost of changes before we make them, to assess if changes are feasible or worthwhile, and to see if alternative changes might be more appropriate.

The impact of schema changes upon the database itself have been well researched by the database research community, under the heading of *schema evolution*, and the research has advanced to the point where tools can respond to many changes automatically [Curino et al., 2008]. In comparison, the amount of research into the the impact of database schema changes upon applications is much less. Some of the most recent recommended industrial practice for managing databases, shows that the impacts of schema changes upon applications must often be estimated manually by application experts with years of experience [Ambler and Sadalage, 2006]. Assessing the effect of changes by hand, is a fragile and difficult process, as noted by Law and Rothermel [2003], who show that expert predictions can be shown to be frequently incorrect [Lindvall M., 1998] and impact analysis from code inspections can be prohibitively expensive [Pfleeger, 1998]. For these reasons, we argue that the current methods for assessing the effects of database schema change upon applications are, in many cases, inadequate. It has already been argued that better tools for managing the effects of impacts are required [Ambler and Sadalage, 2006, Sjoberg, 1993], therefore, this dissertation describes the results of research into automated tools for estimating the impacts of databases schema changes upon applications.

Much of our research has been presented before, and we have previously described our automated impact analysis for database schema changes [Maule et al., 2008]. This dissertation expands on upon this earlier presentation of our research, presenting further results, and describing our approach in much greater detail.

This problem is very broad and before we discuss the contributions we describe in this dissertation, we shall discuss the scope of our work.

1.2 Scope

First, we limited this research to focus upon the impact of schema changes upon applications, rather than the impacts upon the database itself. This is because, as discussed above, the effects of change upon the database, have been studied in detail in the schema evolution research, whilst there is comparatively little research into the impacts of change upon applications, yet this is an important problem.

Second, we limited this research to relational DBMSs. According to recent studies [Fabrizio Biscotti, 2006] the majority of revenue in the modern DBMS market comes from DBMSs based upon the relational model [1], and this type of database also has the fastest growing market share. This is despite the emergence of other types of database model such as deductive and object-oriented databases [Delobel

[1] Although such databases are not strictly relational as defined by Codd [1970], we shall use the term relational to signify the class of modern DBMS that is predominantly based on a relational model, typically those that support SQL as a query language.

et al., 1991]. Therefore, relational DBMSs shall be our primary focus, because of their dominance in current industrial practice.

Third, we shall limit this research to applications written using object-oriented(OO) programming languages. OO programming languages are also very popular in industry [Rubin and Tracy, 2006], and are commonly used to write applications that manipulate databases. Accessing relational data from OO programming languages is therefore a very common task. It is, therefore, important that our impact analysis should be able to process OO applications. Creating an impact analysis for other types of programming language, is unfeasible within the scope of this dissertation, because of the fundamental differences between modern OO languages and other paradigms such as functional, or symbolic languages. We have chosen to investigate OO programming languages over other language paradigms, due to their popularity in modern industrial practice.

The differences between the OO paradigm and the relational model can make it difficult to process relational data in OO programs and vice versa. The difficulty in overcoming the differences between these two models has become known as the *object-relational impedance mismatch problem* [Atkinson et al., 1990]. The presence of the object-relational impedance mismatch makes for a particularly interesting research problem, because it requires applications to be written in a way that makes it difficult to establish traceability between the application and the database schema.

1.3 Contributions

Our thesis is that automated tool support for predicting the effects of schema change can create a more informed schema change process, providing the stakeholders with beneficial information, at lower costs than currently used industrial practice. Towards the investigation of this thesis, we have developed an automated analysis for assessing the effects that relational database schema changes have upon object-oriented applications. Assessing or predicting the effects of change in software systems is typically known as software change impact analysis [Bohner and Arnold, 1996], and in this dissertation we describe our novel impact analysis. The specific contributions that have resulted from this research are as follows:

1. We have identified that the closest related work cannot be applied to impact analysis of database applications, without having the potential for high levels of incorrect predictions. This inaccuracy[2] is due to the lack of precision in the techniques used for analysing procedural programs, and the way in which database applications are written in practice. To solve these problems, we have extended related work to create a novel *query analysis*, for finding the parts of an application that are dependent upon a database schema.

2. We have defined a novel technique called *impact calculation* where the results of a query analysis can be used to predict the effect of database schema change, and investigated how impact calculation can be practically and efficiently implemented.

[2] We define accuracy in more detail in Chapter 2

3. Due to the accuracy required by our query analysis, it is more expensive than related work. The closest related work on efficient dataflow analyses is not applicable to our approach, due to the complexity of our query analysis. Therefore, we have developed a novel approach to reducing the overall cost of our impact analysis, by using *data-dependence program-slicing*, which is a variant of a more common software analysis called program slicing [Tip, 1994].

4. We have investigated the combination of novel analyses, such as query analysis and impact calculation, with established techniques, such as program slicing and garbage collection, to create an accurate and computationally tractable analysis with high impact-precision[3]. We have combined these approaches to create a novel *database schema change impact analysis*.

5. We have created a prototype tool called *SUITE* that implements our database schema change impact analysis.

6. We have applied SUITE to a large industrial case-study to evaluate its potential usefulness, and to evaluate the overall feasibility of our approach.

1.4 Dissertation Outline

The remainder of this Dissertation is organised as follows:

Chapter 2 We introduce a scenario to be used as a running example, which will further motivate the need for schema change impact analysis, and will further illustrate why this is an important and interesting problem.

Chapter 3 We build upon established analyses to create a *query analysis* for identifying all the places in an application which might interact with a database, therefore establishing dependency relationships between the application and the schema. We describe this analysis formally in a language neutral way. We discuss related work.

Chapter 4 We introduce a novel *impact calculation* technique, which uses the results of a query analysis to predict the effects of database schema change. We show how this analysis can be formally specified, and how it can be practically and efficiently implemented. We discuss alternative approaches and related work.

Chapter 5 We present novel techniques to reduce the cost of query analysis, so that the required level of accuracy can be maintained, whilst allowing the analysis to scale to programs of real-world size. We also describe the major performance optimisations that are used in the prototype implementation. We discuss alternative approaches and related work.

Chapter 6 We discuss how the work from previous chapters can be combined to create a database schema change impact analysis. We describe our prototype implementation tool SUITE. We discuss related work and alternative and complementary approaches for managing the schema change process.

[3] We shall define *impact-precision* as the level of detail of the analysis in Chapter 2.

Chapter 7 We present an evaluation of the feasibility of our impact analysis using a large industrial case-study, and we discuss the accuracy, impact-precision, cost and scalability of this analysis. We discuss the feasibility and potential usefulness of our analysis as well as the threats to validity for our case-study.

Chapter 8 We discuss the conclusions of our research with respect to our thesis and goals, and also discuss the contributions and future directions for this research.

Chapter 2

Motivating Example

In this chapter we introduce example applications to describe the problems caused by schema change in more detail, and to explain further our motivation for addressing these problems. These applications will be used as a running example throughout the rest of this dissertation.

2.1 The UCL Coffee Company

Our example applications are used by a fictitious company called UCL Coffee, a coffee wholesalers specialising in providing coffee to academic institutions. The clients of UCL Coffee are particularly demanding, requiring a variety of specialist coffees, such as coffee with unusually high caffeine levels or coffee produced and sold according to very specific economic principles. The company therefore has to source its coffee from specialist suppliers from all over the world.

UCL Coffee has two software applications that it uses for managing its business, the sales application and the inventory application. The sales application is for managing customer orders and customer records. The inventory application is for updating stock details, managing supplier information and adding and removing products. These applications require access to the same data concerning customers, orders, inventory and suppliers, and use a database to store these data.

The database schema for these applications is shown by an entity relation (ER) diagram in Figure 2.1. Each box represents a table and shows the name of the table and the columns that the table contains. The tables store the required information about customers, orders, products and suppliers. The relationships between these entities are enforced by referential constraints. Every customer can have many possible orders, every product has exactly one supplier and every order can consist of multiple products[1].

For illustrative purposes, we have kept this schema, and the applications, deliberately simple. We omit many details that would appear in a real application, such as validation, as they would make the example larger without adding anything to the explanation of the problem we are addressing.

We refer to each interaction with the database as a `query`, and the term query is used for all interactions, regardless of whether they are retrieving or updating information.

The sales application can produce the following queries:

[1] The `OrderProduct` table is used to provide the required many-to-many relationship between products and orders; this is the standard way to implemented this type of relationship in a relational database.

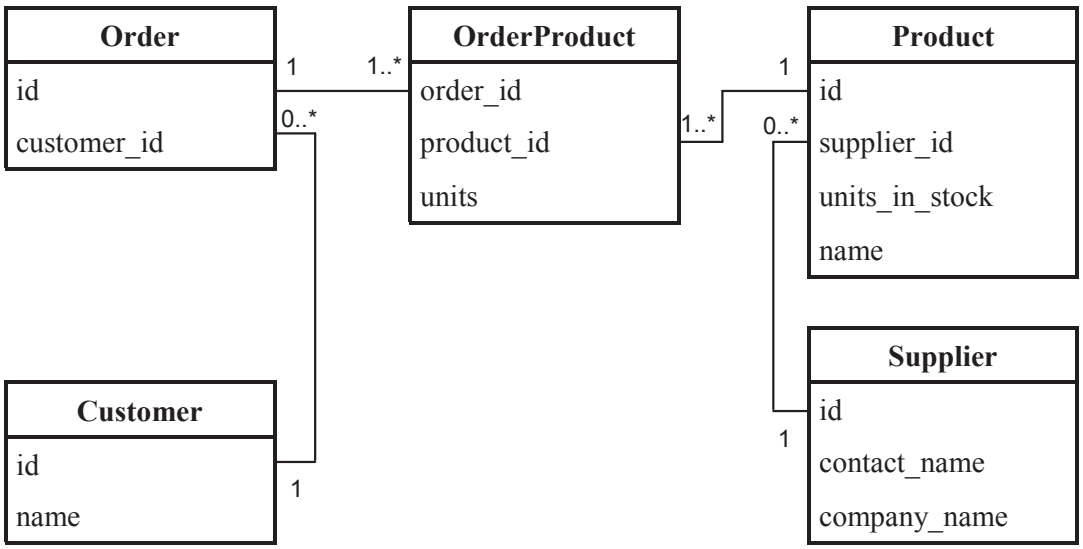

Figure 2.1: UCL Coffee database schema

SALES-Q1 Find all customers

SALES-Q2 Create a new customer

SALES-Q3 Update a customer's details

SALES-Q4 Delete a customer

SALES-Q5 Create a new order

SALES-Q6 Add a product to an order

SALES-Q7 Find all orders for a given customer

The inventory application can produce the following queries:

INV-Q1 Find a product with a given identifier

INV-Q2 Find a supplier with a given identifier

INV-Q3 Create a new product

INV-Q4 Find products where the product or supplier details matches a keyword

INV-Q5 Create a new supplier

Whilst we have expressed these queries here in English, they would typically be implemented in some kind of query language, for example SQL [Date, 1989]. We shall see examples of how these queries are actually implemented in detail, in Section 2.2.1.

The development of the applications and the database are managed by 3 separate teams. The sales application is developed by a team of software engineers, as is the inventory application, whilst the

database is managed by a team of database administrators. Each team is responsible for maintaining and modifying their respective system.

2.2 The Requirement for a Better Impact Analysis

Recently UCL Coffee has been getting complaints from customers about the incorrect use of their names, titles and salutations. As UCL Coffee mainly supplies coffee to academic institutions, it has a high number of customers with titles such as Dr. or Professor, who can sometimes object to being called Mr, Mrs or Miss.

The `Customer` and `Supplier` tables, shown in Figure 2.1, both use a single column to store the name of the customer or supplier contact respectively. Storing the names in this way has been sufficient up until now; whenever names have been used, they have been used in their entirety, and any titles have been added as appropriate, by hand.

UCL Coffee has grown beyond all expectations and now deals with far too many customers for the staff to know when to add salutations and titles by hand. Because of this, it is decided that the applications need to be able to display the constituent parts of people's names, such as first name and title, so that they can be used more accurately and stop the complaints of customers.

The database team proposes the following changes to the schema:

SchemaChange1 Drop the column Customer.name

SchemaChange2 Add the required column Customer.title

SchemaChange3 Add the required column Customer.first_name

SchemaChange4 Add the required column Customer.last_name

SchemaChange5 Add the optional column Customer.other_names

SchemaChange6 Drop the column Supplier.contact_name

SchemaChange7 Add the required column Supplier.contact_title

SchemaChange8 Add the required column Supplier.contact_first_name

SchemaChange9 Add the required column Supplier.contact_last_name

SchemaChange10 Add the optional column Supplier.contact_other_names

We make the distinction that a *required* column is a column with no default value specified and where null values are not allowed, whereas an optional column either has a default value or null values are allowed.

The requirements change for UCL Coffee is obviously contrived, for illustrative purposes, but real-world software projects are often similarly affected. As software engineers, we still have problems eliciting requirements correctly, and even when we do get the requirements correct they are often changed during the lifetime of a project [Pfleeger, 1998]. As software projects increase in size, and as they become

more complex with more stakeholders involved, change becomes more and more probable. In almost all modern software development, we have to accept changing requirements as inevitable; in fact, modern software practices such as iterative and agile methodologies have been developed, in part, to cope with this very problem [Fowler and Highsmith, 2001]. Once we accept change as inevitable the problem we are left with is how best to deal with change.

UCL Coffee has two applications using a shared database and, due to changes in requirements, the developers would like to change the database schema. How can we deal with this situation using modern software engineering and database administration techniques, and what problems can arise? We shall discuss this in the remainder of this section.

2.2.1 The impacts of Schema Change

Database schema changes can affect the database itself and anything which uses the database. In this dissertation we are only concerned with the effects of change that affect applications which use the database. We shall use the term `impact` to refer to these effects upon applications; we define an impact as *any location in the application which will behave differently, or may be required to behave differently as a direct consequence of a schema change*.

The most obvious form of impacts are the locations in the application where runtime errors will occur. Our definition of an impact, limits the types of errors we focus on to runtime errors which are a direct consequence of a schema change, typically resulting in a runtime error being returned from the DBMS query execution engine, as we shall discuss. We note that application errors that occur elsewhere in the program, as an indirect consequence of a schema change, are not considered to be impacts because we are focusing only on the direct consequences of the change. We shall show some example impacts using our UCL Coffee applications, but first we must show the way in which the queries of the application are implemented.

We present these following queries, using standard SQL query language [Date, 1989]. We use the notation '?' in these queries to indicate a parameter that is supplied at runtime. Whilst this is only one possible way to implement these queries, we will discuss others in Section 2.3, we note that SQL is a very common way in which queries to relational databases can be represented. Each SQL query carries out one interpretation of the English language description of the queries introduced above.

SALES-Q1 SELECT * FROM Customer;

SALES-Q2 INSERT INTO Customer (name) VALUES (?);

SALES-Q3 UPDATE Customer SET name=? WHERE id=?;

SALES-Q4 DELETE FROM Customer WHERE id=?;

SALES-Q5 INSERT INTO Order (customer_id) VALUES (?);

SALES-Q6 INSERT INTO OrderProduct (order_id, product_id, units) VALUES (?, ?, ?);

SALES-Q7 SELECT * FROM Order WHERE customer_id = ?;

INV-Q1 SELECT id, name, supplier_id FROM Product WHERE id=?;

INV-Q2 SELECT id, contact_name, company_name FROM Supplier WHERE id=?;

INV-Q3 INSERT INTO Product (supplier_id, name) VALUES (?, ?);

INV-Q4 SELECT * FROM Product, Supplier WHERE Product.name LIKE ? OR Supplier.company_name LIKE ? OR Supplier.contact_name LIKE ?;

INV-Q5 INSERT INTO Supplier (contact_name, company_name) VALUES (?, ?);

These queries were written against the schema in Figure 2.1. If we apply the ten schema changes proposed above, and run these queries against the updated schema, the following runtime errors will occur[2]:

INV-Q1	err1	references invalid `Customer.name` column	SchemaChange1
INV-Q2	err2	references invalid `Supplier.contact_name` column	SchemaChange6
INV-Q4	err3	references invalid `Supplier.contact_name` column	SchemaChange6
INV-Q5	err4	references invalid `Supplier.contact_name` column	SchemaChange6
	err5	no value for required field `Supplier.contact_title`	SchemaChange7
	err6	no value for required field `Supplier.contact_firstname`	SchemaChange8
	err7	no value for required field `Supplier.contact_lastname`	SchemaChange9
SALES-Q2	err8	references invalid `Customer.name` column	SchemaChange1
	err9	no value for required field `Customer.title`	SchemaChange2
	err10	no value for required field `Customer.firstname`	SchemaChange3
	err11	no value for required field `Customer.lastname`	SchemaChange4
SALES-Q3	err12	references invalid `Customer.name` column	SchemaChange1

The places in the application where these errors will occur all fit our definition of impacts, because they behave differently as a result of the schema change, in this case the difference in behaviour causing a runtime error. We classify this type of impact as an error impact.

Runtime errors are obviously undesirable, so one of the more conservative UCL Coffee DBAs proposes an alternative set of changes that tries to avoid any runtime errors. The changes do not drop any columns and only add optional columns to tables. When executing an *INSERT* SQL query, such as `SALES-Q2`, the absence of values specified for required columns will cause a runtime error, whereas the absence of values for optional columns will not. Therefore, we could avoid runtime errors by substituting our initial changes for the following:

NonBreakingChange1 Add the optional column Customer.title

NonBreakingChange2 Add the optional column Customer.first_name

NonBreakingChange3 Add the optional column Customer.last_name

NonBreakingChange4 Add the optional column Customer.other_names

[2]The exact error messages produced depend upon the SQL engine being used, but the errors shown here are an example of errors that would be expected to occur.

NonBreakingChange5 Add the optional column Supplier.contact_title

NonBreakingChange6 Add the optional column Supplier.contact_first_name

NonBreakingChange7 Add the optional column Supplier.contact_last_name

NonBreakingChange8 Add the optional column Supplier.contact_other_names

Making these alternative changes instead would avoid runtime errors in the SQL queries, however, this does not mean that there would be no impacts. For example, whenever we insert a new record, such as in `SALES-Q2`, the application is behaving differently, *null* or default values are being inserted into the database that were not being inserted before. This may or may not be the desired behaviour, but it is clear that the application is behaving differently as a result of the schema change, which fits our definition of an impact even though it does not cause a runtime error. We classify this type of impact as a warning impact. An example of this kind of impact would be the impact caused by SchemaChange5 upon `SALES-Q2`, where the `other_names` field will be populated with a null or default value after the query has been executed.

A further type of warning impact is caused by new data being returned from a *SELECT* SQL query. For example, if we decide to use the non-breaking changes, `NonBreakingChange5` adds the `Supplier.contact_title` column, and this will not affect the validity of `INV-Q2`, which will therefore not behave differently as a result of the schema change. The application developer may wish to add the `Supplier.contact_title` field to the result set of this query, because the requirements of the application may require the use of this new information, meaning the query itself and the application logic need to be altered. This fits our definition of an impact, as although the query would not behave differently, it is required to behave differently following the change; the second clause of our impact definition specifies that impacts are also locations that are required to behave differently as the result of a schema change.

In summary, we have error impacts, which produce runtime errors, and warning impacts, which are changes which do not cause runtime errors but where the application behaves differently, or is required to behave differently as the result of a schema change.

2.2.2 The Difficulty of Schema Change

We argue that discovering and predicting impacts is particularly important both before and after the schema change is made.

Before a Schema Change is Made

The UCL Coffee company requires that the sales and inventory applications should be aware of its customer's and supplier's full names, including their titles. To achieve this, should the database development team make the breaking changes, *SchemaChange1-10*? If these changes have a large impact on the applications, then it may be more suitable to use the non-breaking changes *NonBreakingChange1-8*. But, as we have discussed, even these non-breaking changes have impacts, and it may be the case that using non-breaking changes does not justify the cost. The DBAs also have other possible ways in which they

could change the database, so they require access to good information about the possible impacts of change, in order to help inform the choice between these options.

For example, we may wish to decrease the amount of storage used by our database by removing unessential indexes from columns[3]. The impacts of this schema change could be very subtle, and unlikely to cause errors, however, the queries that use this column may now be less efficient, so alternative queries may be more suitable. If these queries are used often, it may be best to keep the index in place, whilst if these queries are not often used then it may be better to delete the index and accept the small loss of performance. This decision could drastically affect the performance of a system.

The decision to change a schema must be made using much more information than whether or not an error will occur. Schema changes have varied and important effects upon the functional and non-functional properties of a system, such as performance, security and maintainability. The changes must also be considered within the context of the project, and may be affected by urgent deadlines or budget constraints. With this in mind, the DBA must make decisions that are suitable at the time, based upon all the information available. Being able to estimate the impacts of schema changes upon applications is important in the schema change process because the impacts upon applications are inextricably linked to the various properties of the application and the development process.

The book "Database Refactoring" [Ambler and Sadalage, 2006] describes the current state of the art in industrial database administration methodologies; it describes the latest techniques designed to cope with schema change in highly iterative and agile software development. When it comes to the point of estimating the effects of a schema change this book describes how a fictitious DBA, Beverley, should proceed:

> The next thing that Beverley does is to assess the overall impact of the refactoring. To do this, Beverley should have an understanding of how the external program(s) are coupled to this part of the database. This is knowledge that Beverley has built up over time by working with the enterprise architects, operational database administrators, application developers and other DBAs. When Beverley is not sure of the impact she needs to make a decision at the time and go with her gut feeling or decide to advise the application developer to wait while she talks to the right people. Her goal is to ensure that she implements database refactorings that will succeed - if you are going to need to update, test and redeploy 50 other applications to support this refactoring, it may not be viable for her to continue. Even when there is only one application accessing the database, it may be so highly coupled to the portion of the schema that you want to change that the database refactoring simply is not worth it. [Ambler and Sadalage, 2006]

In this scenario, Beverley assessed the cost of the schema changes, but then went on to make the schema changes even though the changes affected many applications. The changes were made because the estimated benefits of the changes outweighed the estimated costs.

[3] indexing is a optimisation that uses extra storage to create an index of a column, with the advantage of providing faster lookups of values in this column.

If the UCL Coffee database team proceeded in the same way, they would also have to assess the changes based on their experience and knowledge. The DBAs would consult the developers of the sales and inventory applications. The developers would have to estimate the effects of the changes, and estimate the cost of reconciling the impacts. To get these estimates they would rely on their knowledge of the applications and expert judgement. This could also involve using manual code inspection to investigate the source code that could be affected, to get a better estimate of the cost of reconciling the potential impacts. The DBAs would then use this information, and their discretion to decide upon the most suitable changes. It is clear that this process relies heavily on subjective expert judgement, but is typical of modern industrial practice.

We shall not discuss the process of how the DBA estimates cost of change upon the rest of the database, because it has been studied by the schema evolution community, and many modern DBMSs have tool support to manage changes within the boundary of the database itself. We shall discuss this in detail in Section 6.3.

After a Schema Change is Made

After a schema change has been made, the developers of any affected applications need to reconcile the application with the changed schema. The standard software engineering approaches for finding these impacts are manual code inspection or regression testing [Pfleeger, 1998].

Manual code-inspection involves a developer examining the code, by hand, or by using standard tools such as source code editors and integrated development environments (IDEs). Regression testing is a way of testing the application in response to a change, to make sure it still exhibits the required behaviour. These kind of tests are usually automated.

We argue that, although very valuable for other purposes, regression testing is not suitable for change impact analysis. If test coverage is sufficiently high, running a regression test suite against the changed schema will find impacts that cause runtime errors. In order to find the remaining warning impacts, the test suite must be updated to reflect the new desired behaviour of the program. We are now left with the problem of how to update the regression test suite. If we view the test suite as part of the application, a required change to the test suite is simply another impact, and finding impacts in the test suite is just as problematic as finding impacts in the main application. This makes regression testing unsuitable for impact analysis. This does not diminish the usefulness of regression testing in any way, however, we simply note that it is not adequate for the purposes of impact analysis, and can be considered orthogonal. We shall discuss how specific regression testing approaches apply to our research in more detail in Section 6.3.

We are inevitably left with the choice of using expert judgement and manual code inspections to reconcile the applications with the new schema.

In the case of UCL Coffee sales and inventory applications, the developers will update their tests, using their experience and judgement and manual code inspection to find any tests that need to be changed, and to write any new tests that are needed. They will then modify the application as required to make these tests pass. Different software engineering methodologies might perform these steps in different

orders, but the developers at some point would need to assess the impacts of the changes using manual code inspection. The cost of reconciling each different impact will vary. We define the the cost of reconciling an impact as the cost of locating the impact and altering the application to function correctly with the new schema.

The Need for an Automated Analysis

We have shown that before and after a schema change is made, we make estimations about impacts using expert judgement, and by using manual code inspection. Assessing the effect of changes in this way, has been shown to be a fragile and difficult process, as noted by Law and Rothermel [2003], who show that expert predictions can be shown to be frequently incorrect [Lindvall M., 1998], and impact analysis by-hand, using code inspections, can be prohibitively expensive [Pfleeger, 1998]. For these reasons, we argue that these current methods are, in many cases, inadequate, especially as applications become larger and more complex, and changes become more frequent.

Whilst the impacts must be reconciled, once they have been identified, the process of making the required changes falls into more well-established software engineering territory. The real problem we have identified here is, that estimating and predicting impacts in an accurate and cost-effective way is difficult.

We wish to improve this process, which is currently conducted by using manual code inspection and using expert judgement. Can we provide a solution that is more cost-effective? The alternative to a manual process is an automated process, and if such a process is feasible, automation has the potential to solve these problems. Our thesis is that an automated impact analysis is feasible, and can provide beneficial information to the stakeholders of the schema change process.

2.3 The Requirements of an Automated Impact Analysis

An automated schema change impact analysis has many technical challenges to overcome. In this section we discuss some of the functional requirements that make this an interesting research problem, before discussing the trade-offs that can be taken when satisfying these requirements. We end this section by discussing what it means for such an analysis to be considered ‘useful’, and how the trade-offs could be balanced to provide a practically useful analysis.

2.3.1 Finding Schema Dependent Code

In Chapter 1 we discussed the scope of our work, limiting it to object oriented programs that use relational databases. The are many ways in which an object oriented program can interact with a database, and a variety of data access practices must be considered by our analysis. In this section we shall discuss our requirements for analysing applications, and how common data access practices are potentially problematic for an automated impact analysis.

Ho do we find impacts in an application? So far we have only discussed, and defined, how impacts relate to individual database queries. Given that we can find the impacts within queries, then in order to find the impacts in a given application, we need to know all queries an application can produce. The problem that we face is that queries can be defined, queries can be executed and query results can be

used, all in different parts of the program.

To provide an estimate of the cost of an impact, we need to examine the application to see how it currently operates, estimate what needs to be altered, estimate the cost of making this alteration and estimate the new behaviour that will exhibited by the application. The impacts could affect locations where queries are defined, where queries are executed or where query results are used, therefore when we analyse an application we need to find all of these locations.

An example of where we need to examine the definition of a query, is the impact of `SchemaChange10` on `INV-Q5`. `SchemaChange10` adds an optional column to the `Suppliers` table, whilst `INV-Q5` is inserting new records into this table. The impact will be a warning impact, because the query does not specify this new data, and as a result the behaviour of the query will be changed, inserting default or null data. The application's requirements will tell us whether or not this behaviour is required or not, and if it is not required, the location we need to alter is the query definition. It is intuitively obvious that the most likely place to require alteration is the location(s) where the query is defined, therefore by being able to identify these locations quickly and easily, developers could save a lot of time compared to searching for these definition locations manually.

An example of where we need to examine a query execution location, is the impact of `SchemaChange9` on `INV-Q5`. Whilst `SchemaChange10` created an optional column, `SchemaChange9` creates a required column. This impact will cause a runtime error, and for the same reasons it would be helpful to track down the definition of the query it would also be beneficial to track down its execution point. To assess the cost of change, the developers may need to know how the execution site currently operates, if it could cope with a runtime-error or if it requires changes to the way in which the query is executed.

An example of where we need to examine the use of the returned query results locations, is the impact of `SchemaChange7` upon `INV-Q2`. `INV-Q2` returns information about suppliers, but `SchemaChange7` has added new information about suppliers to the schema. This change requires that the inventory application must now return this new information to the user, and where we are using the result of `INV-Q2` we must alter the program to process and display this new information. In order to assess the costs associated with this alteration we need to examine the locations in the code where the results of `INV-Q2` are currently used and estimate how they need to be altered to process this new information.

The three example changes described above, have shown us that to assess the costs associated with an impact, we may need to examine the locations where a query is defined, where it is executed and where its results are used. The developers and DBAs would use this information before the schema change is made to estimate the cost and effects of changing the query, and after the change it would help them find the exact locations that require changes.

This observation leads us to define our first requirement for an automated impact analysis:

Requirement-1 for each query in the application, we should identify the locations where the query is defined, where the query is executed and where the results of the query are used.

Now that we have defined this requirement, the following sub sections will discuss common data access practices, why they are important to consider, and how they affect impact analysis. We shall define a requirement for each type of data access practice that we require our analysis to be able to analyse.

To help illustrate these requirements we present code examples from an implementation of the UCL Coffee inventory and sales applications. The full code listing for these applications can be found in Appendix D. The applications are written using C# using a SQL Server 2005 database and whilst they represent only one possible implementation, they have been purposefully constructed to include examples of several important features for analysis, each of which we shall discuss in more detail next.

Static and Dynamic Queries

The simplest way that an application can execute a query against a database, is to create a representation of the query, usually using a string data type, and then, using an interface API such as ODBC or OLEDB, send that query to be executed by the DBMS query engine. This is known as a call level interface (CLI) and simply involves creating a query and sending it directly to the DBMS for execution.

An example of this can be seen in an implementation of `INV-Q1`:

```
1 string cmdText = "SELECT * FROM Product WHERE id=@ID;";
2 SqlCommand command = new SqlCommand(cmdText, conn);
3 command.Parameters.Add(new SqlParameter("@ID", id));
4 SqlDataReader reader = command.ExecuteReader(); // INV-Q1 executed
5 reader.Read();
6 result = Load(reader);
7 return result;
```

Listing 2.1: Implementation of `INV-Q1`

On Line 1 of this example we can see the query defined as a string variable `cmdText`, and passed as an argument to the constructor of the `SqlCommand` object. This command object represents a command that will be sent to the database and executed. It is executed on Line 4, and the results of the query are passed to the `Load` method, on Line 6 where they will be processed. The `SqlCommand` and `SqlDataReader` objects are part of the ADO.NET libraries, which are the standard libraries used by C# programs for database access. Almost all modern programming languages and environments include some form of data access libraries, similar to ADO.NET, for accessing relational databases. The simplest, and most common, of these libraries usually involve CLI based APIs that use string data types to specify SQL queries. Java has the JDBC libraries [Hamilton et al., 1997] and similar libraries exist for many other programming environments.

Creating queries using strings is simple yet powerful, because string data types are easy to define and manipulate. It is therefore very easy to create program logic that dynamically creates strings, which are then used as queries. This technique can be used to easily build and execute complex queries.

A common example of how dynamically generated queries are used, are queries which perform complex searches such as keyword based searching. The inventory application has a query that searches

products and suppliers by keyword, INV-Q4, that can be implemented as follows:

```
public static ICollection<Product> Find(string[] supplierKeywords,
    string[] contactKeywords, string[] productKeywords)
{
    string sql = "SELECT * FROM Product";

    if (supplierKeywords.Length > 0 || contactKeywords.Length > 0)
        sql += ", Supplier";

    sql += " WHERE ";

    using (DB db = new DB())
    {
        SqlCommand cmd = db.Prepare();

        sql += "(";
        sql = AddKeyWordClause(cmd, supplierKeywords, sql,
            "Supplier.company_name", "supplier");
        sql += " OR ";
        sql = AddKeyWordClause(cmd, contactKeywords, sql,
            "Supplier.contact_name", "contact");
        sql += " OR ";
        sql = AddKeyWordClause(cmd, productKeywords, sql,
            "Product.name", "product");
        sql += ") AND Product.supplier_id = Supplier.id;";
        cmd.CommandText = sql;

        System.Diagnostics.Debug.WriteLine("SQL: " + sql);

        SqlDataReader reader = cmd.ExecuteReader();

        List<Product> products = new List<Product>();
        while (reader.Read())
        {
            products.Add(Load(reader));
        }
        return products;
    }
}

private static string AddKeyWordClause(SqlCommand cmd, string[] keywords,
    string sql, string fieldName, string paramId)
```

```
42 {
43     for (int i = 1; i <= keywords.Length; i++)
44     {
45         if (i == 1)
46             sql += "("; // first loop
47 
48         string paramName = String.Concat("@", paramId.ToString(), "Key",
49              i.ToString());
50         sql += String.Concat(fieldName, " LIKE ", paramName);
51 
52         if (i != keywords.Length)
53             sql += " OR ";
54         else
55             sql += ")"; // last loop
56 
57         string valWithWildcards =
58             String.Concat("%", keywords[i - 1], "%");
59         cmd.Parameters.AddWithValue(paramName, valWithWildcards);
60     }
61     return sql;
62 }
```

Listing 2.2: Implementation of `INV-Q4`

Here we see the SQL string being modified extensively before being executed, even using the auxiliary method `AddKeyWordClause` to alter the string. It starts off as a simple *SELECT* SQL query, but its *WHERE* clause is built up dynamically, every keyword specified adds another item to the *WHERE* clause, checking for columns that might match that particular value.

There are advantages and disadvantages to using highly dynamic queries, but because of their ease of use and power, they are commonly used, as shown by their use in design patterns and recommended practices [Fowler, 2003, Microsoft Patterns, Java Patterns]. Therefore, our automated analysis should be able to process dynamic and static queries, but as we shall see, in Chapter 3, the way in which queries can be represented as variables and then manipulated dynamically, is the main challenge to creating an automated analysis, as it makes tracking the queries difficult.

The application logic that creates these queries adds complexity that makes analysis by hand more difficult. As these queries become more complex, the queries become hidden behind the application logic. Because dynamic queries are common, we require that our automated impact analysis must be able to analyse the values of query representing types, even if the queries are generated dynamically.

Requirement-2 The impact analysis should be able to analyse queries that are composed statically and dynamically.

DBMS Features

Modern DBMSs often have many different features such as stored procedures, views, triggers, indexes and constraints, that may affect our impact analysis. We shall show an example of how UCL Coffee uses some of these features, and how it adds requirements for our change impact analysis.

The inventory and sales applications were designed and developed by separate teams who chose to use different data access strategies. The inventory application, as we have seen, uses string based SQL queries. The architect of the sales application decided that for reasons of performance, security and maintainability, it would be better if all queries were composed using stored procedures.

A *stored procedure* is a query that is stored by the database and can be called by name, it can consist of many sub queries, and can be called with variable parameters, much like calling a function or method in a programming language. Stored procedures, for a number of reasons, are very popular and are used in many recommended data access practices [Fowler, 2003].

In the sales application all data-access is through stored procedures. The following example shows the implementation of `SALES-Q2`:

```
1 OleDbCommand cmd = new OleDbCommand("insCustomer", conn);
2 cmd.Parameters.Add(
3     new System.Data.OleDb.OleDbParameter("@name", customerName));
4 cmd.CommandType = CommandType.StoredProcedure;
5 return (decimal)cmd.ExecuteScalar(); // SALES-Q2 executed
```

Listing 2.3: Implementation of SALES-Q2

Here we see the construction of an *OleDbCommand* object on Line 1. The object represents a command that will be sent to the database for execution, just like the *SqlCommand* object we saw earlier, however, in this case the object is created with the name of the stored procedure that should be executed, rather than a SQL query. The command is executed on Line 5. where the result of the execution is used as the return value.

The stored procedure itself is defined as follows:

```
1 CREATE PROCEDURE [dbo].[insCustomer]
2         @name varchar(255)
3 AS
4 BEGIN
5         INSERT INTO Customer ([name]) VALUES (@name);
6         SELECT @@IDENTITY;
7 END
```

Listing 2.4: SALES-Q2 stored procedure

This stored procedure uses T-SQL(the variant of SQL used by Microsoft SQL Server) to define a stored

procedure called `insCustomer`. The name of the customer is the single input parameter. The body of the stored procedure inserts a new customer with the specified name, then returns the primary key of the newly inserted record, in this case the value of the newly inserted `id` column.

An advantage of using stored procedures, is that the logic of the query itself is now under the control of the DBMS, and large scale DBMSs often have software that can predict the impact of schema changes upon stored procedures. As we are only interested in impacts upon applications, we ignore the effects of schema changes upon stored procedures in this dissertation, however, changing a stored procedure can also have impacts upon the application.

Reconciling an application with a changed stored procedure, is the same as reconciling the application against a changed table, therefore, for the purposes of this dissertation, we consider stored procedures to be part of the schema, and our analysis must be capable of analysing the impacts that changing them will cause. Equally, there are other DBMS features that can be treated in the same way, such as views. The same problems apply when predicting the effects of impact by hand, so our automated analysis should include support for these kinds of DBMS features, as they cause similar problems but are just as significant as changes to other parts of the schema. As illustrated by our example application, these kinds of features may be used extensively, and so it is important that they are considered by our analysis.

Requirement-3 The impact analysis should be able to predict impacts caused by changes to tables, columns, stored procedures and any other DBMS feature (vendor specific or otherwise) that may have a direct impact upon the application.

Query Representing Data Types

The example code for executing `INV-Q1`, in Listing 2.1, showed the query being generated as a string and passed as a parameter in the constructor of an *SqlCommand* object. Our analysis could simply analyse string data types and look for all definition locations like this, noting the value of strings where they are used as queries. Requirement-1, however, requires that we do more than this because we also want to know where this query is executed and where its results are used. To achieve this, our analysis needs to be able to analyse the uses of *SqlCommand* objects so that it can associate the definition of a query at Line 2 with an execution on Line 4, and then we must also be able to analyse how the *SqlDataReader* object is used within the *Load* method on Line 6, to find out how and where the results of the query are used. The *SqlCommand* and *SqlDataReader* objects are not strings, but are part of the data access libraries that we must be able to process. They are not the only other objects we must consider, there are many other data access library objects that we may need to analyse, in fact, there are too many to specify, and because new libraries are created all the time, our analysis must be flexible and extensible so that it can include any required data access libraries.

As a further example, Listing 2.1 uses the *SqlCommand* type to define a command, whereas Listing 2.4 use the equivalent *OleDbCommand* type. The function of these types is the same, but their implementations use different protocols when communicating with the database[4].

[4]One using a SQL server proprietary protocol, the other using the OLEDB protocol.

These data access libraries are not always CLIs, they could be other persistence technologies, such as object relational mappings (ORMs) like Hibernate [King and Bauer, 2004]. ORMs are libraries that use some sort of implicit or explicit mapping between the database and the application, to automate many data access tasks. ORMs often have much more complex libraries for interacting with the databases, commonly using their own proprietary query languages. They can be treated as just another persistence library, with an interface that is different from the CLI style interface.

In Listing 2.2 we see how the *StringBuilder* is used to create dynamic query strings. This is a common technique as the *StringBuilder* class is more efficient at concatenation than the standard string class[5]. Our impact analysis should be able to analyse the use of the *StringBuilder* and predict the value of the query it produces, because this class is so commonly used in creating dynamic queries. Just as we need to analyse the *StringBuilder* there are other types from base libraries, and other libraries, that we may wish to make our analysis aware of. It should be possible to include in our analysis, any types that can be involved in the definition, execution, or use of queries.

Analysing the impacts of schema change can be made more difficult by the different query libraries, persistence technologies and base libraries that are used. Each different type that must be analysed adds a layer of complexity, and whilst some of these technologies aim to make some forms of change easier, they can make others, like estimating the impact of schema change, more difficult. Because these types and libraries are so important, and so commonly used, we require that our analysis be able to analyse them.

Requirement-4 The impact analysis should be able to, extensible, analyse types which can be used to represent queries or query results, where these types are required to accurately estimate impacts.

Architectural Patterns

The UCL Coffee inventory and sales applications were designed and developed by separate teams who chose to use different data access strategies, and correspondingly, they also chose to use different application architectures.

The sales application uses the Table Data Gateway design pattern from Fowler's Patterns of Enterprise Application Architecture [Fowler, 2003]. We shall not describe this pattern in detail here, for the sake of brevity, but we shall highlight the consequences of using such design patterns. By using this pattern in our example application, the queries are defined, executed and the results are used, all in different methods, as shown by our implementation of `SALES-Q1` in Listing 2.5.

The query is initially defined as a literal string on Line 3. The call to *db.Prepare* calls the method shown on Line 9. This method creates a new object that wraps the query in a command and returns it. The command is executed on Line 4. The results of the query are then returned, to be used as appropriate by the initial caller of the *FindAll* method.

As we have discussed, knowing the locations where queries are defined, where queries are executed and where their results are used, is very important for our analysis. The use of design patterns, such as the Table Data Gateway pattern we use in the Sales application and the Active Record pattern we

[5] Equivalents exist in other languages such as Java(*java.lang.StringBuffer*)

```
1  public IDataReader FindAll()
2  {
3      var cmd = db.Prepare("selCustomersAll");
4      return cmd.ExecuteReader(); // SALES–Q1 executed
5  }
6
7  ...
8
9  public OleDbCommand Prepare(string cmdText)
10 {
11     OleDbCommand command = new OleDbCommand(cmdText, conn);
12     return command;
13 }
```

Listing 2.5: SALES-Q1 implementation

use in the Inventory application, often separate the query definitions from their executions, and from their use of the query results. When assessing the impact of change using manual code-inspection, this adds further cost and chance of error, as these relationships must be traced by-hand, especially in large applications. This separation may be simple to analyse by hand in the code shown in Listing 2.5, but in practice, definitions of queries and uses of results can be separated from query executions by several layers of different methods, and tracing these by hand, even using modern IDEs can be error-prone and difficult, especially if it must be done repeatedly for many queries. This problem is a very important consideration, and results in our final data-access requirement.

Requirement-5 The impact analysis should be able to establish relationships between query definitions, query executions and the use of query results, even when design patterns separate these locations across method boundaries.

Summary of Data Access Practices

We require our analysis to cope with each of the data access practices described above, defined as Requirement2-5, because each one is in widespread use, and excluding any of these practices would greatly limit the applications to which our analysis could be applied. Although we have no empirical evidence to support this claim, recommended architectural patterns and practice in several prominent works circumstantially indicate that such practices are in widespread use [Fowler, 2003, Microsoft Patterns, Java Patterns]. By making our impact analysis satisfy these requirements, we make sure our analysis is applicable to a broad range of applications.

The data access practices we cover with Requirement2-5 are not exhaustive, but we argue that these practices represent the majority of data access practice in modern applications, although, again, we can only use the descriptions of recommended practice to justify this claim. Moreover, we expect that different data access practices that are not covered by our requirements, will be similar to the practices

we do cover, and that extending our analysis to new data access practices will be possible. We discuss this in the Section 8.2.

2.3.2 Trade-offs for Schema Change Impact Analysis

The requirements, specified in the previous section, simply define the applicability of our analysis, making sure it can be applied to many types of application. Whilst this is important, it does not achieve our goal of creating an analysis that is more useful than the current practice of impact analysis via manual code-inspection. An analysis satisfying all these applicability requirements could be created, but how do we define if our analysis is better or worse than prediction using manual code inspection and expert judgement? How do we know if our analysis is useful?

There is no single criterion by which to measure usefulness, it must be measured using using several different criteria. We shall now describe the criteria by which we shall judge usefulness, and discuss the trade-offs involved in satisfying them.

The usefulness of the information returned by a change impact analysis can be measured by the following criteria:

1. Accuracy
2. Impact-Precision
3. Cost

Accuracy

Accuracy is the measure of how correct and free from error the analysis is. Typically accuracy is measured as a percentage of correct predictions against incorrect predictions; we define the categories of correct/incorrect predictions as follows:

Any predicted impact that requires a change in the application is a *true positive*. Any predicted impact that requires no corresponding change in the application is a *false positive*. Conversely, a *false negative* is a change in the application which has no corresponding predicted impact. A *true negative* is the correct prediction of no change in the application.

The standard measures for accuracy of a binary classification are precision and recall. Precision is defined as:

$$\text{precision} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false positives}}$$

Precision is a measure of fidelity and is important for impact analysis as it gives us a measure of the level of false positive that are occuring. Every false positives the user has to examine could potentially detract from the usefulness of the analysis, adding extra cost to the schema change process.

Recall is defined as follows:

$$\text{recall} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of true negatives}}$$

Recall is a measure of completness and is important for impact analysis as it gives us a measure of the level of false negatives that are occuring. Every false negative means that our analysis has failed to predict an impact, which would potentially decrease the users confidence in our tool and decrease its usefulness.

We define a high level accuracy as being high levels of recall and precision.

Impact-Precision

Impact-precision defines the level of granularity with which an impact can be identified. For example we could identify impacts at the level of affected modules, classes, methods, lines of code or individual program statements. The highest impact-precision analysis would pinpoint the exact locations involved in a impact, down to the affected statements. A very low impact-precision analysis could identify the entire program as simply being affected or not. Likewise, a very high impact-precision analysis would give an estimation of queries with large amounts of the query text intact and with few unknown values or approximations, whereas low impact-precision analysis may over approximate queries as unknown values. For example the query $SELECT\ id\ FROM\ table1;$ can be predicted with higher impact-precision by the regular expression "$SELECT\ .^*\ FROM\ .^*$", and lower impact-precision by ".*".

Related work for impact analysis of object-oriented database schema change, shows that identifying impacts at the finer granularity can reduce the time needed to conduct schema changes and reduce the number of errors [Karahasanovic and Sjoberg, 2001]. Accuracy and impact-precision are closely related, and they both need to be of a sufficient level to allow the stakeholders to identify, and deal with, impacts.

Cost

Cost is the processing cost and memory costs of executing the analysis. These costs affect the time in which the analysis can be executed, and the resources which are required to make it run. Scalability is a measure of how these costs vary as the size of the application and the size of the database schema increases. It is important that the analysis is executable in a reasonable time, on reasonable hardware, and scales to programs of significant size; of the size that are typically found in commercial enterprise software development. If the analysis becomes too expensive, it will become unfeasible for large programs, or the results may take too long to produce on affordable hardware, and the analysis will not be useful in real world development.

2.3.3 Conservative Analysis

A *conservative* analysis will always predict all true positives and has no false negatives, and therefore will not miss any impacts. However, conservative analyses can be inaccurate, in that they have a high number of false positives. While many program analyses are required to be conservative, as the cost of false negatives is very high, we do not require our analysis to be conservative. The cost of a false negative in our analysis is still high, and we do not wish to miss any potential impacts, but the requirement for a conservative analysis could make our analysis too costly, too inaccurate or have levels of impact-precision that are too low to be useful. For an analysis to be conservative it must consider all possible cases, even ones which are very unlikely. A non-conservative analysis has the option of ignoring un-

likely cases, potentially making conservative analyses more expensive than non-conservative analyses. Considering these extra cases can also make conservative analyses less accurate because they can produce more false positives from the extra considered cases. The extra cases could also decrease the level of impact-precision of the analysis, resulting in predicted queries which are more general, showing the entire possible range of queries, rather than the queries which are likely.

We must also consider that the cost of producing a conservative analysis could be very high, and therefore not feasible to implement within the context of our research. Guaranteeing that the analysis is conservative typically requires a formal description of the analysed language, which can be related to formal descriptions of the analysis, and proved to be conservative. This is an expensive and difficult process, and is too expensive to be considered with in the scope of our research, as we shall discuss further in Section 7.1.

Instead of a conservative analysis, we would like our analysis to be as accurate as possible, whilst still being scalable enough to analyse large, real-world size programs, and with impact-precision levels high enough to be useful. Allowing a small number of false negatives can be acceptable, if it allows an accurate, high impact-precision and scalable analysis to be developed, that otherwise would be impossible. We discuss what can be an acceptable level of false negatives, and whether a useful analysis can be made that is not conservative, in Section 7.4. An important consideration is that we are aiming to provide an *estimation* tool, for estimating the impacts of change, and if the presence of false negatives does not sufficiently alter the accuracy of the estimates, the analysis could still be useful. However, even very accurate or conservative results may not be useful if the impact-precision is too low, or the cost too high. We therefore relax the constraint of requiring a conservative analysis, so that we can more freely explore the accuracy, impact-precision and cost trade-offs available.

2.3.4 Trade-offs and Usefulness

Ideally we want an analysis that is very accurate, with high impact-precision, with very low cost and with high scalability. This is, of course, not possible; increasing accuracy and impact-precision will usually increase the cost of the analysis, and conversely reducing the cost and improving scalability can require a reduction in accuracy and impact-precision. These trade-offs do not always hold true, and in Chapter 5 we discuss ways of maintaining accuracy and impact-precision whilst reducing cost by creating an efficient analysis. There comes a point, however, at which we cannot increase accuracy or impact-precision without increasing cost and vice versa.

These trade-offs should be made to create a useful analysis, but as we have discussed, usefulness is a subjective term. We therefore define a useful analysis as: *an analysis which has the accuracy, impact-precision and cost, suitable to inform schema change in typical commercial enterprise application development, with benefit to the stakeholders*. This is still a subjective measure, but it provides us with a clear goal, and one by which we can start to judge the usefulness of our analysis.

2.4 Summary

In this chapter we have introduced the UCL Coffee Company example, and used it to show the effects that schema changes can have upon applications. We discussed how current software engineering practice deals with these effects, and showed that it typically relies on expert judgement and manual code inspection, which is both prohibitively expensive and error prone. This motivates the need for an automated change impact analysis.

We discussed the requirements for making an automated analysis widely applicable, by examining the data access practices that it should be capable of analysing. We then defined the criteria of accuracy, impact-precision and cost and the implicit trade-offs that exist between them. We showed that by using these criteria we can judge the usefulness of our analysis.

The effects of schema change upon applications are difficult to deal with, and in the context of real-world software development, there are many facets to making this a difficult problem. The research described in this dissertation does not solve this problem in its entirety; instead we provide the stakeholders with a better way of estimating the impacts of schema change. We hope that this will give developers and DBAs access to better quality information than is currently available, and that this will allow them to make more informed decisions. This will not transform the database schema change process into an easy process, but has the potential to make the process more disciplined and better informed.

Chapter 3

Query Analysis

In Chapter 2 we motivated the need for an automated schema change impact analysis. To achieve this, we need to analyse an application to establish how it interacts with the database, and examine the queries that the application might produce. This analysis should establish where these queries are defined, where they are executed and where the results of the queries are used, as defined by *Requirement-1* discussed in Chapter 2. We define the term *query analysis* to mean, an analysis which can extract query definitions, query executions and the use of query results from an application. A query analysis can be used as independently of the analyses we describe in the remainder of this dissertaion. The input to the query analysis is the target application. The output of the query analysis is a prediction about runtime behaviour of the application, predicting possible queries and where these queries are defined, where they are executed and where their results are used.

In this chapter, we shall describe a novel query analysis that is useful for impact analysis. We shall start by discussing some background on program analysis, and related analyses, before defining the requirements specific to our problem, and showing how we create a query analysis that satisfies these new requirements and the requirements previously defined in Chapter 2. We shall introduce the closest related work, and show how this work is insufficient for our needs. We shall then show how this related work can be extended, in terms of impact-precision and functionality, to create a novel query analysis which satisfies all of our requirements. In Chapter 4, we shall show how the results of a query analysis can be used to predict the impacts of schema change.

3.1 Background

We require an estimation of the possible queries that an application can produce, and to do this we need to predict the runtime behaviour of the application. There are two fundamentally different approaches to doing this. First, we can analyse the program statically, at compile-time, to predict what it will do at runtime, this is static analysis. Second, we can observe the actual runtime behaviour of the program as it is executed against some representative set of inputs, this is dynamic analysis. Both static and dynamic analyses have their advantages and disadvantages. Static analyses are good for analysing programs for all possible executions and can have high recall, but can suffer from prohibitively high costs or low precision [Nielson et al., 1999]. Dynamic analyses can be very precise and relatively low cost, but can

suffer from problems with low levels of recall if inputs do not exercise all possible behaviours of the application [Orso et al., 2004a].

In this dissertation we have chosen to investigate only static analysis, primarily for its advantages in accuracy, although static analysis is not prescriptive. Dynamic analysis techniques could also be used and would involve an alternative set of techniques and different trade-offs, but it is outside the scope of this research to investigate both approaches. We discuss the possibilities for dynamic analysis in the related work Section 3.5, and for the remainder of this chapter we shall be concerned only with static analyses.

The particular type of static analysis we shall use here is dataflow analysis. Whilst many types of static program analysis exist, we shall discuss our reasons for choosing dataflow analysis, and shall discuss some alternative approaches we could have used, in Section 3.5.

3.1.1 Dataflow Analysis

Dataflow analysis is a type of static program analysis that uses the control flow graph (CFG) of the program. Each node in the CFG represents an instruction in the program, with the incoming and outgoing edges representing the possible execution paths that the program could take. The data in the analysis are represented as *flow states*; sets of values that represent some properties of interest. Dataflow analysis propagates flow states around the CFG, with each node altering the incoming state to produce an outgoing state.

For example, consider a dataflow analysis to predict the runtime value of integer variables, applied to the code in Listing 3.1. We shall call this *integer analysis*.

```
1 int x = 1;
2 while(conditionalVariable)
3 {
4   x = x + 1;
5 }
6 Console.WriteLine(x.ToString());
```

Listing 3.1: Dataflow example

The control flow graph for Listing 3.1 is shown in Figure 3.1, showing the possible execution paths that the program can take. For an integer analysis, we shall estimate the runtime values of integer variables at each node in the flow graph, and propagate the resulting flow state along the edges in the call graph. The flow state of the integer analysis would be a mapping of variable identifiers to integer values, $\mathbf{VarId} \rightarrow \mathcal{P}\{\mathbf{Z}\}$. Each node will modify the incoming flow states as appropriate to calculate the outgoing flow state. Nodes that define integer variables, like Node 1, will add new variables to the flow state with a corresponding initial value. Nodes that redefine integer values such as Node 3 will alter the value in the mapping as appropriate, accounting for any integer arithmetic that has occurred. We shall see examples of how we can specify this analysis more formally in the remainder of this chapter.

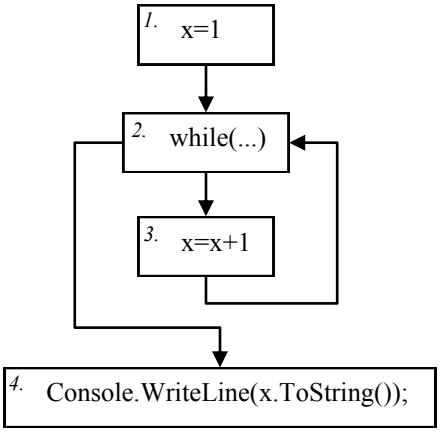

Figure 3.1: Dataflow example control-flow graph

The input flow state for Node 1 would be empty, representing the program starting in an empty state. The output for Node 1 would be:

$$x \rightarrow \{1\}$$

When this data gets propagated to the input of Node 3, the output of Node 3 could become:

$$x \rightarrow \{1, 2\}$$

The inputs are merged together at Node 2, so that the sets mapped to by variable identifiers are unioned together. The eventual output of this is that the input to Node 3. will be:

$$x \rightarrow \mathbb{Z}$$

This shows that the variable x could have the value of any positive integer, represented by the set $\mathbb{Z}$. The result of an integer analysis of Figure 3.1 could be:

Node No.	Incoming	Outgoing
1	$\{\}$	$\{x \rightarrow 1\}$
2	$\{x \rightarrow \mathbb{Z}\}$	$\{x \rightarrow \mathbb{Z}\}$
3	$\{x \rightarrow \mathbb{Z}\}$	$\{x \rightarrow \mathbb{Z}\}$
4	$\{x \rightarrow \mathbb{Z}\}$	$\{x \rightarrow \mathbb{Z}\}$

This shows that, for nodes 2,3 and 4 the x variable is a member of the positive integers, whilst in Node 1, the only possible value is 1. This is a prediction of the possible runtime values that may occur, and in a real execution, the actual values would be dependent upon how the *conditionalVariable* variable, on Line 2, is defined and modified, which we have not included in Figure 3.1. It can very difficult, or even undecidable [Nielson et al., 1999], to predict the value of all conditional expressions in a program, therefore, for most analyses, it is assumed that the conditional paths are feasible, and dataflow analyses

over-approximate the flow state accordingly. Therefore the actual runtime values are typically a subset of the values predicted by the program analysis.

We have discussed that dataflow analysis can predict the runtime properties of a program, but we have not discussed how such an analysis can be defined and executed. To define the analysis, each type of node in the CFG is associated with a *transfer function* that calculates the outgoing state from the incoming state. A solution for the analysis can be found by representing the program and transfer functions as a set of *dataflow equations*. Solving these equations can be achieved by propagating flow state around the CFG in an iterative way, which can be thought of as simulating the execution of the program. The solution to a set of dataflow equations, is a point where the incoming and outgoing states for each node do not change when the transfer functions are evaluated, this is known as a fixed point [Nielson et al., 1999].

There are several ways to calculate a fixed point solution for a dataflow analysis, some resulting in many solutions, but if not correctly specified, an analysis cannot be guaranteed to have a solution at all. Likewise, when the program contains loops or recursion, the analysis algorithms may not terminate. It is important to make sure that the analysis has a solution, and that the solution to the analysis can be found. It can be shown that this is the case by proving properties about the value domain of the flow states.

A monotone framework is a mathematical framework for specifying and reasoning about dataflow analyses [Nielson et al., 1999]. A monotone framework lets us specify the essential details of the analysis, such as the transfer functions, in a uniform way, thus allowing us to calculate a fixed point solution using one of the generic algorithms that can be applied to instances of a monotone framework. If we provide transfer functions and a definition of the flow state value domain, which satisfy mathematical properties such as monotonicity and the ascending chain condition, then we can also guarantee that the algorithm will terminate, and that there exists a solution.

We shall not describe monotone frameworks in detail here, and instead we refer interested readers to [Nielson et al., 1999]. Instead we shall describe an analysis that can be built using dataflow analysis and monotone frameworks, and how this analysis can be used as part of our automated impact analysis.

3.1.2 Widening-based String Analysis

String analysis is a form of program analysis where the runtime values of string variables are predicted. String analysis by Choi et al. [2006], is the most suitable analysis for our purposes, as we shall discuss in this section, before describing why and how we extend it in the following sections. We discuss alternative string analyses in the related work Section 3.5.

The string analysis proposed by Choi et al. [2006] uses the standard monotone framework for abstract interpretation [Cousot, 1996]. As such we shall describe the details required to specify the dataflow analysis here, and the analysis can be implemented using the standard monotone framework; we shall specify the transfer functions for the analysis, and the operations for partial ordering and joining. A standard algorithm applicable to monotone frameworks can then be used to find a fixed point solution for the dataflow equations that result from these functions for a given program.

Collecting domain:

$\mathbf{Var}$	x	
$\mathbf{Char}$	c	
$\mathbf{Str}$	s	$\in \{c_1c_2\cdots c_n \mid n \geq 0, c_i \in \mathbf{Char}\}$
$\mathbf{State^{col}}$	S	$\in \mathcal{P}(\mathbf{Var} \to \mathbf{Str})$

Abstract domain:

$\mathbf{Chars}$	C	$\in \mathcal{P}^N(\mathbf{Char})$ where $\mathcal{P}^N(A) = \mathcal{P}(A)/\{\emptyset\}$
$\mathbf{Atom}$	a	$::= C \mid r^*$
$\mathbf{Reg}$	p, q, r	$\in \{a_1a_2\cdots a_n \mid n \geq 0, a_i \in \mathbf{Atom}, \neg\exists i.(a_i = p^* \wedge a_{i+1} = q^*)\}$
$\mathbf{State}$	σ	$\in (\mathbf{Var} \to \mathcal{P}^N(\mathbf{Reg}))_\perp$

Meaning:

$\mathbf{Atom} \to \mathcal{P}(\mathbf{Str})$	$\gamma_a(C)$	$= C$
	$\gamma_a(r^*)$	$= \{s_1s_2\cdots s_n \mid 0 \leq n, s_i \in \gamma_r(r)\}$
$\mathbf{Reg} \to \mathcal{P}(\mathbf{Str})$	$\gamma_r(a_1a_2\cdots a_n)$	$= \{s_1s_2\cdots s_n \mid s_i \in \gamma_a(a_i)\}$
$\mathcal{P}^N(\mathbf{Reg}) \to \mathcal{P}(\mathbf{Str})$	$\gamma_R(R)$	$= \bigcup\{\gamma_r(r) \mid r \in R\}$
$\mathbf{State} \to \mathbf{State}^{col}$	$\gamma_S(\perp)$	$= \emptyset$
	$\gamma_S(\sigma)$	$= \{\lambda x.s_x \mid s_x \in \gamma_R(\sigma(x))\}$

Order:

$\mathbf{Reg}$	$p \sqsubseteq q$ **iff** $\gamma_r(p) \subseteq \gamma_r(q)$
$\mathcal{P}^N(\mathbf{Reg})$	$P \sqsubseteq Q$ **iff** $\gamma_R(P) \subseteq \gamma_R(Q)$
$\mathbf{State}$	$\sigma \sqsubseteq \sigma'$ **iff** $\gamma_S(\sigma) \subseteq \gamma_S(\sigma')$

Figure 3.2: Abstract Domain [Choi et al., 2006]

Figure 3.2 is replicated from Choi et al. [2006], and shows the abstract domain for the string analysis. The collecting domain describes the concrete properties of the application that we are aiming to predict, or collect. Each string consists of a sequence of characters, whilst the collecting state, $\mathbf{State^{col}}$, is simply a mapping of variable identifiers to strings.

The abstract domain defines the value domain for the flow state of the dataflow analysis, which provides an approximation of the collecting domain. Abstract characters are represented by the set $\mathbf{Chars}$, a non-empty set of characters. Choi et al. [2006] define the term *atom*, where an atom is either an abstract character or the repetition of a regular string[1]. A regular string consists of a series of atoms without consecutive repetitions. The abstract state is a mapping of variable identifiers to non-empty sets of regular strings, and the abstract state has a least element, as defined by a partial ordering.

We shall describe regular strings further with some examples. As Choi et al. [2006] we shall use the notation "$\{a, b, c\}$" to define an abstract character that represents an instance of any one of three characters, a, b or c. Regular strings can be concatenations of abstract characters, so "$\{a\}\{b\}\{c, d\}$" represents the string "abc" or "abd". For abstract characters that are singleton sets, we shall usually omit the braces, so the string "$\{a\}\{b\}\{c, d\}$" is equivalent to "$ab\{c, d\}$". We show repetitions using the star character, as in regular expressions, so the string "abc^*" is the character a then b followed by

[1] Repetition here has the same meaning as in regular expressions i.e. this regular string may be repeated 0 or more times

zero or more occurrences of c. Consecutive repetitions are not allowed, in order to maintain properties which guarantee the termination of the analysis. This guarantee comes at the cost of a slight lack of expressiveness, because consecutive repetitions are grouped as a single repeated abstract character; the string "$ab\{c,d\}^*e^*f$" is illegal, but can be approximated as "$ab\{c,d,e\}^*f$" without using consecutive repetition. It is also worth noting that, the set of characters can include the "." character which represents an instance of any character, therefore we can create the regular string ".*" which represents the repetition of any character, and therefore, any possible string in the collecting domain.

The *Meaning* section of Figure 3.2 shows how the abstract domain is mapped to the collecting domain, and how each abstract value can represent the corresponding set of values in the collecting domain. The γ functions will return the possible concrete values for any supplied abstract values.

The *Order* section of Figure 3.2 shows the partial ordering for the abstract domain. The partial ordering allows us to compare regular strings and abstract states, to see if one is a subset of the other. For example, the regular string "$abbb$" is partially ordered before "$a.^*$", because the set of concrete strings it represents, is a subset of the concrete strings represented by "$a.^*$". The partial ordering is important for instances of the monotone framework, because it is used by the algorithms that solve the dataflow analysis for deciding if merged flow states contain new information or not.

In the remainder of this chapter we shall use the bottom operator $\perp$, to classify sets that always contain a least element as defined by the ordering. This is used to specify sets that have a unique least element to avoid ambiguity.

Regular strings, as defined here, are a subset of regular expressions. They are required because they can be used to create a widening operator for approximating the upper-bound of any two given regular strings. This widening operator is essential in enabling the string analysis. For example, consider the source code shown in Listing 3.2.

```
1 string x = "a";
2 for(int i=0; i<10; i++)
3 {
4   x = x + "b ";
5   x = x.Trim();
6 }
7 Console.WriteLine(x);
```

Listing 3.2: Dynamically constructed string

The value of *x* at Line 7 will be "*abbbbbbbbb*". It is clear from this example that the loop will execute ten times, however for the purposes of program analysis, it is often undecidable how many times a loop will iterate, and therefore it is common to approximate that loops will be executed any number of times.

Line 4 can have two possible reaching definitions of *x*, the definition coming from Line 1 or the definition from the result of the assignment on Line 5. During the first iteration of the loop *x* will have the value "a". During the second it will be "a" or "ab", during the third "a" or "ab" or "abb" etc. The

analysis will not terminate, because the algorithms used to evaluate the dataflow will loop until the flow states reach a fixed point, and because the string could be infinitely long [2], a fixed point will never be reached.

The approach used by Choi et al. [2006] to solve this problem, is the creation of a widening operator for regular strings. During the second iteration of the loop, we know that the value of *x* can be "a" or "ab". At the point we where the control flow graph merges, we also join the values of the incoming flow states. The widening operator is used to find the least upper bound of these two states, "a" $\bigtriangledown^k$ "ab" = "ab^*". During the third iteration, the analysis will compare "ab^*" to "ab^*b" and conclude that because "ab^*" is partially ordered before "ab^*b" - i.e. the string "ab^*" can contain the string "ab^*b" - no merging needs to take place. Every stage of the string analysis that requires widening, as the result of a joining or merging of flow states, only merges if the right hand side argument is not partially ordered before the left hand side. This is because, if the right hand side is partially ordered before the left hand side, then the left hand side is already a superset of the right hand side value. The use of partial ordering in this way is standard practice in the implementation of algorithms to solve monotone framework instances.

The results of the analysis are that on Line 7 the value of the variable *x* will be approximated by the regular string "ab^*".

The full semantics of the widening operator $\bigtriangledown^k$, are specified in the paper by Choi et al. [2006]. The k value of the widening operator specifies that the widening will be precise where the nesting of repetitions is less than or equal to k, otherwise the widening may involve some approximation. Various other properties about the widening operator are also shown in the original paper. The paper also specifies semantics for concatenation of regular strings, specified by the $\cdot$ operator, as well as the operations *replace*, *trim* and *substr*.

The semantics for the string analysis are specified in Figure 3.3. We omit the extensions for integers, provided by Choi et al. [2006], for the sake of brevity.

The core language consists of the following expressions; a constant s, a variable x, a string concatenation $e + e$, or a string operation $x.op$. The semantics for expressions are denoted by $\mathcal{E}$. A string constant expression, $\mathcal{E}[\![s]\!]$, returns a regular string equivalent of the supplied concrete string s. A variable expression, $\mathcal{E}[\![x]\!]$, evaluates to the value of the variable x from the abstract state. A concatenation expression, $\mathcal{E}[\![e_1 + e_2]\!]$, uses the regular string concatenation operator to concatenate the regular strings, which result from evaluating the two sub-expressions e_1 and e_2 respectively. The string operation expression, $\mathcal{E}[\![e.op]\!]$, produces a set, in which the appropriate string operation has been applied to all regular strings resulting from the evaluation of the sub expression. The string concatenation operators and other string operations are specified in the original paper by Choi et al. [2006], and are not replicated here for the sake of brevity.

The statements of the language each have a transfer function denoted by $\mathcal{T}$. In the core language a

[2] An accurate concrete domain might limit the string length to less than infinity, for example Java strings are limited to 2,147,483,647 characters. This limit would allow the analysis to terminate, but the huge size of the sets used to reach termination would be computationally unfeasible, meaning that these limits can effectively be considered as infinite.

Abstract Domain:

Loc	l	
Value	V	$\in \mathcal{P}^N(\mathbf{Reg} + \mathbf{Loc} + \{nil\})$
State	σ	$\in \mathbf{Var} \to \mathbf{Value}$
Uniqueness	u	$\in \{1, \omega\}$
Content	V^u	$\in \mathbf{Value} \times \mathbf{Uniqueness}$
Heap	h	$\in \mathbf{Loc} \rightharpoonup \mathbf{Content}$

Order:

Value	$V \sqsubseteq V'$	**iff** $V, V' \subseteq \mathbf{Reg}$ and $\gamma_R(V) \subseteq \gamma_R(V')$
		or $V, V' \subseteq \mathbf{Loc} \cup \{nil\}$ and $V \subseteq V'$
State	$\sigma \sqsubseteq \sigma'$	**iff** $\sigma(x) \sqsubseteq \sigma'(x)$ for all $x \in \mathbf{Var}$
Uniqueness	$1 \sqsubseteq \omega$	
Content	$V_1^{u_1} \sqsubseteq V_2^{u_2}$	**iff** $V_1 \sqsubseteq V_2$ and $u_1 \sqsubseteq u_2$
Heap	$h_1 \sqsubseteq h_2$	**iff** $dom(h_1) \subseteq dom(h_2)$
		and $h_1(l) \sqsubseteq h_2(l)$ for all $l \in dom(h_1)$
$(\mathbf{State} \times \mathbf{Heap})_\perp$	$\perp \sqsubseteq (\sigma, h)$	
	$(\sigma_1, h_1) \sqsubseteq (\sigma_2, h_2)$	**iff** $\sigma_1 \sqsubseteq \sigma_2$ and $h_1 \sqsubseteq h_2$

Expressions:

$\mathcal{E}[\![e]\!] : \mathbf{State} \to \mathcal{P}(\mathbf{Reg})$ for $e ::= s \mid x \mid e + e \mid e.op$

$$
\begin{aligned}
\mathcal{E}[\![s]\!]\sigma &= \{C_1 \cdots C_2 | s = c_1 c_2 \cdots c_n, C_i = \{c_i\}\} \\
\mathcal{E}[\![x]\!]\sigma &= \sigma(x) \\
\mathcal{E}[\![e_1 + e_2]\!]\sigma &= \mathcal{E}[\![e_1]\!]\sigma \cdot \mathcal{E}[\![e_2]\!]\sigma \\
\mathcal{E}[\![e.op]\!]\sigma &= \bigcup\{[\![op]\!]p | p \in \mathcal{E}[\![e]\!]\sigma\}
\end{aligned}
$$

Statements:

$\mathcal{T}[\![t]\!] : \quad (\mathbf{State} \times \mathbf{Heap})_\perp \to (\mathbf{State} \times \mathbf{Heap})_\perp$

for $t ::=$ skip $\mid x := e \mid t; t \mid$ if $t\ t \mid$ while $t \mid x :=$ new$^l \mid x := [y] \mid [x] := y$

$$
\begin{aligned}
\mathcal{T}[\![t]\!]\perp &= \perp \\
\mathcal{T}[\![\text{skip}]\!](\sigma, h) &= (\sigma, h) \\
\mathcal{T}[\![x := e]\!](\sigma, h) &= \begin{cases} (\sigma[\mathcal{E}[\![e]\!]\sigma / x], h) & \text{if } \mathcal{E}[\![e]\!]\sigma \neq \emptyset \\ \perp & \text{if } \mathcal{E}[\![e]\!]\sigma = \emptyset \end{cases} \\
\mathcal{T}[\![t_1; t_2]\!](\sigma, h) &= \mathcal{T}[\![t_2]\!](\mathcal{T}[\![t_1]\!](\sigma, h)) \\
\mathcal{T}[\![\text{if } t_1 t_2]\!](\sigma, h) &= \mathcal{T}[\![t_1]\!](\sigma, h) \sqcup \mathcal{T}[\![t_2]\!](\sigma, h) \\
\mathcal{T}[\![\text{while } t]\!](\sigma, h) &= \text{fix}^\nabla \lambda(\sigma', h').(\sigma, h) \sqcup \mathcal{T}[\![t]\!](\sigma', h') \\
\mathcal{T}[\![x := \text{new}^l]\!](\sigma, h) &= \begin{cases} (\sigma[\{l\}/x], h[\{nil\}^1/l]) & \text{if } l \notin dom(h) \\ (\sigma[\{l\}/x], h[(V \cup \{nil\})^\omega/l]) & \text{if } l \in dom(h) \text{ and } h(l) = V^u \end{cases} \\
\mathcal{T}[\![x := [y]]\!](\sigma, h) &= \begin{cases} (\sigma[V'/x], h) & \text{if } V' \neq \emptyset \\ \perp & \text{if } V' = \emptyset \end{cases} \\
& \quad \text{where } V' = \bigcup\{V | l \in \sigma(y), h(l) = V^u\} \\
\mathcal{T}[\![[x] := y]\!](\sigma, h) &= \begin{cases} (\sigma, h[\sigma(y)^1/l]) & \text{if } \sigma(x) = \{l\} \text{ and } h(l) = V^1 \\ (\sigma, h') & \text{otherwise} \end{cases} \\
& \quad \text{where } h' = \lambda l. \begin{cases} h(l) & \text{if } l \in dom(h) \text{ and } l \notin \sigma(x) \\ (\sigma(y) \cup V)^u \text{ where } h(l) = V^u & \text{if } l \in dom(h) \text{ and } l \in \sigma(x) \\ \text{undefined} & \text{if } l \notin dom(h) \end{cases}
\end{aligned}
$$

Figure 3.3: String Analysis for Core Language with Heap Extensions [Choi et al., 2006]

statement is one of the following; a no-operation skip, an assignment $x := e$, a sequence of statements $t; t$, a conditional statement if t t or a while loop while t. The no-operation transfer function has no effect. The assignment statement replaces the value in the abstract state that x maps onto, with the value of the evaluated expression. The sequence of statements transfer function uses the result of the the first sub statement as the input for the second. The conditional statement creates a union of the evaluation of both sub statements that represent the conditional branches. The loop transfer function evaluates to the fixed point of the evaluation of the transfer function for the body sub statement. Note, that the boolean expression in the loop and conditional statements are not considered. This is because loop conditions with boolean expressions can be produced from the statements defined here; a conditional with a boolean can be considered as a compound statement, consisting of an expression statement followed by a basic conditional expression. In this way the statements defined here represent a minimum set that can be used to create more complex control structures and language statements.

In addition to the expressions and statements of the core language, Choi et al. [2006] define the semantics for an abstract heap, to model modern OO programming languages that store objects as items on the heap, such as C++, Java and C#. The heap is represented in the abstract domain as a partial mapping of Loc to $Content$. Each member of Loc represents a pointer or reference, that can map to a heap value that represents an object. The heap value is represented by a tuple consisting of a set of values, and a flag to indicate if the object is unique. A heap value typically represents an object, but only has one set of values, as this analysis considers all objects to be of size 1, with this one value being an approximation of all instance variables for the object. This may seem like a crude generalisation, but we shall show how this affects the analysis in Chapter 7, being accurate enough for our purposes, whilst maintaining a low cost.

The definition of $\mathbf{Value}$ has also been extended, from the definition in Figure 3.3 to include Loc or nil values, meaning that the abstract state can now contain string values or pointers to items in the abstract heap.

The added transfer functions for the heap extensions are as follows; an allocation statement $x := new^l$, a load statement $x := [y]$ and a store statement $[x] := y$.

The allocation transfer function creates a new item on the heap if the location l does not already exist in the domain of the heap, and updates the values and uniqueness of the existing value otherwise. If a new heap value is created, the content of the object is initialised to the nil value with a uniqueness of 1. In both cases the value of x in the abstract state will mapped to l.

The transfer function for the load statement will load the values from any heap objects pointed to by the locations of y. The brackets around y can be thought of as dereferencing y, or following the locations values to the appropriate heap values.

The transfer function for the store statement will store the value specified in y into the content of the locations pointed to by x. If the object is unique, the content can be strongly updated, otherwise it is weakly updated.

The core language, specified here, is missing many features of modern programming languages.

Most of these are not discussed here as they would start to involve language specific code, and could be added to the analysis using standard techniques [Nielson et al., 1999]. One key omission, however, is how this analysis can be extended to include interprocedural analysis, which we shall discuss in the following section, including why impact analysis imposes specific requirements upon how this is achieved.

3.2 Interprocedural Analysis

```
1  class Program
2  {
3      public static void Main()
4      {
5          string name1 = "Wolfgang";
6          string name2 = "David";
7
8          SayHello(name1);
9          SayHello(name2);
10     }
11
12     private static void SayHello(string name)
13     {
14         Console.WriteLine("Hello " + name);
15     }
16 }
```

Listing 3.3: A simple hello world program

The analysis we have discussed so far is capable of analysing sequential code, without method calls, and could only be used to analyse the program in Listing 3.3, one method at a time. The incoming flow state of the *Main* method, can be initialised with an empty abstract state and abstract heap because the method signifies the beginning of the program, the root method, and will have no prior values in the state or heap.

What should the initial flow state value be for the *SayHello* method? The *SayHello* method is called from two separate locations, and there are a number of ways in which the flow state could be initialised. We could start the analysis with an empty abstract state and heap, but clearly this excludes information about the value of the *name* parameter on Line 12. We need to take into account the values specified by the parameters on Lines 8 and 9. One approach is to approximate the *name* parameter to be a set containing both possible strings, or a single regular string that represents the upper bound of the two. A *context-insensitive analysis* is an interprocedural analysis where all possible calling contexts of a procedure or method are merged into a single context that approximates them all [Nielson et al., 1999]. The different calling contexts of the procedures are not being taken into account separately, so it is called context-insensitive.

A *context-sensitive analysis* would, to some degree, take into account the separate calling contexts, so in Listing 3.3, we could analyse the *SayHello* method twice, once with the *name* variable initialised in the state as "$Wolfgang$", and once with it initialised as "$David$". This would be a context-sensitive analysis. We argue that our analysis requires context sensitivity, and we shall describe the reasons for this, in detail, in the following subsection.

The details of how to implement an interprocedural analysis are numerous. How do we handle return values, different parameter passing mechanisms and global and static variables? Fortunately, this problem is well-studied and standard solutions exist. We describe the particular type of interprocedural analysis that is suitable next.

3.2.1 k-CFA

A k-CFA analysis is a standard program analysis solution for implementing interprocedural dataflow analysis [Jagannathan and Weeks, 1995]. In a k-CFA analysis, items in the property space are labelled with context strings to distinguish one dynamic instance from another. A context string consists of a string of identifiers representing the last k call sites, which would represent the top k items on the call-stack in a standard procedural programming language. For example, in Listing 3.3, the abstract state variables for the `SayHello` method could be represented as:

$$8\text{:name} \rightarrow "Wolfgang"$$
$$9\text{:name} \rightarrow "David"$$

The name variable has two instances in the state, the first is in the context of a call from Line 8, the second is in the context of a call from Line 9. This is a 1-CFA naming scheme, using the last call site to distinguish between the variables.

We shall illustrate the k-CFA approach, in more detail using an example from our implementation of the UCL Coffee applications. The following example shows the pertinent methods from three classes in the sales application:

The application was implemented according to the *Table Data Gateway* pattern, as described by Fowler [2003], and as such we argue that this is a realistic design pattern, not a contrived example. Notably, the Table Data Gateway pattern is described as one of the simpler patterns for data access, and we can expect to see much more complex patterns in use, in practice.

The *CustomerGateway.Findall()* and *OrderGateway.SelectByCustomer(int)* methods, both call *DB.Prepare(string)*, with a parameter specifying the name of the stored procedure to execute. The *DB.Prepare(string)* method is an overloaded method, and calls the *DB.Prepare(string,OleDbConnection)*, which constructs an appropriate *OleDbCommand* object and returns it. The returned command object is executed in the *CustomerGateway.Findall()* and *OrderGateway.SelectByCustomer(int)* methods respectively.

Figure 3.4 is a graph showing the method calls to *DB.Prepare(string)* and *DB.Prepare(string, OleDbConnection)*, in a dataflow analysis, illustrating how the incoming flow states might be approximated for different calling contexts using the k-CFA approach. The boxes with the gray header show methods with their callsites, whilst the boxes with white headers show the incoming flow states for the methods.

```
1  class DB : IDisposable{
2      // ... (other methods omitted for brevity)
3
4      public OleDbCommand Prepare(string cmdText){
5        return Prepare(cmdText, conn);
6      }
7
8      public static OleDbCommand Prepare(string cmdText, OleDbConnection conn){
9          OleDbCommand command = new OleDbCommand(cmdText, conn);
10         return command;
11     }
12 }
13
14 public class OrderGateway : IDisposable{
15     // ... (other methods omitted for brevity)
16
17     public IDataReader SelectByCustomer(int customerId){
18         var selOrderByCustomer = db.Prepare("selOrderByCustomer");
19         selOrderByCustomer.Parameters.Add(
20           new System.Data.OleDb.OleDbParameter("@customerId", customerId));
21         selOrderByCustomer.CommandType = CommandType.StoredProcedure;
22         return selOrderByCustomer.ExecuteReader(); // SALES-Q7 executed
23     }
24 }
25
26 public class CustomerGateway : IDisposable{
27     // ... (other methods omitted for brevity)
28
29     public IDataReader FindAll(){
30         var cmd = db.Prepare("selCustomersAll");
31         return cmd.ExecuteReader(); // SALES-Q1 executed
32     }
33 }
```

Listing 3.4: 'Sales application design pattern example'

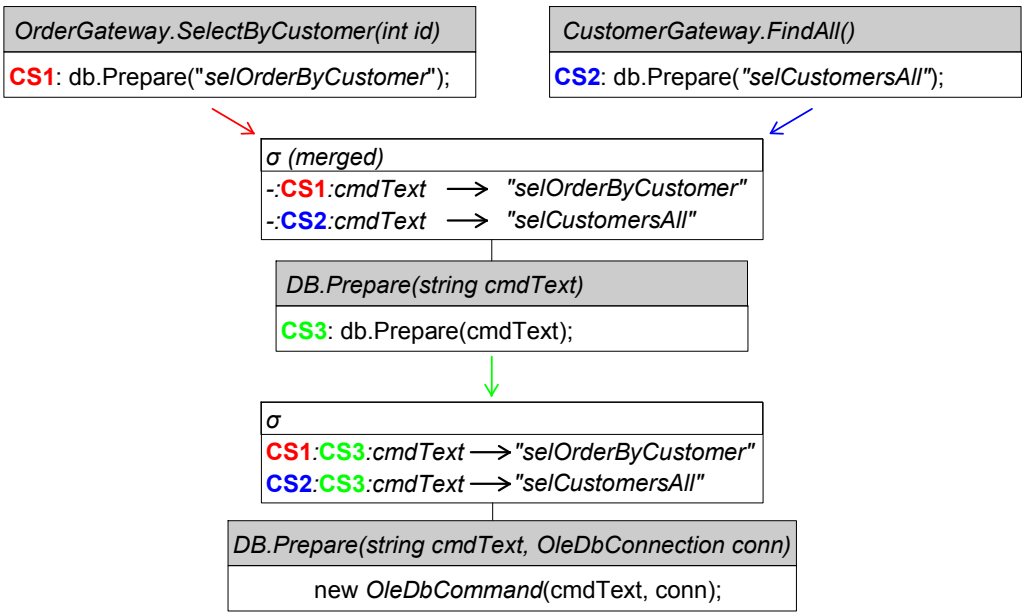

Figure 3.4: 2-CFA analysis example

The coloured arrows represent calls, and are coloured in the same colour as the originating call site identifiers.

Figure 3.4 shows the case where $k = 2$. The *OrderGateway.SelectByCustomer(int)* method, shows a call to *DB.Prepare(string)* at a callsite labelled `CS1`. The *CustomerGateway.FindAll()* method, shows a call to *DB.Prepare(string)* at a callsite labelled `CS2`. The abstract states at these two callsites are used to create incoming states for the *DB.Prepare(string)* method, and are merged together to create a single incoming abstract state, where we see two instances of the *cmdText* parameter. The call to *DB.Prepare(string, OleDbCommand)* is made at the callsite labelled `CS3`, and we see how this callsite is used to label the items in the incoming abstract state for the *DB.Prepare(string, OleDbCommand)* method.

In this case the precision of the context-sensitive analysis is adequate to correctly predict the queries. However, we shall see next what happens when we decrease the precision of the analysis.

3.2.2 Requirement for Context-Sensitivity

The results of using a 1-CFA analysis, as opposed to a 2-CFA analysis, are shown in Figure 3.5.

When the 1-CFA approach is used, we see that each variable identifier in the abstract state is prefixed by a single callsite identifier. The incoming abstract state for *DB.Prepare(string)* can still correctly disambiguate between the two parameters supplied at `CS1` and `CS2`. When we calculate the incoming abstract state for the *DB.Prepare(string, OleDbCommand)* method, we see that the *cmdText* parameter gets approximated as either of the query values. This is because the k-CFA naming scheme only uses

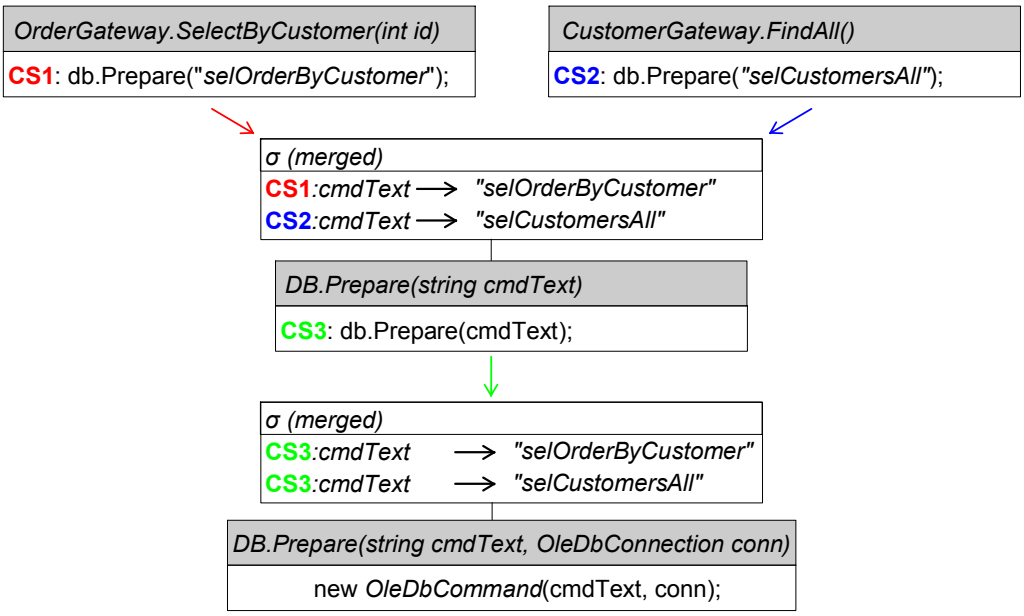

Figure 3.5: 1-CFA analysis example

the top k callsites in the identification, and in this case the top 1 callsite consists only of `CS3`, and in the context of a call from `CS3` there are two possible values that *cmdText* could have.

In Figure 3.5 we have shown the *cmdText* variable as simply being mapped to two possible values, however, this is not how the analysis would be implemented in practice. Using the analysis of Choi et al. [2006], as described above, the *cmdText* variable would be mapped to the upper limit of these two possible values, found by using the widening operator, which would give us the string $\{s, e, l, O, r, d, B, y, C, u, t, o, m, A\}*$.

This approximation is required to make the analysis terminate, as discussed previously. In the approximated string, it is hard to distinguish the separate, constituent queries that constitute the string, and in a real world application this string could be an approximation of hundreds of queries, instead of just two, obfuscating the original queries further.

Additionally, a lack of context sensitivity causes the use of query results to be over-approximated. In Figure 3.5 we over-approximate the *cmdText* variable that represents a query string. Consider, however, the case where the parameter being approximated is a query results object, such as an *OleDbDataReader*. The query results object will be approximated as an object that could result from the execution of multiple queries, and this will cause our analysis to identify individual queries as being executed and used, in places where they are not, effectively adding more false positives.

If we have large applications with hundreds or thousands of queries, and the analysis is approximated due to a lack of context-sensitivity, our analysis could produce many false positives. If the

analysis is unable to correctly associate query definitions, query executions, and the use of query results with each other without over-approximation, then query definitions could be associated to more executions and more results uses than are correct. In the worst case, every query definition will be associated with all execution locations, and this could result in a situation where we simply group all queries in the application into a single regular string approximation.

As we shall see, in the remainder of this chapter, and in Chapter 4, traceability and the ability to associate a query definition with its execution and the use of its results are very important. The over-approximation caused by an insufficient level of context-sensitivity, also affects the ability to maintain this traceability. In the worst case all the results objects would be associated with all of the query definitions, making it impossible to correctly associate the definition of an individual query with its execution and the use of its results. We shall describe in more detail how the results of the string analysis are used to analyse these associations in Chapter 4, showing how being able to make such associations is important to our approach, and how over-approximation of these associations could be detrimental to the accuracy of our analysis.

The lack of context-sensitivity, could potentially lead to a level of false positives that leaves the user to examine large parts of the application by hand, and our analysis would be providing little benefit. This kind of approximation can be very bad for impact analysis, reducing accuracy and impact-precision and decreasing the overall usefulness of the analysis.

The approximations we have discussed here occur when the value of k in the k-CFA analysis is not high enough to analyse programs that contain design patterns with complex control flow and deeply nested procedure calls. In Chapter 2 we required that our analysis should be able to cope with real-world design patterns, as specified in Requirement-4, and therefore we require that our analysis is context-sensitive to a precise enough level to achieve this.

The implementation described by Choi et al. [2006] provides an implementation of a 1-CFA analysis. We argue that given the above example, this is not sufficient for the purposes of impact analysis. Although we do not mandate a specific level of context sensitivity, we require that the level should at least be variable, and it may need to be greater than 1. The example shown here requires that $k \geq 2$ to avoid over-approximation, and we shall discuss in Chapter 7 that the levels of context-sensitivity that are required in practice can be $k \geq 4$ or higher. For now, we argue that a variable level of k-CFA analysis is required, and typically analyses will be at least 2-CFA. The increase in k can be achieved using the standard algorithms, and we discuss the increase in computational cost this has in Chapter 5 and Chapter 7.

3.3 Traceability

3.3.1 Requirement for Traceability

In Section 2.3.1, we saw how the definition of a query, the execution of a query and the use of the results of a query, can be spread across many different methods, and we defined Requirement-5 to make sure that we preserved the traceability between these locations for a given query.

The string analysis, as described above, is not built with this traceability in mind; instead, it is required only to predict the value of a string at a given location. Our impact analysis requires that we know the value of a query data type at a given location, but also that we know where it was defined or modified. The analysis described so far, has been the standard widening-based string analysis of Choi et al. [2006], but the extensions that we shall now describe are novel contributions of our research.

For example, in Listing 2.2, a dynamic query is created and executed. The string analysis creates the following approximate query at the point of execution:

$$SELECT\ \ FROM Product\{,\ Supplier\} * WHERE\ (.*\ .*)\ AND\ Product.supplier\_id\ =\ Supplier.id;$$

The data propagated through the dataflow analysis only represents the *value* of these strings, and when the strings are altered, the values are simply updated. The analysis does not keep track of where the original definitions were, or where the modifications took place. When a fixed point is reached, we cannot tell where a given string was defined. We shall describe the modifications that are necessary to add this required traceability to the analysis next.

3.3.2 Traceability Extensions

The analysis of Choi et al. [2006] represents strings as *mutable* values in the abstract state; meaning that when strings are altered, their values are mutated in place. If we make strings *immutable*, every time we modify a string, we must create a new string without destroying the constituent strings.

Strings can be made immutable by representing them as objects on the heap, instead of values in the state, and this can be achieved by altering the semantics of the string manipulating operations to create new heap objects when performing string modifications[3].

Once we have immutable heap string objects, every string will have an identifier, defined by its location in the heap, and it is these identifiers that can be used to provide the traceability that we desire.

The abstract heap of the analysis is changed as follows:

Abstract Domain:

$\mathbf{Loc}$	l	
$\mathbf{RegLoc}$	l^{reg}	
$\mathbf{Value}$	V	$\in \mathcal{P}^N(\mathbf{RegLoc} + \mathbf{Loc} + \{nil\})$
$\mathbf{State}$	σ	$\in \mathbf{Var} \to \mathbf{Value}$
$\mathbf{Uniqueness}$	u	$\in \{1, \omega\}$
$\mathbf{Content}$	V^u	$\in \mathbf{Value} \times \mathbf{Uniqueness}$
$\mathbf{Heap}$	h	$\in \mathbf{Loc} \rightharpoonup \mathbf{Content}$
$\mathbf{RegHeap}$	h^{reg}	$\in \mathbf{RegLoc} \to \mathbf{Reg}$

Most of the abstract domain is similar to the original definition by Choi et al. [2006], shown in Figure 3.3. In our extensions, we alter the set of values, $\mathbf{Value}$, so that instead of referring to a set of regular strings directly, the value can refer to a set of locations for regular strings, $\mathcal{P}\{l^{\text{reg}}\}$. Each regular string location, $\mathbf{RegLoc}$, is used by the regular string heap, h^{reg}, to identify a particular regular string.

[3] In fact, this closely resembles the concrete semantics of C# and Java, where strings are immutable heap objects.

Instead of adding the regular string heap, immutable strings could have been added by reusing the previously defined abstract heap, but this would have made the definition of the semantics more complex, especially the partial ordering operators, so we have chosen to create a separate string heap to keep the semantics simple, and to show clearly, the differences between our extensions and the analysis of Choi et al. [2006].

With these extensions to the abstract domain, the ordering operators can be updated:

Order:

$(\mathbf{Value} \times \mathbf{RegHeap})$	$(V, h_1^{\text{reg}}) \sqsubseteq (V', h_2^{\text{reg}})$	**iff** $V, V' \subseteq \mathbf{RegLoc}$ and $\gamma_R(\text{lookup}(V, h_1^{\text{reg}})) \subseteq \gamma_R(\text{lookup}(V', h_2^{\text{reg}})$ or $V, V' \subseteq \mathbf{Loc} \cup \{nil\}$ and $V \subseteq V'$
$(\mathbf{State} \times \mathbf{RegHeap})$	$(\sigma, h_1^{\text{reg}}) \sqsubseteq (\sigma', h_2^{\text{reg}})$	**iff** $(\sigma(x), h_1^{\text{reg}}) \sqsubseteq (\sigma'(x), h_2^{\text{reg}})$ for all $x \in \mathbf{Var}$
$\mathbf{Uniqueness}$	$1 \sqsubseteq \omega$	
$(\mathbf{Content} \times \mathbf{RegHeap})$	$(V_1^{u_1}, h_1^{\text{reg}}) \sqsubseteq (V_2^{u_2}, h_2^{\text{reg}})$	**iff** $(V_1, h_1^{\text{reg}}) \sqsubseteq (V_2, h_2^{\text{reg}})$ and $u_1 \sqsubseteq u_2$
$(\mathbf{Heap} \times \mathbf{RegHeap})$	$(h_1, h_1^{\text{reg}}) \sqsubseteq (h_2, h_2^{\text{reg}})$	**iff** $dom(h_1) \subseteq dom(h_2)$ and $(h_1(l), h_1^{\text{reg}}) \sqsubseteq (h_2(l), h_2^{\text{reg}})$ for all $l \in dom(h_1)$
$(\mathbf{State} \times \mathbf{Heap} \times \mathbf{RegHeap})_\perp$	$\perp \sqsubseteq (\sigma, h, h^{\text{reg}})$	
	$(\sigma_1, h_1, h_1^{\text{reg}}) \sqsubseteq (\sigma_2, h_2, h_2^{\text{reg}})$	**iff** $(\sigma_1, h_1^{\text{reg}}) \sqsubseteq (\sigma_2, h_2^{\text{reg}})$ and $(h_1, h_1^{\text{reg}}) \sqsubseteq (h_2, h_2^{\text{reg}})$

We introduce an auxiliary function to lookup a set of values in the regular string heap:

$$\text{lookup} : (\mathbf{Value} \times \mathbf{RegHeap}) \rightarrow \mathcal{P}^N(Reg)$$

$$\text{lookup}(V, h^{\text{reg}}) \quad = \quad \begin{cases} \bigcup\{h^{\text{reg}}(l^{\text{reg}}) | l^{\text{reg}} \in V\} & \textbf{iff } V \subseteq dom(h^{\text{reg}}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

This function is used to lookup the actual regular strings from the regular string heap that correspond to the supplied set of values.

The altered partial orderings, are effectively the same, except that they now have to lookup the values of the regular strings, rather than comparing regular string values directly.

We also introduce the set $\mathbf{UnassignedRegLoc}$ which contains all regular string heap location identifiers that are yet to be assigned; these can be thought of as representing free memory locations in the concrete domain. $\mathbf{UnassignedRegLoc}$ is a subset of $\mathbf{RegLoc}$, representing the regular heap location that are yet to be assigned and used as identifiers.

The major difference in the semantics, is the string expressions, as follows:

$$\mathcal{E}[\![e]\!] : (\mathbf{State} \times \mathbf{RegHeap}) \rightarrow (\mathcal{P}(\mathbf{RegLoc}) \times \mathbf{RegHeap}) \text{ for } e ::= s, l^{\text{reg}} \mid x \mid e + e, l^{\text{reg}} \mid e.op$$

$$
\begin{array}{lcl}
\mathcal{E}[\![s, l^{\text{reg}}]\!](\sigma, h^{\text{reg}}) & = & (\{l^{\text{reg}}\}, h^{\text{reg}}[\{C_1 \cdots C_2 | s = c_1 c_2 \cdots c_n, C_i = \{c_i\}\}/l^{\text{reg}}]) \\
\mathcal{E}[\![x]\!](\sigma, h^{\text{reg}}) & = & (\sigma(x), h^{\text{reg}}) \\
\mathcal{E}[\![e_1 + e_2, l^{\text{reg}}]\!](\sigma, h_1^{\text{reg}}) & = & (V'', \bigsqcup(h_3^{\text{reg}}[p/l^{\text{reg}}] | p \in P, [\text{REMOVED}] l^{\text{reg}} \in \mathbf{UnassignedRegLoc})) \\
 & & \text{where } P = h_3^{\text{reg}}[\text{lookup}(V, h_3^{\text{reg}}) \cdot \text{lookup}(V', h_3^{\text{reg}})/l^{\text{reg}}] \\
 & & \text{where } (V', h_3^{\text{reg}}) = \mathcal{E}[\![e_2]\!](\sigma, h_2^{\text{reg}}) \\
 & & \text{where } (V, h_2^{\text{reg}}) = \mathcal{E}[\![e_1]\!](\sigma, h_1^{\text{reg}}) \\
 & & \text{where } V'' = \text{ all } l^{reg} \text{ added from } \mathbf{UnassignedRegLoc} \\
\mathcal{E}[\![e.op]\!](\sigma, h_1^{\text{reg}}) & = & (V', \bigsqcup(h_2^{\text{reg}}[p/l^{\text{reg}}] | p \in P, l^{reg} \in \mathbf{RegLoc}, l^{\text{reg}} \in \mathbf{UnassignedRegLoc})) \\
 & & \text{where } P = \bigcup\{[\![op]\!]p | p \in \text{lookup}(V, h_2^{\text{reg}})\} \\
 & & \text{where } (V, h_2^{\text{reg}}) = \mathcal{E}[\![e]\!](\sigma, h_1^{\text{reg}}) \\
 & & \text{where } V' = \text{ all } l^{reg} \text{ added from } \mathbf{UnassignedRegLoc}
\end{array}
$$

The signature of the expressions function is changed, so that it now requires the state and the regular heap as arguments, and returns a set of regular string locations and an updated regular string heap.

The altered string constant expression adds a new string to the regular heap for the specified location. The variable expression, is almost unaltered, looking up the values in the state as before, but has been altered to comply with the new function signature.

The concatenation expression creates all possible concatenated strings, P, then each of these strings is added to the regular string heap with a new unassigned location identifier. These new locations identifiers are remembered, and returned as the part of the result. In this way, we create new strings, rather than mutating strings in place, and each new string is given a unique identifier.

The expression for string operations is very similar to the concatenation expression. We create a set of all the regular strings resulting from the expression, then assign each one a new identifier in the regular string heap. Again this means we are creating new strings rather than mutating them in place.

We do not show the full semantics for the widening operator here for the sake of brevity. But we must also extend the widening operator in this way, to create new strings in the regular string heap. We can no longer widen an abstract state alone, instead the signature of the widening operator becomes:

$$\bigtriangledown^{k} : (\mathbf{State} \times \mathbf{RegHeap}) \rightarrow (\mathbf{State} \times \mathbf{RegHeap})$$

The corresponding semantics definitions are straightforward to change; every time a string is widened, we simply add a new string to the regular string heap, replacing the existing locations in the state[4], just as we did for the concatenation expression defined above.

The transfer functions remain largely the same, except that they are altered to include the regular string heap:

$$
\begin{array}{ll}
\mathcal{T}[\![t]\!] : & (\mathbf{State} \times \mathbf{Heap} \times \mathbf{RegHeap})_\perp \rightarrow (\mathbf{State} \times \mathbf{Heap} \times \mathbf{RegHeap})_\perp \\
 & \text{for } t ::= \text{skip} \mid x := e \mid t; t \mid \text{if } t\ t \mid \text{while } t \mid x := \text{new}^l \mid x := [y] \mid [x] := y
\end{array}
$$

[4]We consider a new string to be created for each point where the total relation operator is evaluated in the widening, as opposed to every minor mutation that occurs in the calculation of the widened regular string.

$$
\begin{array}{lcl}
\mathcal{T}[\![t]\!]\bot & = & \bot \\
\mathcal{T}[\![\text{skip}]\!](\sigma,h,h^{\text{reg}}) & = & (\sigma,h,h^{\text{reg}}) \\
\mathcal{T}[\![x := e]\!](\sigma,h,h_1^{\text{reg}}) & = & \begin{cases} (\sigma[V/x],h,h_2^{\text{reg}}) & \text{if } V \neq \emptyset \\ \bot & \text{if } V = \emptyset \end{cases} \\
 & & \text{where } (V,h_2^{\text{reg}}) = \mathcal{E}[\![e]\!](\sigma,h_1^{\text{reg}}) \\
\mathcal{T}[\![t_1;t_2]\!](\sigma,h,h^{\text{reg}}) & = & \mathcal{T}[\![t_2]\!](\sqcup[\![t_1]\!](\sigma,h,h^{\text{reg}})) \\
\mathcal{T}[\![\text{if } t_1\ t_2]\!](\sigma,h,h^{\text{reg}}) & = & \mathcal{T}[\![t_1]\!](\sigma,h,h^{\text{reg}}) \sqcup \mathcal{T}[\![t_2]\!](\sigma,h,h^{\text{reg}}) \\
\mathcal{T}[\![\text{while } t]\!](\sigma,h,h^{\text{reg}}) & = & \text{fix}^{\nabla}\lambda(\sigma',h',h_2^{\text{reg}}).(\sigma,h,h_1^{\text{reg}}) \sqcup \mathcal{T}[\![t]\!](\sigma',h',h_2^{\text{reg}}) \\
\mathcal{T}[\![x := \text{new}^l]\!](\sigma,h,h^{\text{reg}}) & = & \begin{cases} (\sigma[\{l\}/x],h[\{nil\}^1/l],h^{\text{reg}}) & \text{if } l \notin dom(h) \\ (\sigma[\{l\}/x],h[(V \cup \{nil\})^{\omega}/l],h^{\text{reg}}) & \text{if } l \in dom(h) \text{ and } h(l) = V^u \end{cases} \\
\mathcal{T}[\![x := [y]]\!](\sigma,h,h^{\text{reg}}) & = & \begin{cases} (\sigma[V'/x],h,h^{\text{reg}}) & \text{if } V' \neq \emptyset \\ \bot & \text{if } V' = \emptyset \end{cases} \\
 & & \text{where } V' = \bigcup\{V | l \in \sigma(y), h(l) = V^u\} \\
\mathcal{T}[\![[x] := y]\!](\sigma,h,h^{\text{reg}}) & = & \begin{cases} (\sigma,h[\sigma(y)^1/l],h^{\text{reg}}) & \text{if } \sigma(x) = \{l\} \text{ and } h(l) = V^1 \\ (\sigma,h',h^{\text{reg}}) & \text{otherwise} \end{cases} \\
 & & \text{where } h' = \lambda l. \begin{cases} h(l) & \text{if } l \in dom(h) \text{ and } l \notin \sigma(x) \\ (\sigma(y) \cup V)^u \text{ where } h(l) = V^u & \text{if } l \in dom(h) \text{ and } l \in \sigma(x) \\ \text{undefined} & \text{if } l \notin dom(h) \end{cases}
\end{array}
$$

We have identified four points where new strings are created, given that strings are now immutable. These are constant string expressions, string concatenation expressions, string operation expressions and widening of abstract states. At each of these points a new string can be created from one or more constituent strings. For our analysis, it is necessary that we can trace the constituent strings for any given regular string. Being able to do this, gives us a traceability between query definitions, query executions and the use of query results. In Chapter 4 we shall show exactly why this traceability of constituent strings is necessary.

To record the original constituent strings of every regular string, we need to store information every time a new string is created. Every time we create a new string, we can identify the constituent elements of the string. Whenever a new string is added to the regular string heap, we record the constituent strings that were used to build it. We create a mapping called the *original regular string mapping* to store this information. We can add this mapping to the abstract domain as follows:

$$\textbf{OriginalReg} \quad O \quad \in \textbf{RegLoc} \to \mathcal{P}(\textbf{RegLoc})$$

When a new string is created in the regular string heap, a new entry is added to the original regular string mapping, mapping the identifier of the newly created string to the identifiers of the original strings that were part of the operation, if any. We do not include the semantics of how these updates occur, as it is very straightforward to implement, and would greatly increase the complexity of the semantics shown. We can use this original string mapping to trace the origins of any string transitively, and this will be enough information to provide the traceability that we require. We shall see how this information is used in practice, as we describe how the result of the string analysis are processed, in Chapter 4.

We shall explain these traceability extensions further, with some examples.

```
1 string y = "abc";
2 string z = "def";
3 string x = y + z;
```

Listing 3.5: A string concatenation

Listing 3.5 shows a program which performs a simple string concatenation. Line 3 shows the variable x being defined as the concatenation of the y and z variables. The incoming flow state for the query analysis of at Line 3 would be as follows:

$$\begin{aligned} \sigma &= \{y \rightarrow l_1^{\text{reg}},\ z \rightarrow l_2^{\text{reg}}\} \\ h &= \{\} \\ h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"}abc\text{"},\ l_2^{\text{reg}} \rightarrow \text{"}def\text{"}\} \\ O &= \{\} \end{aligned}$$

The abstract state contains the variable identifiers x and y, which are mapped to the regular string heap location identifiers l_1^{reg} and l_2^{reg} respectively. The abstract heap is empty, as is the original regular string mapping. The regular string heap, contains the actual strings stored for the identifiers l_1^{reg} and l_2^{reg}.

If we execute the appropriate transfer function for Line 3, then we will return the following output flow state:

$$\begin{aligned} \sigma &= \{y \rightarrow l_1^{\text{reg}},\ z \rightarrow l_2^{\text{reg}},\ x \rightarrow l_3^{\text{reg}}\} \\ h &= \{\} \\ h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"}abc\text{"},\ l_2^{\text{reg}} \rightarrow \text{"}def\text{"},\ l_3^{\text{reg}} \rightarrow \text{"}abcdef\text{"}\} \\ O &= \{l_3^{\text{reg}} \rightarrow \{l_1^{\text{reg}},\ l_2^{\text{reg}}\}\} \end{aligned}$$

This outgoing flow state now includes another item in the abstract state, mapping the x variable to the l_3^{reg} regular string heap location identifier. The abstract heap also includes the string value for this new identifier, which is a concatenation of the other two strings. The original regular string mapping now contains an entry for l_3^{reg}, showing that the constituent components for this string were the strings with the identifiers l_1^{reg} and l_2^{reg}. This resulting flow state shows how the transfer functions alter the incoming flow state to produce an outgoing flow state.

```
1 string x = "a";
2
3 while(conditionalVariable)
4 {
5   x = x + "b";
6 }
```

Listing 3.6: A string widening

Listing 3.6 shows a program which performs a simple string concatenation in a loop. Line 5 shows the variable x being defined as the concatenation of itself plus the literal string "b". The loop will continue

indefinitely, depending upon how the variable *conditionalVariable* is modified. During a typical dataflow analysis implementation, Line 3 will have two incoming states, one from the previous line, and one from the repetition of the loop body. The incoming flow states for the query analysis of Line 5 would be as follows:

InState1:

$$\begin{aligned}\sigma &= \{x \rightarrow l_1^{\text{reg}}\}\\ h &= \{\}\\ h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"}a\text{"}\}\\ O &= \{\}\end{aligned}$$

InState2:

$$\begin{aligned}\sigma &= \{x \rightarrow l_3^{\text{reg}}\}\\ h &= \{\}\\ h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"}a\text{"},\ l_2^{\text{reg}} \rightarrow \text{"}b\text{"},\ l_3^{\text{reg}} \rightarrow \text{"}ab\text{"}\}\\ O &= \{l_3^{\text{reg}} \rightarrow \{l_1^{\text{reg}}, l_2^{\text{reg}}\}\}\end{aligned}$$

InState1 contains a value of x which maps to the regular string "a"; this flow state represents the outgoing flow state from Line 1. *InState2* contains a value of x which maps to the regular string "ab"; this flow state represents the outgoing flow state of the body of the loop. We can see that concatenation in the body of the loop has stored the original constituent strings in the original regular string mapping. These two flow states will be merged into a single incoming flow state for Line 3.

$$\begin{aligned}\sigma &= \{x \rightarrow l_4^{\text{reg}}\}\\ h &= \{\}\\ h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"}a\text{"},\ l_2^{\text{reg}} \rightarrow \text{"}b\text{"},\ l_3^{\text{reg}} \rightarrow \text{"}ab\text{"},\ l_4^{\text{reg}} \rightarrow \text{"}ab^*\text{"}\}\\ O &= \{l_3^{\text{reg}} \rightarrow \{l_1^{\text{reg}}, l_2^{\text{reg}}\},\ l_4^{\text{reg}} \rightarrow \{l_1^{\text{reg}}, l_2^{\text{reg}},\ l_3^{\text{reg}}\}\}\end{aligned}$$

This widened flow state has widened the x variable so that it now maps to the string with the identifier l_4^{reg}, and this identifier maps to the string "ab^*" in the regular string heap. This identifier l_4^{reg} is also added to the original regular string mapping, to record the constituent strings of the widening. Every time a new string is created, the constituent strings are stored in the original string mapping in the same way as shown here.

The previous two examples are typical of the way flow states are altered by transfer functions and widening operators during the course of the dataflow analysis.

3.4 Query Data Types

In Section 2.3 we defined Requirement-4, which specified that our analysis should be able to, extensibly, analyse types which can be used to represent queries or query results. The formalised analysis shown previously, only includes the string data type and a generic heap data type, therefore, in order to analyse a wide range of applications, we need to add many more data types to the analysis.

Consider the implementation of the INV-Q2 query from the UCL Coffee inventory application, as shown in Listing 3.7. On Line 3 an *SqlCommand* object is created with the SQL string as a parameter. To implement this in our analysis we create a transfer function for the constructor of the *SqlCommand*

```
1 string cmdText = "SELECT id, contact_name, company_name FROM " +
2                  "Supplier WHERE id=@ID;";
3 SqlCommand command = new SqlCommand(cmdText, conn);
4 command.Parameters.Add(new SqlParameter("@ID", id));
5 SqlDataReader reader = command.ExecuteReader(); // INV-Q2 executed
6 reader.Read();
7 result = Load(reader);
8 return result;
```

Listing 3.7: Implementation of INV-Q2

object.

$$\mathcal{T}[\![x := \text{new}^l \text{SqlCommand.ctor}(y)]\!](\sigma, h, h^{\text{reg}}) \quad = \quad \mathcal{T}[\![[x] := y]\!](t_1) \quad \text{where } t_1 = \mathcal{T}[\![x := new^l]\!](\sigma, h, h^{\text{reg}})$$

This transfer function, creates a new object, then assigns the value of the parameter y to be the value of the newly created heap object. This transfer function was simply built by concatenating two existing transfer functions; first a new object is created using the object instantiation statement transfer function, then the value of the parameter is assigned to the newly created object using the store statement transfer function. This shows how we can create semantics for new query representing data-types by aliasing and combining the existing string and heap type language constructs. As long as these new data types can be modelled as strings and heap variables, the analysis defined here will be suitable for analysing them.

Another group of data types we must represent, are returned query results. For example consider Line 5 of Listing 3.7, where we execute the query and return the result object. The transfer function for the *SqlCommand.ExecuteReader()* method is:

$$\mathcal{T}[\![x := \text{SqlCommand.Exec}, l]\!](\sigma, h, h^{\text{reg}}) \quad = \quad \mathcal{T}[\![x := new^l]\!](\sigma, h, h^{\text{reg}})$$

This transfer function simply creates a new object. We are not concerned here with the value of the query, only that an object has been returned, which identifies the returned result set. We shall show how this simple information is used to relate the executed query to the place where the results are used, in Chapter 4.

The transfer functions defined in the string analysis, are expressive enough to create most required transfer functions for other data-types. We can create objects on the heap with string values, and manipulate them arbitrarily. This is sufficient for our analysis because any type that represents a query can be represented as a string, and any operation upon that query can simply be replicated as a manipulation of the string-based query. We can also represent other types, such as returned data sets, simply as empty heap items. In Chapter 4 we shall show how the results of this query analysis can be used to predict the impacts of schema change.

3.5 Related Work

There has been work on dynamic analysis for the purpose of impact analysis [Law and Rothermel, 2003, Orso et al., 2004a], however, as dynamic analysis is a fundamentally different approach to static analysis, it is not feasible to investigate both approaches within the scope of this research. We chose to investigate static program analysis because the closest related work was more promising and more suitable for our particular problem. Dynamic analysis must overcome the problem of providing input data with adequate coverage, which is a problem that may involve a static analysis such as the query analysis we have described in this dissertation. Static analysis has the advantage of being able to be performed off-line with guaranteed coverage, therefore we argue that static impact analysis is more suited to this particular impact analysis problem.

For static program analysis we could have chosen to use dataflow analysis, constraint based analysis, abstract interpretation, type and effect systems or some other kind of established program analysis approach, as described by Nielson et al. [1999]. Dataflow analysis is the most suitable representation for our purposes because it allows us to concisely define and implement our analysis, and directly compare it to closely related work. The other program analysis approaches could be used to implement similar analyses, but are more suitable in different contexts and for different purposes. Constraint based analysis is typically used for the analysis of functional programming languages, but the scope of this research only focuses on object oriented programming languages. Abstract interpretation is used to define rigorous analyses where certain mathematically properties need to be guaranteed, but the expense of creating an analysis of this kind is prohibitive in our context, as discussed in Section 2.3.3.

There has been previous work on extracting queries from OO applications; string analysis was used by Gould et al. [2004], to estimate the values of strings passed to the Java JDBC library methods, and to check that the queries were type safe with respect to the database schema. This approach initially seems similar, but the goal of type-checking requires a fundamentally different analysis to our goal of impact analysis. The approach of Gould et al. [2004] is based upon a string analysis that is similar to the approach of Choi et al. [2006] in that it does not satisfy the requirements of context-sensitivity or traceability, that we have described in this chapter. Instead, the focus of this work is on type checking strings with unknown values, rather than accurately tracing the definition of strings to where they are used.

The Gould et al. [2004] analysis can type check automata, where the automata include unknown strings and approximations. This approach could be used to analyse the queries that are extracted using our query analysis, type checking and providing more information about the structure of the queries that are extracted. By combining our approach with the approach of Gould et al. [2004], we would improve the applicability of their analysis to range beyond string data types, and would add further functionality to our query analysis, which could be leveraged by the later stages of our analysis. This is a possible direction for future work, but we note that our query analysis and the work of Gould et al. [2004], are largely complementary to each other, as they address fundamentally different problems.

The string analysis used by Gould et al. [2004] was based on the earlier Java String Analyser (JSA)

created by Christensen et al. [2003]. This approach was shown to be very useful and powerful, but was not intended to be used for impact analysis, in particular the analysis was context-insensitive, and was therefore not suitable for our purposes. The JSA tool was also used as the basis for other approaches which validated SQL query strings for the purpose of finding security vulnerabilities in the form of SQL injection attacks [Halfond and Orso, 2005]. Any work based upon string analysis that is context-insensitive, and does not satisfy our traceability requirements, is not suitable for impact analysis of schema changes, because the analysis could have an unacceptable level of false positives.

A similar analysis to the JSA was the work of Choi et al. [2006], a string analysis that added support for fields and an interesting new widening approach. This widening based string analysis was more suited to our analysis, and as such, used as the basis for our query analysis. The field sensitivity was required for good impact analysis, and the analysis was already shown to be amenable to higher levels of context-sensitivity. The widening approach was also shown to be more precise in many cases, than other related work. For example, in a program similar to Listing 3.2, the JSA analyser predicts a regular expression '$(a + b +' ')^*$', whilst the widening based analysis predicts string of 'ab^*', which is clearly a more precise approximation.

Wiedermann et al. [2008] investigate the analysis of transparent persistence queries in Java. Transparent persistence, is similar to the concept of orthogonal persistence, where persistence is managed automatically by the program environment; in much the same way that memory management and garbage collection automatically manages memory in languages such as Java and C#. Wiedermann et al. [2008] propose a technique called *query extraction*, which uses a path-based analysis to identify the the persistent values that will be accessed on corresponding execution paths. This research has the goal of creating query optimisations for transparent persistence, by pre-loading required data and avoiding unnecessarily large queries. Query extraction is used to analyse transparent persistence, but it is unclear if the results would be useful for impact analysis. Although this work might initially appear similar, it is largely orthogonal, and the query extraction and query analysis, examine very different properties about queries, with little overlap.

We are unaware of any other string analysis, or similar analysis, that solves the requirements we have discussed in this chapter, and whilst other similar analyses exist, none were entirely suitable for our purposes.

In this chapter, we motivated the need for a context-sensitive analysis, of which there are two main approaches, the call-string approach and the functional approach [Sharir and Pnueli, 1981]. The call string approach, uses a string of identifiers that represent the callsites to represent the context. The functional approach uses some other information available at the call site to distinguish context. We chose to use k-CFA [Shivers, 1991], which is an example of the call string approach to context-sensitive interprocedural analysis. k-CFA is well recognised, and suitable for our purposes. We are unaware of any other context-sensitivity approaches which would work as well as the k-CFA approach we have adopted. We argue that, for this type of analysis, context sensitivity is required, even though context sensitivity in other analyses can be insignificant [Lhoták and Hendren, 2006]. The price we pay for using k-CFA is that

it can be very expensive, with possible exponential worst case behaviour, as k increases [Jagannathan and Weeks, 1995]. We shall discuss the costs of this analysis, and how they can be minimised, in Chapter 5.

3.6 Summary

We defined the term *query analysis* to mean a program analysis which can extract query definitions, query executions and the use of query results. For a query analysis to be useful for impact analysis, it must also address the specific requirements for context-sensitivity, traceability and the use of arbitrary database query data-types. In the chapter we have shown why these requirements exist, and we have shown that the closest related work does not define a suitable query analysis.

To address these requirements, we have extended an existing string analysis and added variable levels of context-sensitivity, added extensions to make strings immutable heap variables and shown how arbitrary query data types can be added. This resulting analysis is substantially different from related work and represents a novel contribution.

In the following chapter, we will show how the results of our query analysis can be used to calculate impacts of schema change.

Chapter 4

Impact Calculation

In this chapter we shall explain how the results of the query analysis, described in Chapter 3, can be used to calculate the impacts of schema changes. We shall show how the fixed point solution from a query analysis of a program, can be processed. We call this process *fact extraction*, and the result of the process is a set of facts about the program, describing where queries are defined, where queries are executed and where the results of queries are used. We will show how these facts can be processed and used to calculate the impacts of schema change using a process we call *fact processing*, and the output of this process is an impact report detailing the impacts caused by schema changes.

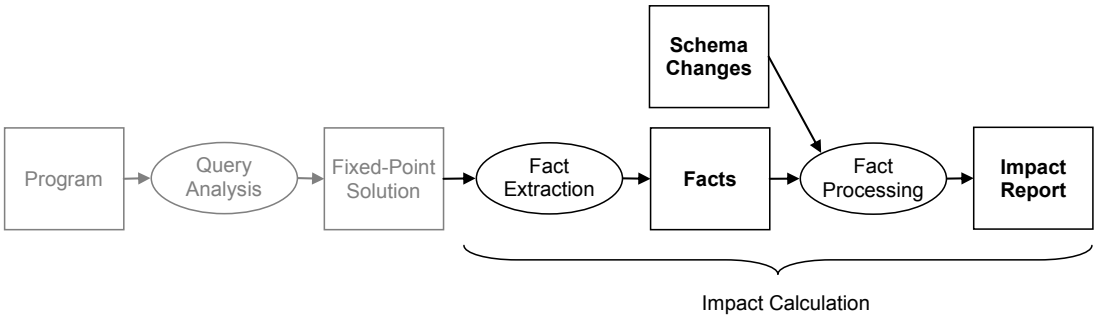

Figure 4.1: Impact Calculation Overview

These two processes are used together to form an approach that we call *impact calculation*, as shown in Figure 4.1. The input to the fact extraction stage is the results of a query analysis, and the output of fact extraction is a set of facts about the target program. Like the query analysis before it, fact extraction is independent of the schema changes and applies to the entire application. We shall describe fact extraction in Section 4.1, describing more formally, facts and how they are calculated.

The inputs to the fact processing stage are the facts resulting from the fact extraction stage, and information about what changes to the schema should be processed. The fact processing stage is the stage where we move from analysing the application in a general context, to analysing within the context of a specific set of schema changes, as driven by the users of the analysis. We shall describe fact processing in Section 4.2, showing how abstract facts can be processed to calculate impacts using first order predicate calculus.

We shall describe how this impact calculation approach applies to our example scenario and the general case in Section 4.3, showing how the analysis can be implemented in an extensible and efficient way.

4.1 Fact Extraction

In Chapter 3 we defined a query analysis that predicts the runtime values and dataflow for query data types in a given program. We shall now describe how we use this fixed-point solution to extract all the information that we require, such as where queries are defined, where they are executed, and where their results are used.

```
1 var selOrderByCustomer = new OleDbCommand("selOrderByCustomer", conn);
2 selOrderByCustomer.Parameters.Add(
3     new OleDbParameter("@customerId", customerId));
4 selOrderByCustomer.CommandType = CommandType.StoredProcedure;
5 return selOrderByCustomer.ExecuteReader();
```

Listing 4.1: Implementation of SALES-Q7

After the query analysis is complete, for any given program, we will have a fixed point solution, where we know the incoming and outgoing flow state for each program statement. Consider the example code shown in Listing 4.1, on Line 5 we have the execution of the query. The instruction that performs this execution will have incoming and outgoing flow states, given by the solution to the analysis. We wish to use these data to find out what effects this statement has upon the queries that the application could perform.

The execution of a call to *OleDbCommand.ExecuteReader()* on Line 5, is represented by a single program instruction in the resulting compiled code. For every context in which this instruction occurs in the resulting fixed point solution, we define a *statement occurrence of interest* (SOI). SOIs consist of a source code location (given by the full path to the file and the line number), a unique identifier and a set of facts. A *fact* is defined as a formula in first order predicate calculus, that consists of a *predicate* and some *terms*, and is assumed to be true[1]. For example the following is a fact:

$$Wasted(youth, young)$$

Here, the predicate $Wasted$ takes two terms, and represents the semantic meaning that the first term is wasted upon the second. The term $youth$ represents the semantic object meaning the English word youth, and the term $young$ represents the semantic object meaning the English word young. Because this formula is a fact, it is always considered to be true, therefore this fact represents the semantics that: youth is always wasted upon the young [Shaw, unknown year].

[1]Facts are not part of first order predicate calculus. We base our notion of facts upon the concept from the programming language Prolog, which is itself based on first order predicate calculus.

The predicates and terms in facts can be arbitrary, but for the remainder of this dissertation we shall consider only facts with two terms, where the first term is always an identifier that represents a SOI, and the second term represents some information that is related to the SOI by the predicate. These SOI facts can be thought of as something interesting that we can observe about a particular SOI.

We represent SOIs and facts more formally, as follows:

SourceLoc	sourceLoc	
Predicate	d	$\in$ { ConcAtLine, Concats, CreatesStr, CreatesStrWithId, ExecutesQuery, ExecutesQueryWithId, Executes, ExecutedAtLine, LdStr, LdStrAtLine, PrmAddName, PrmAddNameWithId, PrmAddType, PrmAddLine, PrmAddTo, ReadsColumn, ReadsColumnWithId, ReadAtLine, ReadsResultObject, ReturnsResult, UsesParams }
Term	t	$\in \{\mathbf{Reg} + \mathbf{RegLoc} + \mathbf{Loc} + \mathbf{SourceLoc}\}$
Fact	e	$\in \mathbf{Predicate} \times \mathbf{Term}$

A sourceLoc is a token that represents a location in the program source code; this would be specified in our analysis using the full path to the file and the line number of the location within the file.

A predicate d, is member of the set $\mathbf{Predicate}$. We define predicates for the constant values shown, and the semantic meanings of these predicates are described in Appendix A, but as an example, the *ConcAtLine* predicate indicates that a concatenation SOI occurred at the location in the program specified by the second term, and the *Concats* predicate indicates that a concatenation SOI involves the concatenation of the regular string identifier specified by the second term. These predicates are not exhaustive and consist of the predicates were identified during the implementation of our analysis. We created change impact analysis scripts for a set of likely schema changes, as we shall discuss in Section 4.2.2. We then created predicates that were suitable for supplying all the information required to perform this required impact calculation. The predicates defined here are suitable for analysing libraries such as ADO.NET, and were suitably expressive to perform the case study described in Chapter 7. We believe that these predicates will be sufficient for the majority of standard data access libraries. These predicates are not sufficient when impact calculation scripts need more information. For example, if we wanted to create an impact analysis script that searched for queries that were in a transaction, we might need a predicate such as *InTransaction* that denotes if a given query is executed within a transaction. We can add more predicates arbitrarily, as required, simply by expanding the definition of the $\mathbf{Predicate}$ set, then they can be used in the same way as the predicates we have already defined, as we shall discuss in the remainder of this section. Our analysis must be able to cope with new and varied DBMS features and different persistence technologies, as specified by *Requirement-4*, Chapter 2. Being able to add new predicates as required, helps satisfy this requirement.

A term is a regular string (r in the query analysis abstract domain), a heap location identifier (l in the query analysis abstract domain) or a regular string heap location identifier (l^{reg} in the query analysis abstract domain). These identifiers are required to establish the traceability relationships between SOIs

(as we shall describe below), whilst the regular strings represent the values of queries at given locations. A member of **Term** will always represent the second term in the fact, as the first term is always specified by the identifier of the SOI to which the facts belong. For example, consider the SOI with the identifier **SOI1**, which has a represented as $\{\text{ExecutesQuery}, "SELECT..."\}$. This representation actually describes the fact:

$$ExecutesQuery(SOI1, "SELECT...")$$

We shall see more examples of facts and SOIs in the remainder of this chapter.

A statement occurrence in the query analysis becomes an SOI if it produces facts. Thus for every type of interesting statement we define a function to discover these facts, of the form:

$$Facts[\![t]\!] : (\sigma \times h \times h^{\text{reg}} \times OriginalReg \times \mathsf{sourceLoc}) \rightarrow \mathcal{P}(\mathbf{Fact})$$

The $Facts$ function, supplied with the outgoing flow state from the fixed point solution, and a source location parameter, will calculate the fact produced by the SOIs of type t. Continuing our example, we define a $Facts$ function for calls to *OleDbCommand.ExecuteReader()* as follows:

$$\begin{aligned} Facts[\![x = \text{call OleDbCommand.ExecuteReader}(y)]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\ \bigcup\{(\text{ExecutesQuery}, p) \mid p \in \text{lookup}(V', h^{\text{reg}})\}+ \\ \bigcup\{(\text{ExecutesQueryWithId}, o) \mid l^{\text{reg}} \in V', o \in O(l^{\text{reg}})\}+ \\ \bigcup\{(\text{Executes}, l) \mid l \in \sigma(y)\}+ \\ \bigcup\{(\text{ReturnsResult}, l) \mid l \in \sigma(x)\}+ \\ \{(ExecuteAtLine, \mathsf{sourceLoc})\} \\ \text{where } V' = \bigcup\{V \mid l \in \sigma(y), h(l) = V^u\} \end{aligned}$$

The function returns a set of facts, based upon the values in the supplied flow state. A fact with the *ExecutesQuery* predicate is created for every string value in the content of the heap variable y; it is assumed that y is the variable containing a reference to the receiver object upon which the method is being called, in this case an instance of the *OleDbCommand* class.

The fact with the *ExecutesQueryWithId* predicate, lists the original strings' identifiers for each string that was found by the *ExecutesQuery* fact. This allows us to trace the constituent strings of each possible query that could be executed here; we shall discuss exactly how this traceability is established using these values, later in Section 4.1.1.

The fact with the *Executes* predicate, is added for every possible query representing heap object that is being executed here, allowing us to trace this query execution to its original definition.

The fact with the *ReturnsResult* predicate, is added for the values of the x variable that represents the query results object, in this case an *OleDbDataReader*. This lets us trace the returned results of the query to the places where they are used.

The fact with *ExecutedAtLine* predicate, is added with the supplied source code location parameter. Almost every SOI will include a similar location fact, so that we can find the place where it occurred in

the original source code.

The where clause for the function contains the value V^u which is used to represent the value of a heap location, consisting of a value with a uniqueness. In this case we are only interested in the value V, so the uniqueness, u, is a free variable that is discarded.

Evaluating this fact extracting function for our example, on Line 5 of Listing 3.7, we would obtain the following:

$$
\begin{aligned}
&\mathbf{QueryExecSOI1} = \\
&\{ \\
&\quad (\text{ExecutesQuery}, \text{"SELECT id, contact\_name, company\_name FROM Supplier WHERE id=@ID;"}), \\
&\quad (\text{ExecuteQueryWithId}, l_1^{\text{reg}}), \\
&\quad (\text{Executes}, l_1), \\
&\quad (\text{ReturnsResult}, l_2), \\
&\quad (\text{ExecutedAtLine}, \text{"}Example.cs : 05\text{"}) \\
&\}
\end{aligned}
$$

These tuples represent facts that provide information about the SOI. We can extract facts by evaluating fact extracting functions, as we have shown for the the *OleDbCommand.ExecuteReader* method, but we will also need to extract the facts for other interesting statements. For example, consider the statement for evaluating a string expression:

$$
\begin{aligned}
&Facts[\![x := (s, l_1^{\text{reg}})]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \{(\text{LdStr}, l_2^{\text{reg}}) | l_2^{\text{reg}} \in V)\}+ \\
&\quad \{(\text{LdStrAtLine}, \mathsf{sourceLoc})\} \\
&\quad \text{where } (V, h_2^{\text{reg}}) = \mathcal{E}[\![s, l_1^{\text{reg}}]\!](\sigma, h^{\text{reg}})
\end{aligned}
$$

This function creates two facts, the first fact relates the SOI to the term using the *LdStr* predicate, and uses the regular-string-heap location-identifier of the newly added string as the term. This first fact simply represents that a constant string has been loaded at this location. The second fact relates the SOI to the term using the *LdStrAtLine* predicate, and uses the source code location of the SOI as the term. This fact simply represents where in the program this SOI occurred.

Just as we have defined fact extraction for calls to *OleDbCommand.ExecuteReader()*, and for string literal instructions, we have also defined fact extraction functions for significant items in the C# language and the .NET framework, as used by our prototype implementation, which we will describe in Chapter 6. The fact extraction functions we have defined, can be found in Appendix A. For any new API or data access features we wish to analyse, we can add new fact extraction functions as required.

4.1.1 Relating Facts

Some of the predicates we have described, are used to create facts for providing traceability to other facts or other SOIs. In Section 3.3.1 we discussed the need for traceability in our analysis, which we can achieve by establishing relationships between the facts of different SOIs. We shall illustrate how facts

can establish these traceability relationships using an example.

The facts extracted from the string definition statement, from Line 1 of the example, are:

$\mathbf{StringDefSOI1} =$
$\{$
$(\text{LdStr}, l_1^{\text{reg}}),$
$(\text{LdStrAtLine}, \text{"Example.cs:01"})$
$\}$

Although not shown in the example code, we shall also introduce the facts extracted from a use of the query results given by the following:

$\mathbf{UseOfResultsSOI1} =$
$\{$
$(\text{ReadsColumn}, \text{"contact\_name"}),$
$(\text{ReadsColumnWithId}, l_2^{\text{reg}}),$
$(\text{Reads}, l_2),$
$(\text{ReadAtLine}, \text{"AnotherFile.cs:123"})$
$\}$

We now have the extracted facts of 3 SOIs $\mathbf{QueryExecSOI1}$, $\mathbf{StringDefSOI1}$ and $\mathbf{UseOfResultsSOI1}$. Using the facts we can establish relationships between these SOIs, showing how they are related to each other, if at all. First, we can relate the $\mathbf{QueryExecSOI1}$ to the $\mathbf{StringDefSOI1}$, by matching the identifiers in the terms of the *ExecuteQueryWithId* predicate and the *LdStr* predicate. By relating these two SOIs we can establish that the query executed at Example.cs Line 5, was originally defined at Example.cs Line 1. Similarly, we can establish a relationship between $\mathbf{QueryExecSOI1}$ and $\mathbf{UseOfResultsSOI1}$, by matching the term of the *ReadsColumnWithId* predicate with the term of the *ReturnsResult* predicate. With this relationship, we can establish where the results of the query are used, and we can also establish information like the name of the column that is read by this use.

By establishing these relationships between SOIs using facts, we can identify the locations where queries are defined, where they are executed and where their results are used. Conversely, if the identifiers cannot be matched, we know that these SOIs are unrelated.

For every interesting statement in the application, we can extract similar facts. The resulting facts define relationships between query definitions, query executions, and the use of query results.

String Concatenation and Widening Facts

In the query analysis described in Chapter 3, the string concatenation and widening operators create new immutable strings in the regular string heap. The operations also record all of the original regular strings that were used to create the new string by creating the appropriate entries in the original regular string mapping. When we are extracting facts, we shall show how important the original regular string mapping is for the sake of traceability, and how without it we would not be able to establish the chain of relationships that can associate a query execution with its definitions.

For example, the code shown in Listing 4.2 shows a program where the SQL query is built dynamically in a loop. Every string in the *tables* collection is added to the SQL string, before the string is

executed.

```
1  var tables = new string[] {"table1", "table2", "table3", "table4"};
2
3  string query = "SELECT * FROM ";
4
5  for (int i = 0; i < tables.Length; i++ )
6  {
7      query += tables[i];
8
9      if (i < tables.Length - 1)
10     {
11         query += ", ";
12     }
13 }
14
15 OleDbCommand cmd = new OleDbCommand(query, conn);
16 cmd.ExecuteNonQuery();
```

Listing 4.2: Concatenation and Widening Traceability Example

If we execute our query analysis and fact extraction on the code shown in Listing 4.2, we might obtain the facts shown in Figure 4.2.

We can see that the *ExecutesQueryWithId* facts for the **QueryExecSOI2** SOI, allow us to establish a relationship with the **StringDefSOI1** SOI, using the term l_3^{reg}. We could expect that this fact should not occur, and that the result of the loop should be a single widened string. Why is the regular string l_9^{reg} not the only string that is recorded using the *ExecuteQueryWithId* fact? This is because, instead of creating *ExecuteQueryWithId* facts only for the string in the *ExecuteQuery* fact, we create *ExecuteQueryWithId* facts for all the possible literal strings that could have created the query string. In this case, this includes the strings l_3^{reg} and l_5^{reg}. We do this because otherwise we could lose traceability when widening occurs. For example, the **StringConcatSOI1** does not have a fact for the term l_3^{reg} because l_3^{reg} is replaced during the the evaluation of the loop, and the widening operator creates a widened string with the identifier l_6^{reg}. The regular string l_6^{reg} would have an unknown definition if we had not stored its constituent strings, and it would be impossible to establish that part of the query was defined on Line 3. For this reason, it is important that the query analysis keep track of the original elements of widened and concatenated strings, and that the fact extraction functions have access to the original regular string mapping.

We can add more fact extraction functions for various libraries or APIs. We can also added new predicates as required, making our analysis extensible, and being able to customise our approach to different persistence technologies.

So far we have shown how we can establish relationships between SOIs. These relationships allow us to link definitions of queries with query executions and the use of query results, but we have not yet

$\mathbf{StringDefSOI2} =$
{
(LdStr, l_3^{reg}),
(LdStrAtLine, "TraceabilityExample.cs:03")
}
$\mathbf{StringDefSOI3} =$
{
(LdStr, l_5^{reg}),
(LdStrAtLine, "TraceabilityExample.cs:011")
}
$\mathbf{StringConcatSOI1} =$
{
(Concats, l_5^{reg}),
(Concats, l_6^{reg}),
(CreatesStr, l_7^{reg}),
(ConcAtLine, "Example.cs:07")
}
$\mathbf{StringConcatSOI2} =$
{
(Concats, l_7^{reg}),
(Concats, l_8^{reg}),
(CreatesStr, l_9^{reg}),
(ConcAtLine, "Example.cs:11")
}
$\mathbf{QueryExecSOI2} =$
{
(ExecutesQuery, "SELECT * FROM .*"),
(ExecuteQueryWithId, l_9^{reg}),
(ExecuteQueryWithId, l_5^{reg}),
(ExecuteQueryWithId, l_3^{reg}),
(Executes, l_3),
(ExecutedAtLine, "$TraceabilityExample.cs : 16$")
}

Figure 4.2: Facts produced from the analysis of Listing 4.2.

LdStr($StringDefSOI2, l_3^{reg}$)

LdStrAtLine($StringDefSOI2$,"TraceabilityExample.cs:03")

LdStr($StringDefSOI3, l_5^{reg}$)

LdStrAtLine($StringDefSOI3$,"TraceabilityExample.cs:011")

Concats($StringConcatSOI1, l_5^{reg}$)

Concats($StringConcatSOI1, l_6^{reg}$)

CreatesStr($StringConcatSOI1, l_7^{reg}$)

ConcAtLine($StringConcatSOI1$,"Example.cs:07")

Concats($StringConcatSOI2, l_7^{reg}$)

Concats($StringConcatSOI2, l_8^{reg}$)

CreatesStr($StringConcatSOI2, l_9^{reg}$)

ConcAtLine($StringConcatSOI2$,"Example.cs:11")

ExecutesQuery($QueryExecSOI2$,"SELECT * FROM .*"),

ExecuteQueryWithId($QueryExecSOI2, l_9^{reg}$),

ExecuteQueryWithId($QueryExecSOI2, l_5^{reg}$),

ExecuteQueryWithId($QueryExecSOI2, l_3^{reg}$),

Executes($QueryExecSOI2, l_3$),

ExecutedAtLine($QueryExecSOI2$,"$TraceabilityExample.cs : 16$")

Figure 4.3: Facts produced from the analysis of Listing 4.2.

shown how these relationships and facts can be used to actually predict the impacts of schema change, which we shall discuss in the following section.

4.2 Fact Processing

Fact extraction, described in the previous section, takes a fixed point solution from a query analysis, and by evaluating a fact extracting function for each SOI, we obtain a set of facts. We can use these facts to predict impacts of schema changes, by analysing the details of the facts, and the relationships that exist between them. We shall show how the set of facts can be examined to predict the effects of schema change, by using first order predicate calculus. We call this part of the impact calculation, *fact processing*.

In our UCL Coffee example, *SchemaChange6* dropped the *Supplier.customer_name* column. How can we examine the facts that we have extracted to find the impacts of this change? For *SchemaChange6*, as defined in Chapter 2, we could ask many different questions to determine the impact, but we shall start with a simple example.

First, we must represent the facts shown in Figure 4.2 using a different representation. Figure 4.2 uses a representation of facts that is suitable for fact extraction functions over the domain of our query analysis. These facts can be alternatively represented in first order predicate calculus as shown in Figure 4.3. Each of these sentences will evaluate to true, as facts are propositions that we know to hold.

Given the facts in Figure 4.3, we can use first order predicate calculus to examine the relationships between these facts, and to ask questions.

$ColumnMatches(x) \leftarrow TermContainsString(x, "contact\_name")$
$AffectedQueries(x) \leftarrow \exists y(ColumnMatches(y) \wedge EXEC\_QUERY(x, y))$

Figure 4.4: Fact processing for 'where is `contact_name` specified?'

Figure 4.4 finds all query execution SOIs where a column `contact_name` is specified. Line 1 finds all nodes that contain `contact_name`. We introduce the predicate $TermContainsString$ that is true if the first term contains a string supplied by the second term. Because the first term, x, is a free variable, all the terms from our previously defined facts will be possible values of x. This defines the $ColumnMatches$ predicate as being true for any term that contains the string "$contact\_name$". These terms will be query values that contain the column.

Line 2 uses the existential quantifier to identify all query SOI identifiers that have a query value which is true for the $ColumnMatches$ predicate; the free variable x is used in the EXEC_QUERY predicate to represent the SOI identifier. This defines the predicate $AffectedQueries$ as being true for any terms that are query SOIs that execute queries that reference the column `contact_name`.

The results of evaluating Figure 4.4, using the facts shown in Figure 4.3, is that the $AffectedQueries$ predicate will only evaluate to true for the term $QueryExecSOI1$. Therefore, we have found that the only query that uses the column `contact_name` in our example, is the query SOI with the identifier $QueryExecSOI1$.

If we wanted to find out where the SOI $QueryExecSOI1$ is defined, we could use the relational formulae shown in Figure 4.5.

$ColumnMatches(x) \leftarrow TermContainsString(x, "contact\_name")$
$AffectedQueries(x) \leftarrow \exists y(ColumnMatches(y) \wedge EXEC\_QUERY(x, y))$
$AffectedQueryRegId(x) \leftarrow \exists y(AffectedQueries(y) \wedge EXEC\_QUERY\_REG\_ID(y, x))$
$AffectedDefOccurences(x) \leftarrow \exists y(AffectedQueryRegId(y) \wedge LDSTR(x, y))$
$AffectedDefLocations(x) \leftarrow \exists y(AffectedDefOccurences(y) \wedge LOCATION(y, x))$

Figure 4.5: Fact processing for 'where is contact_name query defined?'

Evaluating the formulae in Figure 4.5, against the facts shown in Figure 4.3, produces a predicate $AffectedDefLocations$ that evaluates true, only for the term "Example.cs:01". Therefore, we have found that the definition of the Query that uses the `contact_name` column, is defined in the file Example.cs at Line 1.

Alternatively we could also find out where the results of the query are used, by using the formulae shown in Figure 4.6.

Evaluating the formulae in Figure 4.6, against the facts shown in Figure 4.3, produces a predicate $AffectedRetUseLocations$ that evaluates true only for the term "AnotherFile.cs:123". Therefore, we have found that the results of the Query that uses the `contact_name` column are used in the file

$ColumnMatches(x) \leftarrow TermContainsString(x, "contact\_name")$
$AffectedQueries(x) \leftarrow \exists y(ColumnMatches(y) \wedge EXEC\_QUERY(x, y))$
$AffectedQueryReturns(x) \leftarrow \exists y(AffectedQueries(y) \wedge EXEC\_RETURNS\_ID(y, x))$
$AffectedRetUseOccurences(x) \leftarrow \exists y(AffectedQueryReturns(y) \wedge READS(x, y))$
$AffectedRetUseLocations(x) \leftarrow \exists y(AffectedRetUseOccurences(y) \wedge LOCATION(y, x))$

Figure 4.6: Fact processing for 'where is contact_name query used?'

AnotherFile.cs at Line 123.

We can process facts using first order predicate calculus because it provides the expressive power to create complex queries. The resulting predicates can be used to inform us about the facts, but in order to process this information we define *change scripts*.

4.2.1 Impact Calculation Scripts

Impact calculation scripts are small programs that perform fact processing, examining the facts and outputting the resulting information.

The user could examine the facts manually to predict impacts, but this process would be time consuming and repetitive. By creating suites of common queries, in the form of impact calculation scripts, we allow the user to examine the extracted facts and predict impacts more quickly and easily.

Typically we would expect a library of impact analysis of scripts to exist, from which the user can choose. The user would choose to run a selection of impact calculation scripts against the information extracted by the fact extraction stage.

The script performs some fact processing, querying the extracted facts and predicting some impacts for a particular schema change. The user only needs to select which scripts to execute, and supply any parameters required to configure the scripts.

If an impact calculation script is not available for the required change, it can be quite simple to add additional custom scripts. Creating impact calculation scripts for our example implementation, as described in Chapter 6, involves writing the new query as a short program in the RML programming language (described in Section 4.3) and placing the program in a scripts folder. The new script can then be executed against the results of fact the extraction, by using the tool. The impact calculation scripts used by our prototype implementation can be found in Appendix B.

In our example, for *SchemaChange6* we create the script shown in Listing 4.3.

This script, includes first order predicate calculus, as we have discussed already, but also includes a pseudocode implementation of how to process and output the resulting data. The script processes the facts looking for any queries where the table name and column name match the supplied runtime parameters, *Param1* and *Param2*. For any matching queries, we output the definition, execution and use locations.

By running this script against the output of the query analysis for the inventory application, with the parameters *Param1*="Supplier" and *Param2*="contact_name", we would get the output shown in Figure 4.3.

```
1  tableQueries(x) ← TermContainsString(x, Param1)
2  columnQueries(x) ← TermContainsString(x, Param2)
3  AffectedQueries(x) ← ∃y(columnQueries(y) ∧ ExecutesQuery(x, y))∧
4      ∃y(tableQueries(y) ∧ ExecutesQuery(x, y))
5
6  FOR q IN AffectedQueries(x) {
7      PRINT "ERROR: These queries reference a dropped column."
8      ExecutionLocations(x) ∈ ExecutedAtLine(q, x)
9      FOR exec IN ExecutionLocations(x) {
10         PRINT " EXECUTED AT:" + exec
11     }
12
13     AffectedDefinitions(x) ∈ ExecutesQueryWithId(q, x)
14     Result(x) ∈ AffectedDefinitions(x)
15     PrevResult(x) ∈ FALSE(x)
16     WHILE (PrevResult(x) ↔ Result(x)) {
17         PrevResult(x) ← Result(x)
18         AffectedStrConcats(y) ← ∃x(CreatesStr(y, x) ∧ Result(x))
19         Result(x) ← Result(x) ∨ ∃y(Concats(y, x) ∧ AffectedStrConcats(y))
20     }
21
22     AffectedLdStrs(x) ← ∃y(Result(y) ∧ LdStr(x, y))
23     AffectedConcats(x) ← ∃y(Result(y) ∧ Concats(x, y))
24     LdstrLocs(x) ← ∃y(AffectedLdStrs(y) ∧ LdStrAtLine(y, x))
25     ConcLocs(x) ← ∃y(AffectedConcats(y) ∧ ConcAtLine(y, x))
26     DefinitionLocations(x) ← ∃y(AffectedLdStrs(y) ∧ LdStrAtLine(y, x))∨
27         ∃y(AffectedConcats(y) ∧ ConcAtLine(y, x))
28
29     FOR def IN DefinitionLocations(x) {
30         PRINT "  DEFINED AT:" + def
31     }
32
33     AffectedResultLocs(x) ← ReturnsResult(q, x)
34     AffectedReads(x) ← ∃y(AffectedResultLocs(y) ∧ ReadsResultObject(x, y))
35     ResultLocations(x) ← ∃y(AffectedReads(y) ∧ ReadAtLine(y, x))
36
37     FOR use IN ResultLocations(x) {
38             PRINT "  USED AT:" + use
39     }
40 }
```

Listing 4.3: Fact processing for 'where is contact_name query used?'

```
ERROR:These queries reference a dropped column.
 EXECUTED AT:C:\dev\...\Supplier.cs:48
  DEFINED AT:C:\dev\...\Supplier.cs:46
  USED AT:C:\dev\...\Supplier.cs:23
  USED AT:C:\dev\...\Supplier.cs:29
  USED AT:C:\dev\...\Supplier.cs:30
ERROR:These queries reference a dropped column.
 EXECUTED AT:C:\dev\...\Product.cs:98
  DEFINED AT:C:\dev\...\Product.cs:114
  DEFINED AT:C:\dev\...\Product.cs:117
  DEFINED AT:C:\dev\...\Product.cs:120
  DEFINED AT:C:\dev\...\Product.cs:122
  DEFINED AT:C:\dev\...\Product.cs:76
  DEFINED AT:C:\dev\...\Product.cs:79
  DEFINED AT:C:\dev\...\Product.cs:81
  DEFINED AT:C:\dev\...\Product.cs:87
  DEFINED AT:C:\dev\...\Product.cs:89
  DEFINED AT:C:\dev\...\Product.cs:91
  DEFINED AT:C:\dev\...\Product.cs:93
  USED AT:C:\dev\...\Product.cs:27
  USED AT:C:\dev\...\Product.cs:34
  USED AT:C:\dev\...\Product.cs:35
  USED AT:C:\dev\...\Product.cs:36
ERROR:These queries reference a dropped column.
 EXECUTED AT:C:\dev\...\Supplier.cs:64
  DEFINED AT:C:\dev\...\Supplier.cs:59
```

Figure 4.7: Output of running change script shown in Listing 4.3 on UCL Coffee inventory application.

The output shown in Figure 4.3 predicts *err2*, *err3* and *err4* from our example in Chapter 2, with an appropriate error message, and showing the definition, execution and query results usage locations. The convention used in our implementation is to output the following information for each distinct query:

1. Warning or error
2. Descriptive message
3. The predicted query
4. Query execution locations
5. Query definition locations
6. Query results usage locations

Warnings or errors correspond to whether the impact causes a runtime error or not; the message is a descriptive message to help the user understand the impact, and the remaining locations show the source code that could be affected.

The convention for impact calculation information, that we use here, is not prescriptive, and we could potentially output much more information. For example, we could attach confidence ratings to impact reports, stating the probability of an impact being a true positive, or we could include advice about necessary remedial action for this particular type of impact. The impact calculation scripts can easily be modified to produce the necessary output, but we believe that the convention we describe here, defines the minimum amount of information that should be output from impact calculation.

We could also make the impact calculation conservative. The impact calculation shown above, searches the facts to determine likely impacts, but any query that contains unknown or approximated values in the regular strings is not conservatively accounted for. The impact scripts could be made conservative, taking into account any unknown or approximated query values, but this would require much more complex impact calculation scripts. The impact calculation scripts we have shown, and the scripts we shall discuss in the remainder of this dissertation, are not conservative, and we shall discuss how this affects the cost and accuracy of the analysis in Chapter 7, but for now we note that a conservative analysis is not a requirement, and that we believe useful analyses can be made without conservative impact calculation.

4.2.2 Impact Calculation Suites

We can estimate the impacts of a set of database schema changes by combining impact calculation scripts. In the UCL Coffee example, one set of proposed changes is represented by the impact calculation scripts shown in Table 4.1, each producing a number of warnings and errors. The name of the script gives an intuitive description of its function, and each impact calculation script is shown with the parameters it was executed against. The full details of these scripts, and the other scripts required for the example application, are listed in Appendix B. It is clear that the impact calculation scripts named in Table 4.1, are only sufficient for a small number of cases, but what constitutes a sufficient set of scripts? Each different persistence technology or DBMS could require different impact calculation scripts. The range of persistence technologies and DBMSs is too high to account for them all, and they are constantly being updated with new versions and features. Therefore, it is unfeasible to create impact calculation scripts for every possible change, and the alternative is creating a library of scripts for likely changes, or creating a library of scripts for changes in which we are particularly interested.

We should create scripts for the most likely database schema changes, such as additions [Sjoberg, 1993]. Then we can use taxonomies of recommended changes, such as the catalogue of refactorings supplied in the book Database Refactoring [Ambler and Sadalage, 2006], or the schema modification operators of $\mathcal{PRISM}$ workbench tool by Curino et al. [2008]. The work involved in creating such libraries is outside the scope of this research, and our only requirement is that our approach allows the creation of such libraries. As an example, we describe the scripts used for our example application and case study in Appendix B.

Impact Calculation Script	Errors	Warnings
csAddNewOptionalParamsToStoredProc(”insCustomer”)	0	1
csAddNewOptionalParamsToStoredProc(”updCustomer”)	0	1
csAddNewRequiredParamsToStoredProc(”insCustomer”)	1	0
csAddNewRequiredParamsToStoredProc(”updCustomer”)	1	0
csAddOptionalColumns(”Customer”, ”other_names”)	0	0
csAddOptionalColumns(”Supplier”, ”contact_other_names”)	0	3
csAddRequiredColumns(”Customer”, ”first_name”)	0	0
csAddRequiredColumns(”Customer”, ”last_name”)	0	0
csAddRequiredColumns(”Customer”, ”title”)	0	0
csAddRequiredColumns(”Supplier”, ”contact_first_name”)	1	0
csAddRequiredColumns(”Supplier”, ”contact_last_name”)	1	0
csAddRequiredColumns(”Supplier”, ”contact_title”)	1	0
csDropColumnRequired(”Customer”, ”name”)	0	0
csDropColumnRequired(”Supplier”, ”contact_name”)	2	0
csDropRequiredParamsToStoredProc(”insCustomer”)	1	0
csDropRequiredParamsToStoredProc(”updCustomer”)	1	0
csReturnNewDataFromStoredProc(”selCustomersAll”)	0	1

Table 4.1: UCL Coffee example change scripts

4.3 A Practical Implementation of Impact Calculation

The description so far, has been predominantly abstract, but in this section we will describe a more concrete example of how facts can be represented and processed in practice. The implementation we describe here is just one example of how impact calculation could be implemented.

We require a format for representing the SOIs and their facts. We want to be able to reason about these facts in a efficient and flexible way, and to be able to easily add new change scripts as required. We have chosen to use the CrocoPat tool [Beyer et al., 2005] as the basis for our impact calculation implementation, as it satisfies these requirements.

CrocoPat is a tool for relational programming; it is efficient, being based on binary decision diagrams (BDDs) [Bryant, 1986] for efficient graph based querying, and it is expressive, being able to query arbitrary graph based data, using scripts based upon first order predicate calculus. There are other tools and approaches that could be used to perform impact calculation, and we shall discuss some of these in Section 4.4, but for the remainder of this chapter we shall discuss our chosen implementation based upon the CrocoPat tool.

The input to CrocoPat must be in the Rigi Standard Format (RSF) [Wong, 1998]; this is a text based file format for representing graph based data, which lends itself well to representing facts. Each line consists of three optionally quoted values, the first value representing an edge label, and the second and third values, representing start and end nodes respectively.

```
1  EXEC_QUERY              QueryExecOccurence1   "SELECT id, contact_name, ..."
2  EXEC_QUERY_REG_ID       QueryExecOccurence1   reg1
3  EXEC_ID                 QueryExecOccurence1   loc1
4  EXEC_RETURNS_ID         QueryExecOccurence1   loc2
5  LOCATION                QueryExecOccurence1   "Example.cs:04"
6
7  LDSTR                   StringDefOccurence1   reg1
8  LOCATION                StringDefOccurence1   "Example.cs:01"
9
10 READS_COLUMN            UseOfResultsOccurence1   "contact_name"
11 READS_COLUMN_REG_ID     UseOfResultsOccurence1   reg2
12 READS                   UseOfResultsOccurence1   loc2
13 LOCATION                UseOfResultsOccurence1   "AnotherFile.cs:123"
```

Listing 4.4: Example RSF for QueryExecSOI1

Listing 4.4 shows how the facts defined in Figure 4.3 can be represented in RSF. Line 1 shows how a node representing the SOI identifier *QueryExecSOI1*, is linked, via an *EXEC_QUERY* edge, to a node representing the query value. Listing 4.4 shows the value of the query as a truncated value, using an ellipsis to show the missing text, however the full query in the RSF would be:

"SELECT id, contact_name, company_name FROM Supplier WHERE id=@ID"

Every other fact, is represented in a similar way, by creating an edge labelled with the name of the predicate, from the SOI identifier, to the second term of the fact.

Relation Manipulation Language (RML) is CrocoPat's language for processing relational data. RML is small language for querying and processing relational data, designed to be efficient and expressive. The RML language is based upon predicated calculus with extensions for standard programming features like conditional expressions, loops and input/output. Because RML is based upon predicate calculus it is easy to translate the examples we have seen so far, into RML. For example, the change script shown in Listing 4.3, can be represented in RML as the program shown in Listing 4.5. We shall not describe the syntax and semantics of RML here, for the sake of brevity, and refer interested readers to Beyer et al. [2005].

```
1  Seperator          :=   "($|^|[^a-zA-Z0-9]+)";
2  columnQueries(x)   :=   @Seperator + $2 + Seperator(x);
3  tableQueries(x)    :=   @Seperator + $1 + Seperator(x);
4
5  AffectedQueries(x) :=  EX(y, columnQueries(y) & ExecutesQuery(x, y)) &
6                         EX(y, tableQueries(y) & ExecutesQuery(x, y));
7
8  FOR q IN AffectedQueries(x) {
9
10     PRINT "ERROR:";
11     PRINT "These queries reference a dropped column.", ENDL;
```

```
12
13      ExecutionLocations(x) :=  ExecutedAtLine(q, x);
14
15      FOR exec IN ExecutionLocations(x) {
16         PRINT "␣EXECUTED␣AT:";
17         PRINT exec, ENDL;
18      }
19
20      AffectedDefinitions(x) := ExecutesQueryWithId(q, x);
21
22      // Find fixed point for all concats, and widenings
23
24      Result(x) := AffectedDefinitions(x);
25      PrevResult(x) := FALSE(x);
26      WHILE (PrevResult(x) != Result(x)) {
27          PrevResult(x) := Result(x);
28          AffectedStrConcats(y) :=
29            EX(x, CreatesStr(y, x) & Result(x));
30          Result(x) := Result(x) |
31            EX(y, Concats(y, x) & AffectedStrConcats(y));
32      }
33
34      AffectedLdStrs(x) := EX(y, Result(y) & LdStr(x, y));
35      AffectedConcats(x) := EX(y, Result(y) & Concats(x, y));
36
37      LdstrLocs(x) := EX(y, AffectedLdStrs(y) & LdStrAtLine(y, x));
38      ConcLocs(x) := EX(y, AffectedConcats(y) & ConcAtLine(y, x));
39
40      DefinitionLocations(x) :=
41            EX(y, AffectedLdStrs(y) & LdStrAtLine(y, x)) |
42            EX(y, AffectedConcats(y) & ConcAtLine(y, x));
43
44      FOR def IN DefinitionLocations(x) {
45         PRINT "␣␣DEFINED␣AT:";
46         PRINT def, ENDL;
47      }
48
49      AffectedResultLocs(x) := ReturnsResult(q, x);
50      AffectedReads(x) :=
51        EX(y, AffectedResultLocs(y) & ReadsResultObject(x, y));
52      ResultLocations(x) :=
53        EX(y, AffectedReads(y) & ReadAtLine(y, x));
```

```
54
55      FOR use IN ResultLocations(x) {
56                  PRINT "  USED AT:";
57                  PRINT use, ENDL;
58      }
59  }
```

Listing 4.5: RML for 'where is contact_name query used?'

Executing the script in Listing 4.5 against the RSF extracted from our sample application[2], produces exactly the same output as we predicted in Figure 4.7.

We can arbitrarily query the RSF because the RML language provides the same expressive power to create complex queries found in first order predicate calculus, and we can use the results to create arbitrary text output. With this approach we can implement all the impact calculation that is necessary.

4.4 Related Work

We chose to represent the facts related to each SOI using RSF, but several alternative solutions exist. We could have stored to the facts in a relational database, and queried them using SQL. We could have stored the facts using XML and queried them using XPATH[Bray et al., 1998]. We could have stored the facts using in-memory C# objects and queried them using LINQ[Meijer et al., 2006]. Many possible choices exist, and the only requirements are that the approach is extensible, expressive and efficient enough to be useful. We believe that the use of CrocoPat, using RSF to represent facts and RML to create impact calculation scripts, satisfies these requirements, and provides a good choice for impact calculation, although no particular approach is prescriptive, or more suitable in the general case. The choice of an impact calculation implementation technology should be made based upon the trade-offs present in the design of the overall impact analysis tool. For our prototype impact analysis tool implementation, described in Chapter 6, the CrocoPat approach was very suitable.

Sjoberg [Sjoberg, 1993] conducted a case study of the development of a database over an 18 month period. The database was used in several UK hospitals, and by several applications; it reached a maximum of size of 55 tables with 666 constituent columns. The conclusions of this study were that additions and deletions were more common than renamings, but this study tells us little about the importance of these changes. The types of changes that were identified by this study are basic additions, deletions or alterations of tables and columns, and the only conclusion we can draw from this, related to impact calculation, is that our approach should be able to cope with all the types of change that were observed in the study. One interesting question arising from this research is; why are additions and deletions more common than renamings? We could presume that this is because the impacts of renamings are more difficult to deal with, although we have no direct evidence to support this claim. An interesting future research direction could be to test this claim, by comparing database development with and without impact analysis, and see if the presence of impact analysis tools affects the types of changes that are made

[2]This RSF is not included here for the sake of brevity as it is over one thousand lines long.

to the schema. Will the schema be changed in more complex ways, or more often with impact analysis tools, i.e. do developers currently avoid changing the schema? We identify this as a possible direction for future research.

The book Database Refactoring by Ambler and Sadalage [2006], details the latest methodologies for dealing with database schema change in highly iterative software engineering practices. It encourages the use of refactorings, which are small, semantics preserving changes, that improve the non-functional properties of the system. Each of these refactorings is named and described in detail, giving details about the reasons for using the refactoring and the effects it might have. Because the refactorings are clearly named, and well described, implementing impact calculation scripts for these refactorings could be very useful; the terms define a useful vocabulary for talking about schema changes, and they are well-described and will be understood by many developers and DBAs. The only drawback to this work is that it focuses on refactorings, and does not describe as many non-semantics preserving schema changes. The refactorings described by this book, however, are strong candidates for including in any impact calculation script library.

The schema modification operators (SMOs) of the $\mathcal{PRISM}$ workbench tool by Curino et al. [2008], define another set of possible schema change classifications. This work defines a language, based upon SMOs, to express changes to the schema, which has been applied to large real-world case studies. Therefore, we expect that the changes that are expressible by the $\mathcal{PRISM}$ workbench, are a good example of changes that could occur to a schema in practice. These changes could be used as yet more guidance for creating suitable impact calculation script libraries. We shall discuss how the $\mathcal{PRISM}$ workbench tool relates to our analysis as a whole, in Chapter 6.

The impact analysis we have described in this chapter is highly dependent upon the results of the query analysis defined in Chapter 3, therefore there is little directly related work. The closest related work to assessing impacts is the traditional impact analysis work, using dependence analysis or transitive closure algorithms [Bohner and Arnold, 1996]. We shall discuss such work in general, as a comparison to our overall impact analysis approach in Chapter 6.

4.5 Summary

In this chapter we have shown how the results of a query analysis can be used to calculate the impacts of schema changes. We have shown how the fixed point solution of the query analysis can be used to find SOIs and facts, in a process called fact extraction. We then showed how these SOIs and facts can be processed to find potential impacts using a process we called fact processing. We then showed how such an approach can be used to predict impacts by creating impact calculation scripts. We showed how facts resulting from fact extraction can be practically represented, and how the fact processing of these data can be efficiently and practically implemented using the CrocoPat tool.

We have argued that it is not possible to define a complete set of impact calculation scripts, meaning that we must build up libraries of impact calculation scripts as required. It is likely that we will have libraries of impact calculation scripts for different DBMS vendors and for specific query analysis implementations. For example the impact calculation scripts for SQL Server or Oracle databases could be

very different, having to deal with the vendor specific differences in the SQL based query languages.

This necessitates an extensible approach, with the ability to create impact calculation scripts for all required scenarios. We also require an efficient implementation, as the amount of data produced by the query analysis could be very large, and the impact calculation scripts could be complex. We believe that the approach outlined in this chapter, with a practical implementation using a tool such as CrocoPat, achieves these goals.

Chapter 5

Efficient Analysis

The query analysis, presented in Chapter 2, can be expensive, because we require high levels of impact-precision and accuracy to reduce the amount of false positive predictions. For example, we identified that the context-sensitivity of the analysis should be variable, and that using the standard k-CFA approach, we require $k >= 2$, but it has been shown that k-CFA analysis can be exponentially expensive where $k > 0$ [Jagannathan and Weeks, 1995], which could potentially render our analysis unfeasible in practice.

We wish to perform a query analysis with high accuracy and impact-precision, as described by our requirements, but with acceptable time and space costs, and in a scalable manner. To achieve this, we can do two things. First, we can make the query analysis itself as computationally inexpensive and scalable as possible. We do this by implementing two optimisations, liveness analysis described in Section 5.1.1 and abstract garbage collection described in Section 5.1.2. While we do not claim that either of these techniques are novel, they are important to describe, as they provide the groundwork for an efficient implementation of the query analysis that we evaluate in Chapter 6 and Chapter 7. The second approach to improving the efficiency of the analysis is creating optimisations that reduce the parts of the program that we need to use the query analysis upon; this is described in Section 5.2.

Although there are numerous ways optimise query analysis, we describe these particular optimisations because they had the most dramatic impact upon our prototype tool. Abstract garbage collection and liveness analysis were developed as a response to particular problem points in the prototype application. By profiling the application, we found that flow states were becoming large and much of the time was spent comparing them to each other. This observation lead us to realise that reducing the size of flow states wherever possible, could dramatically improve the performance of our tool, and these two optimisations provided the most significant reductions in flow state size.

The optimisation that we shall describe in Section 5.2, was not based upon a particular problem area, but was largely envisaged before our prototype tool was developed. It was clear that query analysis would be expensive, and it became appparent that any uneccessary analysis should be avoided. Therefore, techniques to reduce the parts of the application that need to be analysed were always seen to be an important part of our research.

5.1 Efficient Query Analysis

The solutions to dataflow analyses, as described in Chapter 3, are calculated by highly iterative algorithms that merge, duplicate, update and compare flow states. The larger and more complex these flow states are, the more expensive the merge, duplication, update and compare operations become, and even a small increase in the average size of the flow states, can lead to a large decrease in performance. Often, dataflow analyses are specified or implemented in a way that leads to unnecessary items in the flow state, such as items that have been used previously but will no longer be required in the remainder of the analysis. If we can remove some of these extraneous items, and keep the flow states small, and minimal, then the memory and computational efficiency of the algorithm can be increased. The goal of the approaches in the following two subsections is to remove unnecessary items from the flow state.

5.1.1 Cleaning Dead State

A variable is defined as *live* if its value will be used at some future point in the program; conversely, if the value of a variable will not be used at some future point, the variable is said to be *dead*. Liveness analysis, or live variable analysis, is a classic program analysis that determines the liveness of variables at any given point in a program [Aho et al., 2006, Nielson et al., 1999], and is commonly used in compilers for purposes such as register allocation.

If a variable that represents a string or query is dead at some point in a program, then it may also be considered dead by our query analysis. If the variable will be not be used by the concrete semantics, then the abstract semantics of our query analysis will also not use the variable. If we can remove dead variables from the flow state, then we can reduce the size of the flow state and achieve a faster and more efficient query analysis.

For example, the code listed in Listing 5.1 composes a dynamic SQL string, and then prints it to the console. The query searches the `Products` table, and has a dynamic *WHERE* clause, which searches for product names that contain any of the supplied strings in the *args* parameter. As specified in Chapter 3, the query analysis will create an abstract state containing entries for the string variables *table* and *sql*. During the iteration of Lines 6-13, the abstract state will be compared to previous states and copied or merged multiple times, as the widening of the *sql* variable occurs. The abstract state, from Line 3 onwards, will always contain a value for the *table* variable, even though it will not be used beyond this point in the program. After Line 4 we can remove the *table* variable from the abstract state because it is dead. This process of cleaning dead variable from the abstract state can be defined as follows:

$$\text{CleanDeadState}(\sigma, \text{live}) = \bigsqcup(\sigma(v)|v \in \text{live})$$

This function takes an abstract state (σ), and the results of a liveness analysis (live), and returns the least upper bound of the abstract state, containing only variables that are still live. The liveness analysis can be efficiently calculated using standard live-variable analysis algorithms, as described by Aho et al. [2006], Nielson et al. [1999].

We could perform the CleanDeadState function on the abstract state after every statement, but intraprocedural analysis algorithms typically operate at the level of basic blocks; therefore it will often

```
1  static void Main(string[] args)
2  {
3      string table = "Products";
4      string sql = "SELECT * FROM " + table;
5
6      for(int i=0; i < args.Length; i++)
7      {
8          if (i == 0)
9              sql += " WHERE ";
10         sql += "name LIKE '%" + args[i] + "%'";
11         if (i + 1 < args.Length)
12             sql += " OR ";
13     }
14
15     sql += ";";
16     Console.WriteLine(sql);
17 }
```

Listing 5.1: Dead State example

be more practical to clean the dead state before calculating the outset of each basic block.

We show how the cleaning of dead state affects the size of the flow state in Chapter 7. It is clear that the cleaning function is simple and can be implemented in an efficient way, so we would expect this almost always provide improved performance.

5.1.2 Abstract Garbage Collection

In object-oriented programming languages, *garbage collection* is the process of de-allocating allocated regions from memory that are no longer required by the program [Jones and Lins, 1996]. This is used by programming languages to manage memory automatically, relieving the programmer from the burden of having to manually allocate and deallocate memory. Garbage collection can be achieved in many ways, but typically involves removing objects from memory that are no longer reachable by any variables in the local or global scope.

The abstract domain for our query analysis, as defined in Chapter 3, has an abstract state and abstract heap that are abstract representations of the memory in the concrete domain. There are many situations in which the items contained in the abstract heap, regular string heap and original regular string mapping, will no longer be used by the analysis. These unused items can be removed by a process similar to garbage collection, reducing the size of the flow states in our dataflow analysis, and increasing performance.

For example, Figure 5.2 shows code that executes the INV-Q3 query, defined in Chapter 2. The incoming flow state for Line 6 is:

```
1 SqlCommand command = db.Prepare(insertStatement);
2
3 command.Parameters.Add(new SqlParameter(PRM_SUPPLIER_ID, supplierId));
4 command.Parameters.Add(new SqlParameter(PRM_NAME, name));
5
6 object result = command.ExecuteScalar(); // INV–Q3 executed
7 return Decimal.ToInt32((decimal)result);
```

Listing 5.2: Garbage collection example

$$
\begin{aligned}
\sigma &= \{\text{command} \rightarrow l_1\} \\
h &= \{l_1 \rightarrow \{l_1^{\text{reg}}\}\} \\
h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"INSERT INTO ..."}\} \\
O &= \{...\}
\end{aligned}
$$

The ellipses in the flow state represent data that have been omitted for brevity. The abstract state contains a variable referencing an item on the heap that represents the *SqlCommand* object. The *command* variable is dead on Line 7, so between Line 6 and Line 7 it is removed from the state by the dead state removal optimisation described in the previous section.

The incoming flow state for Line 7 is:

$$
\begin{aligned}
\sigma &= \{\text{result} \rightarrow l_2\} \\
h &= \{l_1 \rightarrow \{l_1^{\text{reg}}\},\ l_2 \rightarrow \{\text{nil}\}\} \\
h^{\text{reg}} &= \{l_1^{\text{reg}} \rightarrow \text{"INSERT INTO ..."}\} \\
O &= \{...\}
\end{aligned}
$$

The abstract state contains the *result* variable, which references an item on the heap with a nil value. Because the *command* variable was removed from the state, the l_1 location identifier on the abstract heap is no longer referenced by anything in the state, it is left dangling. An item in the abstract heap is reachable if the location identifier is in the range of the abstract state. Also if there is a reachable item on the heap, any heap identifiers contained in the contents of the heap item are also reachable. Heap items can be transitively reachable in this way, because the heap contents can contain pointers to the location identifiers of other items on the heap. Any items that are unreachable, are extraneous to the analysis, using up memory and add unnecessary calculations to the query analysis. Therefore, in abstract garbage collection we remove all unreachable items from the abstract heap, to improve efficiency.

We can formalise this as follows:

$$
\begin{aligned}
\text{GarbageCollect}: \quad & (\mathbf{State} \times \mathbf{Heap} \times \mathbf{RegHeap} \times \mathbf{OriginalReg}) \rightarrow \\
& (\mathbf{Heap} \times \mathbf{RegHeap} \times \mathbf{OriginalHeap})
\end{aligned}
$$

$$
\begin{aligned}
\text{GarbageCollect}(\sigma, h, h^{\text{reg}}, O) \quad = \quad & (h/V,\ h^{\text{reg}}/V',\ O/V') \\
& \text{where } V' = \begin{array}{l} \bigcup\{l_2^{\text{reg}} | l_1^{\text{reg}} \in V, l_2^{\text{reg}} \in \mathbf{TC}_O(l_1^{\text{reg}})\} \cup \\ \bigcup\{l_4^{\text{reg}} | l_3^{\text{reg}} \in \sigma(x), x \in dom(\sigma), l_4^{\text{reg}} \in \mathbf{TC}_h(l_1^{\text{reg}})\} \end{array} \\
& \text{where } V = \bigcup\{l_2 | l_1 \in \sigma(x), x \in dom(\sigma), l_2 \in \mathbf{TC}_h(l_1)\} \\
& \text{where } \mathbf{TC}_O = \text{transitive closure function for } l^{\text{reg}} \text{ values in } O \\
& \text{where } \mathbf{TC}_h = \text{transitive closure function for } l^{\text{reg}} \text{ values in } h
\end{aligned}
$$

The abstract heap is restricted by V, where V represents the set of all heap location identifiers that are reachable from the variables in the abstract state. The regular string heap, and the original regular string mapping, are restricted by V', where V' represents the set of regular string location identifiers that are reachable from values in the abstract state and the values of any of the reachable heap objects. By restricting the domains of these mappings, we perform our abstract garbage collection, removing any items that are unreachable.

When abstract garbage collection is applied to the interprocedural version of our query analysis, it is assumed that the state contains static or global variables. If we include all static and global variable in the abstract state, then their values will be counted as reachable, because otherwise, items that are only reachable by static or global variables could be removed erroneously.

The abstract garbage collection could be applied after every transfer function, but this may be unnecessarily often. There are various strategies by which abstract garbage collection could be triggered. These are related to strategies for when to call real garbage collection. We suggest that calling the garbage collection periodically is sufficient, as we shall discuss in Chapter 7. The only part of the abstract garbage collection that could be expensive, is the transitive closure functions, and calling at very frequent intervals may become slow when the flow states are large. Calling the analysis periodically is a simple approach that provides a good performance increase, as we shall discuss in Chapter 7.

We show how the abstract garbage collection affects the size of the flow state in Chapter 7, and we shall show that it can be implemented in an efficient way, with only a small time and space cost.

5.2 Program Slicing

The previous sections described how to improve the efficiency of the dataflow analysis by eliminating unused items in the flow state, thereby improving the performance of the query analysis itself. In this section we take a complementary approach, and investigate a technique for reducing the amount of query analysis that needs to be done. We propose a technique for eliminating parts of the program that do not need to be included in the query analysis, as they will not affect the impact analysis results. If this technique has a lower time and space cost than the gained reduction in execution time of memory usage, then we will get an overall improvement in the effiency of the query analysis. Likewise the reduction technique could lead to a more scalable query analysis. We shall discuss this in more detail in Chapter 7.

5.2.1 Background

We wish to find out which parts of the program are involved in database queries. If there are parts of the program that will never affect the value of a query, and will never use the results of a query, then it is

unnecessary to include them in our analysis.

To discover which parts of the program are related to database queries we use *program slicing*:

> A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. [Tip, 1994]

Program slicing was originally proposed by Weiser [1979], and used for program debugging. Since this initial work, slicing has been used for a wide range of applications [Tip, 1994, David Binkley, 2004].

```
1  int n = Console.Read();
2  int i = 1;
3  long sum = 0;
4  long product = 1;
5
6  while (i <= n)
7  {
8      sum = sum + i;
9      product = product * i;
10     i = i + 1;
11 }
12
13 Console.WriteLine(sum);
14 Console.WriteLine(product);
```

Listing 5.3: Slicing example code

The example code shown in Listing 5.3, shows a simple program that reads from the command line a number, and then loops, calculating two values which are printed to the console[1]. The control flow graph (CFG) for this program is shown in Figure 5.1. The nodes of a CFG represent statements, expressions, or instructions in the program, and the edges represent the possible execution paths that exist between these nodes. This control flow graph can be used to establish dependencies between nodes.

A reaching definition of a node is a definition instruction whose target variable may reach that node without being redefined [Aho et al., 2006]. For example, Node 6 is data dependent on Node 3, because Node 6 uses variable i, for which Node 6 is a reaching definition. Node j is *data-dependent* on Node i if, the definition of x at Node i is a reaching definition [Aho et al., 2006] for Node j[2].

A node i post-dominates a node j if every path from i to the Stop node, contains j. A node j is *control dependent* on a node i, if there is a path from i to j such that j post-dominates every node in the path(excluding i and j), and, i is not post-dominated by j. For example, Node 10 is control dependent on Node 6, because the path between Node 6 and Node 10 does not contain any extra nodes that are

[1] Listing 5.3 is based on Figure 1. from Tip's program slicing survey [Tip, 1994]

[2] This type of data-dependency can also be known as flow dependence [Tip, 1994] and other definitions of data-dependence for slicing exist [Orso et al., 2004b, Ranganath et al., 2007], but the differences are not relevant here.

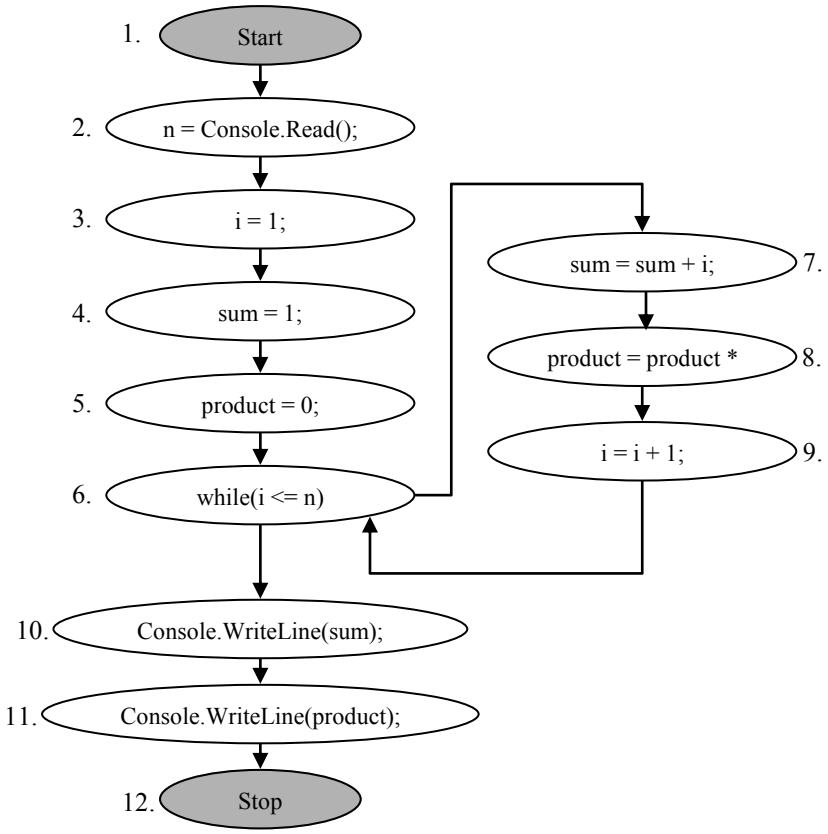

Figure 5.1: CFG for Listing 5.3

not post dominated by Node 10, and Node 6 does not post dominate Node 10. This can be explained alternatively by saying Node 6 controls the branch that dictates whether the execution will pass through Node 10 or not [3].

Data-dependence and control-dependence can be used to build a procedure dependence graph (PDG[4]) [Ferrante et al., 1987, Kuck et al., 1981]. PDGs are useful, as they can be used to accurately and efficiently implement program slicing, however; other techniques do exist and we shall discuss alternative program slicing techniques in Section 5.3.

A PDG for Listing 5.3 is shown in Figure 5.2. It consists of a procedure entry node, plus a node for each instruction, statement or expression. The edges connecting the nodes represent data dependence and control dependence relationships, the dotted lines showing data dependencies and the solid lines showing control dependence. Nodes that will be executed every time the procedure is run, are control dependent upon the procedure entry node. This is the most basic form of PDG, and many other variations exist, usually with additional information represented [Tip, 1994]. We shall discuss how the PDG is changed for interprocedural and object-oriented programs, but first we will discuss how the PDG is used for program slicing.

The slicing criteria, for PDG based slicing, are nodes in the PDG. A backwards slice, includes all

[3]This type of control-dependency can also be defined in other ways [Tip, 1994, Ranganath et al., 2007], but again, the differences are not relevant here.

[4]Sometimes also called the program dependence graph.

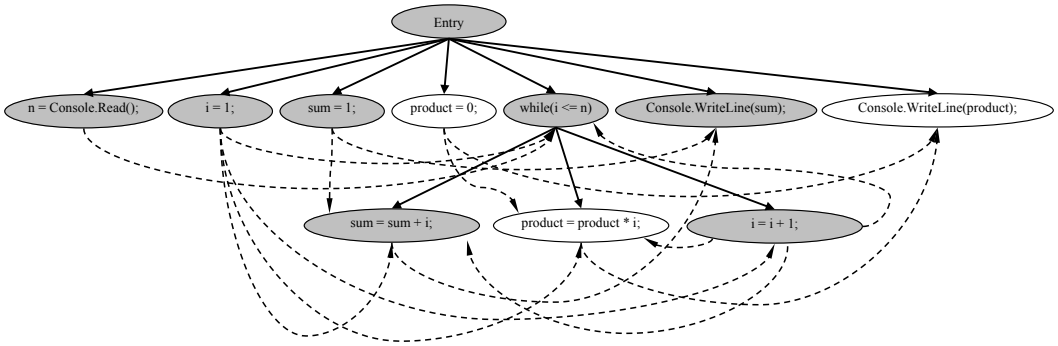

Figure 5.2: PDG for Listing 5.3

nodes that can reach the slicing criteria node(s). The nodes which can reach the criteria, are all part of the slice because they have some dependency relationship with the criteria. Conversely, a forward slice includes all nodes that are reachable from the criteria. For example, consider a backwards slice on the node labelled $Console.WriteLine(sum)$; the grey nodes in Figure 5.2 show the nodes that become part of the slice, as they all reach the criterion node. This slice corresponds to the lines (1,2,3,6,8,10,13), highlighting all parts of the program which might affect the value of the *sum* variable. The PDG, shown in figure 5.2, represents a program consisting of a single procedure, but when we need to slice programs with multiple procedures, it is not so simple.

Slicing a single PDG is very simple, and it might not be clear why the PDG is needed at all. The original slicing approaches were based upon dataflow analysis [Weiser, 1979]. However, problems were identified with this approach, making interprocedural slicing inaccurate. Because of this, PDG based slicing, and other similar approaches, are now more commonly used [Horwitz et al., 1988]. We shall explain these problems and describe interprocedural slicing in more detail, in the following section.

Interprocedural Program Slicing using the SDG

To extend the PDG based slicing approach to be interprocedural, we can use one of the most well-known slicing techniques, the approach of Horwitz et al. [1988]. This approach involves creating a system dependence graph (SDG), computing interprocedural information and a two-pass algorithm for extracting a slice from an SDG.

The System Dependence Graph

The SDG is a representation of a program that can capture interprocedural dependencies. The SDG is derived from the PDG of each procedure in the program, with some extra information added to capture the call and return semantics for the procedures of the program.

For each procedure call in a program, the SDG contains a *call* node. Each call node is associated with nodes that represent the actual parameter values passed to the call, *actual-in* nodes, and any out-parameter, or returned values, *actual-out* nodes. The actual-in and actual-out nodes are represented as control dependent children of the call node.

Each PDG entry node is also associated with *formal-in* and *formal-out* nodes, which represent the

formal parameters that are passed to the procedure, and the parameters and return values that are output, respectively. The formal-in and formal-out nodes are represented as control dependent children of the procedure entry node.

The individual PDGs are linked by three new kinds of edge, that represent interprocedural dependencies. First, each actual-in node is joined to the corresponding formal-in node in its target PDG, by a *parameter-in* edge. Second, each formal-out node is joined to each of its corresponding actual-out nodes by a *parameter-out* edge. Third, each call node is joined to the procedure entry node for the called procedure by a *call* edge. In fact, we classify the parameter-out and parameter-in edges as types of call edge, where call edges are any edges that interprocedurally connect PDGs. The SDG also has one more kind of edge, the *summary edge*, that represents transitive interprocedural dependencies between actual-in and actual out nodes, but we shall describe this type of edge and its use in more detail, later in this section.

Listing 5.4 shows an interprocedural version of the code in Listing 5.3. The SDG for this program is shown in Figure 5.3. Both Listing 5.4 and Figure 5.3 are based upon an example by Tip [1994]. Here we see the addition of call, actual-in and actual-out nodes at each call site, and we can see how these are linked to the corresponding formal-in, formal-out and procedure entry nodes. In the SDG shown in Figure 5.3, the calls to *Console.Read()* and *Console.WriteLine()*, are treated as normal instructions, and not as separate call sites, for simplicity.

Summary Edges and Calling Context

The main problem with computing a program slice for interprocedural programs, is the problem of correctly modelling the call-return structure of the program in order to give precise slices.

Horwitz et al. [1990] identified problems with existing interprocedural slicing approaches, showing that existing approaches over-approximated the call-return structure of a program. The problem is that each procedure can be called in different contexts, and the control-flow of the call site to the return site will be different in each context.

For example, the call to the *Add* procedure on Line 10 of Listing 5.4, will execute the procedure and always return execution to Line 10. Slicing algorithms developed prior to the work of Horwitz et al. [1990], were unable to distinguish this control flow correctly, and could over-approximate the possible return sites to include lines 12, 26 and 27. If this happens, the slice may be larger than necessary. The problem of only including the correct return sites and calling state, has been termed the *calling context problem*[Horwitz et al., 1990].

The summary edge, in conjunction with the SDG slicing algorithm we shall describe next, is used to alleviate the calling context problem. A summary edge represents that an input parameter may influence the value of an out parameter. Summary edges, therefore, only ever exist between actual-in nodes and actual-out nodes. The calculation of these edges was initially expensive, but more efficient algorithms for computing them, have been developed [Reps et al., 1994]. The SDG in Figure 5.3 shows summary edges using the bold lines. We shall describe how summary edges can be used in slicing algorithms, to avoid the calling context problem, next.

```
1  static void Main(string[] args)
2  {
3      int n = Console.Read();
4      int i = 1;
5      int sum = 0;
6      int product = 1;
7
8      while (i <= n)
9      {
10         sum = Add(i, sum);
11         product = Multiply(i, product);
12         i = Add(i, 1);
13     }
14
15     Console.WriteLine(sum);
16     Console.WriteLine(product);
17 }
18
19 private static int Multiply(int c, int d)
20 {
21     int j = 1;
22     int k = 0;
23
24     while (j <= d)
25     {
26         Add(k, c);
27         Add(j, 1);
28     }
29
30     return k;
31 }
32
33 private static int Add(int a, int b)
34 {
35     return a + b;
36 }
```

Listing 5.4: InterproceduralCode

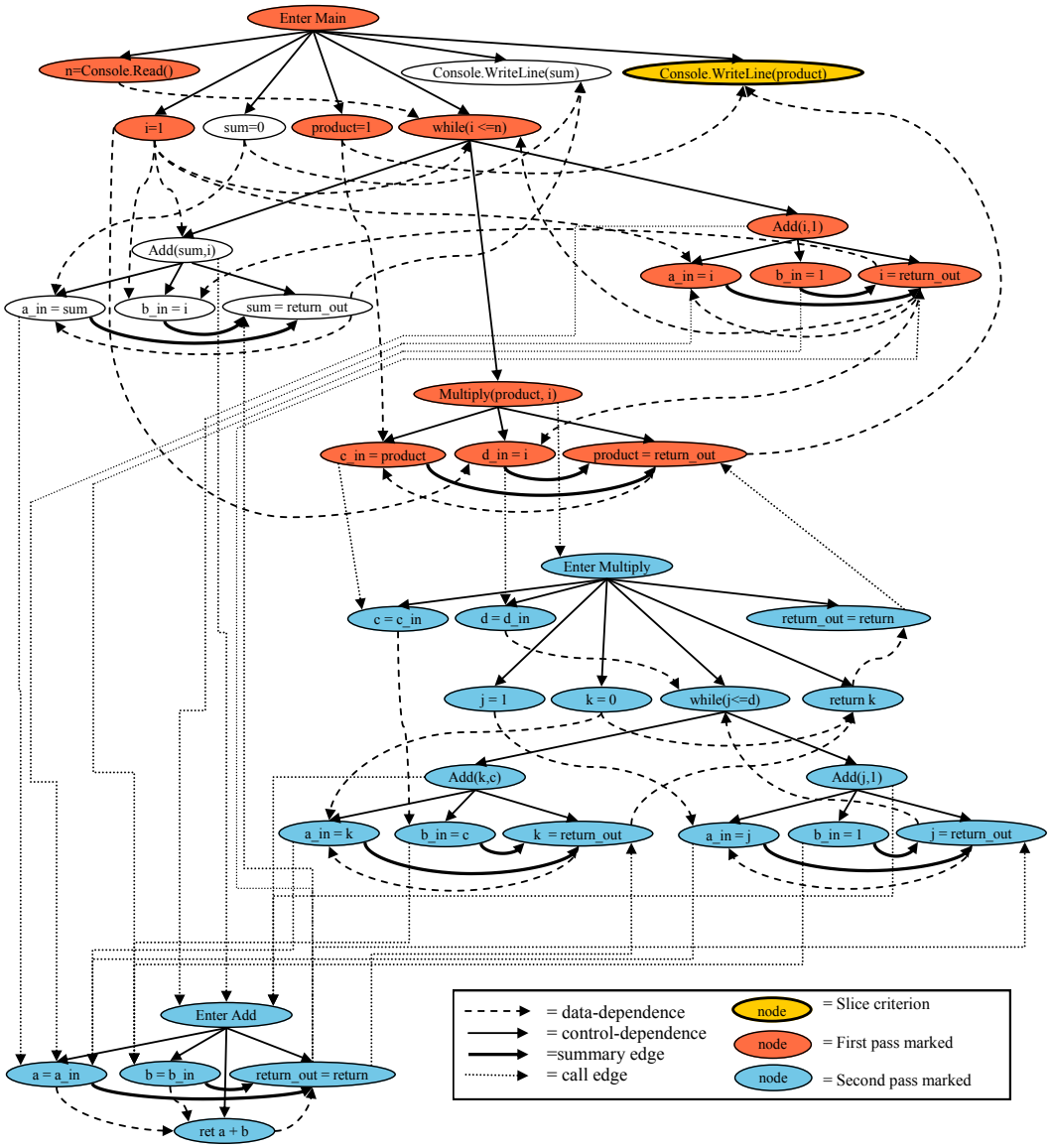

Figure 5.3: SDG for Listing 5.4

Slicing a SDG

Horwitz et al. [1988] propose a two part slicing algorithm to precisely and efficiently extract program slices from an SDG. We shall refer to this as the HRB algorithm.

The HRB algorithm consists of two passes of the SDG, each pass marking particular nodes. The slice consists of the union of nodes marked by each pass of the algorithm.

The first pass of the algorithm traverses backwards along vertices, marking all nodes that are backwards-reachable from the slicing criteria, but ignoring parameter-out edges. This effectively marks all backwards-reachable nodes, without descending into callsites, but the summary edges guarantee that the effects upon out parameters are considered, even if the body of the calls that made them are not. The calls that are not considered, the ones which effect out parameters, are then processed in the second pass.

The second pass of the algorithm, marks all backwards reachable nodes, starting from any node

marked in the first pass, but ignoring call and parameter-in edges. This pass marks nodes that effect out parameters, but without escaping from the context that produces these effects. This creates a program slice, without violating the calling context problem.

```
1  function BackwardsSlice(criteriaNodes)
2          markedFirstPassNodes = FirstPass(criteriaNodes)
3          return SecondPass(markedFirstPassNodes)
4
5  function FirstPass(nodes)
6          edges = GetIncomingEdges(nodes)
7          edges = RemoveParamOutEdges(edges)
8          parentNodes = GetStartNodes(edges)
9          return nodes + FirstPass(parentNodes)
10
11 function SecondPass(nodes)
12         edges = GetIncomingEdges(nodes)
13         edges = RemoveParamInEdges(edges)
14         edges = RemoveCallEdges(edges)
15         parentNodes = GetStartNodes(edges)
16         return nodes + SecondPass(parentNodes)
```

Listing 5.5: HRB SDG Backwards slicing algorithm

A pseudocode example implementation of the slicing algorithm is shown in Listing 5.5. The *FirstPass* function, finds all incoming edges from the supplied nodes, thereby performing a backwards traversal of the graph, and removes the parameter-out edges. The supplied nodes are returned, showing they have been marked, and the function is called recursively to find the transitive parents of the supplied nodes. The results of the *FirstPass* are used as the input to the *SecondPass* algorithm. The *SecondPass* function proceeds in the same way, only removing parameter-out and call edges.

Figure 5.3 shows an example of this algorithm in practice. The slice criterion is the *Console.WriteLine(product)* node, and the marked nodes for the first and second passes are shown. The slice is the union of these marked nodes, and we can clearly see that the nodes involved with the *sum* variable, in the main method, will be excluded from the slice.

The HRB algorithm can also be modified to perform a forward slice, find all parts of the program that are affected, rather than all parts which affect, the slice criteria. Instead of tracing backwards along dependence edges, it traces forwards, the first pass ignoring parameter-in and call edges, and the second pass ignoring parameter-out edges.

The slicing algorithms we have described here work for simple interprocedural programs, but slicing can also be applied to programs with object-orientation [Larsen and Harrold, 1996], different parameter passing mechanisms [Binkley, 1993], concurrency [Chen and Xu, 2001a] and many other modern programming language features. These techniques are available, but creating a program slicing tool for a

modern programming language is still an expensive and complex software engineering task, as we shall describe in Chapter 6.

5.2.2 Data-dependence slicing

The problem with standard slicing

Our goal in using program slicing was to find all parts of the program that affect, or are affected by, the database queries. We can obtain this subset of the program, by unioning the forward and the backward slices for each call to a query executing method in the application. This combination of program slices will contain all parts of the program which potentially affect queries, or are affected by queries. If we only perform query analysis on this subset of the program, then the query analysis will be less costly. If the time and space cost of the query analysis on the program subset and slicing to obtain the subset, is less than the cost of running the query analysis over the entire program, then we shall obtain an overall increase in efficiency.

The problem with this approach, is that program slices can be large, especially in programs with dependence clusters [Binkley et al., 2007a]. If slices are very large, then they will eliminate less computation from the query analysis; therefore, the slice sizes should be as small as possible to obtain the greatest possible reduction in computation.

Program slices follow transitive control and data dependences, meaning that we include in the slice, code which makes the slicing criteria reachable, as well as code which will affect the value of the query itself. For the purposes of impact analysis, we do not need to know if the query is reachable, we simply need to know the value of the query itself.

As an example, consider Listing 5.6, which shows some user interface (UI) code for our UCL Coffee inventory application. The *ChooseCommandDialog()* method on Line 1 activates the part of the UI which allows the user to choose whether to search by product, or by a supplier, then returns an enumeration type which represents the chosen command. If the user selected product, then Line 6 will launch the part of the UI where the user enters the product ID. This ID is then passed to the *Product.Find()* method which executes the query against the database and returns a populated *Product* object. This object is then displayed by the UI on Line 8. The same process takes place for suppliers, if the user selects suppliers instead of products.

The query-executing methods on lines 7 and 12 are control dependent upon Line 1, therefore Line 1 is included in the program slices of query executions. Analysing the *ChooseCommandDialog* on Line 1 will not affect the query analysis, as no data from this method will ever reach the query executing methods, so it could be safely ignored by the query analysis, even though it is included in the program slice.

Data-dependence slicing

We wish to find a minimal subset of the program that needs to be analysed, in order to reduce the parts of the program that need to be analysed with the query analysis described in Chapter 3. By definition forward and backward slices using all query executing methods as the criteria will cover a subset of the

```
1  Command command = ChooseCommandDialog();
2
3  switch (command)
4  {
5      case Command.Product:
6          int id = EnterIdDialog();
7          Product product = Product.Find(id); // INV–Q1
8          DisplayProduct(product);
9          break;
10     case Command.Supplier:
11         int id = EnterIdDialog();
12         Supplier supplier = Supplier.Find(id); // INV–Q2
13         DisplaySupplier(supplier);
14         break;
15     default:
16         throw new ApplicationException("Unknown Command");
17         break;
18 }
```

Listing 5.6: Inventory application UI example

program involved in database queries. Traditional program slicing, however, can still include large parts of the program that are irrelevant for the query analysis. In this section we shall introduce a type of slicing that can be used to yield much smaller slice sizes, by making the observation that we are only interested in the *values* of the queries, as opposed to the *control-flow* of the program that executes them.

In Listing 5.6, the key observation is that the calls on lines 7 and 12 are only *control* dependent upon the call to *ChooseCommandDialog()* on Line 1. The control dependence means that Line 1 decides if lines 7 or 12 will be executed. Our query analysis, and most program analyses, over-approximates control flow, and will simply consider both Line 7 and 12 as possible executions; therefore, the effect of a control dependence on whether or not the query executing method gets called is largely irrelevant, because all possible paths are considered by the query analysis anyway.

Conversely, whilst control-dependence is largely irrelevant, query analysis will be greatly affected by data-dependence between locations that can affect the values of the queries, and by the locations where the query results could be used. We observe that control dependence is always irrelevant, while data dependence is significant. Therefore, we would like to achieve a program slice, based upon data-dependence only, which we shall call *data-dependence slicing*.

To create a data-dependence slice, we create a standard SDG[Reps et al., 1994], with one difference: the standard summary edge algorithm ignores control-dependence edges. This gives us summary edges that only represent transitive interprocedural data dependencies, rather than transitive interprocedural control or data dependencies. To slice the SDG we ignore control-dependence edges in the standard

slicing algorithm, whilst the rest of the algorithm remains unchanged.

```
1  function BackwardsDataDepSlice(criteriaNodes)
2          markedFirstPassNodes = FirstPass(criteriaNodes)
3          return SecondPass(markedFirstPassNodes)
4
5  function FirstPass(nodes)
6          edges = GetIncomingEdges(nodes)
7          edges = RemoveParamOutEdges(edges)
8          edges = RemoveControlEdges(edges)
9          parentNodes = GetStartNodes(edges)
10         return nodes + FirstPass(parentNodes)
11
12 function SecondPass(nodes)
13         edges = GetIncomingEdges(nodes)
14         edges = RemoveParamInEdges(edges)
15         edges = RemoveCallEdges(edges)
16         edges = RemoveControlEdges(edges)
17         parentNodes = GetStartNodes(edges)
18         return nodes + SecondPass(parentNodes)
```

Listing 5.7: Data-dependence backwards slicing algorithm

The algorithm in Listing 5.7 shows pseudocode for the backwards data-dependence slicing algorithm. This algorithm is identical to the algorithm in Listing 5.5 except that it filters the list of edges further by calling *RemoveControlEdges*. Intuitively, this modification is very minor, and can be implemented with no significant increase in complexity; in fact, it is likely to cause a decrease in the average complexity, as less nodes will become part of the *parentNodes* variables, reducing the amount of computation. We also expect that the modifications of this type, to the summary edge calculation algorithm, will have almost no increase in performance in the worst case, and a reduction in complexity in the average case, for the exact same reasons.

Improving the Efficiency of Query Analysis Using Data-Dependence Slicing

A backwards and forwards data-dependence only slice of every query executing method in the program will cover the parts of the program associated with database queries. The results of the slicing will be a set of marked nodes. A mapping between each node in the SDG and its corresponding instruction in the program, is used to convert this set of nodes into a usable subset of the program that can be used as the input to our query analysis. It is important that we maintain mappings between all nodes in the graph, including nodes such as actual-in and actual-out nodes, so that we include all required instructions for the query analysis.

We conduct the query analysis for the original control flow graphs of any method in the program which contains at least one instruction in the program subset. During the query analysis, we only con-

sider instructions that are part of the subset. We ignore methods which are not included in the subset. Otherwise, the query analysis is executed as normal. In this way, we can execute the query analysis only over the subset of the program included in the data-dependence slicing.

The backwards data-dependence slicing ignores all parts of the program that will not have any effect on the value of the queries. The parts of the code that are ignored might have complex string operations that are very expensive to analyse; therefore by ignoring them we can improve the efficiency of the query analysis. We shall discuss the performance improvements this optimisation acheives in practice, in Chapter 7.

5.3 Related Work

5.3.1 Dead state removal

Dead state removal is not a novel technique, and liveness analysis is used by the string analysis of Christensen et al. [2003][5] and can be used for similar compiler optimisations [Aho et al., 2006]. Even though this technique is not novel, it is important that we describe how it relates to our query analysis exactly, in order to give a complete picture of the optimisations we are making. In Chapter 7, we analyse an implementation of our impact analysis that uses these optimisation techniques. We claim our implementation is efficient, and for the sake of external validity and reliability, we need to show exactly the optimisations that we have made, to describe the performance shown in our evaluation. It also serves to help others replicate our results by using the same optimisations, and as a guide for efficient implementation of our query analysis.

5.3.2 Abstract garbage collection

We are not aware of any other similar program analyses that use abstract garbage collection, but we do not claim novelty. Again, the importance of specifying the abstract garbage collection is not to explain a novel contribution, but to show how an efficient implementation of a query analysis can be created, and also as a reference to help explain the performance results described in Chapter 7. It also serves as a guide to aid the external validity of our results, allowing others to replicate similar efficient implementations.

Garbage collection is well studied, and many advanced techniques for garbage collection exist [Jones and Lins, 1996]. The approach we have chosen is lightweight and effective, as we shall discuss in Chapter 7, but there is the possibility of utilising more advanced garbage collection techniques and applying them to our abstract domain. It is unlikely that many of the more advanced garbage collection techniques will be appropriate, because our abstract domain is much simpler than the concrete domain that we are modelling, and is not subject to the same constraints. For example, our current garbage collection effectively pauses the dataflow analysis, and continues when it has completed. Whilst this is not a problem for our query analysis, it would be a problem for garbage collection in many real programming environments, and techniques designed to overcome this pausing will have little or no benefit for our approach.

[5] presumably for the same purpose, although the exact purpose of the liveness is not specified in the paper.

5.3.3 Program slicing

One of the most notable approaches to reducing unnecessary computation in dataflow analysis, is the graph-reachability approach of Reps et al. [1995]. This approach, uses the call-graph and reachability information to eliminate parts of the application which do not need to be analysed. The drawback to this approach is that it is only suitable for analyses where the set of dataflow facts is a finite set, and where the dataflow functions satisfy some mathematical properties. Our dataflow analysis functions do not satisfy these properties, the set of dataflow facts is not finite, therefore the analysis described in Chapter 3 is not amenable to this approach. Whilst not suitable for our query analysis, this approach can be seen as very similar to our approach; whereas Reps et al. [1995] use graph reachability to eliminate parts of the program which are irrelevant for the analysis, our approach uses program slicing. It is much more expensive to calculate a program slice than to calculate reachability from a call graph, so our approach is only suitable when slicing is likely to give enough of a reduction in size to achieve an overall increase in performance. The approach of Reps et al. [1995] can be thought of as a similar conceptual idea to our program slicing reduction approach, but the two approaches are applicable to mutually exclusive types of dataflow analysis.

It may seem unnecessary to build an SDG, with control dependence edges, and then ignore them during slicing. A naive assumption would be that, by ignoring the control dependencies, we could use a much simpler representation. Unfortunately this is not the case. A data dependence slice can be compared directly to a interprocedural def-use analysis [Harrold and Soffa, 1994]. A def-use analysis identifies relationships between the definitions and uses of variables, similar to a reaching definitions analysis, and is typically used in single static assignment (SSA) representations of programs [Aho et al., 2006]. Def-use relationships are directly comparable to data-dependencies and therefore, interprocedural def-use analysis is directly comparable to data-dependence slicing. They both have no requirement for directly considering control-dependence, and they both identify the relationships between the definitions and uses of variables. If we examine related work on interprocedural def-use analysis [Harrold and Soffa, 1994], we see that this work uses an interprocedural flow graph (IFG) with interreaching edges to mark interprocedural dependencies, that serve a similar purpose to summary edges. This solves the calling context problem in a similar way to the HRB program slicing algorithm, and building the IFG requires similar auxiliary analyses, such as side-effects analysis. Interprocedural def-use analysis is remarkably similar to data-dependence slicing, and is not significantly simpler, despite having no requirement to explicitly consider control-dependence. We therefore argue that current comparable techniques to data-dependence slicing are not significantly simpler or more efficient than data-dependence slicing, and largely solve very similar problems using very similar techniques.

If interprocedural def-use analysis is directly comparable to data-dependence slicing, then why use data-dependence slicing at all? We argue that the problem we are addressing could be phrased in terms of slicing or def-use analysis, but we have chosen to use slicing for the following reasons. Slicing is arguably more advanced and more actively researched than interprocedural def-use analysis, meaning that tools and techniques are appearing that can apply slicing techniques to popular commercial

programming languages [Durga Prasad Mohapatra, 2006, Hammer and Snelting, 2004, Willmor et al., 2004, Ranganath et al., 2007, Anderson and Zarins, 2005]. We can build upon this slicing research and these techniques, allowing our data-dependence slicing to be applied to a wide range of important programming languages. Slicing is being studied to increase its performance and scope [Binkley et al., 2007b, Binkley, 2007], and our work can potentially utilise much of this research. Alternatively the study of interprocedural def-use analysis is not based upon a standard representation, unlike SDG based slicing, and must solve the same problems independently. Either approach is suitable; however, they are variations upon a single theme, and the phrasing of our analysis in terms of SDG-based program slicing allows us to utilise the research and resources for SDG based analysis.

Our approach can effectively be applied to any SDG, and there is an abundance of research into creating SDGs; particularly relevant to this work, is the work on creating SDGs for object-oriented programs [Larsen and Harrold, 1996, Durga Prasad Mohapatra, 2006, Hammer and Snelting, 2004, Chen and Xu, 2001b]. We explain how our prototype implementation takes advantage of some of these techniques in Chapter 6.

The definitions of control and data dependence that we have used are very simplistic, but more accurate definitions have been proposed by Ranganath et al. [2007]. These definitions can be used to build up more accurate SDGs, with which we can obtain more accurate slices. Although we do not currently take advantage of this work, we expect research such as this to be used in many slicing algorithms, and as such we expect our analysis to be built using slicing based on these concepts in future.

Whilst we have chosen to use static program slicing, several dynamic slicing approaches exist, which could be applied in a similar way. However, if we are using dynamic slicing, we shall encounter the same problems that affect dynamic program analysis, which we discussed in Chapter 3, such as problems with coverage of execution paths. If these problems exist, and we are using dynamic runtime data, then it may not be suitable to use dynamic slicing as an optimisation for a static query analysis. This could lead to exposing the disadvantages of both approaches. Rather than use dynamic slicing to find a program subset, it would be more efficient to simply use dynamic program analysis, instead of our static query analysis. A dynamic query analysis would, likely, not suffer the performance problems of static query analysis, so would not need to use program slicing as an optimisation. However, it would suffer from other problems that are not present for static query analysis.

5.4 Summary

In this chapter we have described two techniques for efficient implementation of query analysis. Dead state removal uses a standard liveness analysis to remove items from the abstract state, as early as possible, so that they are not used in unnecessary computations. Abstract garbage collection, removes unreachable items from the abstract heap and regular string heap, again minimising unnecessary computations. These two techniques provide the groundwork for an efficient query analysis for use in our prototype implementation that will be discussed in Chapter 6 and Chapter 7.

The second half of this chapter introduced a technique for discovering the parts of the program

which are involved with database queries. We use program slicing to discover this subset of the program, and introduce a novel data-dependence slicing technique, which creates much smaller slices than traditional program slicing. By calculating data-dependence slices, forwards and backwards, for all database query executing methods, we obtain a subset of the program involved in database queries. This program subset can be used to limit the analysis performed by the query analysis described in Chapter 3, eliminating expensive parts of the analysis, with the goal of improving efficiency and scalability. We shall discuss the empirical evaluation of the efficiency and scalability of this approach in Chapter 7.

In the following chapter, we shall describe how the techniques we have described so far, can be used to produce a practical implementation of database schema change impact analysis.

Chapter 6

Impact Analysis

In this chapter we shall describe how the techniques presented so far, can be combined to produce a database schema change impact analysis. We shall then describe our prototype implementation tool and discuss how it implements the query analysis, impact calculation and data-dependence program slicing. We shall also discuss the algorithms used and the implementation decisions that will be significant for discussing the evaluation in Chapter 7.

6.1 Database Schema Change Impact Analysis

Software change impact analysis is the process of predicting the effects of change in a software system. The categories and frameworks of software change impact analysis are described by Bohner and Arnold [1996]. This dissertation describes a type of software change impact analysis, which fits into the sub-category of dependency analysis, being a low-level source code analysis rather than a higher level analysis of other artefacts produced in the software life cycle.

Database schema change impact analysis predicts the impacts of database schema change upon the applications that interact with the database. Figure 6.1 shows the stages involved in database schema change impact analysis, which we shall discuss in turn. Database schema change impact analysis includes query analysis, described in Chapter 3, impact calculation, described in Chapter 4, and optionally includes the implementation optimisations described in Chapter 5.

6.1.1 Data-dependence slicing

This initial stage is an optional optimisation stage, as described in Chapter 5. The input to this stage is the application program, either as source code or compiled machine or byte code. We extract a subset of the program that is involved with database queries by using data-dependence slicing over each known query-executing method. The output of this stage is the program subset that was identified, which consists of a union of the forward and backward program slices, resulting in the subset of the program that interacts directly with the database.

6.1.2 Query Analysis

The inputs to this stage are either, the subset of the program identified by Stage 1, or the full program if Stage 1 was omitted. This stage executes the query analysis, described in Chapter 3, which is a static program analysis that predicts the values of all the queries in the application, maintaining the details of

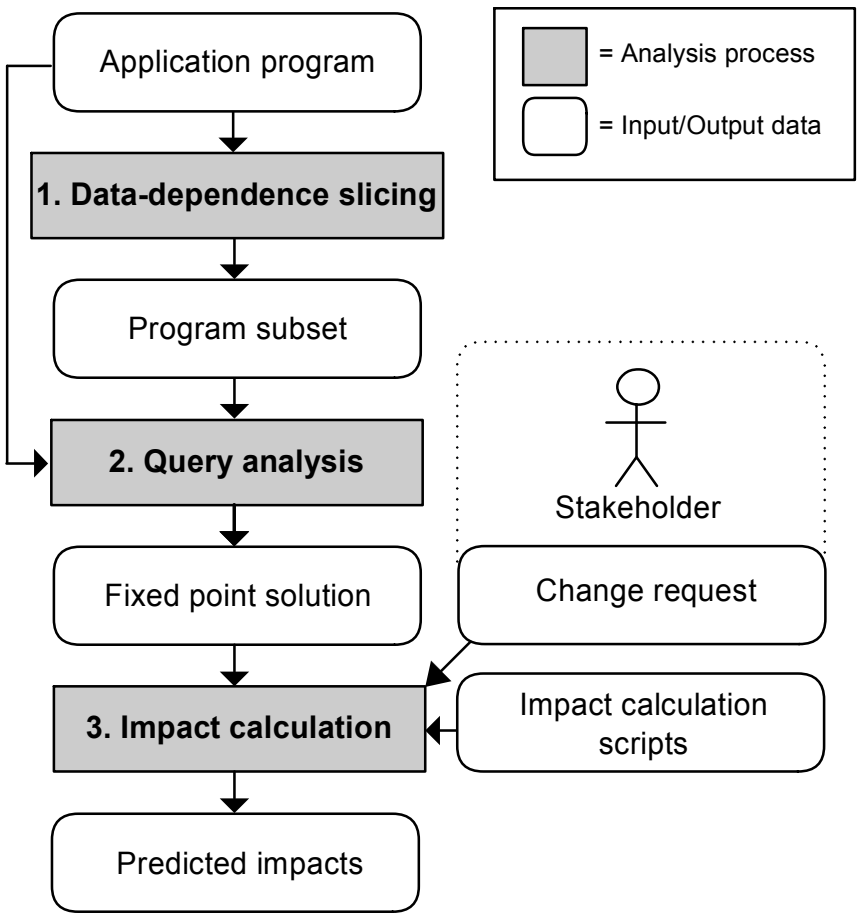

Figure 6.1: Database Schema Change Impact Analysis

where individual queries are defined, executed and where the results of the queries are used. The output of this stage is a fixed point solution of the query analysis for the supplied program.

6.1.3 Impact Calculation

The input to this stage is the fixed point solution from Stage 2, which is processed to extract information about where queries are defined, where they are executed and where their results are used. This information can then be processed, using the impact calculation process described in Chapter 4, to predict the impacts of a given set of impacts. This process is driven by the requests of stakeholders, specifying which parts of the database schema are being changed, and choosing from impact calculation scripts that are available. The output of this stage is the final result of the analysis, a set of predicted impacts.

6.2 SUITE

The implementation of a schema change impact analysis tool, is a significant engineering task, as we shall discuss, and we regard the construction and evaluation of this implementation as a major contribution of our research. We shall describe this prototype implementation because it serves as a guide for others who wish to implement similar tools, but more importantly, because it describes in detail the system with which we evaluate our impact analysis approach in Chapter 7.

Our prototype implementation is the Schema Update Impact Tool Environment (SUITE) and was initially presented in our earlier work [Maule et al., 2008]. SUITE is built using the Microsoft .NET Framework and written in C#. It currently consists of approximately 32 thousand lines of code (KLOC).

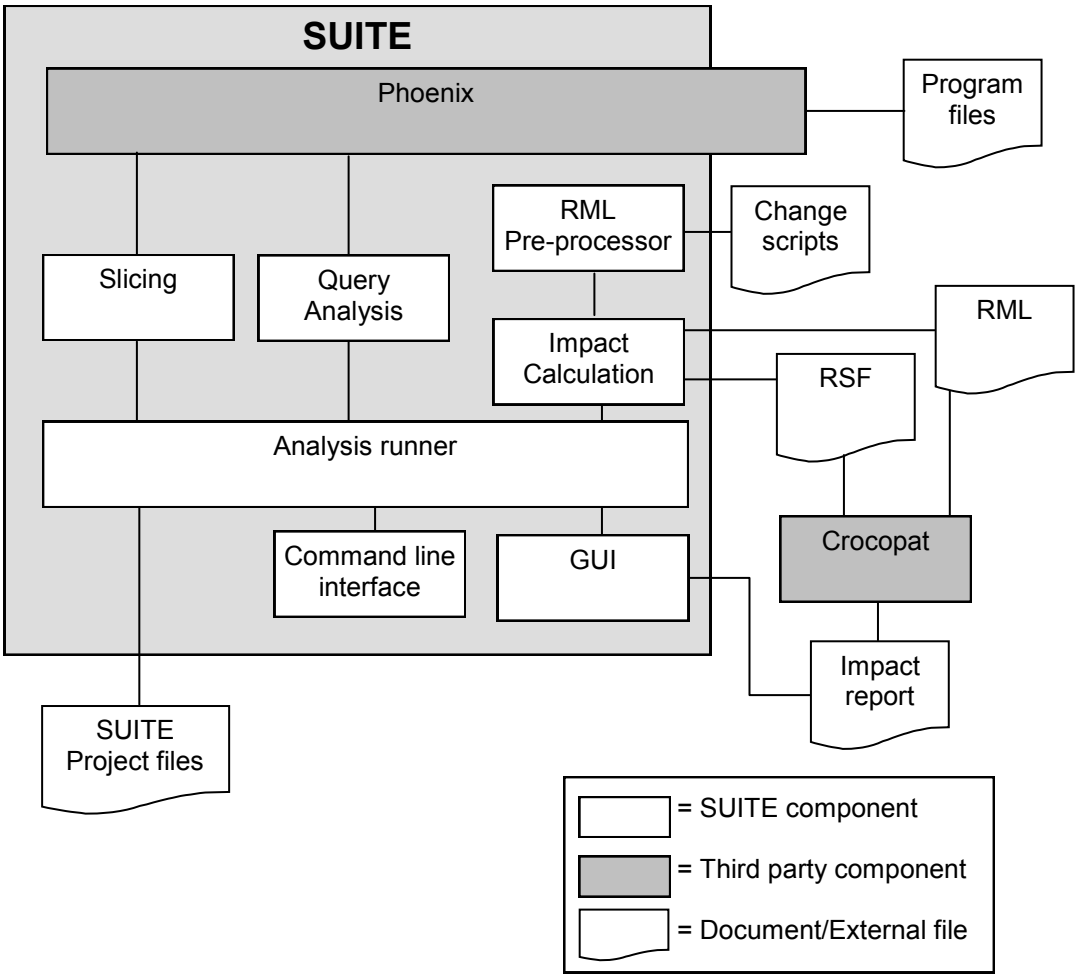

Figure 6.2: SUITE Architecture

Figure 6.2 shows the main architectural components of SUITE. The slicing, query analysis and impact calculation components carry out the analyses that we have described in previous chapters. The analysis runner component coordinates the interactions between the impact analysis stages. SUITE uses the Microsoft Phoenix and CrocoPat as third party components. The interactions with external files and documents, and external components, are all controlled by SUITE and require no extra actions from the user.

SUITE is based around a GUI that we shall use to illustrate the application, but also has a command line interface.

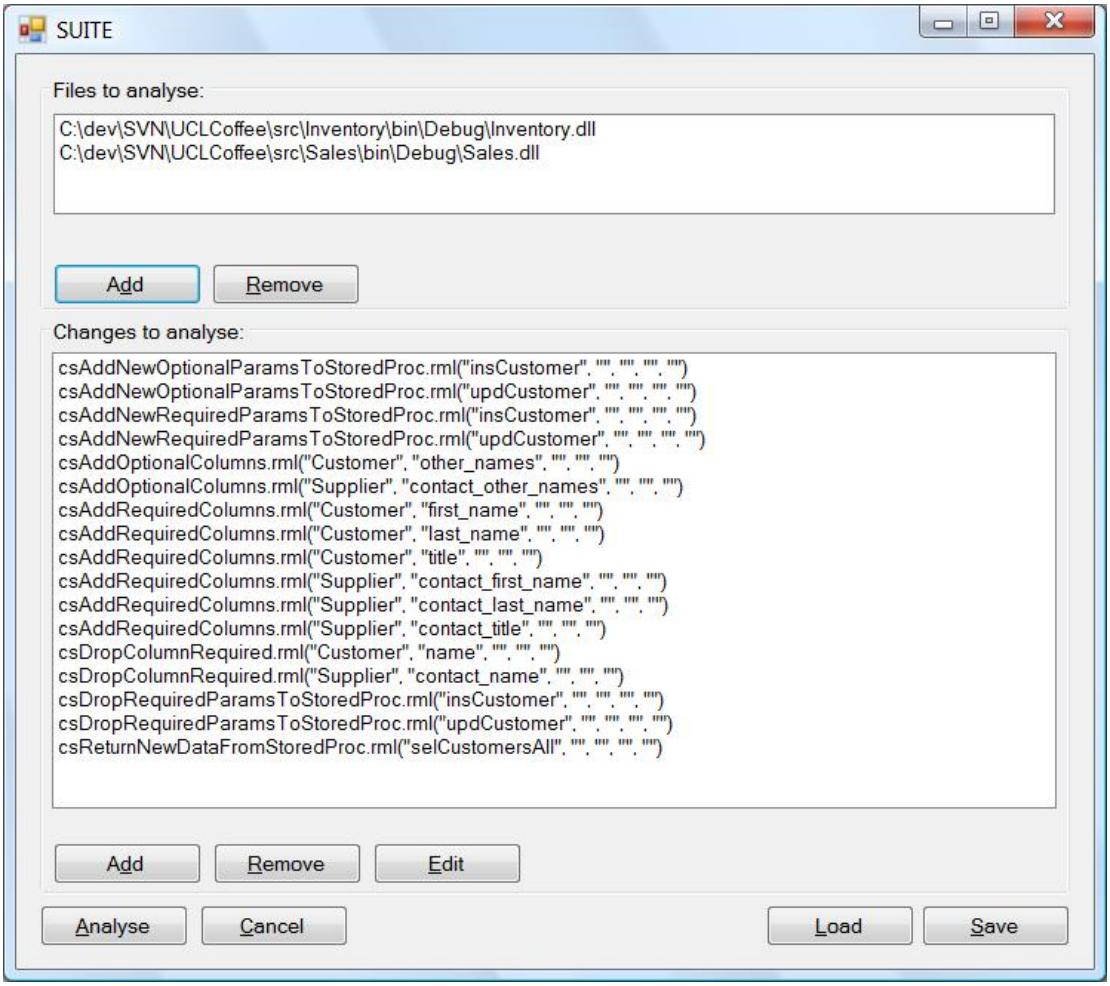

Figure 6.3: SUITE Main Window

Figure 6.3 shows the main screen for the SUITE GUI. The top part of the screen shows the files that are to be analysed, with the option to add and remove files. Any compiled .NET library or executable can be analysed by SUITE, and in the main window of the application we can see that the DLLs for the UCL Coffee inventory and sales applications have already been added to the list for analysis.

The bottom part of the application window shows the impact calculation scripts that will be executed during the impact calculation, with controls to add, remove and edit. The scripts shown, are from the sample impact calculation suite in Section 4.2.2.

The application will execute all three stages of the analysis, and display the results upon pressing the 'Analyse' button in the bottom left hand corner. SUITE can also save and load the state of the application into a SUITE project, and this file can also be used as an input for the command line version of SUITE.

We shall now describe what happens when the analyse button is pressed, and the analysis itself is started.

6.2.1 Data-dependence Slicing

When the analysis starts, SUITE will perform data-dependence slicing of the input applications and libraries. We use the Microsoft Phoenix Framework [Microsoft Phoenix] for much of the low level analysis, such as construction of the CFG and the creation of an SSA representation [Cytron et al., 1991]. We have implemented a slicing algorithm using this framework, based upon the algorithm of Liang and Harrold [1998]. Although, there exist improvements to the Liang and Harrold slicing algorithm [Hammer and Snelting, 2004], the implementation of these extensions are outside of the scope of this research.

The construction of the program slicing algorithm was extremely time-consuming to implement, requiring significant engineering effort. This is mainly due to the amount of auxiliary analyses upon which slicing is based, such as side-effects analysis [Landi et al., 1993]. Also, the expense is affected by the complexity of modern commercial programming environments, such as .NET. To create a high quality slicing algorithm implementation, for the full C# language, is well beyond the scope of this research and would require significantly more resources. Because of this, we do not claim that the algorithm used by SUITE is safe, conservative or complete. We argue that SUITE is still suitable for use by our case study, as the language features that we do not support (reflection, generics, lambda expressions) are rarely used in practice, and we discuss what this means for the external validity of our results in Chapter 7.

The main expense in creating a slicing algorithm is the creation of the SDG. Using the Microsoft Phoenix Framework [Microsoft Phoenix] provides only part of the information required for the SDG, including use-def relationships [Cytron et al., 1991] and type information. The rest of the information must be extracted manually during the construction of the SDG, and for a language built on a complex modern virtual machine, such as the .NET of Java virtual machine, this is a complex process.

For example, call sites and procedure entry nodes have to have the correct actual and formal, ‘in’ and ‘out’ nodes; these nodes represent the incoming and outgoing parameters of the calls. Different calling conventions order these parameters differently, based on whether it is a static call, or a message sent to an object, access to an implicit property, or another type of call. Distinguishing between these types of call to create the parameter nodes correctly, for every different case can involve complex heuristics.

Another example of extra information that must be calculated is that some method calls modify the receiver object, and this must be accounted for as a definition of the variable that stores a reference to the receiver object. These extra definitions must be accounted for in the SDG[1] and included in reaching definitions calculations that calculate dependencies. This is non-trivial because it requires creating a reaching-definitions analysis for the possible modification of objects, which involves creating additional dataflow analyses, and transparently combining this information with the information already provided by Phoenix.

We shall not describe this extra SDG information in more detail, or how SUITE calculates this information, as our research does not offer any novel contributions to this area. We make note of these technical issues to show that the construction of an SDG is non-trivial, and to illustrate that, in our

[1] This is discussed as the object flow sub-graph in the paper by Liang and Harrold [1998]

experience, much of the development time cost of this project was dedicated to experimenting with and developing the SDG construction and slicing algorithms.

6.2.2 Query Analysis

The query analysis stage is an implementation of the analysis described in Chapter 4. Again, we use the Microsoft Phoenix Framework [Microsoft Phoenix] for much of the low level analysis, and implement the dataflow analysis on top of this. We use standard [removed] work list algorithms [Nielson et al., 1999] to obtain the fixed point solution.

Worklist iteration algorithms can be summarised as follows. During the intraprocedural analysis of each method, if the method calls another method, we add the callee to the worklist if it has not already been processed in this context. Likewise, if at the end of the intraprocedural analysis of a method, the returning flow state has changed, we add the caller methods to the worklist. Worklist algorithms proceed in this way, by removing then evaluating each item in the worklist. The evaluation of each item can add further items to the worklist. The algorithm continues until the worklist is empty, and a fixed point solution has been reached. The partial ordering operators we discussed in Chapter 3 play an important role here in judging whether the incoming and outgoing flow states of the methods have changed.

Worklist algorithms start with a initial list of items, and in the case of SUITE we use the root methods of the interprocedural flow graph to populate the worklist. We could simply populate the worklist with all methods, but that would result in unecessary calculations, analysing procedures in contexts in which they can never occur. Populating the worklist with the root nodes of the interprocedural flow graph ensures that all the dependent methods will get added to the worklist in time whilst maintaining an efficient analysis.

We use a standard queue data structure for our worklist in SUITE, having experimented with other approaches, such as selecting the method with the most dominant call graph node. We have found that the queue datastructure performs just as well, and often better, on our case study, than the other approaches we tried.

We have discussed the worklist algorithm that we use have used here, to better explain the results in Chapter 7. We do not, however, discuss these results here further, as it is not the main focus of our research.

The query analysis is extensible, in that the API for SUITE allows arbitrary known methods to be specified. For each known method we can add a new transfer function to the analysis. SUITE, by default, specifies the known methods that are required to process the UCL Coffee applications as well as the case study application used in Chapter 7. The new known methods can be added by editting the settings of SUITE to map a fully qualified method name to an implementation of an abstract class; this implemenation can specify the semantics of the transfer function. New fact calculating functions for impact calculation are also added in this way. This plugin approach process is availabe in our current prototype implementation, and all the transfer functions and fact calculating functions we specify in Appendix A are included using this approach.

The query analysis can accept the whole program, or a subset of the program, as input. If a subset is

specified the analysis performs the intraprocedural analysis for any method where at least one instruction is included in the subset. The intraprocedural analysis uses the full control flow graph for each method, but ignores any instructions not in the program subset by treating them as no-operation (NOP) nodes. This approach maintains the correct dataflow, whilst ignoring parts to the application that are not included in the program subset. In this way, we can execute SUITE with or without the data-dependence program slicing optimisation, as we shall discuss in Chapter 7.

We implement the optimisations of dead state removal and abstract garbage collection, as specified in Chapter 5. The dead state removal is applied during the intraprocedural part of the query analysis, and applied to the outgoing flow state calculated for each basic block. The abstract garbage collection is applied periodically, at intervals of every 1000 methods analysed.

An interesting observation about the implementation of the query analysis is that, after the optimisations have been implemented, the dominant cost of the algorithm is the partial ordering comparison between regular strings, caused by the iterative widening of large regular strings. As discussed previously, the partial ordering operator is used by the chaotic iteration worklist algorithms that solve the dataflow analysis, deciding if merged flow states contain new information or not. We shall discuss the expense of the partial ordering operations, the data behind this observation, and what it means in Chapter 7.

Because partial ordering is so significant in the cost of the analysis, and significantly effects our evaluation, we shall describe the algorithms we use to calculate partial ordering.

For regular strings, r_1 is partially ordered before r_2 if the set of concrete strings for r_1 is a subset or equal to the set of concrete strings for r_2. To test this we use an approach from model checking [Clarke et al., 1999], which states:

$$\mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$$

is equivalent to:

$$\mathcal{L}(A) \subseteq \mathcal{L}(B)$$

This formula states that if the language produced by $\mathcal{L}(A)$ is a subset of the language produced by $\mathcal{L}(B)$, then we also know that the intersection of $\mathcal{L}(A)$ and the complement of $\mathcal{L}(B)$ will be the empty set. This means, that if we can perform the operations to find the complement of a regular string, to union two regular strings, and to test if a regular string is empty, then we can evaluate if one regular string is partially ordered before another. These operations can be applied to deterministic finite automata (DFA) using well known algorithms, and regular strings can be also be represented as DFAs.

We first convert both regular strings to non-deterministic finite automata (NFA) using the McNaughton-Yamada-Thompson algorithm [Aho et al., 2006]. Then we convert each NFA to a deterministic finite state automaton (DFA), using the subset construction algorithm [Aho et al., 2006]. We then find the complement of the DFA of language B [Hopcroft et al., 2006], and combine it with the DFA of language A using the product construction algorithm [Hopcroft et al., 2006]. We can then test

the resulting automata for emptiness, by checking to see if there exists a path from a starting state to an accepting state. If it is empty, then language A is partially ordered before, or equal to, language B.

There are many separate algorithms and stages in calculating the partial ordering of regular strings, and each stage carries its own expense. Whilst these algorithms are generally well-studied, it is easy to see that repeated application of the partial ordering operator could be expensive. As we shall show, in Chapter 7, this partial ordering operation has become the dominant cost of the analysis.

Because regular strings may contain the any character ".", this precludes us from using some of the more efficient data-structures. With a finite alphabet, shortcuts can be taken in the approach defined above, but the inclusion of the any character "." requires a more expensive approach. Other options for creating more efficient algorithms for calculating the partial ordering of regular strings may be found in the model-checking research, for example using different data structures such as push-down automata [Hopcroft et al., 2006]. Our current approach use well-studied efficient algorithms, and the study of alternative approaches would involve a significant research effort, so is outside of the scope of our research.

One important implementation optimisation is *regular string approximation*, where we approximate very long regular strings as the repetition of the any character, ".*". In our current implementation of SUITE, whenever a string is widened that is greater than one thousand characters in length, we abbreviate the string to ".*". We do this because the widening of very long strings can cause iterations in the worklist algorithms where the partial ordering operation is called repeatedly at great expense. The resulting widened string is often a repetition of a large set of characters, and has already lost much of its useful impact-precision. By replacing the long regular string with ".*", we remove the majority of the widening that occurs in long strings, and reduce the cost. Therefore, to enable a more efficient analysis, at the cost of a small loss of impact-precision, we use regular string approximation during the widening of large strings. We shall discuss how the analysis is effected by the use of this approximation in Chapter 7, showing how it affects accuracy, impact-precision and cost.

6.2.3 Impact Calculation

Figure 6.4 shows the SUITE GUI window for adding an impact calculation script to be executed during the analysis. We can choose any of the change scripts available in the drop down list; in this case we have chosen the scripts to add a new optional column to a table. Each change script has a description, and a set of up to 4 parameters. The parameters are named and each have a description. The details of changes are created and edited using this screen.

We discussed an implementation of impact calculation using the CrocoPat tool in Chapter 4. We use the same approach in SUITE, representing the output of the effect calculation using the RSF file format, and executing change scripts written using RML using the CrocoPat tool.

SUITE searches pre-specified folders in the file-system, and loads any change scripts found in these folders. In this way it is simple to add new change scripts, simply by adding them to these folders.

Because RML currently has little support for modularisation, we added a preprocessor which can add 'include' files to RML. For instance, consider the following line of RML:

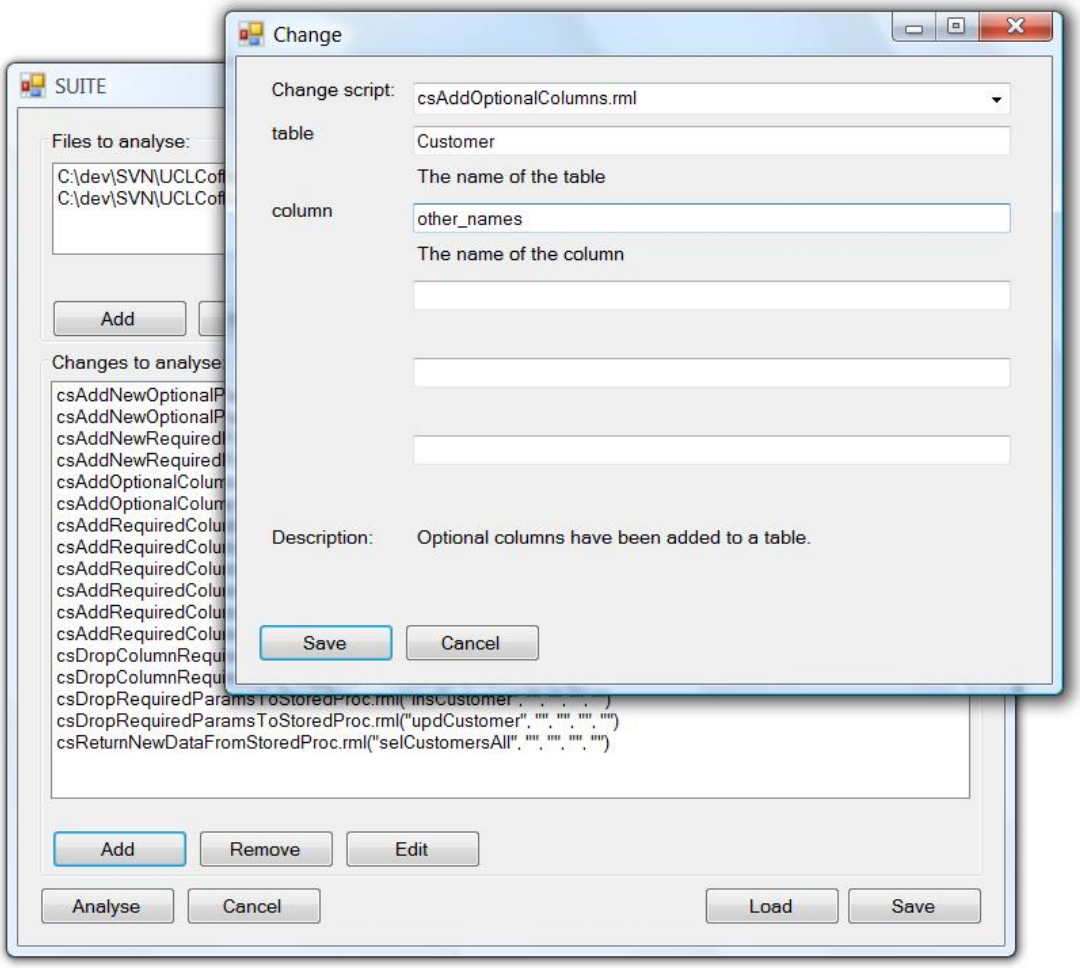

Figure 6.4: SUITE Add Change

```
\\ INCLUDE: x.inc
```

This line is interpreted by CrocoPat as a comment, but our preprocessor will replace the line with the contents of the *x.inc* file. This 'include' file mechanism allows for greater modularity amongst change scripts.

The second function of the pre-processor is to extract meta-data from the RML file. In the example shown in Figure 6.4, the description of the change script and the name and description of each of the parameters are specified on the form. This meta data is extracted from the change script by the pre-processor. The meta data is specified in RML comments using *description*, *argXName* and *argXDesc* markers. An example complete RML change script is shown in Listing 6.1; it contains meta data for the description of the change script, meta data for the names and descriptions of the two arguments and it has an include file at the beginning and end of the script. An extensive list of change scripts, including the meta data and included files, can be found in Appendix B.

The output of the impact calculation stage is a set of impacts, but these must be represented in some kind of textual format. We have chosen to standardise our changescripts so that they produce XML in a particular format. The SUITE GUI can process the impacts from this XML and display the impacts to

```
1  // INCLUDE: header.inc
2
3  //
4  // description = Optional columns have been added to a table.
5  //
6  // $1
7  // arg1Name = table
8  // arg1Desc =  The name of the table
9  //
10 // $2
11 // arg2Name = column
12 // arg2Desc = The name of the column
13 //
14
15 ImpactType    := "warn";
16 ImpactMessage := "These queries reference a changed table that now has "+
17                   "new unrequired columns. Check to see whether these "+
18                   "new data should be supplied or returned here.";
19
20 tableQueries(x)    := @ Seperator + "$1" + Seperator (x);
21 AffectedQueries(x) := EX(y, tableQueries(y) & ExecutesQuery(x, y));
22
23 // INCLUDE: output.inc
```

Listing 6.1: AddOptionalColumns Change Script

the user. The command line tool can output the XML as a file for later use.

```
<output>
  ...
  <change id='57140728'>
    <impact type='warn' message='These queries reference a changed ...'>
      <query>SELECT id\, contact_name\, company_name FROM ... </query>
      <queryexec location='C:\...\Supplier.cs:48' />
      <querydef location='C:\...\Supplier.cs:46' />
      ...
    </impact>
    <impact type='warn' message='These queries reference a changed ...'>
      <query>SELECT \* FROM Product{\, Supplier}* WHERE ... </query>
      <queryexec location='C:\...\Product.cs:98' />
      <querydef location='C:\...\Product.cs:114' />
      ...
    </impact>
    <impact type='warn' message='These queries reference a changed ...'>
      <query>INSERT INTO Supplier (company_name\, contact_name) ... </query>
      <queryexec location='C:\...\Supplier.cs:64' />
      <querydef location='C:\...\Supplier.cs:59' />
    </impact>
  </change>
  <change id='30272055'>
    <impact type='error' message='This query now has a new required ...'>
      <query>insCustomer</query>
      <queryexec location='C:\...\CustomerGateway.cs:29' />
      <querydef location='C:\...\CustomerGateway.cs:26' />
    </impact>
  </change>
  ...
<output>
```

Listing 6.2: Impact Calculation XML Output

The output of the impact calculation could look something like Listing 6.2. The elipses in this figure show where the text has been omitted for brevity.

Figure 6.5 shows how the GUI displays all the warning and error impacts in a list at the top of the window, each impact with its corresponding error message, and the change that caused it. The center of the window shows how the GUI displays the definitions of the queries, the executions and where the results are used. Each time one of these locations is selected, the bottom part of the window highlights the source code location that is affected.

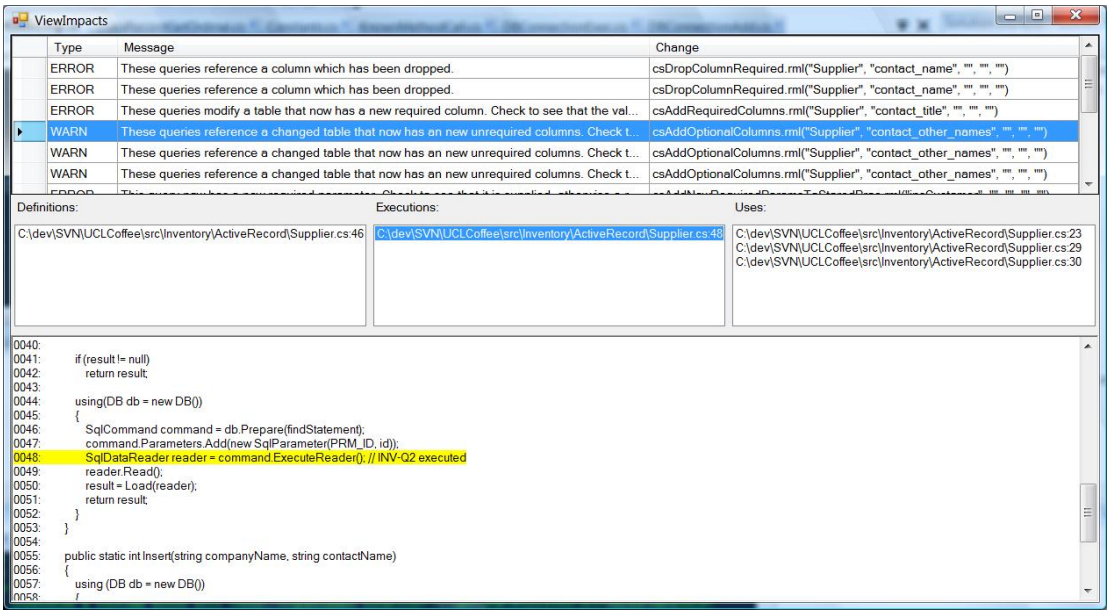

Figure 6.5: SUITE GUI Impact Report

6.3 Related Work

We have already discussed the related work for each stage of our impact analysis in the previous chapters. In this section we shall discuss other work that relates to our impact analysis approach in general.

6.3.1 Database Change Impact Analysis

There is a great deal of work related to software change impact analysis [Bohner and Arnold, 1996, Ren et al., 2004, Law and Rothermel, 2003]. However, we are only aware of two similar projects that focussed on impact analysis of database schemas upon applications.

The first of these works is by Karahasanovic [2002] and focuses on object-oriented databases, whereas we focuse on relational databases. Karahasanovic [2002] present the SEMT tool which uses an efficient implementation of the transitive closure approach [Bohner and Arnold, 1996] for impact analysis. We argue that the object-relational impedance mismatch makes this a significantly different problem, and the approach taken by SEMT is unapplicable to our particular problem. The transitive closure of objects in the application, which match objects in the database, will find all locations where DB objects are used. For applications that do not have objects that are labelled as being part of the database, this approach is not applicable. Because of the object-relational impedance mismatch [Atkinson et al., 1990], the mapping between objects and relations can be difficult or impossible to identify, especially when the data access approach used is based upon call level interfaces. However we take inspiration from the general approach defined by Karahasanovic [2002], especially the work on visualisation of results, that could be complementary to our approach, and we argue that we are solving a similar problem, but in a significantly different context.

The second related research project is the DaSIAN tool proposed by Gardikiotis and Malevris [Gardikiotis and Malevris, 2006]. This tool addresses the problem of schema change in a

very similar context to our work. However, it does not address the specific program analysis problems which we address here, such as dynamic queries and complex interprocedural control flow. The DaSIAN tool uses pattern matching to search the source code for query strings (unfortunately this paper does not ellaborate on this pattern matching approach, and how it might deal with some of the issues described here). If all database queries are embedded as static strings in the code, then pattern matching based upon regular expressions or something similar, may be very suitable; however, if dynamic queries, complex design patterns, object-relational mappings (ORMs) and other features are used, then the accuracy of the approach will suffer. Our work seeks to address these issues, providing a way of extracting potential database queries from program code with high impact-precision. However, our work is arguably more complex and expensive, and the DaSIAN tool could be more applicable in situations where dynamic queries are not used, and where a lightweight tool is more appropriate.

Some related work has been made in analysing transparent persistence [Wiedermann and Cook, 2007] using program analysis, although the focus here is on providing optimisation of queries, based on abstract interpretation [Cousot, 1996]. This work bears similarity to work on extracting queries from legacy applications [Cohen and Feldman, 2003], however these techniques are tailored to languages where queries are defined statically. This ignores many of the problems we have described in this dissertation, but it provides an insight into how embedded persistence technologies could be incorporated into our approach in the future, and insights into formalisation of similar techniques. This work was extended by Wiedermann et al. [2008], which we discussed in Chapter 3, but because this work is applied to optimisation of transparent queries, it addresses different problems. Wiedermann et al. [2008] use a context-insensitive analysis for analysing queries written using transparent persistence, which is entirely suitable for their purposes, but as we discussed in Chapter 3 we require our query analysis to be context-sensitive and applicable to a wider range of data-access practice. Therefore, although initially appearing similar to our work, the work of Wiedermann et al. [2008] addresses a significantly different problem and is largely orthoganal.

6.3.2 Dynamic Analysis and Database Testing

There has also been related work produced by the testing community. There are papers which define criteria by which the quality of database testing can be measured [Kapfhammer and Soffa, 2003, Willmor and Embury, 2005b]. These criteria are useful in creating good tests, but for our purposes are largely orthoganal. If we were to do a dynamic instead of a static impact analysis, we would need inputs that are representitive of the real application, and these adequacy criteria could help to judge the quality of these input data, maybe by creating a coverage metric for the tests [Suárez-Cabal and Tuya, 2004]. However, we have chosen to limit the scope of our research to static program analysis, so the techniques required to create good test coverage of database applications are of little direct benefit to the problems we are addressing.

Willmor and Embury define a program slicing technique [Willmor et al., 2004] which can be used to treat the database as part of the program state when creating the program slice. This is very different to the slicing approach that we defined in Chapter 5, creating a potentially larger slice, rather than creating

a minimal one. This work on slicing was later used to create a regression test selection technique for database tests [Willmor and Embury, 2005a]. Another similar regression test selection technique was proposed by Haraty et al. [2001]. Both regression test selection techniques can be viewed as very similar to impact analysis. They both identify which tests are affected, after a schema change has been made. When we consider that the tests are part of the application, database regression test selection is a subset of the impact analysis problem we solve here. But, despite the similarities, the main difference between our work and these test selection techniques is that our impact analysis can examine dynamic queries in complex interprocedural control flow, whereas these techniques expect the queries to be static. Therefore, our work presents a contribution that is not addressed by this related work. In future work we could apply our query analysis to these approaches, potentially improving the range of application to which they can be applied.

6.3.3 Schema Evolution

One of the most advanced schema evolution approaches is the $\mathcal{PRISM}$ system of Curino et al. [2008]. $\mathcal{PRISM}$ provides tools to express a schema change, estimate the effects of a schema change, and perform the schema change itself. Schema changes are proposed in terms of Schema Modification Operators (SMOs) and, as discussed in Chapter 4, SMOs could be used to help inform the change scripts we are using.

The way $\mathcal{PRISM}$ predicts effects differs from our work in two major ways. First, $\mathcal{PRISM}$ uses a predefined set of common queries for the schema, which is typically collected from the actual use of the database by logging all queries sent to the database. This is opposed to our approach of finding queries by static analysis of the application source code. Second, the prediction of the effects of change are only made upon these queries, and the applications that generate these queries are not considered.

The approach taken by $\mathcal{PRISM}$ can be seen as a database centric approach whilst our approach is application centric. They could also be thought of as solving two different parts of the same problem, and could be both used to mutual benefit. The input to $\mathcal{PRISM}$ could be better informed by the queries we discover from our query analysis, instead of queries from a log, potentially improving the coverage of queries. $\mathcal{PRISM}$ could still automate the changes that are required, and SUITE could be used to pinpoint any remaining queries in the application that require alteration, allowing the application developer to identify the specific places that require attention.

There is little overlap between our impact analysis approach and the approach of $\mathcal{PRISM}$ and they could be used together to mutual advantage. Our approach is a tool for predicting the parts of an application that will be affected by a schema change, whilst $\mathcal{PRISM}$ is a tool for minimising the effects of change and managing the change process within the database as much as possible.

Whilst there is a large body of work into database schema evolution, it is usually more database focused than $\mathcal{PRISM}$, and focused on reconciling the database itself, or focused on minimising impacts upon applications [Roddick, 1995]. There is very little work that we are aware of that directly measures the impacts of schema change upon applications, and any significant work in this direction has been discussed in this section. We see the estimation of impacts upon applications as a complementary goal to

that of schema evolution and, as such, the majority of schema evolution research is largely orthogonal to our work. Ultimately the approach outlined in this paper does not aim to replace any existing techniques for database change, but helps to extract tacit information to aid these techniques and better inform the schema change process.

6.4 Summary

We introduced this chapter by describing how the work in the previous chapters can be combined to create a database schema change impact analysis. We described how each stage was related to the others, in the general case. We then described our prototype implementation tool, SUITE, describing each of its significant components in turn. We finished with a discussion of related work.

In the following chapter we shall use SUITE to evaluate our database schema change impact analysis.

Chapter 7

Evaluation

In this chapter we shall present an evaluation of the impact analysis described in Chapter 6. We shall describe the reasons for choosing to use a historical case study, before describing the case study approach in detail. We shall first discuss the results of the case study with respects to accuracy, impact-precision and cost, as defined in Chapter 2. We shall then present a further discussion of these results, describing how useful our impact analysis would have been in practice. We conclude the chapter by discussing the threats to validity of the case study, and summarising the results with respect to the goals of the case study.

7.1 Evaluation Method

Our thesis is that the database schema change impact analysis described in Chapter 6 can be applied to real-world software projects, and will provide beneficial information when schema changes occur. This requires the evaluation of two claims:

1. The database schema change impact analysis, as described in this dissertation, is sufficiently efficient that it is feasible to apply the analysis to large commercial applications.

2. The information provided by database schema change impact analysis is useful for the stakeholders to understand the impacts of schema change.

To evaluate these claims we use a case study, as described by Yin [1989]. The reasons for using a case study are, first, a case study can be applied to real-world software projects, allowing us to investigate the validity of the first claim; we shall refer to this as the investigation of the *feasibility* of our analysis. Second, a case study can evaluate whether the results of the analysis have the potential to be useful in practice; we shall refer to this as the investigation of *potential usefulness*.

We have chosen not to use controlled experiments as the primary evaluation method. The problems we described in this dissertation, are caused by many contributing factors that occur in large and complex software projects. Replicating these problems requires replicating large and complex projects, and doing this in a controlled manner would be prohibitively expensive, if not altogether impossible. Controlled software experiments are generally not suited to replicating complex real-world situations that have many contributing variables [Pfleeger, 1994], and as such are not suitable as the primary method for evaluation of our impact analysis.

We have also chosen not to use theoretical evaluation because the results of theoretical analysis would not, in this case, justify the cost. A theoretical analysis would allow us to ask two important questions of our analysis:

First, we could use a theoretical evaluation to ask: is the query analysis safely conservative and sound? To answer this question, we could use a technique such as abstract interpretation [Cousot, 1996] to relate the formal semantics of our target language to our analysis, and show that the query analysis is guaranteed to find all queries, and that the analysis is safe, complete and sound. But, our analysis is not safe or sound and the effort of conducting a full scale proof of this kind would involve creating a full formal semantics of C# [1] and relating it to our analysis using some abstract interpretation framework. This is a complex task which is outside of the scope of our research. Moreover, we have argued in Chapter 2 that safety and soundness are not required in order to create a useful analysis, and we shall discuss the validity of this claim in Section 7.4.

The second question that can be answered by a theoretical analysis is; do the scalability enhancements, as described in Chapter 5, provide a reduction in the complexity of the overall algorithm? To answer this question, we would need to show that the reduction in program size, given by our data-dependence slicing, can be obtained at a cost that is algorithmically more scalable than our query analysis alone. The size of the slice can be the entire program in the worst case or, in the best case, the slice will consist of only the slicing criteria. Having these best and worst case slice sizes tells us very little; we either provide no benefit at all, or reduce the problem to a single program statement. Meaningful answers about the scalability of our approach would therefore require us to find the average case costs of program slicing, and to find the average slice sizes. We could then measure these average costs of slicing against the costs of query analysis with and without the reduction. It is unclear how to do this theoretically, as it is unclear what constitutes an average program or how to develop a reasonable approximation of an average database application. The alternative is to run the application on a number of real-world applications that are similar to other real-world programs, which is the approach we take by conducting a case-study.

There is, however, one property of our analysis which needs to be evaluated using a theoretical analysis. We need to prove that the dataflow analysis that we use is guaranteed to terminate. This can be proved by showing that the abstract domain of the analysis satisfies certain mathematical properties [Nielson et al., 1999]. Our query analysis is based upon the string analysis of Choi et al. [2006], where the properties of the partial ordering and widening operators are shown to guarantee termination. The widening of a finite sequence of regular strings is guaranteed to involve a finite number of iterations, and produce a finite set of regular strings. Because our extensions to the analysis do not alter the widening or partial ordering operator, and only create regular string variables with finite sets of values, we preserve the conditions required by Choi et al. [2006] to guarantee termination of the analysis. We do not replicate the proof further here, and refer interested readers to the argument presented by Choi

[1] there is no known publicly available semantics of C#, and it would be a considerable effort to create one because C# contains many advanced features such as generics, lambda expression, type inference, and other features which are complicated to correctly formalise.

et al. [2006].

Within the scope of this dissertation, controlled software experiments and theoretical validation are unsuitable as primary evaluation techniques. The cost of using controlled software experiments is prohibitive, because the class of applications we wish to evaluate is too complex to simulate in a controlled environment. Theoretical analysis can be expensive for such a complex analysis, and the guarantee of safety, completeness and soundness do not justify this cost. Instead we wish to show that our impact analysis can be applied to an important class of real applications, in practice, giving potentially useful results, investigating if our analysis is feasible and potentially useful.

Evaluation using a case study is an analysis technique that is suitable for our purposes, being able to objectively investigate complex real world situations, and evaluate the effects of real and contrived phenomena within these situations [Pfleeger, 1994]. It is a misconception that case studies are less valuable than controlled experiments or theoretical evaluation, they are simply an alternative technique that can be applied with similar effect when suitable. It is very important that a case study is carried out in a methodical and rigorous manner, just as with controlled experiments or theoretical evaluation, systematically addressing the threats to validity of any findings. If not conducted carefully, there is no guarantee that the results will hold beyond the specific case that is studied. We describe the case study approach in detail below, before discussing the results and the threats to validity.

7.2 Case Study

We conduct an exploratory case study, as describe by Yin [1989], based on the following research question:

> Is an automated database schema change impact analysis feasible for large commercial applications?

A research question is needed to limit the scope of the case study and to define a context in which the more specific questions can be addressed. These more specific questions allow us to investigate the effect our impact analysis might have within the context of the research question, and are defined using the following propositions:

1. The impact analysis will find the same impacts that are found manually by developers.
2. The impact analysis can be successfully executed on standard hardware within a reasonable time.
3. The impact analysis can be applied to the data-access APIs and program architecture used in the project.

We shall discuss the results of these propositions, resulting from the case study, in Section 7.6.

The data that we are studying in this case study are the historical records of a commercial software project. Our subject application is the irPublish™content management system (CMS), produced by Interesource Ltd. At the time we conducted the case study, the application had been in development for five years and was used by FTSE 100 companies and leading UK not-for-profit organisations. It is

a web-based CMS application built using Microsoft's .NET framework, C#, ADO.NET and using the SQL Server DBMS. It currently consists of 127KLOC of C# source code, and uses a primary database schema of up to 101 tables with 615 columns and 568 stored procedures.

For our evaluation to be generalisable, the subject application had to be representative of real world practice for database driven applications. irPublish has been developed using many well-established and commonly used techniques. For example, we see applications of design and architectural patterns proposed by Gamma et al. [1995] and Fowler [2003]. It is also important to note that irPublish has been developed using established software engineering practices such as automated testing, source code revision control, continuous integration and bug tracking. We shall discuss how this relates to the external validity of our findings in Section 7.5.3.

The case study data are the historical versions of the application as stored in the source code repository. For each version in this version history, we know the changes that occurred and the comments of the developer who checked-in the changes. We are interested in versions of the application where changes to the database schema have been made. All database schema changes are stored, in the irPublish repository, as SQL scripts so by identifying any updates, deletions, or creation of SQL scripts, we can identify all changes to the database schema that have occurred. We use the term *schema-version change* to refer to the schema changes occurring between two consecutive versions of the application. Each schema-version change may consist of several smaller changes, the atomic elements of which are individual SQL DDL queries (we refer to these atomic changes simply as schema changes). We refer to the version of the application before the schema-version change as the *before application-configuration*, and to the version after as the *after application-configuration*. The unit of analysis for this case study is a schema-version change with its corresponding before and after application-configurations.

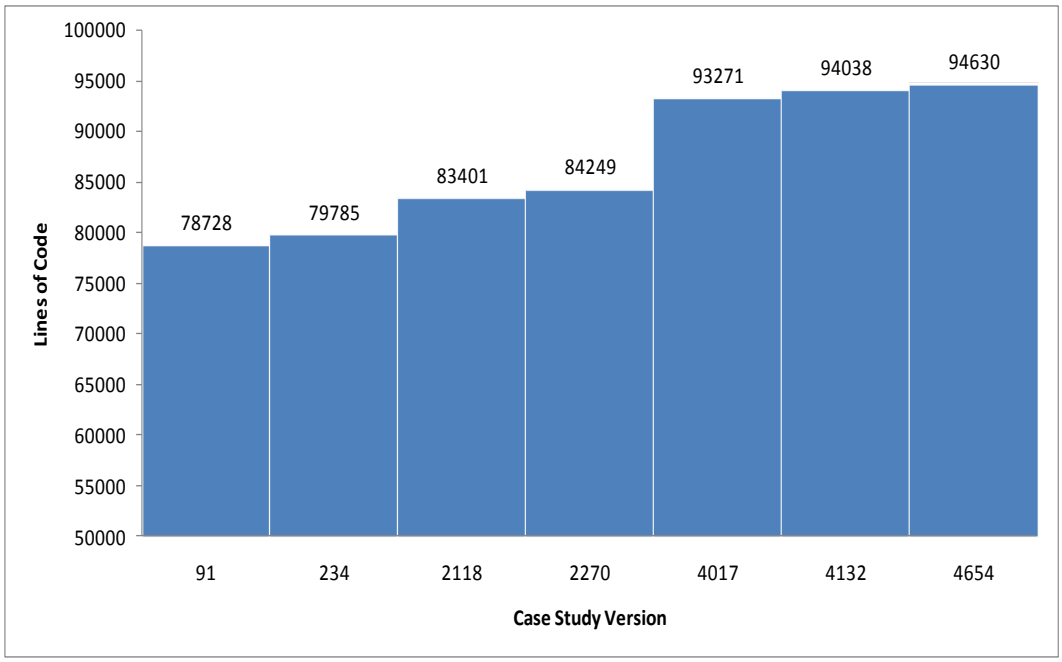

Figure 7.1: Total lines of code by schema version.

We analysed seven separate schema-version changes; each version analysed consists of multiple individual schema changes and corresponding observed changes, as listed in Appendix C. The schema version number is represented by the identifier of the source control check-in that contains the schema change scripts. For example, schema version change 91 is the version of the schema that resulted from the changes of the source code revision number 91; it does not represent the 91st version of the schema. The relative size of these applications can be seen in the graph shown in Figure 7.1. We chose seven schema change versions that had significant changes to the schema and that represent a variety of schema modifications and corresponding application changes.

For each schema-version change, we applied our impact analysis to the before application-configuration and compared the impacts predicted by SUITE, our prototype tool described in Chapter 6, to the actual changes made in the after application-configuration. Any predicted impact that had a corresponding change in the after application-configuration was a *true positive*. Any predicted impact that was missing a corresponding change in the after application-configuration was a *false positive*. Conversely, a *false negative* was a change in the after application-configuration which was missing a corresponding predicted impact. A *true negative* was the correct absence of a predicted impact because no changes were made in the after application-configuration.

7.3 Results

We interpret the data with various quantitative measures that will be used to inform our qualitative measures. We will use the number of true positives, false positives, and false negatives as the quantitative measures for informing accuracy and impact-precision, and we shall use averaged timed executions to inform the scalability and cost measures. We shall present a summary of results here, as appropriate, but the full, raw results of this case study can be found in Appendix C. All tests were run on a virtual machine with a single 32bit, 2.5Ghz CPU and 3GB of RAM, running Microsoft Windows XP.

7.3.1 Accuracy

Table 7.1 shows how the predictions of our tool compare to the observed changes made by the developers of irPublish.

Version	**True Positives**		**False Positives**		**No. Changes**	**Precision**		
	Errors	Warnings	Errors	Warnings		Errors	Warnings	Overall
91	0	0	0	8	7	-	0%	0%
234	0	4	0	17	8	-	19%	19%
2118	0	0	0	18	3	-	0%	0%
2270	0	0	0	0	1	-	-	-
4017	2	3	0	22	31	100%	12%	19%
4132	0	1	0	2	6	-	33%	33%
4654	0	0	0	0	3	-	-	-

Table 7.1: Case Study Impact Accuracy

We omit false negatives from the table for the sake of brevity, but note that there were no false negatives, which means that every change that the developers actually made was correctly predicted by SUITE. This results in a consistent rate of 100% recall, therefore we also omit recall from the table. We omit true negatives from the table as they are too time-consuming to accurately calculate, because they would require repeated analysis of the case-study applications by hand, verifying all points which are unaffected, and this is unfeasible for such large applications.

Table 7.1 shows the number of true positive and false positive impact predictions for each version of the schema in the case study. We record error and warning impact predictions separately, to show the difference in accuracy between error and warning predictions. We also display the number of individual schema changes involved in the schema-version change, because this illustrates the relative size of the change. We provide further information about accuracy using the standard statistical measure of precision, as defined in Chapter 2, and we show the precision rates for errors, warnings and overall accuracy separately. Some of the precision figures could not be calculated because they involved a division by zero. This resulted from situations where there were no true positives or false positives, such as schema-version changes 2270 and 4654, where the schema changes required no application changes, and our tool correspondingly did not predict any true or false positives.

The data in Table 7.1 were obtained using a version of SUITE that included all the optimisations described in Chapter 5; including the data-dependence slicing optimisation. SUITE was executed with a context sensitivity of $k = 4$, which is an average level of context-sensitivity for our case study, and suitable for maintaining a high level of accuracy in the case study application.

The values in Table 7.1 remained the same whether our program was run with, or without the data-dependence slicing optimisation. This shows that our data-dependence slicing optimisation did not omit any parts of the application that were required in order to predict these impacts. This is an important validation of our data-dependence slicing approach, showing that in a real-world application of a significant size, data-dependence slicing can be used to find a subset of the program involved in database query executions, without omitting relevant parts of the application. This is no guarantee that this result will hold in the general case because accuracy of program slicing in general is an open research question [Ranganath et al., 2007]. We can only conclude that data-dependence slicing appears feasible for applications of this type, and therefore appears useful in practice.

The figures in Table 7.1 were also unaffected by altering the values of k, for the k-CFA analysis, between 1 and 8. We might expect to get more false positives as k is decreased, but this was not the case. Our requirement for using high levels of context-sensitivity, was to avoid unacceptable levels of inaccuracy. The changes observed in this case study do not validate this requirement, showing that the level did not affect the accuracy. We believe that despite these results, our requirement for high levels of context sensitivity still hold, for the following reasons.

Although decreasing k did produce more approximations, none of the approximated values affected the case study change scripts. By manually studying the case-study application we found several places where a k value of at least 4 was required to relate a query definition correctly to the use of the query

results. There are also several places in the application where the design patterns used, would result in over-approximated queries when $k \leq 2$. These approximations exist in the extracted facts, when k is low, and would result in a loss of accuracy if a different set of changes occurred in the case study. Therefore, we can see that a loss of accuracy can occur when k is low, even though it was not produced by changes analysed in our case study.

We still believe that given the architectural patterns that are in use in practice, that variable levels of context sensitivity are required, even though our case study does not show this. If an application uses some architectural patterns, and the impact analysis uses low levels of context-sensitivity, there is still the potential for the results to be so inaccurate that they are unusable. This did not happen in our case-study, but we believe that it is not unlikely in practice. Therefore despite the results of our case study, we still maintain that a variable level of context-sensitivity is required.

On average, in this case study, each impact calculation script produced 0.1 true positives and 1.5 false negatives. We expect these figures to vary quite widely in other cases, dependent upon the change scripts and the types of change that are observed in the application, although we can observe from these numbers that the amount of predicted impacts is low. We can see that the maximum number of predicted impacts is never higher than 25, even with complex schema-version changes consisting of up to 31 individual changes. We can compare this to the amount of queries that we discover for each application-configuration which, in the largest case, is 1431 possible queries. We have no guarantee that these results will hold in the general case, but these results indicate that our approach can produce a small number of predictions from a large number of potential queries.

We have purposely created our impact calculation scripts to predict errors with higher precision , than predicting warnings. We generally predict errors when there is a very high level of confidence that an error will be a true positive, and we predict warnings often when there is a much lower chance of the impact being a true positive. In this case study we predicted only two errors, both of which were true positives, whilst we predicted many warnings that were false positives. It is important that our approach has the ability to predict impacts with varying levels of confidence in this way, allowing the stakeholders to make more informed judgements, paying more attention to impact predictions with a high probability of being true positives. Whilst we only use the categories of errors and warnings, we discuss future work of creating further confidence levels in Section 8.2.

Whilst not shown in Table 7.1, the case study exhibited no false negatives. This means that every observed change was successfully predicted by our analysis resulting in 100% recall for all schema-version changes. This does not however mean that our analysis is conservative; there are several places in the applications that could be affected by change and SUITE would not have been able to predict the effect of these impacts. The majority of queries were predicted accurately, but manual inspection of the analysis results indicates where false negatives could occur. Queries that are defined using features such as reflection and meta-programming are generally not be amenable to current static analysis techniques [Ryder, 2003], and could cause false negatives. Queries that are defined, modified or executed by APIs which are unknown to the impact analysis, could cause false negatives. Advanced language

features such as generics, lambda expressions and closures, type inference and dynamic typing could all cause problems for the analysis as well. Many of these programming language features are amenable to program analysis in general, but are not currently implemented in SUITE. Extending SUITE to analyse to the full features of a version of C# is beyond the scope of this dissertation, due to the required engineering effort. It is also an open question as to what language features can be included in such an analysis[2]. However, the fact that our case study observed no false negatives indicates that this approach appears feasible, and has the potential to be useful in real-world applications, as we shall discuss in Section 7.4.

7.3.2 Impact-precision

As defined in Chapter 2, impact-precision defines the level of detail with which an impact can be identified. Impact-precision can be measured in two dimensions. First, the precision of predicted queries and, second, the precision of predicted impact locations in the source code.

Impact-Precision of Predicted Queries

Each regular string is an abstract representation of a set of concrete strings, as defined by the concretisation functions in Figure 3.2. The impact-precision of predicted queries can be defined as the degree to which the set of predicted concrete strings differs from the strings that can actually occur during the execution of the program, and the amount of over-approximation present in the regular string.

For example, the regular string "$SELECT\ x\ FROM\ table1$;" evaluates to the set of concrete strings consisting only of the string "$SELECT\ x\ FROM\ table1$;". If the set of queries that can actually occur in the application is also a set containing only the string "$SELECT\ x\ FROM\ table1$;", then this is an optimally precise prediction. When a query is dynamic, its actual values are likely to be a set of possible values. For example a query "$SELECT\ x\ FROM\ table\{0,1,2,3,4,5,6,7,8,9\}^*$;" could be the most precise prediction possible for a dynamic query where the table name consists of the word *table*, followed by a number. A slightly lower impact-precision regular string prediction "$SELECT\ x\ FROM\ table.^*$;", over-approximates the arbitrary number as a repetition of any character. An even lower impact-precision prediction would be ".*", which simply marks the entire query as being any possible concrete string. All the predicted queries we have described here are accurate, in that the set of concrete strings that they represent, include the actual queries that might occur. They have high impact-precision if they clearly resemble the actual queries with few over-approximations. They have low impact-precision if they do not clearly resemble the actual queries with many over-approximations.

Schema version 234 has a true positive impact for the change `ChangeSp1`. This impact is predicted by the `csAddNewOptionalParamsToStoredProc.rml` change script with the table parameter set to "$cp\_ContentItemRating\_ins$". The change script predicts that the query "$cp\_ContentItemRating\_ins$" will be executed. This regular string evaluates to only one concrete string that is the exact query that will be executed, and therefore has high impact-precision. Because the

[2]languages such as Java 1.4 have been covered completely in some complex program analyses, such as program slicing [Ranganath et al., 2007], but C# 2.0 and later versions of Java contain more advanced features that may prove difficult to include in program analysis.

query is composed statically, it can be predicted with high impact-precision. Out of 39 distinct predicted impact queries in the case study, 27 queries are predicted with optimal impact-precision in this way; these are largely queries that are defined statically, or with minimal dynamism.

Other queries are not predicted so such high impact-precision. For example, the change `ChangeSc1` for version 234 predicts that the query
"$DELETE\ FROM\ t\_ContentItem\ WHERE\ RealContentItemId\ =\ .*$" will be executed, where ".*" represents the repetition of any character. This is because the identifier used in the *WHERE* clause, is supplied at runtime and cannot be predicted by the analysis, because it is not a string, or query representing data type, that is included in the query analysis. Despite this unknown value the remainder of the query is predicted precisely. Out of 39 distinct predicted impact queries in the case study, 8 queries are predicted with a small approximation of a *WHERE* clause value in this way. These are largely queries that include a dynamically supplied identifier in the *WHERE* clause.

Other predicted query strings may become complex regular strings, which include many alternations and repetitions. Such regular strings occur when queries are constructed dynamically, especially in loops or recursive methods. If the regular string includes many repetitions or alternations, then the set of concrete strings it maps to can be very large. Out of 39 distinct predicted impact queries in the case study, 4 queries are predicted with a large amount of approximation, including very complex alterations and repetitions. These are largely queries that are dynamically constructed, typically being built in many loops and involving 30 or more separate substrings.

We have tried to ensure that our analysis has high levels of impact-precision, however, there is one point at which impact-precision of query strings is sacrificed for the sake of performance. String approximation is used when the widening of large regular strings occurs. We approximate very large strings as ".*", as described in Chapter 6, reducing the amount of iterative widening that occurs. We have found that, in practice, performance is dominated by the cost of the partial ordering comparisons produced by iterative widening, and that our analysis cannot scale to large real-world applications without this optimisation, as we shall discuss in Section 7.3.3. This approximation leads to a drop in impact-precision, meaning that parts of very large dynamic queries are approximated by the string ".*". This can produce more false positives and false negatives, affecting accuracy and impact-precision, but as discussed in Chapter 6, we record the value of the strings that were approximated using impact calculation facts, outputting these facts into the RSF, and consequently the approximated information is available to the impact calculation scripts. Therefore, the effect of the approximation of large regular strings, is a slight drop in impact-precision of the query that is presented to the user as being executed at a given location. Only one of the predicted queries in the case study was affected by this approximation of long strings, where the beginning of a very long dynamic query string, in version 4017, gets approximated as ".*". The query is constructed in a method with 50 separate appends to a *StringBuilder* object, many of which are conditional or appended within loops. This was a highly complex query and would be difficult to understand as a fully widened regular string, so the approximation may be considered as only a small decrease in impact-precision. Out of 39 distinct predicted impact queries in the case study, 1 query was

affected by large regular string approximation.

We shall discuss how these impact-precision results affect the potential usefulness of our impact analysis in Section 7.4.

Impact-Precision of Impact Locations

Other than the impact-precision of the regular string queries, the other dimension of impact-precision is the impact-precision of the source code locations that are highlighted. `ChangeSp1` for version 234, predicts that a query will be executed in the file "MembershipData.cs" on Line 283, and that the query was defined in the file "MembershipData.cs" on Line 264. This shows that our analysis locates impacts at the level of individual lines of code. A low impact-precision analysis might only highlight affected methods, or classes. A high impact-precision analysis might highlight individual statements in the compiled program, and thus highlight subexpressions within given lines of source code. Whilst it is possible to change our analysis to achieve this higher level of impact-precision, we believe that the additional benefit to the stakeholders would be minimal, whilst a significant increase in cost and complexity could be incurred by having to maintain the mapping from individual instructions to subexpressions in the source code.

We consider an impact prediction a true positive if the actual line of source code that was changed is identified by the predicted impact, with an appropriate message that could indicate the reason why the change is required. Therefore, each true positive indicates that the results had high enough levels of impact-precision to identify the actual line of code that required changing.

We shall discuss how these impact-precision results affect the potential usefulness of our impact analysis in Section 7.4.

7.3.3 Cost, Efficiency and Scalability

Dead State Removal

The first performance optimisation described in Chapter 5 was dead state removal. We measured the average execution of the dead-state removal functions for all executions where the context-sensitivity was $k = 4$, representing a standard execution of our analysis. The average time dedicated to dead-state removal was 5.9 seconds, which can be compared to the average execution time for all versions of the application, of 21 minutes 18.3 seconds. Therefore, the cost of the dead state removal is on average 0.3% of the total execution time of the analysis.

Figure 7.2 shows the timings of version 234 with and without the dead state removal optimisation. The "data only" series shows the total execution times calculated using all of the optimisations, including data-only dependence slicing, abstract garbage collection and dead state removal. The remaining series shows how the analysis runs without the dead-state removal optimisation. We can see that when run without the dead state removal optimisation, the effects of increasing the context-sensitivity are much more pronounced. The unnecessary operations that are mitigated by dead state removal are not directly related to the level of context-sensitivity, as might be inferred from this graph. Instead, the level of context-sensitivity multiplies the cost of these mitigated operations, therefore when these operations are

removed by the optimisation, the effect of increasing the level of context sensitivity is greatly reduced.

The series without dead state removal plateaus around $k = 6$ because the fixed point solution does not substantially change by increasing the context-sensitivity further.

When $k < 3$ the dead state removal seems to provide little or no benefit. This is because the cost of creating the liveness information is not recouped. Our current implementation of liveness analysis is not efficient, and is based upon a simple intraprocedural dataflow algorithm. If we were to use a dataflow framework using the bit-vector data structure and a more efficient worklist algorithm, such as the intraprocedural analysis framework provided by the Phoenix Framework [Microsoft Phoenix], we estimate that the cost of the liveness analysis could be reduced, and dead-state removal could be used to improve performance even at low levels of context-sensitivity.

If run without the dead-state removal and without the data-dependence slicing optimisation, no version of our case study terminates before running out of memory[3]. This shows that it is important that both optimisations are used, and that they provide a significant reduction in the cost of the analysis.

Figure 7.2 shows data for the schema-version 234, but for the remaining versions, very similar results can also be observed, which we omit for the sake of brevity; they are included in Appendix C.

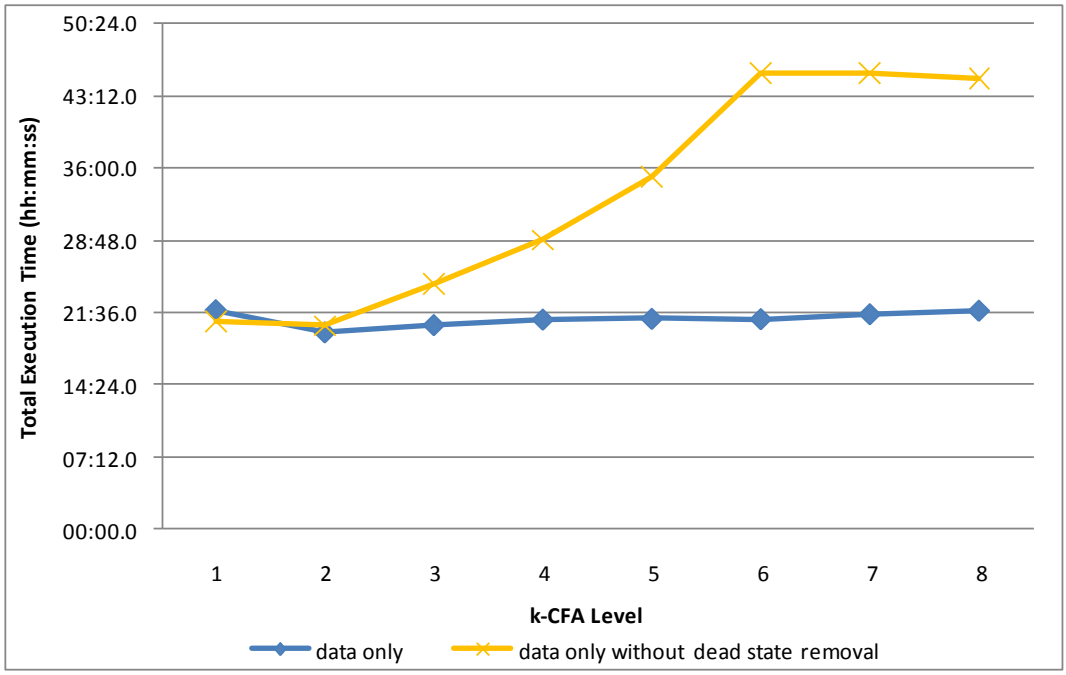

Figure 7.2: Total Analysis With and Without Dead State Removal for Version 234

Abstract Garbage Collection

The abstract garbage collection optimisation described in Chapter 5, has an average time cost of 2.6 seconds for the execution of all versions of the case study where the context sensitivity level is set at $k = 4$. Therefore, the average cost of the abstract garbage collection is 0.2% of the total cost of the

[3]As described, the test hardware used 3GB of RAM, but because we were using the 32bit .NET virtual machine, the memory limit for our application was 2GB.

analysis.

During the analysis of the case study, we garbage collect 60-61% of the total regular strings, and 64-68% of the total heap items. This is a considerable saving, and when run without the abstract garbage collection, our analysis fails to complete on any of the cases, running out memory. Further results about the benefits of abstract garbage collection are therefore difficult to obtain, as the case study cannot be executed without it, providing no basis for comparison. We can conclude that both dead state removal, and abstract garbage collection, provide significant performance improvements to the analysis.

Data Dependence Program Slicing

The program subset obtained by the data-dependence slicing is an average of 20.43% the size of original program. Across all versions of the case-study, this figure only varies between 20.08% and 20.95%. The average cost of obtaining this data-dependence slice is 41.7 seconds, for all versions of the case study, ranging from 37 to 51 seconds. This cost of slicing is an average of 3% of the cost of the total average analysis time of all cases where the context-sensitivity is $k = 4$.

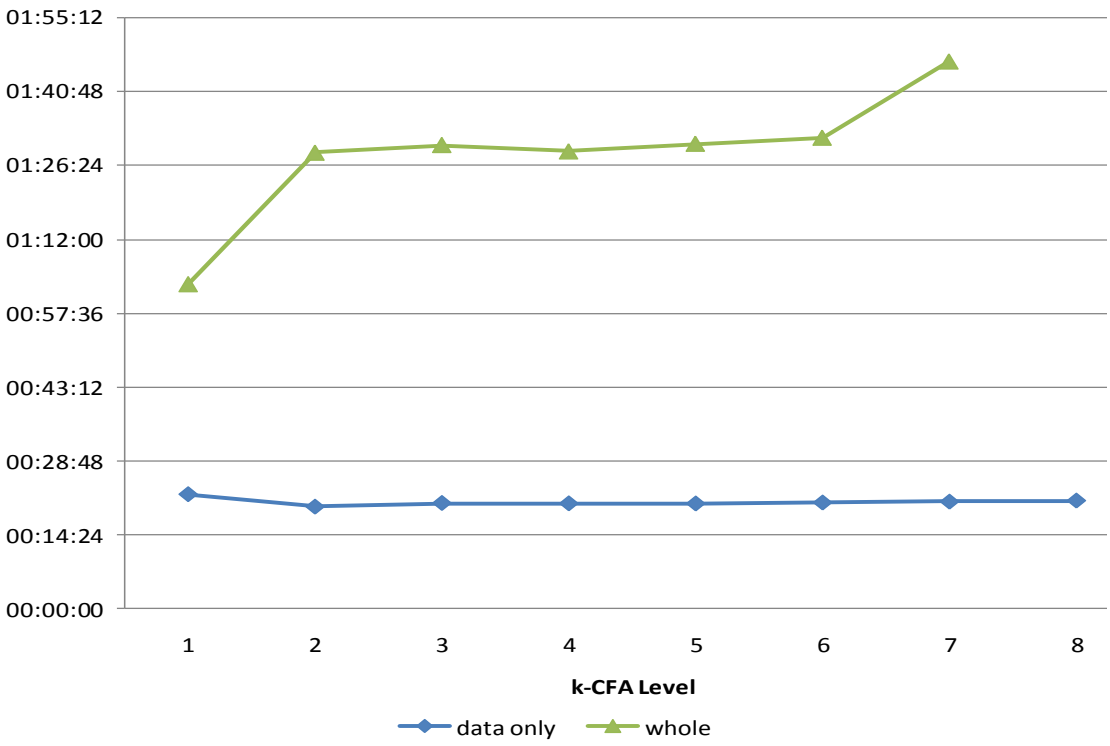

Figure 7.3: Total Analysis Execution Times for Version 2118

Figure 7.3 shows the total analysis times for version 2118 of the case study, as the level of context-sensitivity is varied. The two lines show averaged total execution times, with and without the data-dependence program slicing optimisation.

It is clear that the execution with data-dependence slicing has a lower cost than the whole program analysis. The execution time of whole program analysis is at its minimum at $k = 1$, and plateaus around $k = 3$. This is because the majority of the application has reached a fixed point by $k = 3$, and increasing

the value of k only marginally affects the execution times. The cost of increasing k is also mitigated by the liveness analysis, abstract garbage collection and string approximation of large strings, as previously discussed. This means that the cost of increasing k does not have the dramatic effect that could be expected, and that was observed in our earlier work [Maule et al., 2008].

When $k = 6$ the whole program analysis slows down significantly as SUITE runs out of memory. There is no value for $k = 8$ shown on the graph for whole program analysis, because SUITE failed to terminate before running out of memory. The point at which the memory is exhausted for the largest case study version, 4654, is when $k >= 5$, and in version 4017 and 4132 SUITE runs out of memory at $k >= 4$. The reason why 4017 and 4132 run out of memory earlier than 4654, despite being smaller in terms of lines of code, is that they contain methods with complex dynamic strings that were removed from the project by version 4654. When run with the data-dependence slicing optimisation, SUITE does not run out of memory for any of our case study versions.

When $k = 1$ the analysis with data-dependence slicing is slightly slower. This is because less strings reach the size where the large regular string approximation is used, and as a result, slightly more partial ordering operations are used. The analysis then shows a very slight rise as k is increased, but again, the gradient of this rise is not dramatic because of the effects of dead state removal, abstract garbage collection and large regular string approximation.

The remaining versions of the case study show very similar results for both the whole program analysis and the analysis with the data-dependence slicing optimisation. The timings for the remaining case study versions can be found in Appendix C.

We can conclude from these results that, for our case study application, the data-dependence slicing optimisation results in a less expensive analysis in terms of memory and total execution time, although we have not yet discussed how the analysis scales as the size of the analysed program varies, which we shall discuss next.

Overall Performance

Figure 7.4 shows a graph of the timings of all versions of our case study. The horizontal axis shows lines of code and the vertical axis shows averaged total analysis time. The points correspond to executions of each version of the case study application, executed with a context-sensitivity of $k = 4$.

The general trend, in both lines, is a slight increase in execution time as the lines of code increase. This indicates that as the application size increases, so does the analysis time. The points for version 2118 and 2270 are higher for the whole program analysis, than 4017, 4132 and 4654, despite having less lines of code. This can be explained because versions 2118 and 2270 contain methods with complex string manipulations that do not occur in version 4017 onwards. Even though versions 4017 onwards are larger in terms of overall code size, they do not have as many string manipulating functions, and so are less expensive to analyse. The series for the data-dependence slicing optimisation is not affected by this issue, because the data-dependence slice excludes the methods that make 2118 and 2270 more expensive, because they are not involved in creating database queries.

Lines of code is not an accurate indicator of the cost of our analysis. The cost of the analysis is

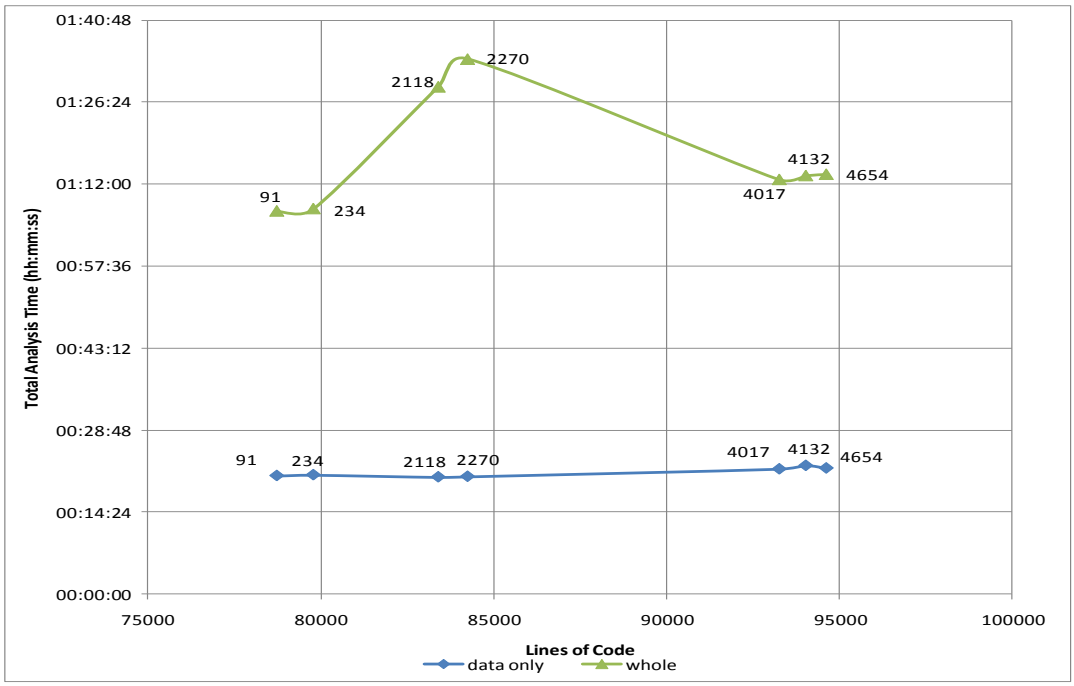

Figure 7.4: Total Analysis Execution Times for All Version by Lines of Code

affected by the amount and complexity of string and query data type manipulating functions, but we have no reasonable metric for measuring this. Instead, we can conclude that our analysis with the data-dependence slicing optimisation, scales at least as well as the whole program analysis (has a similar overall trend), but has a lower overall execution time cost. We cannot determine if the data-dependence slicing optimisation is more or less scalable than the whole program analysis, without conducting further studies. We need to examine the performance of these algorithms on larger and smaller case studies with more variation in size, and different types of applications with different data access practices, and different architectures. This would require significant further research, therefore currently lies outside the scope of our research.

We have found that the cost of the analysis is dominated by the cost of the partial ordering comparisons between regular strings, taking up approximately 70-80% of the total time cost of the analysis. We described how our analysis implements this partial ordering in Chapter 6. The partial ordering operation is complex, and a partial ordering comparison occurs after every widening operation. When very long strings are widened, it can take many iterations to reach a fixed point, and on large regular strings this will induce the partial ordering operation many times. Therefore, programs with very large complex dynamic strings are much more expensive to analyse. Related work in model checking could provide increases in performance by using different algorithms and data-structures for calculating partial ordering, such as pushdown automata [Hopcroft et al., 2006]. Because our current algorithms are already efficient and because the investigation of alternative techniques would require significantly more research, we highlight the investigation of alternative techniques as possible future work.

Because analysis of long dynamic strings is so expensive, the majority of the cost of the analysis

is incurred by only a few analysed methods, whilst the rest of the program is comparatively low-cost to analyse. In our case study, in each version, two methods consume over 60% of the analysis time, whilst the remaining approximately 1900 methods had a much lower time cost. These two, expensive-to-analyse methods use intensive string manipulation, one creating large query strings, the other used for creating dynamic textual reports.

Because dynamic strings are typically built using string concatenation libraries, such as the .NET *StringBuilder*, we can judge the cost of calculating large strings by removing these libraries from the analysis. In our case study, if we remove the *StringBuilder* class from the analysis, the average execution time drops from approximately 20 minutes to under 2 minutes. This shows just how much the analysis time can be increased by using more complex dynamic strings, and shows that it is the widening and partial ordering of complex strings that is the dominant cost.

The observation that the execution time cost of the analysis is dominated by the analysis of a small number of analysed methods leaves two open questions. First, how common are these types of method, and what types of application will they be found in? Second, is there a way we can optimise the analysis to account for a small number of expensive string manipulating methods? We highlight the investigation of these problems as possible future work.

The scalability graph shown in Figure 7.3 differs from the scalability that we described in our earlier work [Maule et al., 2008]. This is because our implementation, SUITE, now includes more libraries and therefore has more dynamic strings in the analysis, increasing the analysis time. We have also increased the impact-precision and accuracy of the analysis, increasing the total cost further. The trends of the graphs are also different because of the introduction of liveness analysis and abstract garbage collection, which have mitigated some of the more obvious effects of increasing context-sensitivity. This makes Figure 7.3 generally higher and flatter than the result shown in our previous work [Maule et al., 2008], and the results presented here are more mature than our previous work.

7.4 Discussion

In Chapter 2 we defined a *useful* analysis as:

> An analysis which has the accuracy, impact-precision and cost, suitable to inform schema change in typical commercial enterprise application development, with benefit to the stakeholders.

To determine whether our analysis is useful, by the definition above, we need to establish that the analysis is beneficial to the stakeholders. In the following sections we shall discuss how the accuracy, impact-precision, cost and efficiency of the impact analysis relate to its overall usefulness, and whether the analysis could have provided benefit for the stakeholders.

7.4.1 Accuracy

If accuracy is low, there may be a large number of false positives and false negatives. If there are too many false positives, then the cost of confirming/denying each predicted impact might not improve upon the cost of performing manual code inspection without our impact analysis. This means that if there

are too many false positives then there will be no benefit to the stakeholders. If there are too many false negatives then the stakeholders may need to perform manual source code inspection to catch the missing false negatives. If the cost of the false negatives is high enough then there may be no benefit to the stakeholders in using the impact analysis because they will have no confidence in the results. This means, that in order to have a useful level of accuracy, we need a low level of false positives and false negatives.

The case study shows that no false negatives occurred. This is an encouraging result, and shows that results we obtained in the case study, would likely have been useful in practice. The case study shows that in industrial applications we can achieve a low level of false negatives, but we need further evaluation to see if this result holds in the general case. The absence of false negatives here, does not show that they will never occur, but does indicate that there is a low probability that false negatives will affect the overall analysis. This low probability could mean that false negatives are rare enough, so that the results are still useful even if some false negatives occasionally occur. But to objectively evaluate this, we need to evaluate the benefit of a predicted impact and the cost of a false negative. To find these figures is difficult, as they are highly dependent upon the context of the software project, and can only be reliably measured in a real industrial project. It is impossible to say whether the level of false negatives that could be produced by our approach is useful or not, without running more investigation on real projects.

The maximum number of predicted impacts was 25 for a particular schema-version change, which resulted from an analysis that found 1431 possible queries. Without further evaluation we cannot say that this level of false positives would be useful in practice, and we cannot say that these results will always apply to the general case. However, 25 predicted impacts is an encouragingly low number when compared to the 1432 possible queries, and indicates that this level of predicted impacts is likely to have been useful in practice.

Although the impact analysis is not conservative, it has still maintained a high level of accuracy, and appears to be useful without being conservative. It is unclear if making the analysis conservative will increase the time and space costs and decrease the accuracy and impact-precision, to the point where the analysis is no longer useful. If the levels of cost, accuracy and impact-precision can be maintained, whilst also making the analysis conservative, then this is an ideal solution. To investigate if this is possible, would require a significant effort in adapting the analysis to show that it is safely conservative.

These results indicate that our impact analysis appears to have a level of accuracy that could make it useful in practice, and that it is likely that our tool would have been useful during the development of the irPublish system. It is likely that similar results will be observed in similar applications, although further investigation is required to confirm this.

7.4.2 Impact-Precision

Impact-precision, with respect to usefulness, means that the results are detailed enough to allow the stakeholders to correctly identify if an impact is a true or false positive. The two dimensions of impact-precision we consider are the precision of the predicted query, and the precision of the source code

locations identified as impacts.

The results of our case study have shown that the majority of queries, around 70% of those in the case study, were predicted with perfect impact-precision. 20% of the queries have a small amount of approximation for a value in the where clause. 10% involve complex dynamic queries that have lower impact-precision.

With respects to usefulness, this means that 90% of the queries are easily identifiable, containing little or no string approximation. This would allow the stakeholders to identify the semantics of the query quickly, and they could use this information to manually judge if the impact is a true or false positive. The remaining 10% of queries that are highly dynamic, will require more work by the stakeholders, in order to identify the full range of queries that can be produced, and identify whether the impact is a true or false positive. Even when the query has been widened and significantly approximated, the regular string approximation may still contain useful information to aid the stakeholders, and can still be beneficial compared to analysing the source code manually. Therefore, we argue that it is likely that the level of impact-precision we have seen for predicted query strings is useful to stakeholders in practice.

The impacts highlight individual lines of code, and because each true positive in our case study includes the lines of code that were actually changed by the developers, we suggest that this is sufficiently high level of impact-precision. Less impact-precision would not highlight the lines of code that were changed, and therefore, we would have false negatives, which were not observed in the case study. The only way to be have higher impact-precision is to highlight impacts to the statement/instruction level, and it is unclear if this will result in an increase in usefulness, and whether the additional cost would be justified. We argue that the results observed in our case study, indicate that a impact-precision level of individual lines of code, would be useful to the stakeholders of the schema change in practice.

The warning or error level, and the message associated with the impact, give the stakeholders more information about the impact. This information should help diagnose whether the impact is a true positive or not, and should help diagnose what needs to be changed in order to reconcile the impact. It is likely that this information would be useful to the stakeholders in identifying if an impact is a true positive or not.

All the levels of impact-precision that we have described here appear to indicate that our analysis would have been useful in practice. However, to determine if this is really the case we need to conduct a real-time case study or conduct controlled experiments to determine the true usefulness of our analysis, by observing how this information is actually used in practice.

7.4.3 Cost

Cost with respect to usefulness, means that the time and space costs of performing the analysis must not outweigh the benefit gained. The cost of performing the analysis for our case study, was on average approximately 20 minutes of execution time on a standard desktop computer, where the largest of the applications analysed was 95 KLOC. We have no way of effectively measuring the benefit of the impact analysis using our current results, although, intuitively the cost of 20 minutes execution time is not high, and we would suspect that, in practice, this would be an acceptable cost. Further evaluation in real-world

development is required to confirm this.

We do not yet know if our impact analysis scales to much larger applications, but 95 KLOC was the largest version of our case study, which represents a large industrial application, and gives a good indication that our analysis is likely to be useful on similar real projects. The trend in Figure 7.4 also indicates that our analysis will scale to much larger applications, but to prove this, we need further evaluation of larger applications and different types of application that may have different analysis costs.

7.4.4 General Usefulness

The evaluation so far has focused on feasibility, and likely usefulness. We have shown using a historical case study that the application of impact analysis appears to be feasible, and is likely to be useful in practice.

To objectively determine if our analysis is useful in practice, we need to further investigate and examine how our analysis can be used in practice, and how the stakeholders would actually make use of the information provided. This future work could be in the form of a real-time case study, using our impact analysis on a real world development project, or it could be based upon a series of controlled software experiments.

This does not mean that our choice of using a historical case study was bad or incorrect, and we argue that the choice of using a historical case study was the best approach for the evaluation within the constraints of our research. Our evaluation has shown that our approach is feasible for large commercial programs, and has potentially useful results, and by showing this, we justify a case for using more resources for evaluating the usefulness of our impact analysis further. The further evaluation of usefulness is likely to be much more expensive than the evaluation of feasibility and potential usefulness that we have presented here, so it was important that we established that the overall impact analysis approach is feasible and potentially useful, before committing resources to a full evaluation of the usefulness of impact analysis.

An important result of our evaluation is that our approach does not have conservative query analysis, or conservative impact calculation, yet the results still appear to be useful (no false negatives, adequate accuracy, impact-precision and computational cost). Again, we need further evaluation to show that this result holds beyond our case study, but the results are a positive indication that this approach appears to be useful and scalable without being conservative. This observation leads to the following question; would a conservative analysis be feasible? We would expect a conservative analysis to reduce the accuracy and impact-precision of the analysis, whilst increasing the cost. It is, therefore, unclear whether a conservative analysis would be more or less useful than the analysis we have evaluated here. To objectively evaluate this, we need to substantially alter our analysis to be conservative, prove its soundness, and re-evaluate it. Again, due to the cost, we highlight this investigation as future work.

7.5 Threats to Validity

7.5.1 Construct Validity

Do our measurements correspond to the real-world phenomenon we are trying to measure?

For each schema-version change we examined the set of individual schema changes that occurred, and manually attributed schema changes to impacts in the after application-configuration. The main threat to construct validity in this case study is the question of whether the measured impacts really do correspond to the actual schema-change-to- impact relationships that occurred in practice. Although many of the schema changes were trivially attributable to impacts, when there was ambiguity or uncertainty, we used the comments supplied with the schema-version change from the source code repository, and then verified with the lead developer of irPublish that our assumptions were correct. We are therefore confident that all schema changes and corresponding impacts that we measured are valid, in that they relate to schema changes and impacts that occurred in practice.

7.5.2 Internal validity

Do we correctly establish causal relationships, by considering all possible causes?

We have tried to mitigate all threats to internal validity for the measurements we present here. For the execution times in Appendix C, we present averaged execution times to eliminate any spurious timings that may be caused by competing operating system processes. We also run the applications in a controlled way, on the same system, with only the minimal operating system processes running. We have verified on other machines that similar timings hold, although we do not present these data here.

The accuracy results, such as true and false positive rates, were extracted from SUITE using automated test scripts, and the results can be deterministically reproduced. The final accuracy results we present here were also verified by hand to make sure they were correct.

7.5.3 External Validity

Is our case study application representative of other real-world applications?

irPublish has been developed using many well-established and commonly used techniques. For example, we see applications of design and architectural patterns proposed by Gamma et al. [1995] and Fowler [2003]. It is also important to note that irPublish has been developed using established software engineering practices such as automated testing, source code revision control, continuous integration and bug tracking. We argue that because these patterns and practices are in widespread use, this case study is a good example of real world practice, and therefore, our findings may also apply to other similar applications.

The DBMS changes that we have observed in this case-study are typical of the types of changes that we might also see in other database driven applications, as shown by the research of Sjoberg [1993] and the work of Ambler and Sadalage [2006]. Our results should, therefore, generalise to other database applications that use relational databases.

For other types of database, such as object-oriented or deductive databases, we cannot expect the same types of changes to occur, and consequently the findings of our case-study only apply to applications that use relational databases. The models being fundamentally different, may not exhibit the same kind of impedance mismatch that exists between relational databases and object-oriented applications. Therefore a different approach, with different trade-offs, may be more applicable, such as the approach of Karahasanovic [2002] for impact analysis of object oriented database schema change.

For applications that use relational databases, but use different database access technologies, different query languages and different DBMS vendors and features, we expect our results may be applicable, in whole or in part. The impact calculation scripts, as described in Chapter 5, are extensible and customisable. It is quite likely that a different DBMS vendor, a different data-access API or a different query language would require very similar impact calculations to that used in our case study. The fact extraction functions and impact calculation scripts can be customised to accommodate these differences, and we are not aware of any fundamental differences in these approaches that would prevent our findings from applying to other data-access practices.

Another question that pertains to external validity of our results is: *Are the costs of obtaining program slices, and the size of the program slices, representative of what could be obtained in other real-world applications and using other slicing tools?*

To replicate the size and cost of the data-dependence slicing we attempted to use the Indus [Ranganath et al., 2007] and CodeSurfer [Anderson and Zarins, 2005] slicing tools. The results of our experiments were inconclusive because the case-study applications of similar size and complexity often had problems being analysed by these slicing tools. Even though it has been shown that these tools can scale to large applications [Binkley et al., 2007b] we had trouble replicating these results due to problems with the libraries involved in the case-study applications, and the tools encountering bugs or problems that prevented us from obtaining reliable results. These slicing tools are both very complex and relatively immature, both being research prototypes. The large applications we are analysing (70 KLOC and higher) usually involve complexities and external dependencies that caused problems with these tools, and as such we did not obtain reliable results by which to replicate our findings. Even if we could obtain reliable results from these applications, slicing tools such as Indus and CodeSurfer are research prototypes, designed for numerous analyses that are not necessary for data-dependence only slicing.

Related work that uses slicing based optimisation techniques, only use full slicing on much smaller applications [Dwyer et al., 2006], therefore this work cannot be used to justify our claims about the costs and size of data-dependence only slicing. Therefore, one of the main open questions of this research is how slicing algorithms scale, and the engineering effort required to answer that question places it outside of the scope of our research.

If an SDG is already built, slicing can be executed very cheaply [Binkley et al., 2007b] and we suspect that even if slicing scales poorly, even for imprecise data-dependence slicing, then we may still be able to amortise the cost of building the SDG against our analysis, and other analyses that make use of the SDG.

To see if our results for the cost and size of data-dependence slicing can be replicated is a priority open question, and the expense of conducting this research means that it is outside the scope of our research and regarded as future work.

7.5.4 Reliability

Could someone else repeat this experiment and obtain the same results?

We have not provided SUITE as a publicly available tool, neither is the source code for the irPublish

CMS publicly available. This means, that the exact results we present here cannot be directly replicated. This does not mean that our overall results cannot be replicated. We have provided the details of our analysis in Chapter 3-6, using precise notations that are sufficiently detailed to allow a full implementation in many modern programming languages. We also provide the full source code for our example application in Appendix D, with which an implementation can be verified and tested. This means that although our exact case study results cannot be replicated, there is more than sufficient information to replicate our analysis and verify that it is functioning correctly by analysing the provided sample application. We have also described performance optimisations, in Chapter 5, that will help guide an efficient implementation, and that will help to replicate similar results.

7.6 Summary

In this chapter we have presented an evaluation of our impact analysis approach using a historical case study of the irPublish application. We discussed why a historical case study was the most appropriate form of evaluation, and presented the case study approach.

The case study investigated the following propositions:

1. The impact analysis will find the same impacts that are found manually by developers.
2. The impact analysis can be applied with a reasonable time and memory cost.
3. The impact analysis can be applied to the relevant data-access approaches.

The propositions can be confirmed as follows:

1. The impact analysis did find the same impacts that were found by the developers, with no false negatives, but with a tolerable number of false positive warning impacts.
2. The time and memory cost for the analysis of a maximum 95 KLOC, was on average 20 minutes on a standard desktop machine.
3. The analysis was applied to a real-world project and was capable of analysing all the required data-access approaches, including static and dynamic queries, stored procedures and various data-access APIs.

These propositions were proposed in the context of a research question, which was:

> Is an automated database schema change impact analysis feasible for large commercial applications?

The answer to this question, is yes, it is feasible that automated database schema change impact analysis can be used upon large commercial applications. We base this answer upon the evaluation of the accuracy, impact-precision and cost of our approach that we have discussed in this chapter. We have discussed that the levels of accuracy, impact-precision and cost appear to be feasible, and indicate that our tool is likely to have been useful in practice.

After identifying that our approach is feasible and likely to be useful, the open research questions are:

1. How useful is this analysis in practice, can the stakeholders make beneficial use of the information that our impact analysis provides?

2. What are the limits of this analysis, in terms of efficiency, scalability, accuracy and impact-precision, and applicability?

We shall discuss these open questions and other future work in Chapter 8.

Chapter 8

Conclusions

In this chapter we shall describe the contributions that have been made before discussing the open questions and directions for future work.

Some of the most recent recommended industrial practice for managing databases shows that the impacts of database schema changes upon applications must often be estimated manually by application experts with years of experience [Ambler and Sadalage, 2006]. Performing change impact analyse manually, by hand, is a fragile and difficult process, as noted by Law and Rothermel [2003] who show that expert predictions can frequently be incorrect [Lindvall M., 1998] and impact analysis from code inspections can be prohibitively expensive [Pfleeger, 1998]. For these reasons, we have argued that the current methods for assessing the effects of database schema change upon applications are, in many cases, inadequate.

With this problem in mind the original goal of this research was to investigate alternative solutions and, in particular, to investigate automated tools for estimating the impacts of database schema changes upon applications.

8.1 Contributions

In this dissertation we have described the thesis that an automated impact analysis for database schema changes is possible, and could be beneficial in practice. We have described such an automated impact analysis for estimating the impacts that relational database schema changes have upon object-oriented applications. We have also shown, using a historical case study, that the approach scales to code sizes of practical relevance and that the analysis gives potentially useful results. The specific contributions we have made to the investigation of this thesis are as follows:

8.1.1 Query Analysis

We have presented a novel dataflow analysis that extracts query information from an application, suitable for use in impact analyses, which we call query analysis. We showed how query analysis has three major requirements. First, a variable level of context-sensitivity is required. With a context-sensitivity level of less than $k = 2$ (or an equivalent context-sensitivity using an approach other than k-CFA) it is very likely that the accuracy of the analysis will suffer. The potential loss of accuracy is very high, because many design patterns that are used in practice would otherwise cause the analysis to over-approximate

the results. For impact analysis, this loss of precision has the potential to cause an unacceptable level of false positives, therefore we require a higher level of context-sensitivity than is usually required by related analyses.

The second requirement is the requirement for traceability. Our requirements for an automated impact analysis, defined in Chapter 2, state that we require the ability to associate a predicted query with the locations where the query is executed, where the query is defined and where the results of the query are used. This information is potentially important and beneficial to the stakeholders. Therefore, our query analysis must preserve this information where possible.

The third requirement is that the query analysis should be able to analyse queries written in a variety of data-access technologies. Queries may be composed dynamically and statically using strings, but may also involve many other data-access types to be included in the analysis. For example, queries, and the result sets resulting from query executions, can be represented by many different types of object. There is no single standard data access API, so it is important that our query analysis be applicable to a wide range of potential data-access approaches.

The closest related work to our desired query analysis was the string analysis of Choi et al. [2006]. This analysis is context-sensitive to a level of 1-CFA, but our requirements mandate a variable level of context-sensitivity. Also, this analysis did not satisfy our requirements for traceability, and is focused on string data types. We extend this analysis to satisfy our impact analysis requirements, creating the query analysis described in Chapter 3. Increasing the level of context-sensitivity and adding further data-types to the analysis are only minor contributions. The significant contribution we make over related work is the identification of the requirements of query analysis for impact analysis, and the extensions for traceability.

8.1.2 Impact Calculation

Impact calculation takes the results of a query analysis, and uses them to predict the impacts of schema change. Impact calculation is a novel approach, and we described it in detail and showed that it can be implemented practically and efficiently.

The first stage of impact calculation is fact extraction, which processes the fixed point solution from a query analysis, to produce a set of facts about the program, describing where queries are defined, where queries are executed and where the results of queries are used. We described fact extraction formally, showing how fact extraction functions can be specified and applied to the fixed point solution of a query analysis.

The second stage of impact calculation is fact processing, which processes the extracted facts to predict impacts of schema change. We described how first order predicate calculus can be used to query the details of the extracted facts and the relationships between the facts. We showed how these queries can be used to compose impact calculation scripts that predict the impacts of schema changes.

We identified the requirement that impact calculation should be extensible, in that it must be able to deal with a variety of schema changes and query analysis results. The schema changes cannot be exhaustively defined because of the variety of DBMS vendors and different query languages and features

that they exhibit, meaning that our impact calculation must be extensible. The query analysis results will vary, dependent upon the data-access approach used in the application, and as these data-access approaches are numerous and varied, our impact calculation must also be extensible to be able to analyse the variety of query analysis results.

We showed that impact calculation can be practically and efficiently implemented using the CrocoPat tool. CrocoPat allows efficient querying of arbitrary graph based data, represented using the RSF file format. This RSF format is a convenient way of representing the facts extracted during the fact extraction stage. The fact processing stage can be implemented using CrocoPat by creating change scripts written in the RML language. RML is an expressive language based on first order predicate calculus, and can be used to arbitrarily query the data, and produce arbitrary textual output. We proposed libraries of change scripts for different DBMS features and data-access approaches. We provide a reference set of change scripts in Appendix B.

The contribution lies in describing a novel impact analysis approach, describing it formally, and describing an efficient and practical implementation, giving examples of its use.

8.1.3 Efficient Analysis

We describe a novel approach to improving the efficiency of a complex dataflow analysis by using data-dependence program slicing. The closest related work on improving the efficiency of dataflow analyses [Reps et al., 1995] is not applicable to the abstract domain of our query analysis. We investigated the use of program slicing to obtain a subset of the program which is involved in database queries, and hoped to gain a improvement in the efficiency of the impact analysis, by running the query analysis only over this program subset. Program slicing has been used to similar effect before in model checking [Schaefer and Poetzsch-Heffter, 2008, Dwyer et al., 2006], but we are unaware of it ever being applied to dataflow analysis. We also use an approach of data-dependence only slicing, to obtain the program subset, which is a novel approach. Data-dependence has been used in other work [Korel et al., 2005] but data-dependence slicing used for the purposes of dataflow analysis optimisation is novel. We used our case study application to show that this optimisation can provide significant reductions in overall execution times on large applications in practice.

8.1.4 Database Schema Change Impact Analysis

We have investigated the combination of novel analyses, such as query analysis and impact calculation, with established techniques, such as program slicing and garbage collection, to create an accurate, high impact-precision and computationally tractable analysis. We have combined these approaches to create a novel *database schema change impact analysis*.

8.1.5 Prototype Implementation

We implemented our database schema change impact analysis in a prototype tool called *SUITE*. At the time of the case study, SUITE consisted of around 32 KLOC. SUITE was written in C# using the Microsoft Phoenix Framework [Microsoft Phoenix]. This complex application was the result of a significant engineering effort, and therefore represents a significant contribution.

8.1.6 Evaluation

We used SUITE to perform a historical case study of a large commercial application. This case study investigated the feasibility and potential usefulness of our impact analysis, showing that the approach is both feasible, and potentially useful. We described the major performance optimisations used by SUITE, dead state removal and abstract garbage collection, in Chapter 5, so that the results we described can be replicated and explained.

The evaluation shows that our database schema change impact analysis is feasible for large real-world applications, and has the potential to be useful in real-world development projects.

8.2 Open Questions and Future Work

Our results so far have shown that our impact analysis is feasible and potentially useful. In this section we shall summarise the open questions and directions for future work.

8.2.1 Usefulness

Because the concept of usefulness is so broad, and has many dimensions, it was not feasible to provide an evaluation of usefulness within the scope of our research. Our results, as discussed in Section 7.4, indicated that our analysis is likely to be useful in practice. Beyond this, any further evaluation of usefulness would have to be very broad. We would need to evaluate the use of impact analysis information during several real projects. These projects would need to vary in size, some being very large, and they would have to use different architectures, approaches and a variety of data-access practices. This investigation would be a substantial undertaking.

This future evaluation could also be useful in answering other questions. The work of Sjoberg [1993] investigates the types of database schema changes that occur in practice. One interesting question arising from this research is; why are additions and deletions more common than renamings? We could speculate that this is because the impacts of renamings are more difficult to deal with. An interesting future research direction could be to compare database development with and without impact analysis, and see if the presence of impact analysis tools affects the types of changes that are made to the schema. Will the schema be changed in more complex ways, or more often with impact analysis tools, i.e. do developers currently avoid changing the schema because it is too difficult to do? We identify this as a possible direction for future research, that could be addressed whilst evaluating the usefulness of our impact analysis.

8.2.2 Efficiency and Scalability

Our initial evaluation of scalability shows that, for a typical enterprise application of up to 95 KLOC, our impact analysis can execute in under 25 minutes on a standard desktop machine. To show how our approach scales beyond this requires extensive further evaluation, which is beyond the scope of this dissertation.

Just as with the further evaluation of usefulness, we would need to conduct more empirical investigations by using our tool on real industrial applications. To do this, we would need to examine

applications larger than our current case-study and applications which have different amounts of queries and varied data-access practices.

The size of data-dependence only slices, in different applications, needs to be replicated and verified. Creating a slicing algorithm for a modern programming language is a significant engineering task and, as such, it is not unreasonable to assume that our implementation contains errors or inaccuracies. Using other slicing tools, such as Indus [Ranganath et al., 2007] and CodeSurfer [Anderson and Zarins, 2005] and using them to do data-dependence only slicing would allow us to gain a more reliable estimate of the costs of slicing and the size of data-dependence only slices in practice. As discussed, this is one of the main threats to external validity for parts of this research, and is a high-priority direction for future work.

There is a future work opportunity for decreasing the size of the slices we use, by investigating chopping [Jackson and Rollins, 1994] where the definition of query variables, or the use of query results are used to stop the forward slice proceeding, thus including unnecessary code. This could potentially decrease the size of our program subset further, without dramatically increasing the cost of calculating the program slices.

As we have discussed, the cost of the analysis is dominated by the partial ordering of regular strings. This is also a problem for model checking, and related work from the model checking community could provide increases in performance by using different algorithms and data-structures for calculating partial ordering, such as pushdown automata [Hopcroft et al., 2006].

8.2.3 Conservative Analysis

To produce a conservative analysis for a full modern program language is a complex task, especially when the analysis is complex, like our query analysis. The goal of this research was to investigate if a useful impact analysis was feasible, and therefore we relaxed the constraint of requiring conservative analysis so that the trade-offs between accuracy, impact-precision and efficiency could be explored more freely. Now we have shown that impact analysis is feasible, we can add more constraints in future work, such as investigating if a conservative impact analysis is feasible.

Creating a conservative analysis will require creating a conservative slicing algorithm, conservative query analysis and conservative impact calculation. This is a significant amount of work, and there is no guarantee that this will lead to useful levels of accuracy, impact-precision and efficiency. The accuracy could be worse, with conservative analysis potentially producing many more false positives. The analysis may have low impact-precision, making conservative assumptions even though more detailed non-conservative information might be available. A conservative analysis might also be much less efficient than our current analysis. It is, therefore, quite possible that after the effort of producing a conservative analysis, it will not be any more useful in practice than the analysis we have described in this dissertation. A conservative analysis would, however, have the guarantee that no false negatives would occur, which could be important for schema change in some industrial situations.

8.2.4 Application to Other Areas

If we regard database queries simply as interactions with an API, the extraction of queries from applications bears similarity to many other similar uses of APIs. For example, web services are defined, called, and then the results of the call are processed. XML documents can also be processed in a similar way. It is likely that our analysis could be used to predict the impact of changes to web services, or changes to the schema of XML documents in the same way as we currently predict the effects of database schema change.

The approach we have defined of using data-dependence slicing to improve the overall efficiency of our analysis could also be applied to other expensive program analyses. Similar approaches have been tried [Schaefer and Poetzsch-Heffter, 2008, Dwyer et al., 2006] in model checking, with similar results, and our data-dependence variation can be seen as another option for optimising complex analyses.

Appendix A

Query Analysis

A.1 Predicates

We define the following impact calculation predicates, as discussed in Chapter 4.

ConcAtLine, an occurence of a string concatenation occured at the source code location specified in the term.

Concats, an occurence of a string concatenation includes the regular string identifier specified in the term.

CreatesStr, an occurence of a string concatenation results in a new regular string with the value specified in the term.

CreatesStrWithId, an occurence of a string concatenation results in a new regular string with the identifier specified in the term.

ExecutesQuery, an occurence of a query execution can execute the query value specified in the term.

ExecutesQueryWithId, an occurence of a query execution can execute the query with the regular string identifier specified in the term.

Executes, an occurence of a query execution can execute the query represented by the heap object with the location identifier specified in the term.

ExecutedAtLine, an occurence of a query execution occurs at the source code location specified in the term.

LdStr, an occurence of a literal string definition occurs, defining the regular string identifier specified in the term.

LdStrAtLine, an occurence of a literal string definition occurs at the source code location specified in the term.

PrmAddName, an occurence of an addition of a parameter to a query adds the parameter with the name specified in the term.

PrmAddNameWithId, an occurence of an addition of a parameter to a query adds the parameter with the identifier specified in the term.

PrmAddType, an occurence of an addition of a parameter to a query adds the parameter with the data type specified in the term.

PrmAddLine, an occurence of an addition of a parameter to a query occurs at the source code location specified in the term.

PrmAddTo, an occurence of an addition of a parameter to a query adds a paremeter to an object with the location identifier specified by the term.

ReadsColumn, an occurence of a reading of a result reads the column with the name value specified by the term.

ReadsColumnWithId, an occurence of a reading of a result reads the column with the name identifier specified by the term.

ReadAtLine, an occurence of a reading of a result occurs at the source code location specified in the term.

ReadsResultObject, an occurence of a reading of a result reads the from the results object with the location identifier specified by the term.

ReturnsResult, an occurence of a query execution creates a result set object with the location identifier specified by the term.

UsesParams, an occurence of a query execution uses a set of parameters represented by the object with the location identifier specified by the term.

A.2 Additional Semantics

A.2.1 AssignFromSecondParam

Description

A method/property of a query representing object that assigns a the value of the parameter to the query.

Transfer Function

$$\mathcal{T}[\![x.\text{setQueryText}(y)]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad \mathcal{T}[\![[x] := y]\!](\sigma, h, h^{\text{reg}}, O)$$

Fact Extraction Function

No effects produced.

A.2.2 AssignReceiver

Description

A method called on an object that represents the result set of a query; the method retrieves another result from the result set and assigns it to the destination variable.

Transfer Function

$$\mathcal{T}[\![x = y.\text{GetNextRecord}()]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad \mathcal{T}[\![x := y]\!](\sigma, h, h^{\text{reg}}, O)$$

Fact Extraction Function

No effects produced.

A.2.3 ExecReceiver

Description

A query is executed. The receiver object represents the query value, and the destination variable will be assigned a new object representing the result set.

Transfer Function

$$\mathcal{T}[\![x = y.\text{Exec}(), l]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad \mathcal{T}[\![x := new^{l}]\!](\sigma, h, h^{\text{reg}}, O)$$

Fact Extraction Function

$$\begin{aligned}
&Facts[\![x = y.\text{ExecuteReader}()]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \bigcup\{(\text{ExecutesQuery}, p) \mid p \in \text{lookup}(V', h^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{ExecutesQueryWithId}, o) \mid l^{\text{reg}} \in V', o \in O(l^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{Executes}, l) \mid l \in \sigma(y)\}+ \\
&\quad \bigcup\{(\text{ReturnsResult}, l) \mid l \in \sigma(x)\}+ \\
&\quad \{(ExecuteAtLine, \mathsf{sourceLoc})\} \\
&\text{where } V' = \bigcup\{V \mid l \in \sigma(y), h(l) = V^{u}\}
\end{aligned}$$

A.2.4 ExecReceiverNonQuery

Description

A query is executed. The receiver object represents the query value but no result object is returned.

Transfer Function

$$\mathcal{T}[\![x = y.\text{GetNextRecord}()]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad \mathcal{T}[\![\text{skip}]\!](\sigma, h, h^{\text{reg}}, O)$$

Fact Extraction Function

$$\begin{aligned}
&Facts[\![x = y.\text{ExecuteReader}()]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \bigcup\{(\text{ExecutesQuery}, p) \mid p \in \text{lookup}(V', h^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{ExecutesQueryWithId}, o) \mid l^{\text{reg}} \in V', o \in O(l^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{Executes}, l) \mid l \in \sigma(y)\}+ \\
&\quad \{(ExecuteAtLine, \mathsf{sourceLoc})\} \\
&\text{where } V' = \bigcup\{V \mid l \in \sigma(y), h(l) = V^{u}\}
\end{aligned}$$

A.2.5 LdStr

Description

A literal string is defined and assigned to a variable.

Transfer Function

Defined in standard semantics.

Fact Extraction Function

$$\begin{aligned}
&Facts[\![x = s]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \bigcup\{(\text{LdStr}, l^{\text{reg}}) \mid l^{\text{reg}} \in \sigma(x)\}+ \\
&\quad \{(LdStrAtLine, \mathsf{sourceLoc})\}
\end{aligned}$$

A.2.6 ReceiverToString

Description

An object that represents a query value, is converted to a string and assigned to a variable.

Transfer Function

$$\mathcal{T}[\![x = y.\text{ToString}()]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad \mathcal{T}[\![x := [y]]\!](\sigma, h, h^{\text{reg}}, O)$$

Fact Extraction Function

No effects produced.

A.2.7 ExecReceiverNonQuery

Description

A query is executed. The receiver object represents the query value but no result object is returned.

Transfer Function

Defined in standard semantics.

Fact Extraction Function

$$\begin{aligned}
&Facts[\![x = y + z]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \bigcup\{(\text{Concats}, l^{\text{reg}}) \mid l^{\text{reg}} \in \{\sigma(y) + \sigma(z)\}\} \\
&\quad \bigcup\{(\text{CreatesStr}, p) \mid p \in \text{lookup}(V', h^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{CreatesStrWithId}, o) \mid l^{\text{reg}} \in V', o \in O(l^{\text{reg}})\}+ \\
&\quad \{(ConcAtLine, \mathsf{sourceLoc})\} \\
&\quad \text{where } V' = \bigcup\{V \mid l \in \sigma(x), h(l) = V^u\}
\end{aligned}$$

A.2.8 StringBuilderAppend

Description

An object that represents a query value, has a string appended to the query value.

Transfer Function

$$\begin{aligned}
\mathcal{T}[\![x.\text{Append}(y)]\!](\sigma, h, h^{\text{reg}}, O) \quad = \quad & \mathcal{T}[\![[x] := a]\!](t_2) \\
& \text{where } t_2 = \mathcal{T}[\![a := z + y]\!](t_1) \\
& \text{where } t_1 = \mathcal{T}[\![z := [x]]\!](\sigma, h, h^{\text{reg}}, O)
\end{aligned}$$

Fact Extraction Function

NOTE: $x^{original}$ signifies a variabe that contains the values of the the x variable before the transfer function took place. This is required because the original values are overwritten, but are required for the effects.

$$\begin{aligned}
&Facts[\![x.\text{Append}(y), x^{\text{original}}]\!](\sigma \times h \times h^{\text{reg}}, O, \mathsf{sourceLoc}) \quad = \\
&\quad \bigcup\{(\text{Concats}, l^{\text{reg}}) \mid l^{\text{reg}} \in \{\sigma(y) + V''\}\} \\
&\quad \bigcup\{(\text{CreatesStr}, p) \mid p \in \text{lookup}(V', h^{\text{reg}})\}+ \\
&\quad \bigcup\{(\text{CreatesStrWithId}, o) \mid l^{\text{reg}} \in V', o \in O(l^{\text{reg}})\}+ \\
&\quad \{(ConcAtLine, \mathsf{sourceLoc})\} \\
&\quad \text{where } V' = \bigcup\{V \mid l \in \sigma(x), h(l) = V^u\} \\
&\quad \text{where } V' = \bigcup\{V \mid l \in \sigma(x^{\text{original}}), h(l) = V^u\}
\end{aligned}$$

Appendix B

Impact Calculation

B.1 Impact Calculation Scripts

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Add new optional parameters to a stored procedure
5  //
6  // $1
7  // arg1Name = sproc
8  // arg1Desc = The name of the stored procedure
9  //
10
11 ImpactType      := "warn";
12 ImpactMessage   := "This query now has a new optional parameter. Check to
       see if it should be used here.";
13
14 sprocQueries(x)     := @ Seperator + "$1" + Seperator (x);
15
16 AffectedQueries(x)      :=      EX(y, sprocQueries(y) & ExecutesQuery(x, y))
       ;
17
18 // INCLUDE:..\output.inc
```

Listing B.1: csAddNewOptionalParamsToStoredProc.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Add new required parameters to a stored procedure
5  //
6  // $1
7  // arg1Name = sproc
```

```
8  // arg1Desc = The name of the stored procedure
9  //
10
11 ImpactType      := "error";
12 ImpactMessage   := "This query now has a new required parameter. Check to
       see that it is supplied, otherwise a runtime error will occur.";
13
14 sprocQueries(x)     := @ Seperator + "$1" + Seperator (x);
15
16 AffectedQueries(x)      :=      EX(y, sprocQueries(y) & ExecutesQuery(x, y))
       ;
17
18 // INCLUDE:..\output.inc
```

Listing B.2: csAddNewRequiredParamsToStoredProc.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Optional columns have been added to a table.
5  //
6  // $1
7  // arg1Name = table
8  // arg1Desc =  The name of the table
9  //
10 // $2
11 // arg2Name = column
12 // arg2Desc = The name of the column
13 //
14
15 ImpactType      := "warn";
16 ImpactMessage   := "These queries reference a changed table that now has an
       new unrequired columns. Check to see whether these new data should be
       supplied or returned here.";
17
18 tableQueries(x)      :=      @ Seperator + "$1" + Seperator (x);
19
20 AffectedQueries(x)      :=      EX(y, tableQueries(y) & ExecutesQuery(x, y))
       ;
21
22 // INCLUDE:..\output.inc
```

Listing B.3: csAddOptionalColumns.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Add a new required column to a table
5  //
6  // $1
7  // arg1Name = table
8  // arg1Desc = The name of the table
9  //
10 // $2
11 // arg2Name = column
12 // arg2Desc = The name of the column
13 //
14
15 ImpactType      := "error";
16 ImpactMessage   := "These queries modify a table that now has a new required
       column. Check to see that the value is supplied here.";
17
18 tableQueries(x)      :=      @Seperator + "$1" + Seperator(x);
19
20 // We only look for modifying queries, as selects will never break the
       constraints
21 modifyingQueries(x)            :=      @Seperator + "UPDATE" + Seperator(
       x) |
22                                        @Seperator + "INSERT" + Seperator(
                                              x);
23
24 AffectedQueries(x)      :=      EX(y, tableQueries(y) & ExecutesQuery(x, y))
       &
25                                 EX(y, modifyingQueries(y) & ExecutesQuery(x,
                                        y));
26
27 // INCLUDE:..\output.inc
```

Listing B.4: csAddRequiredColumns.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Drop a required column
5  //
6  //
7  // $1
```

```
8  // arg1Name = table
9  // arg1Desc = The name of the table
10 //
11 // $2
12 // arg2Name = column
13 // arg2Desc = The name of the column within @table
14 //
15
16 ImpactType      := "error";
17 ImpactMessage   := "These queries reference a column which has been dropped
       .";
18
19 columnQueries(x)     :=      @Seperator + "$2" + Seperator(x);
20 tableQueries(x)      :=      @Seperator + "$1" + Seperator(x);
21
22 AffectedQueries(x)      :=      EX(y, columnQueries(y) & ExecutesQuery(x, y
       )) &
23                               EX(y, tableQueries(y) & ExecutesQuery(x, y)
                                     );
24
25 // INCLUDE:..\output.inc
```

Listing B.5: csDropColumnRequired.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = Drop a previously required parameters to a stored procedure
5  //
6  // $1
7  // arg1Name = sproc
8  // arg1Desc = The name of the stored procedure
9  //
10
11 ImpactType      := "error";
12 ImpactMessage   := "This query now has dropped a required parameter. Check
       to see that it is removed, otherwise a runtime error will occur.";
13
14 sprocQueries(x)     := @Seperator + "$1" + Seperator(x);
15
16 AffectedQueries(x)      :=      EX(y, sprocQueries(y) & ExecutesQuery(x, y))
       ;
17
```

```
18 // INCLUDE:..\output.inc
```

Listing B.6: csDropRequiredParamsToStoredProc.rml

```
1  // INCLUDE:..\header.inc
2
3  //
4  // description = The stored procedure return more data than it did before
5  //
6  // $1
7  // arg1Name = sproc
8  // arg1Desc = The name of the stored procedure
9  //
10
11 ImpactType      := "warn";
12 ImpactMessage   := "This query now returning additional data. Check to see
       if these data should be used here.";
13
14 sprocQueries(x)     := @  Seperator + "$1" + Seperator (x);
15
16 AffectedQueries(x)      :=      EX(y, sprocQueries(y) & ExecutesQuery(x, y))
       ;
17
18 // INCLUDE:..\output.inc
```

Listing B.7: csReturnNewDataFromStoredProc.rml

```
1  // Seperator is series of non alphanumeric characters. i.e. anything that
       can delimit a whole word.
2  Seperator               :=      "($|^|[^a-zA-Z0-9]+)";
```

Listing B.8: header.inc

```
1  //
2  // <------- STANDARD OUTPUT SCRIPT ------->
3  //
4  // Displays the XML required to be parsed by the SUITE GUI program.
5  //
6  // Params:
7  // =======
8  // AffectedQueries = affected query IDs
9  // ImpactType = "error" | "warn"
10 // ImpactMessage = string
11 //
12
```

```
13 FOR q IN AffectedQueries(x) {
14
15     PRINT "<impact type='";
16     PRINT ImpactType;
17     PRINT "' message='";
18     PRINT ImpactMessage;
19     PRINT "'>";
20
21     //
22     // Queries
23     //
24     Queries(x) := ExecutesQuery(q, x);
25
26     FOR query IN Queries(x) {
27        PRINT "<query>";
28        PRINT query;
29        PRINT "</query>";
30     }
31
32     //
33     // Executions
34     //
35
36     ExecutionLocations(x) :=  ExecutedAtLine(q, x);
37
38     FOR exec IN ExecutionLocations(x) {
39        PRINT "<queryexec location='";
40        PRINT exec;
41        PRINT "' />";
42     }
43
44     //
45     // Definitions
46     //
47
48     AffectedDefinitions(x) := ExecutesQueryWithId(q, x);
49
50     // Find fixed point for all concats, and widenings
51
52     Result(x) := AffectedDefinitions(x);
53     PrevResult(x) := FALSE(x);
54     WHILE (PrevResult(x) != Result(x)) {
```

```
55          PrevResult(x) := Result(x);
56          AffectedStrConcats(y) := EX(x, CreatesStr(y, x) & Result(x));
57          Result(x) := Result(x) | EX(y, Concats(y, x) & AffectedStrConcats(y
                ));
58      }
59
60      AffectedLdStrs(x) := EX(y, Result(y) & LdStr(x, y));
61      AffectedConcats(x) := EX(y, Result(y) & Concats(x, y));
62
63      LdstrLocs(x) := EX(y, AffectedLdStrs(y) & LdStrAtLine(y, x));
64      ConcLocs(x) := EX(y, AffectedConcats(y) & ConcAtLine(y, x));
65
66      DefinitionLocations(x) := EX(y, AffectedLdStrs(y) & LdStrAtLine(y, x))
            |
67                                EX(y, AffectedConcats(y) & ConcAtLine(y, x));
68
69      FOR def IN DefinitionLocations(x) {
70          PRINT "<querydef location='";
71          PRINT def;
72          PRINT "' />";
73      }
74
75      //
76      // Results uses
77      //
78
79      AffectedResultLocs(x) := ReturnsResult(q, x);
80      AffectedReads(x) := EX(y, AffectedResultLocs(y) & ReadsResultObject(x,
            y));
81      ResultLocations(x) := EX(y, AffectedReads(y) & ReadAtLine(y, x));
82
83      FOR use IN ResultLocations(x) {
84                  PRINT "<queryuse location='";
85                  PRINT use;
86                  PRINT "' />";
87      }
88
89      PRINT "</impact>";
90  }
91  //
92  // <-------------------------------------->
```

```
93 //
```

Listing B.9: output.inc

Appendix C

Case Study

C.1 Results

TP = True positives, FP = False Positives

C.1.1 Changes to Schema Version 91

Change	Predicted	TP	FP	Change Script
ChangeSc1	2 warns	0	2	csChangeConstraint(”t_ContentItemRelation”)
ChangeSc2	1 warn	0	1	csChangeConstraint(”t_StructureRelation”)
ChangeSp1	1 warn	0	1	csChangedDataFromStoredProc(”cp_ContentItemRelations_sel”)
ChangeSp2	1 warn	0	1	csChangedDataFromStoredProc(”cp_ContentItemRelationsReverse_sel”)
ChangeSp3	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelations_sel”)
ChangeSp4	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelationsContent_sel”)
ChangeSp5	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelationsReverse_sel”)

C.1.2 Changes to Schema Version 234

Change	Predicted	TP	FP	Change Script
ChangeSc1	5 warns	2	3	csAddOptionalColumns(”t_ContentItem”)
ChangeSc2	5 warns	0	5	csAddOptionalColumns(”t_ContentItemRating”)
ChangeSc3	none	0	0	csMakeColumnNullable(”t_ContentItemRating”, ”UserId”);
ChangeSc4	5 warns	0	5	csAddTableConstraint(”t_ContentItemRating”)
ChangeSp1	1 err	1	0	csAddNewOptionalParamsToStoredProc(”cp_ContentItemRating_ins”)
ChangeSp2	1 warns	1	0	csReturnNewDataFromStoredProc(”cp_ContentItemRatings_sel”)
ChangeSp3	1 warn	0	1	csReturnNewDataFromStoredProc(”ap_ContentItemRating_sel”)
ChangeSp4	none	0	0	csMakeStoredProcParamOptional(”cp_ContentItemRating_ins”, ”@UserId”)

C.1.3 Changes to Schema Version 2118

Change	Predicted	TP	FP	Change Script
ChangeSc1	none	0	0	csAddTable(”t_RatingType”)
ChangeSc2	9 warns	0	9	csAddOptionalColumns(”t_ContentItemRating”);
ChangeSc3	9 warns	0	9	csAddForeignKeyConstraint(”RatingTypeId”, ”t_ContentItemRating”, ”RatingTypeId”, ”t_RatingType”)

C.1.4 Changes to Schema Version 2270

Change	Predicted	TP	FP	Change Script
ChangeSc1	none	0	0	csAddIdentityConstraint(”t_RatingType”, ”RatingTypeId”)

C.1.5 Changes to Schema Version 4017

Change	Predicted	TP	FP	Change Script
ChangeSc1	none	0	0	csAddTable(”t_StructureScoreAudit”)
ChangeSc2	5 warns	2	3	csAddOptionalColumns(”t_Structure”)
ChangeSc3	none	0	0	csAddForeignKeyConstraint(
				”StructureId”, ”t_StructureScoreAudit”, ”StructureId”, ”t_Structure”)
ChangeSc4	1 warn	1	0	csAddForeignKeyConstraint(
				”UserId”, ”t_StructureScoreAudit”, ”UserId”, ”t_User”)
ChangeSp1	none	0	0	csReturnNewDataFromStoredProc(”kp_StructureSiblings_sel”)
ChangeSp2	none	0	0	csReturnNewDataFromStoredProc(”kp_StructuresAtRoot_sel”)
ChangeSp3	none	0	0	csReturnNewDataFromStoredProc(”kp_StructurePathToRoot_sel”)
ChangeSp4	none	0	0	csReturnNewDataFromStoredProc(
				”kp_StructureFromVanityPath_sel”)
ChangeSp5	none	0	0	csReturnNewDataFromStoredProc(
				”kp_StructureChildrenWithImageCount_sel”)
ChangeSp6	none	0	0	csReturnNewDataFromStoredProc(”kp_StructureChildren_sel”)
ChangeSp7	none	0	0	csReturnNewDataFromStoredProc(”kp_Structure_sel”)
ChangeSp8	1 warn	0	1	csReturnNewDataFromStoredProc(”cp_StructureWithProperty_sel”)
ChangeSp9	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_StructureSiblingsByStartString_sel”)
ChangeSp10	2 warn	0	1	csReturnNewDataFromStoredProc(”cp_StructureSiblings_sel”)
ChangeSp11	1 warn	0	1	csReturnNewDataFromStoredProc(”cp_StructuresByType_sel”)
ChangeSp12	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelationsReverse_sel”)
ChangeSp13	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelationsContent_sel”)
ChangeSp14	1 warn	0	1	csChangedDataFromStoredProc(”cp_StructureRelations_sel”)
ChangeSp15	1 warn	0	1	csReturnNewDataFromStoredProc(”cp_StructurePathToRoot_sel”)
ChangeSp16	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_StructureElementFromVanityPath_sel”)
ChangeSp17	1 warn	0	1	csReturnNewDataFromStoredProc(”cp_Structure_sel”)
ChangeSp18	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_ListStructuresWithLinksAsReal_sel”)
ChangeSp19	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_ListStructuresForNavigation_sel”)
ChangeSp20	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_ListStructuresByTypeWithLinksAsReal_sel”)
ChangeSp21	1 warn	0	1	csReturnNewDataFromStoredProc(”cp_ListStructuresByType_sel”)
ChangeSp22	1 warn	0	1	csReturnNewDataFromStoredProc(
				”cp_ListStructuresByStartString_sel”)
ChangeSp23	2 warn	0	2	csReturnNewDataFromStoredProc(”cp_ListStructures_sel”)
ChangeSp24	1 warn	1	0	csDropStoredProc(”cp_ContentItemScore_upd”)
ChangeSp25	1 warn	1	0	csDropStoredProc(”cp_ContentItemHasScored_upd”)
ChangeSp26	1 warn	0	1	csAddStoredProc(”cp_ContentObjectScore_upd”)
ChangeSp27	none	0	0	csAddStoredProc(”cp_ContentObjectHasScored_upd”)

C.1.6 Changes to Schema Version 4132

Change	Predicted	TP	FP	Change Script
ChangeSc1	none	0	0	csAddTable(”t_UserCreditHistory”)
ChangeSc2	1 warn	1	0	csAddForeignKeyConstraint(”UserId”, ”t_UserCreditHistory”, ”UserId”, ”t_User”)
ChangeSc3	none	0	0	csAddForeignKeyConstraint(”ChangedById”, ”t_UserCreditHistory”, ”UserId”, ”t_User”)
ChangeSc4	3 warns	0	3	csAddOptionalColumns(”t_User”)
ChangeSc5	none	0	0	csMakeColumnRequired(”t_User”, ”AccountCredit”)
ChangeSc6	none	0	0	csAddOptionalColumns(”t_UserCreditHistory”)

C.1.7 Changes to Schema Version 4654

Change	Predicted	TP	FP	Change Script
ChangeSp1	none	0	0	csNoInOutInterfaceChangeStoredProc(”ap_ContentItemRelation_ins”)
ChangeSp2	none	0	0	csAddStoredProc(”kp_ContentElementRelations_sel”)
ChangeSp3	none	0	0	csAddStoredProc(”kp_StructureRelations_sel”)

C.2 Timings

The timings in the following tables consist of the timed execution of the case study version, averaged over three executions. All tests were run on a virtual machine with a single 32bit, 2.5Ghz CPU and 3GB of RAM, running Microsoft Windows XP. All times displayed with the format hh:mm:ss.

C.2.1 91

k	Data-Dep. Slicing	Whole Program
1	00:21:43	00:58:42
2	00:19:10	01:05:09
3	00:20:33	01:06:22
4	00:20:46	01:07:16
5	00:20:47	01:08:46
6	00:21:07	01:10:23
7	00:21:15	01:10:48
8	00:20:49	01:09:17

C.2.2 234

k	Data-Dep. Slicing	Whole Program
1	00:21:52	00:59:56
2	00:19:38	01:05:21
3	00:20:22	01:06:58
4	00:20:55	01:07:41
5	00:20:59	01:08:34
6	00:20:58	01:12:25
7	00:21:27	01:13:02
8	00:21:48	01:30:02

C.2.3 2118

k	Data-Dep. Slicing	Whole Program
1	00:22:22	01:03:19
2	00:19:53	01:28:54
3	00:20:37	01:30:23
4	00:20:32	01:29:09
5	00:20:31	01:30:38
6	00:20:42	01:31:49
7	00:20:53	01:46:38
8	00:21:06	*Out of memory*

C.2.4 2270

k	Data-Dep. Slicing	Whole Program
1	00:21:55	01:03:03
2	00:19:26	01:27:49
3	00:20:30	01:31:24
4	00:20:38	01:31:56
5	00:21:04	01:32:44
6	00:22:15	01:31:30
7	00:20:44	01:31:33
8	00:20:48	01:31:50

C.2.5 4017

k	Data-Dep. Slicing	Whole Program
1	00:21:49	01:00:19
2	00:20:14	01:09:15
3	00:21:15	01:11:41
4	00:21:57	01:12:50
5	00:22:21	*Out of memory*
6	00:24:20	*Out of memory*
7	00:23:54	*Out of memory*
8	00:23:55	*Out of memory*

C.2.6 4132

k	Data-Dep. Slicing	Whole Program
1	00:21:51	01:00:45
2	00:20:16	01:09:20
3	00:21:41	01:12:25
4	00:22:33	01:13:31
5	00:22:54	*Out of memory*
6	00:23:15	*Out of memory*
7	00:23:28	*Out of memory*
8	00:24:28	*Out of memory*

C.2.7 4654

k	Data-Dep. Slicing	Whole Program
1	00:22:49	01:03:00
2	00:21:20	01:11:43
3	00:21:48	01:12:04
4	00:22:06	01:13:44
5	00:22:44	01:16:28
6	00:23:05	*Out of memory*
7	00:23:21	*Out of memory*
8	00:23:28	*Out of memory*

Appendix D

UCL Coffee Applications

We include the source code for the UCL Coffee Inventory and Sales applications. The unit test projects have been omitted for the sake of brevity.

D.1 Inventory Application

```
1   using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Data;
6
7  namespace Inventory
8  {
9      public class Inventory
10     {
11         public static void Main(string[] args)
12         {
13             // INV–Q1
14             var product = Product.Find(1);
15
16             // INV–Q2
17             var supplier = Supplier.Find(1);
18
19             var supplierId = 0;
20             var productName = "newProductName";
21             // INV–Q3
22             var insertedProductId = Product.Insert(supplierId, productName);
23
24             var supplierKeywords = new string[] { };
25             var contactKeywords = new string[] { };
26             var productKeywords = new string[] { };
27             // INV–Q4
28             var products = Product.Find(supplierKeywords, contactKeywords,
                   productKeywords);
29
30             var supplierName = "";
```

```
            var contactName = "";
            // INV–Q5
            var insertedSupplierId = Supplier.Insert(supplierName, contactName);

        }
    }
}
```

Listing D.1: Inventory.cs

```
 using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.SqlClient;

namespace Inventory
{
    partial class Product
    {
        const string PRM_ID = "@ID";
        const string PRM_NAME = "@NAME";
        const string PRM_SUPPLIER_ID = "@SUPPLIER_ID";

        const string getIdentityStatement = "SELECT @@IDENTITY";

        // INV–Q1
        const string findStatement = "SELECT * FROM Product WHERE id=" + PRM_ID + ";";

        // INV–Q3
        const string insertStatement = "INSERT INTO Product (supplier_id, name)" +
               " VALUES (" + PRM_SUPPLIER_ID + ", " + PRM_NAME + ");"
               + getIdentityStatement;

        public static Product Load(SqlDataReader reader)
        {
            int id = reader.GetInt32(reader.GetOrdinal("id"));

            Product result = Registry.GetProduct(id);

            if (result != null)
                return result;

            string name = reader.GetString(reader.GetOrdinal("name"));
            int supplierId = reader.GetInt32(reader.GetOrdinal("supplier_id"));
            int unitsInStock = reader.GetInt32(reader.GetOrdinal("units_in_stock"));
            result = new Product(id, supplierId, unitsInStock, name);
            Registry.AddProduct(id, result);
            return result;
```

```
40          }
41
42          public static Product Find(int id)
43          {
44              Product result = Registry.GetProduct(id);
45
46              if (result != null)
47                  return result;
48
49              using(DB db = new DB())
50              {
51                  SqlCommand command = db.Prepare(findStatement);
52                  command.Parameters.Add(new SqlParameter(PRM_ID, id));
53                  SqlDataReader reader = command.ExecuteReader(); // INV-Q1 executed
54                  reader.Read();
55                  result = Load(reader);
56                  return result;
57              }
58          }
59
60          public static int Insert(int supplierId, string name)
61          {
62              using(DB db = new DB())
63              {
64                  SqlCommand command = db.Prepare(insertStatement);
65
66                  command.Parameters.Add(new SqlParameter(PRM_SUPPLIER_ID, supplierId));
67                  command.Parameters.Add(new SqlParameter(PRM_NAME, name));
68
69                  object result = command.ExecuteScalar(); // INV-Q3 executed
70                  return Decimal.ToInt32((decimal)result);
71              }
72          }
73
74          public static ICollection<Product> Find(string[] supplierKeywords, string[]
                contactKeywords, string[] productKeywords)
75          {
76              string sql = "SELECT * FROM Product";
77
78              if (supplierKeywords.Length > 0 || contactKeywords.Length > 0)
79                  sql += ", Supplier";
80
81              sql += " WHERE ";
82
83              using (DB db = new DB())
84              {
85                  SqlCommand cmd = db.Prepare();
86
```

```
87                  sql += "(";
88                  sql = AddKeyWordClause(cmd, supplierKeywords, sql, "Supplier.
                        company_name", "supplier");
89                  sql += " OR ";
90                  sql = AddKeyWordClause(cmd, contactKeywords, sql, "Supplier.
                        contact_name", "contact");
91                  sql += " OR ";
92                  sql = AddKeyWordClause(cmd, productKeywords, sql, "Product.name", "
                        product");
93                  sql += ") AND Product.supplier_id = Supplier.id;";
94                  cmd.CommandText = sql;
95
96                  System.Diagnostics.Debug.WriteLine("SQL: " + sql);
97
98                  SqlDataReader reader = cmd.ExecuteReader();
99
100                 List<Product> products = new List<Product>();
101                 while (reader.Read())
102                 {
103                     products.Add(Load(reader));
104                 }
105                 return products;
106             }
107         }
108
109         private static string AddKeyWordClause(SqlCommand cmd, string[] keywords,
                    string sql, string fieldName, string paramId)
110         {
111             for (int i = 1; i <= keywords.Length; i++)
112             {
113                 if (i == 1)
114                     sql += "("; // first loop
115
116                 string paramName = String.Concat("@", paramId.ToString(), "Key", i.
                        ToString());
117                 sql += String.Concat(fieldName, " LIKE ", paramName);
118
119                 if (i != keywords.Length)
120                     sql += " OR ";
121                 else
122                     sql += ")"; // last loop
123
124                 string valWithWildcards = String.Concat("%", keywords[i - 1], "%");
125                 cmd.Parameters.AddWithValue(paramName, valWithWildcards);
126             }
127             return sql;
128         }
129     }
```

```
130 }
```

Listing D.2: ActiveRecord/Product.cs

```
1   using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Data.SqlClient;
6
7  namespace Inventory
8  {
9      public partial class Supplier
10     {
11         const string PRM_ID = "@ID";
12         const string PRM_COMPANY_NAME = "@COMPANY_NAME";
13         const string PRM_CONTACT_NAME = "@CONTACT_NAME";
14
15         // INV-Q2
16         const string findStatement = "SELECT id, contact_name, company_name FROM
               Supplier WHERE id="+PRM_ID+";";
17
18         // INV-Q5
19         const string insertStatement = "INSERT INTO Supplier (company_name,
               contact_name) VALUES ("+PRM_COMPANY_NAME+","+PRM_CONTACT_NAME+"); SELECT
               @@IDENTITY;";
20
21         public static Supplier Load(SqlDataReader reader)
22         {
23             int id = reader.GetInt32(reader.GetOrdinal("id"));
24
25             Supplier result = Registry.GetSupplier(id);
26             if (result != null)
27                 return result;
28
29             string companyName = reader.GetString(reader.GetOrdinal("company_name"));
30             string contactName = reader.GetString(reader.GetOrdinal("contact_name"));
31
32             result = new Supplier(id, companyName, contactName);
33             Registry.AddSupplier(id, result);
34             return result;
35         }
36
37         public static Supplier Find(int id)
38         {
39             Supplier result = Registry.GetSupplier(id);
40
41             if (result != null)
42                 return result;
```

```
43
44              using(DB db = new DB())
45              {
46                  SqlCommand command = db.Prepare(findStatement);
47                  command.Parameters.Add(new SqlParameter(PRM_ID, id));
48                  SqlDataReader reader = command.ExecuteReader(); // INV-Q2 executed
49                  reader.Read();
50                  result = Load(reader);
51                  return result;
52              }
53          }
54
55          public static int Insert(string companyName, string contactName)
56          {
57              using (DB db = new DB())
58              {
59                  SqlCommand command = db.Prepare(insertStatement);
60
61                  command.Parameters.Add(new SqlParameter(PRM_COMPANY_NAME, companyName)
                        );
62                  command.Parameters.Add(new SqlParameter(PRM_CONTACT_NAME, contactName)
                        );
63
64                  object result = command.ExecuteScalar(); // INV-Q5 executed
65                  return Decimal.ToInt32((decimal)result);
66              }
67          }
68      }
69  }
```

Listing D.3: ActiveRecord/Supplier.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Data.SqlClient;
6   using System.Configuration;
7
8   namespace Inventory
9   {
10      class DB : IDisposable
11      {
12          SqlConnection conn;
13
14          public DB()
15          {
16              string connString = ConfigurationManager.ConnectionStrings[1].
                    ConnectionString;
```

```
17              conn = new SqlConnection(connString);
18              conn.Open();
19          }
20
21          public SqlCommand Prepare(string cmdText)
22          {
23              SqlCommand command = new SqlCommand(cmdText, conn);
24              return command;
25          }
26
27          internal SqlCommand Prepare()
28          {
29              SqlCommand command = new SqlCommand();
30              command.Connection = conn;
31              return command;
32          }
33
34          #region IDisposable Members
35
36          public void Dispose()
37          {
38              if(conn != null)
39                  conn.Dispose();
40          }
41
42          #endregion
43      }
44  }
```

Listing D.4: BaseHelpers/DB.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Inventory
7   {
8       class Registry
9       {
10          private static GenericRegistry<Product> productRegistry = new GenericRegistry<
                Product>();
11          private static GenericRegistry<Supplier> supplierRegistry = new
                GenericRegistry<Supplier>();
12
13          public static void AddProduct(int id, Product prod)
14          {
15              productRegistry.Add(id, prod);
16          }
```

```
17
18          public static Product GetProduct(int id)
19          {
20              return productRegistry.Get(id);
21          }
22
23          internal static void AddSupplier(int id, Supplier supplier)
24          {
25              supplierRegistry.Add(id, supplier);
26          }
27
28          internal static Supplier GetSupplier(int id)
29          {
30              return supplierRegistry.Get(id);
31          }
32      }
33
34      class GenericRegistry<T> where T : DomainObject
35      {
36          Dictionary<int, T> items = new Dictionary<int, T>();
37
38          public void Add(int id, T item)
39          {
40              items.Add(id, item);
41          }
42
43          public T Get(int id)
44          {
45              T result = null;
46              items.TryGetValue(id, out result);
47
48              return result;
49          }
50
51
52      }
53  }
```

Listing D.5: BaseHelpers/Registry.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Inventory
7   {
8       public class DomainObject
9       {
```

```
10      }
11  }
```

Listing D.6: DomainObjects/DomainObject.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Inventory
7   {
8       public partial class Product : DomainObject
9       {
10          public int Id { get; private set; }
11          public int SupplierId { get; private set; }
12          public int UnitsInStock { get; private set; }
13          public string Name { get; private set; }
14
15          public Product(int id, int supplierId, int unitsInStock, string name)
16          {
17              Id = id;
18              SupplierId = supplierId;
19              UnitsInStock = unitsInStock;
20              Name = name;
21          }
22      }
23  }
```

Listing D.7: DomainObjects/Product.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Inventory
7   {
8       public partial class Supplier : DomainObject
9       {
10          public int Id { get; private set; }
11          public string  CompanyName { get; private set; }
12          public string ContactName { get; private set; }
13
14          public Supplier(int id, string companyName, string contactName)
15          {
16              Id = id;
17              CompanyName = companyName;
18              ContactName = contactName;
19          }
```

```
20      }
21  }
```

Listing D.8: DomainObjects/Supplier.cs

D.2 Sales Application

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using Sales.Gateways;
6   using System.Data;
7
8   namespace Sales
9   {
10      public class Sales
11      {
12          static void Main(string[] args)
13          {
14              // call SALES-Q1
15              using (CustomerGateway customerGateway = new CustomerGateway())
16              {
17                  using (var reader = customerGateway.FindAll())
18                  {
19                      reader.Read();
20                      UseCustomerName(reader);
21                  }
22
23                  // call SALES-Q2
24                  var insertedId = customerGateway.Insert("new name");
25
26                  // call SALES-Q3
27                  customerGateway.Update(1, "updated name");
28
29                  // call SALES-Q4
30                  customerGateway.Delete(1);
31              }
32
33              using (OrderGateway orderGateway = new OrderGateway())
34              {
35                  // call SALES-Q5
36                  var newOrderId = orderGateway.Insert(1);
37
38                  // call SALES-Q6
39                  orderGateway.InsertOrderProduct(1, 2, 3);
40
41                  // call SALES-Q7
42                  using (var reader = orderGateway.SelectByCustomer(1))
```

```
43                 {
44                     reader.Read();
45                     UseCustomerId(reader);
46                 }
47             }
48         }
49 
50         private static void UseCustomerId(IDataReader reader)
51         {
52             reader.GetString(reader.GetOrdinal("customer_id"));
53         }
54 
55         private static void UseCustomerName(IDataReader reader)
56         {
57             reader.GetString(reader.GetOrdinal("name"));
58         }
59     }
60 }
```

Listing D.9: Sales.cs

```
1   using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Configuration;
6 using System.Data.OleDb;
7 
8 namespace Sales
9 {
10     class DB : IDisposable
11     {
12         OleDbConnection conn;
13 
14         public DB()
15         {
16             string connString = ConfigurationManager.ConnectionStrings[1].
                   ConnectionString;
17             conn = new OleDbConnection(connString);
18             conn.Open();
19         }
20 
21         public OleDbCommand Prepare(string cmdText)
22         {
23             return Prepare(cmdText, conn);
24         }
25 
26         public static OleDbCommand Prepare(string cmdText, OleDbConnection conn)
27         {
```

```
28              OleDbCommand command = new OleDbCommand(cmdText, conn);
29              return command;
30          }
31
32          #region IDisposable Members
33
34          public void Dispose()
35          {
36              if (conn != null)
37                  conn.Dispose();
38          }
39
40          #endregion
41      }
42  }
```

Listing D.10: BaseHelpers/DB.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Data;
6
7   namespace Sales.Gateways
8   {
9       public class CustomerGateway : IDisposable
10      {
11          private DB db;
12
13          public CustomerGateway()
14          {
15              this.db = new DB();
16          }
17
18          public IDataReader FindAll()
19          {
20              var cmd = db.Prepare("selCustomersAll");
21              return cmd.ExecuteReader(); // SALES-Q1 executed
22          }
23
24          public decimal Insert(string customerName)
25          {
26              var cmd = db.Prepare("insCustomer");
27              cmd.Parameters.Add(new System.Data.OleDb.OleDbParameter("@name",
                    customerName));
28              cmd.CommandType = CommandType.StoredProcedure;
29              return (decimal)cmd.ExecuteScalar(); // SALES-Q2 executed
30          }
```

```
31
32          public void Update(int id, string updatedCustomerName)
33          {
34              var cmd = db.Prepare("updCustomer");
35              cmd.Parameters.Add(new System.Data.OleDb.OleDbParameter("@id", id));
36              cmd.Parameters.Add(new System.Data.OleDb.OleDbParameter("
                    @updatedCustomerName", updatedCustomerName));
37              cmd.CommandType = CommandType.StoredProcedure;
38              cmd.ExecuteNonQuery(); // SALES-Q3 executed
39          }
40
41          public void Delete(long id)
42          {
43              var cmd = db.Prepare("delCustomer");
44              cmd.Parameters.Add(new System.Data.OleDb.OleDbParameter("@id", id));
45              cmd.CommandType = CommandType.StoredProcedure;
46              cmd.ExecuteNonQuery(); // SALES-Q4 executed
47          }
48
49          #region IDisposable Members
50
51          public void Dispose()
52          {
53              if (db != null)
54                  db.Dispose();
55          }
56
57          #endregion
58      }
59  }
```

Listing D.11: Gateways/CustomerGateway.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Data;
6
7   namespace Sales
8   {
9       public class OrderGateway : IDisposable
10      {
11          private DB db;
12
13          public OrderGateway()
14          {
15              this.db = new DB();
16          }
```

```
17
18          public IDataReader SelectByCustomer(int customerId)
19          {
20              var selOrderByCustomer = db.Prepare("selOrderByCustomer");
21              selOrderByCustomer.Parameters.Add(new System.Data.OleDb.OleDbParameter("
                    @customerId", customerId));
22              selOrderByCustomer.CommandType = CommandType.StoredProcedure;
23              return selOrderByCustomer.ExecuteReader(); // SALES-Q7 executed
24          }
25
26          public decimal Insert(int customerId)
27          {
28              var insOrder = db.Prepare("insOrder");
29              insOrder.Parameters.Add(new System.Data.OleDb.OleDbParameter("@customerId"
                    , customerId));
30              insOrder.CommandType = CommandType.StoredProcedure;
31              return (decimal)insOrder.ExecuteScalar(); // SALES-Q5 executed
32          }
33
34          public void InsertOrderProduct(int orderId, int productId, int units)
35          {
36              var insOrderProduct = db.Prepare("insOrderProduct");
37              insOrderProduct.Parameters.Add(new System.Data.OleDb.OleDbParameter("
                    @orderId", orderId));
38              insOrderProduct.Parameters.Add(new System.Data.OleDb.OleDbParameter("
                    @productId", productId));
39              insOrderProduct.Parameters.Add(new System.Data.OleDb.OleDbParameter("
                    @units", units));
40              insOrderProduct.CommandType = CommandType.StoredProcedure;
41              insOrderProduct.ExecuteNonQuery(); // SALES-Q6 executed
42          }
43
44          #region IDisposable Members
45
46          public void Dispose()
47          {
48              if (db != null)
49                  db.Dispose();
50          }
51
52          #endregion
53      }
54  }
```

Listing D.12: Gateways/OrderGateway.cs

D.3 Schema

```
1  SET ANSI_NULLS ON
```

```
2  GO
3  SET QUOTED_IDENTIFIER ON
4  GO
5  IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Customer
       ]') AND type in (N'U'))
6  BEGIN
7  CREATE TABLE [dbo].[Customer](
8          [id] [int] IDENTITY(1,1) NOT NULL,
9          [name] [nvarchar](50) NOT NULL,
10  CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
11 (
12         [id] ASC
13 )WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
14 ) ON [PRIMARY]
15 END
16 GO
17 SET ANSI_NULLS ON
18 GO
19 SET QUOTED_IDENTIFIER ON
20 GO
21 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Supplier
       ]') AND type in (N'U'))
22 BEGIN
23 CREATE TABLE [dbo].[Supplier](
24         [id] [int] IDENTITY(1,1) NOT NULL,
25         [contact_name] [nvarchar](50) NULL,
26         [company_name] [nvarchar](50) NOT NULL,
27  CONSTRAINT [PK_Supplier] PRIMARY KEY CLUSTERED
28 (
29         [id] ASC
30 )WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
31 ) ON [PRIMARY]
32 END
33 GO
34 SET ANSI_NULLS ON
35 GO
36 SET QUOTED_IDENTIFIER ON
37 GO
38 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Order]')
       AND type in (N'U'))
39 BEGIN
40 CREATE TABLE [dbo].[Order](
41         [id] [int] IDENTITY(1,1) NOT NULL,
42         [customer_id] [int] NOT NULL,
43  CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
44 (
45         [id] ASC
46 )WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
```

```
47 ) ON [PRIMARY]
48 END
49 GO
50 SET ANSI_NULLS ON
51 GO
52 SET QUOTED_IDENTIFIER ON
53 GO
54 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
       OrderProduct]') AND type in (N'U'))
55 BEGIN
56 CREATE TABLE [dbo].[OrderProduct](
57         [order_id] [int] NOT NULL,
58         [units] [smallint] NOT NULL,
59         [product_id] [int] NOT NULL,
60  CONSTRAINT [PK_OrderProduct] PRIMARY KEY CLUSTERED
61 (
62         [order_id] ASC,
63         [product_id] ASC
64 )WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
65 ) ON [PRIMARY]
66 END
67 GO
68 SET ANSI_NULLS ON
69 GO
70 SET QUOTED_IDENTIFIER ON
71 GO
72 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[Product]
       ') AND type in (N'U'))
73 BEGIN
74 CREATE TABLE [dbo].[Product](
75         [id] [int] IDENTITY(1,1) NOT NULL,
76         [supplier_id] [int] NOT NULL,
77         [units_in_stock] [int] NULL CONSTRAINT [DF_Product_count]  DEFAULT ((0)),
78         [name] [nvarchar](50) NOT NULL,
79  CONSTRAINT [PK_products] PRIMARY KEY CLUSTERED
80 (
81         [id] ASC
82 )WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
83 ) ON [PRIMARY]
84 END
85 GO
86 SET ANSI_NULLS ON
87 GO
88 SET QUOTED_IDENTIFIER ON
89 GO
90 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
       selCustomersAll]') AND type in (N'P', N'PC'))
91 BEGIN
```

```
92  EXEC dbo.sp_executesql @statement = N'-- ========================================
93  -- Author:              Name
94  -- Create date:
95  -- Description:
96  -- ========================================
97  CREATE PROCEDURE [dbo].[selCustomersAll]
98  AS
99  BEGIN
100         -- SET NOCOUNT ON added to prevent extra result sets from
101         -- interfering with SELECT statements.
102         SET NOCOUNT ON;
103
104     -- Insert statements for procedure here
105         SELECT * FROM Customer
106 END
107
108 '
109 END
110 GO
111 SET ANSI_NULLS ON
112 GO
113 SET QUOTED_IDENTIFIER ON
114 GO
115 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
        insCustomer]') AND type in (N'P', N'PC'))
116 BEGIN
117 EXEC dbo.sp_executesql @statement = N'-- ========================================
118 -- Author:              Name
119 -- Create date:
120 -- Description:
121 -- ========================================
122 CREATE PROCEDURE [dbo].[insCustomer]
123         -- Add the parameters for the stored procedure here
124         @name varchar(255)
125 AS
126 BEGIN
127         -- SET NOCOUNT ON added to prevent extra result sets from
128         -- interfering with SELECT statements.
129         SET NOCOUNT ON;
130
131     -- Insert statements for procedure here
132         INSERT INTO Customer ([name]) VALUES (@name);
133         SELECT @@IDENTITY;
134 END
135
136
137
138 '
```

```
139 END
140 GO
141 SET ANSI_NULLS ON
142 GO
143 SET QUOTED_IDENTIFIER ON
144 GO
145 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
        updCustomer]') AND type in (N'P', N'PC'))
146 BEGIN
147 EXEC dbo.sp_executesql @statement = N'-- ============================================
148 -- Author:              Name
149 -- Create date:
150 -- Description:
151 -- ============================================
152 CREATE PROCEDURE [dbo].[updCustomer]
153         -- Add the parameters for the stored procedure here
154         @id int,
155         @updatedCustomerName varchar(255)
156 AS
157 BEGIN
158         -- SET NOCOUNT ON added to prevent extra result sets from
159         -- interfering with SELECT statements.
160         SET NOCOUNT ON;
161
162     -- Insert statements for procedure here
163         UPDATE Customer SET [name] = @updatedCustomerName WHERE id = @id;
164 END
165
166 '
167 END
168 GO
169 SET ANSI_NULLS ON
170 GO
171 SET QUOTED_IDENTIFIER ON
172 GO
173 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
        delCustomer]') AND type in (N'P', N'PC'))
174 BEGIN
175 EXEC dbo.sp_executesql @statement = N'-- ============================================
176 -- Author:              Name
177 -- Create date:
178 -- Description:
179 -- ============================================
180 CREATE PROCEDURE [dbo].[delCustomer]
181         -- Add the parameters for the stored procedure here
182         @id int
183 AS
184 BEGIN
```

```
185         -- SET NOCOUNT ON added to prevent extra result sets from
186         -- interfering with SELECT statements.
187         SET NOCOUNT ON;
188
189     -- Insert statements for procedure here
190         DELETE FROM Customer WHERE id = @id;
191 END
192
193 '
194 END
195 GO
196 SET ANSI_NULLS ON
197 GO
198 SET QUOTED_IDENTIFIER ON
199 GO
200 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[insOrder
    ]') AND type in (N'P', N'PC'))
201 BEGIN
202 EXEC dbo.sp_executesql @statement = N'-- ================================================
203 -- Author:              Name
204 -- Create date:
205 -- Description:
206 -- ============================================
207 CREATE PROCEDURE [dbo].[insOrder]
208         -- Add the parameters for the stored procedure here
209         @customerId int
210 AS
211 BEGIN
212         -- SET NOCOUNT ON added to prevent extra result sets from
213         -- interfering with SELECT statements.
214         SET NOCOUNT ON;
215
216     -- Insert statements for procedure here
217         INSERT INTO [Order] (customer_id) VALUES (@customerId);
218         SELECT @@IDENTITY;
219 END
220
221 '
222 END
223 GO
224 SET ANSI_NULLS ON
225 GO
226 SET QUOTED_IDENTIFIER ON
227 GO
228 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
    selOrderByCustomer]') AND type in (N'P', N'PC'))
229 BEGIN
230 EXEC dbo.sp_executesql @statement = N'-- ================================================
```

```
231 -- Author:                Name
232 -- Create date:
233 -- Description:
234 -- =============================================
235 CREATE PROCEDURE [dbo].[selOrderByCustomer]
236         -- Add the parameters for the stored procedure here
237         @customerId int
238 AS
239 BEGIN
240         -- SET NOCOUNT ON added to prevent extra result sets from
241         -- interfering with SELECT statements.
242         SET NOCOUNT ON;
243
244     -- Insert statements for procedure here
245         SELECT * FROM [Order] WHERE customer_id = @customerId
246 END
247 '
248 END
249 GO
250 SET ANSI_NULLS ON
251 GO
252 SET QUOTED_IDENTIFIER ON
253 GO
254 IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[
        insOrderProduct]') AND type in (N'P', N'PC'))
255 BEGIN
256 EXEC dbo.sp_executesql @statement = N'-- =============================================
257 -- Author:                Name
258 -- Create date:
259 -- Description:
260 -- =============================================
261 CREATE PROCEDURE [dbo].[insOrderProduct]
262         -- Add the parameters for the stored procedure here
263         @orderId int,
264         @productId int,
265         @units int
266 AS
267 BEGIN
268         -- SET NOCOUNT ON added to prevent extra result sets from
269         -- interfering with SELECT statements.
270         SET NOCOUNT ON;
271
272     -- Insert statements for procedure here
273         INSERT INTO OrderProduct (order_id, product_id, units) VALUES (@orderId,
                @productId, @units);
274 END
275 '
```

```
276 END
```

Listing D.13: Schema.sql

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

Scott W. Ambler and Pramodkumar. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison Wesley, April 2006.

Paul Anderson and Mark Zarins. The codesurfer software understanding platform. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 147–148, Washington, DC, USA, 2005. IEEE Computer Society.

Malcolm P. Atkinson, Franois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the* 1^{st} *International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan*, pages 223–240. North-Holland, 1990.

Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), 1996.

Malcom P. Atkinson, P. J. Bailey, Ken J. Chisholm, Paul W. Cockshott, and Ron W. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.

David Binkley. Slicing in the presence of parameter aliasing. In *In Software Engineering Research Forum*, pages 261–268, 1993.

David Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering Methodology*, 16(2):8, 2007a.

David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Language Systems*, 30(1):3, 2007b.

Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. ISSN 0018-9340.

Zhenqiang Chen and Baowen Xu. Slicing concurrent java programs. *SIGPLAN Notices*, 36(4):41–47, 2001a.

Zhenqiang Chen and Baowen Xu. Slicing object-oriented java programs. *SIGPLAN Notices*, 36(4): 33–40, 2001b.

Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In Naoki Kobayashi, editor, *APLAS*, volume 4279 of *LNCS*, pages 374–388. Springer, 2006.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

Edgar. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

Yossi Cohen and Yishai A. Feldman. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Transactions on Software Engineering and Methodology*, 12(3):285–316, 2003.

Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. In *Very Large Data Base (VLDB)*, 2008.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

C. J. Date. *A Guide to the SQL standard*. Addison Wesley, 1989.

Mark Harmann David Binkley. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

Claude Delobel, Michael Kifer, and Yoshifumi Masunaga, editors. *Deductive and Object-Oriented Databases, Second International Conference, DOOD'91, Munich, Germany, December 16-18, 1991, Proceedings*, volume 566 of *Lecture Notes in Computer Science*, 1991. Springer.

Rajeev Kumar Durga Prasad Mohapatra, Rajib Mall. An overview of slicing techniques for object-oriented programs. *Informatica*, 30(2):253–277, Nov. 2006.

Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2006. ISBN 3-540-33056-9.

Dan Sommer Fabrizio Biscotti, Colleen Graham. Market Share: Database Management Systems Software, EMEA, 2005. Technical report, Gartner, June 2006.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987. ISSN 0164-0925.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.

Martin Fowler and Jim Highsmith. The agile manifesto. In Software Development, Issue on Agile Methodologies, `http://www.sdmagazine.com`, last accessed on March 8th, 2006, August 2001.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software*. Addison Wesley, 1995.

Spyridon K. Gardikiotis and Nicos Malevris. Dasian: A tool for estimating the impact of database schema modifications on web applications. In *AICCSA '06: Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 188–195, Washington, DC, USA, 2006. IEEE Computer Society.

Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, Washington, DC, USA, 2004. IEEE Computer Society.

William G. J. Halfond and Alessandro Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.

Graham Hamilton, Rick Cattell, and Maydene Fisher. *JDBC Database Access with Java*. Addison Wesley, 1997.

Christian Hammer and Gregor Snelting. An improved slicer for java. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, New York, NY, USA, 2004. ACM Press.

Ramzi A. Haraty, Nash'at Mansour, and Bassel Daou. Regression testing of database applications. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 285–289, New York, NY, USA, 2001. ACM Press.

Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994. ISSN 0164-0925.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 229–243. ACM, 1988.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990. ISSN 0164-0925.

Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical report, In Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1994.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 393–407, New York, NY, USA, 1995. ACM Press.

Java Patterns. J2EE Patterns. http://java.sun.com/blueprints/patterns/, 2007.

Richard Jones and Rafael D. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, September 1996.

Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proc. of the* 9^{th} *European software engineering conference (ESEC/FSE 2003)*, pages 98–107, New York, NY, USA, 2003. ACM Press.

A. Karahasanovic. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD thesis, Dept. of Informatics, University of Oslo, 2002.

Amela Karahasanovic and Dag I. K. Sjoberg. Visualizing impacts of database schema changes - a controlled experiment. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 358, Washington, DC, USA, 2001. IEEE Computer Society.

Gavin King and Christian Bauer. *Hibernate in Action: Practical Object/Relational Mapping*. Manning, October 2004.

Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 245–254, Washington, DC, USA, 2005. IEEE Computer Society.

David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, New York, NY, USA, 1981. ACM.

William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, 1993.

Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *15th International Conference on Compiler Construction*, volume 3923 of *LNCS*, pages 47–64, Vienna, 2006. Springer.

D. Liang and M.J. Harrold. Slicing objects using system dependence graphs. *ICSM '98: Proceedings of the 14st IEEE International Conference on Software Maintenance*, 00:358, 1998.

Sandahl K. Lindvall M. How well do experienced software developers predict software change? *Journal of Systems and Software*, 43:19–27(9), 1998.

A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *Proc. of the 30th Int. Conference on Software Engineering, Leipzig, Germany*. ACM Press, 2008.

E. Meijer, B. Beckmann, and G. Biermann. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. of SIGMOD 2006*. ACM Press, 2006.

Microsoft Patterns. Microsoft Patterns and Practices Developer Center. http://msdn.microsoft.com/practices/, 2007.

Microsoft Phoenix. Phoenix Framework. http://research.microsoft.com/phoenix, 2007.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Washington, DC,, 2004a. IEEE Computer Society.

Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Transactions on Software Engineering and Methodology*, 13(2):199–239, 2004b.

Shari Lawrence Pfleeger. Design and analysis in software engineering: the language of case studies and formal experiments. *SIGSOFT Softw. Eng. Notes*, 19(4):16–20, 1994.

Shari Lawrence Pfleeger. *Software engineering: theory and practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5):27, 2007. ISSN 0164-0925.

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, 2004. ACM Press.

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 11–20. ACM Press, 1994.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37:383–393, 1995.

Jed Rubin and Linda Tracy. IT Key Metrics Data 2007: Key Applications Measures: Language Usage. Technical report, Gartner, December 2006.

Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented languages, 2003.

Ina Schaefer and Arnd Poetzsch-Heffter. Slicing for model reduction in adaptive embedded systems development. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 25–32, New York, NY, USA, 2008. ACM.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.

George Bernard Shaw. unknown source, unknown year.

Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

Dag I. Sjoberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.

María José Suárez-Cabal and Javier Tuya. Using an sql coverage measurement for testing database applications. *ACM SIGSOFT Software Engineering Notes*, 29(6):253–262, 2004. ISSN 0163-5948.

Frank Tip. A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, Amsterdam, Netherlands, 1994.

Mark D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. *ACM SIGPLAN Notices*, 42(1):199–210, 2007.

Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *ACM SIGPLAN Notices*, 43(10):19–36, 2008.

David Willmor and Suzanne M. Embury. A safe regression test selection technique for database-driven applications. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 421–430, Washington, DC, USA, 2005a. IEEE Computer Society.

David Willmor and Suzanne M. Embury. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Workshop (UKTest 2005)*, sep 2005b.

David Willmor, Suzanne M. Embury, and Jianhua Shao. Program slicing in the presence of a database state. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 448–452, Washington, DC, USA, 2004. IEEE Computer Society.

Kenny Wong. Rigi Users's manual, Version 5.4.4. Technical report, University of Victoria, Dept of Computer Science, ftp://ftp.rigi.csc.uvic.ca/pub/rigi/doc, 1998.

Robert K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, Inc, February 1989.