Impact Analysis of Database Schema Changes[*]

Andy Maule, Wolfgang Emmerich and David S. Rosenblum
London Software Systems
Dept. of Computer Science, University College London
Gower Street, London WC1E 6BT, UK
{a.maule|w.emmerich|d.rosenblum}@cs.ucl.ac.uk

ABSTRACT

We propose a technique for analysing the impact that changes to relational database schemas have on object-oriented application programs. Our technique combines slicing and k-CFA data flow analysis in order to extract all possible insertions, updates, queries and stored procedure executions that an object-oriented application may make. We then perform impact analyses using relational programs over the extracted data. We describe an implementation of our technique in SUITE, an application built using the Microsoft Phoenix framework for .NET byte code and the BDD based CrocoPat relational program interpreter. We subject our implementation to an evaluation with a case-study, a commercially available content management system written in C#. We have investigated 62 versions of this application and its underlying schema and analyse the three versions of this application that have the most significant schema changes in detail with SUITE. The largest of these versions comprises 127,340 lines of C# source code and has about 900 interactions with the underlying relational database, whose schema has a total of 615 columns and 568 stored procedures. The case study demonstrates that inter-procedural call graphs need to be taken into account, that it is insufficient to just consider string-based analyses and moreover that 0-CFA or 1-CFA based data flow analysis techniques are not precise enough for this problem. Our implementation is able to analyse a version in under 2 minutes and highlight all relevant changes.

1. INTRODUCTION

Enterprise IT applications usually rely on databases to persist information. The vast majority of applications use relational rather than object-oriented or deductive database management systems [10]. At the same time the prevailing programming paradigm for constructing components that create, update and query data in such enterprise IT systems are written using object-oriented programming languages, such as C++, Java and increasingly C#. This leads to the so called impedance mismatch [3] that denotes the significant conceptual gap between how applications and databases are structured.

There is a large body of work aimed at bridging that gap. These include object-relational mappings (ORMs), such as Persistence or Hibernate, or a Call Level Interface (CLI) library, such as JDBC or ADO.NET. In the near future, programming languages will include constructs such as statically typed queries, as they were defined for .NET 3.0 in the Language Integrated Native Query framework [17]. Both ORMs and statically typed query languages support the detection of violations of inter-language type constraints. Moreover, there have been a number of advances recently in the area of analyzing the consistency of queries made using a CLI in object-oriented programs against relational database schemas [13]. These have been evaluated using relatively small applications and have been demonstrated to address the problem of how type safety between a query string that is passed to a CLI and the relational database schema against which the query is formulated.

We are interested in a slightly different, but related problem that addresses the identification of statements in an object-oriented application that are affected by, and need to be changed as a result of, a give relational database schema change. Whenever a change occurs in a given component, it may affect other components. The effects of changes ripple out along the dependencies between the components of the system. As these dependencies increase in number and complexity, the effects of change can become difficult to trace. We propose techniques for extracting definition-use relationships across database schemas and CLI or ORM queries that are embedded in object-oriented programs in order to support impact analysis of schema change. We also aim to demonstrate that these techniques are useful and can be applied to large-scale industrial applications.

This paper discusses an analysis technique for object-oriented programs in support of analysing the impact of changes to a relational database schema. We are interested in impact analyses of changes to relational schemas, including table definitions, views, triggers and stored procedures. The principle contribution of this paper is therefore twofold. We firstly present our program analysis approach to extract relationships between where insertions, updates, queries and stored procedure invocations are made using a CLI/ORM library call and the definition of the relevant parameters. We use a combination of program slicing and k-CFA data

[*]Research UK under MRL Contract 2005-054. David Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

flow analysis, that we have implemented using Microsoft's Phoenix framework [19] for .NET byte code. Our program analyzer extracts definition-use relationships and presents them in RSF to the CrocoPat RML interpreter [4], which we use to reason about impact for particular types of changes. The second key contribution is the discussion of a large industrial case study that we have used to evaluate our technique. We have considered a version history of about two years, which had 62 different versions of both schema and application. We have used the implementation of our technique to analyse the three versions with the most significant changes. The largest of these versions had 127,340 lines of C# code. In each of these versions, we analyzed just under 900 ADO.NET invocations that perform insertions, updates, queries and stored procedure invocations. The case study shows that 0-CFA or 1-CFA data flow analyses that were proposed in related work are insufficient and it also shows that our combination of program slicing with k-CFA data flow analysis produces useful results in about 2 minutes for applications of this size.

The paper is further structured as follows. In Section 2, we give a motivating example for impact analysis of relational database schemas and describe in more detail the features of enterprise architecture for which such analyses are significant. In Section 3, we describe our approach to change impact analysis and its implementation is described in Section 4. We then describe related work in the areas of program analysis, software maintenance and analysis of embedded queries in Section 5. In Section 6, we present the results of subjecting a large industrial case study to change impact analyses and we discuss the findings in Section 7. We conclude the paper in Section 8.

2. MOTIVATING EXAMPLE

Consider a group of scientists, who are conducting experiments and are storing the resulting data in a database. The schema defines two database tables, `Experiments` with four columns and `Readings` with three columns. The italicised column names identify the primary keys of their respective tables.

Experiments			
ExperimentId	Date	Name	Description
VARCHAR(30)	DATE	VARCHAR(30)	TEXT
req.	req.	not req.	not req.

Readings		
ReadingId	ExperimentId	Data
INT	VARCHAR(30)	BINARY
req.	req.	req.

There are two classes of stakeholders in our example: application developers, whose applications query and update the database, and database administrators (DBAs), who maintain the database including the database schema. Consider an application that uses the following queries and updates. In these queries and updates '?' represents parameters supplied at run-time by the application.

```
SELECT Experiment.Name, Experiment.Id         // Query Q1
FROM Experiments
WHERE Experiments.Date=?

INSERT INTO Experiments                       //Update Q2
(ExperimentId, Date, Name, Description)
VALUES (?, ?, ?, ?);
```

```
INSERT INTO Readings                          //Update Q3
(ReadingId, ExperimentId, Data)
VALUES (?, ?, ?);

SELECT Readings.ReadingId, Readings.Data      // Query Q4
FROM Readings
WHERE Readings.ExperimentId=?
```

Let us now assume that there is a requirements change. Experiments used to start and finish on the same day, recorded in `Experiments.Date`. Now scientists want to conduct a new type of experiment that lasts longer than a day and requires taking readings over several days. The schema needs to be altered to include a new column, `Readings.Date` that allows readings to contain more detailed information about when they were taken (`Change 1`). Secondly, the introduction of this new reading date requires `Experiments.Date` to be renamed to `Experiments.StartDate` so that it will not be confused with the `Readings.Date` field (`Change 2`). Finally, the DBA recognises that users of the database have not been using the `Experiments.Name` field. They have been relying on the `Experiments.ExperimentId` field to give each experiment a unique name and have been leaving `Experiments.Name` blank. The DBA decides that the `Experiments.Name` column is superfluous and should be deleted (`Change 3`).

Experiments		
ExperimentId	StartDate	Description
VARCHAR(30)	DATE	TEXT
req.	req.	not req.

Readings			
ReadingId	Date	ExperimentId	Data
INT	DATE	VARCHAR(30)	BINARY
req.	req.	req.	req.

2.1 The impacts of schema change

The schema changes will have several impacts on the application queries. We define an impact as *any location in the application which will behave differently, or is required to behave differently as a consequence of the schema change.* The most obvious form of impacts are errors. In our example change scenario, the following errors will occur:

Q1	err1	references invalid `Experiments.Date` column
	err2	references invalid `Experiments.Name` column
Q2	err3	references invalid `Experiments.Date` column
	err4	references invalid `Experiments.Name` column
	err5	no value for req. field `Experiments.StartDate`
Q3	err6	no value for req. field `Readings.Date`

These errors are typical of the types of impacts that can arise from a schema change. They are simply a result of running queries that are no longer valid.

Not all impacts necessarily cause errors. For example, `Readings.Date` is required by our new schema. We make the distinction that 'required' means that no default value is specified and that null values are not allowed. Suppose the DBA could decide to remedy `err6` by giving this column a default value of the current date. In this situation it is very possible that an application developer would overlook `Q3` as being unaffected. When a new reading is inserted the default date would be used as specified by the DBA. If the database was in a different time zone or the query reading was inserted long after it was taken this value could be very wrong. This may or may not be the desired behaviour. And we would classify this type of impact as a warning.

Q3	warn1	Insert semantics changed. `Readings.Date` added with default value.

A second type of warning would be where a column has been added, and it may need to be used in this query. For example, `Change 1` adds the `Readings.Date` field. This will not affect the validity of `Q4`, but the application developer may wish to add the `Readings.Date` field to the result set of this query. Intuitively this would be one of the places where this new data may need to be returned. `Q4` will run without any errors occurring but will not return all of the available data. The requirements change may mandate that all readings must now be displayed with their corresponding date, and therefore `Q4` may need to be altered to return the date information, even though the query is essentially unaffected. This fits our definition of an impact, as although the query would not behave differently, it is required to behave differently following the change.

Q4	warn2	New available data. `Readings.Date` added.

Whilst the impacts are problems that must be reconciled, the problem we are addressing is not the impacts themselves, but rather the difficultly of discovering and predicting them.

2.2 The difficulty of schema change

We argue that in two key stages of the schema change process, the difficulty of discovering and predicting impacts is particularly problematic:

Before the schema change is made: The DBA has to estimate the benefits of each change against the cost of reconciling the existing application with the new schema. If the changes have little or no effect upon the application, then the DBA can make the changes easily. If the changes could have a large impact, then it might be best to leave the schema as is, or consider alternative changes. Without an accurate estimate of these costs, DBAs have to make overly conservative decisions, avoid change, or use long periods of integration testing [1]. This is a major cause of difficulty in the schema change process.

After a change is made: The application developer needs to locate all affected areas of the application and reconcile them with the changed schema. Again, this is a difficult problem, and without this information, the schema change process can cause errors in the application which go unnoticed. This is the second major cause of difficulty that we identify.

If we consider the information required by the DBA and application developers, at these two critical points, we see that they primarily need to know the same thing. Namely, every impact location in the application. The goal of our work is to investigate the feasibility of supplying this missing information.

2.3 Requirements for change impact analysis

Our example identifies some requirements, which we shall summarise. The main requirement concerns the level of information we can obtain about each impact. The impacts described in our example have very simple error descriptions that are missing information that the developers must infer for themselves. All the impacts we have mentioned above, and related work discussed in Section 5, only provide a minimum level of impact information. For example, if the developer knew that a renaming schema change (`Change 2`) caused errors `err1` and `err3`, they could simply rename some source code or mapping file to rectify the problem. If they did not know this, they may have to manually trace which query caused the error, to make sure that it is not a separate addition and deletion. We propose that if the developer knew the semantics of the schema change that caused the impact, and they were given some advice on remedial action, they could make the required alterations in the ORM far more quickly, with less chance of error, especially when this procedure must be carried out repeatedly by hand, or where queries are not easily readable in the source code.

Given the above examples and shortcomings of common solutions, the goal of this paper is to present a change impact analysis technique for database schema change that retrieves information to better inform the schema change process. Firstly, we want to support the DBA to predict the effects of schema change in order to better inform the choice between alternative changes. Secondly, we wish to inform the application developer of all queries and updates whose behaviour will be altered, or require altering, because of a schema change. This requires an automated solution that gives detailed diagnosis for each impact including the causing change, suggested possible remedial action and an indication of the cost/severity of reconciliation that may be available.

2.4 Data access practices

Before describing our approach, we first describe details of how applications interact with databases in practice. This informs the choice of program analysis techniques that we deem viable.

```
01 class Reading{
02   private int _readingId;
03   private byte[] _data;
04
05   Reading(DBRecord rec){
06     _readingId = rec["ReadingId"];
07     _data      = rec["Data"];
08   }
09
10   public static Reading GetReadingById(int id){
11     DBParams params = new DBParams();
12     DBRecordSet queryResult;
13
14     params.Add("@ExpId", ParamType.Integer, id);
15
16     queryResult = QueryRunner.ExecProcedure(
17       "SELECT Readings.ReadingId, Readings.Data" +
18       "FROM Readings " +
19       "WHERE Readings.ExperimentId={@ExpId}",
20       params);
21
22     return Reading(queryResult[0]);
23   }
24 }
25
26 class QueryRunner{
27   public DBRecordSet ExecProc(string pName, DBParams p){
28    return DBConnection.Exec(pName, p)
29   }
30 }
```

Figure 1: C# data architecture for Q4

In Figure 1 we show an example of how database access logic can be implemented. The figure shows how `Q4` from our example scenario, might be executed in practice.

The entry point in this example is the `GetReadingById` method on Line 10, which can be called statically to return a `Reading` object that matches the supplied parameter. This method creates a SQL query as a string, and supplies the missing parameter in the form of a `DBParam` object[1]. The query is then executed by the `ExecProc` method on Line 28. Line 28 executes the query against the database, and returns the results in a `DBRecordSet` object. On Line 22, the first row of the result set is used to instantiate a new `Reading` object. This object is then populated by extracting values from the supplied `DBRecord` in the constructor of the `Reading` object. Please note that we have omitted many details, such as checking the number of returned rows or catching database exceptions, for the sake of clarity.

In this example, all types belonging to the persistence API begin with DB (`DBConnection`, `DBRecord`, `DBRecordSet`, `DBParams`). These classes are typical of those found in the JDBC and ADO.NET libraries. We therefore argue that there is a need for our analysis to consider much more than string based queries. In this example we may need to know the state of the `DBRecord`, `DBRecordSet` and `DBParams` objects, to know the exact query that is being executed, and where/how results are used. This is an important consideration that we discuss in more detail in Section 7.

In Figure 1, the actual execution of the database query happens in the `QueryRunner` class. If we were to add a new class for the `Experiments` table, assuming we preserved this architecture, then it would also use the `QueryRunner` for the execution of queries. This approach of consolidating execution of queries to a small selection of methods is typical of common practice. In fact, many recommended architectural patterns for data access, have several such layers of abstraction and indirection in order to create composable and maintainable programs. Such patterns, and the reasons for using them, are well described, for example Chapters 3 and 10-13 by Fowler [11] are especially descriptive of such patterns for database access. Several other similar references also exist, which discuss architectural patterns, and the reasons for their use [12, 18, 24]. We argue that such patterns and architectures are in widespread use, and therefore, that any analysis techniques that we develop, should be able to cope with complex control flow caused by such layers of indirection and abstraction. This is an important consideration that we discuss in more detail in Section 7.

3. APPROACH

Given the requirements we have discussed for database schema change impact analysis, we are faced with two tasks. Firstly, we need to extract all possible queries from an application to discover any dependencies that might exist between the application and the database schema. Secondly, we must use this information to predict what could possibly be affected as the result of a schema change.

3.1 Extracting Queries

In order to *extract queries* from applications we use a program analysis approach. We now give a brief overview of the process before describing it in more detail in the remainder of this section.

We have identified the need to analyse programs with complex control flow, produced as the result of using various architectural patterns. This complex control flow requires a high level of precision in order to produce useful results, as we shall illustrate with our case study in Section 6. We refer to the levels of precision defined by Ryder [21]. We argue that a context sensitive analysis is required, meaning that for every procedure that we analyse, we should take into account the context in which this procedure is called. We use k-CFA analysis [22] to this effect. However as k increases, this technique can become very expensive. Therefore we take the approach of using inter-procedural program slicing [25] to limit the parts of the application to which we apply this potentially expensive analysis. We use backwards slicing to understand the composition of queries, and forward slicing to see where and how the results of the criterion are used.

We start our analysis with a single static assignment (SSA) [9] representation of our application[2]. We traverse the def-use graph provided by the SSA representation, and mark the definitions and subsequent uses of any interesting datatypes, which are any type that could potentially be involved in a query. This includes strings and text types, persistent library types that represent or execute queries and persistent library types that are containers for query results. We then use this information to build a set of intraprocedural control flow graphs for query data types, by traversing the original control flow graph and only adding nodes that have been marked as interesting.

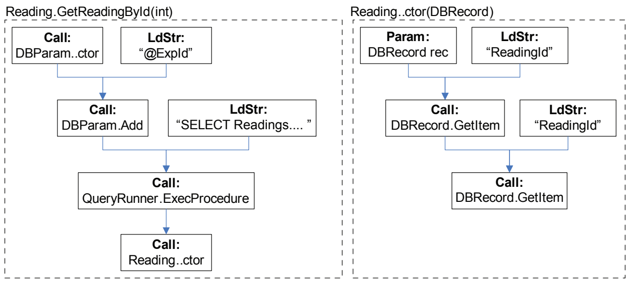


Figure 2: Two example intraprocedural graphs

Figure 2 illustrates two example intraprocedural graphs that result from analysing the `Reading.GetReadingById` method and the `Reading` class' constructor shown in Figure 1. These graphs show that the relevant control flow of the program has been preserved, but we have abstracted away all instructions that do not involve interesting data types and all calls that remain in the intraprocedural graph either have arguments with interesting data types, or return values with interesting data types.

Each node has a type, such as `Call` or `Param` and the node stores any further details that may be required. For example, in most nodes, we store keys that are used to identify which items of the program state will be affected by this node. This extra information is not shown in the diagram for the sake of clarity. Usually, the type of the node corresponds directly to the opcode of the instruction it represents, or some kind of explicit control flow operation, for example we include `PHI` nodes which preserve PHI control flow from the initial SSA representation.

[1]Supplying parameters in this way is common practice to promote modularity and help avoid problems such as SQL injection attacks. The JDBC `PreparedStatement` object is a commonly used example.

[2]In our case this is provided to us by the Phoenix Framework as discussed in Section 4

In the next step we ascertain which procedures may potentially be involved in the execution of a query. In order to know where queries are executed, we must first define a set of calls to methods found in the persistence libraries, which are responsible for executing queries. These are the methods that will be used to create slicing criteria, and we refer to these known query executing methods as *hotspots*. In the example of Figure 1, the method `DBConnection.Exec` is a hotspot. The set of hotspots is defined statically, and will vary, according to the persistence libraries being used.

For each intraprocedural graph, we examine each call node in turn. If a call node represents a call to a hotspot, then it is marked for further analysis. A call node is also marked if it transitively calls a method which has been marked.

The next stage is to create interprocedural program slices of calls to hotspots. The algorithm proceeds as follows. For every call node in the intraprocedural graphs, we have established whether the call is, or may potentially result in, a call to a hotspot. For each of these calls, we start creating a program slice by adding a copy of the last child of this call, as the leaf of a new slice graph. Note that we do not start this traversal from the call site itself, as we may be interested in analysing the data returned from the call.

We traverse from each call node, back up the intraprocedural control flow graph through each parent edge, making copies of any interesting nodes we find, and inserting them into the corresponding position in the new slice graph. Whenever we reach a call node, we resolve the following interprocedural effects.

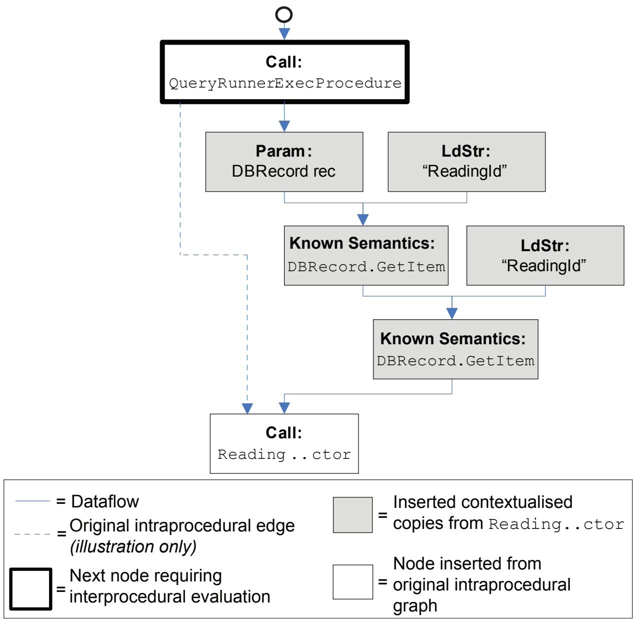
1. For some special methods (such as hotspots or base library methods), we supply our own 'known semantics' nodes, to insert into the slice graph, in place of the actual calls. In the example graph, the node that calls `DBRecord.GetItem` would be replaced with a special known semantics.

2. If the call returns an interesting data type, then we insert a copy of the called procedure into the slice graph, before we insert the call node.

3. If the call has any parameters (that have not already been found in the inserted nodes of the calls return path) then we also insert the intraprocedural graph of the last child for each parameter, into the slice graph. This time these inserted nodes are inserted above the copy of the current call node in the slice graph.

Whenever we resolve such interprocedural effects of a call, we are adding nodes to the slice graph that do not exist in the intraprocedural graph that contains the call, often as new contextualised copies of other intraprocedural graphs. We use a well known approach of maintaining context-sensitivity called k-CFA [22]. Effectively we use a representation of the call site in order to create a context for the call, and label any inserted nodes with this context. This context is important during the evaluation of these program slice graphs, as it allows any parameters in the call to be replaced with their actual values.

In order to actually insert such interprocedural effects into the slice graph at a call site, we stop walking up the current intraprocedural call graph, and make a note of where we have stopped. We resume walking at the last node in the intraprocedural graph for the nodes that we wish to insert, copying each interesting node, just as before. If we find another call node, in this latest intraprocedural graph, we also resolve the interprocedural effects of this call in the same way. We resolve such sub calls to an arbitrary depth[3]. We note that this arbitrary level of context sensitivity is an important feature of our approach, and its implications are discussed in Section 7. When we reach a parameter node, or a node with no parents, then we return to the saved position in the previous intraprocedural graph, continuing where we left off. We do this until we reach nodes with no parents in the intraprocedural graph, and the slice is complete.



Call:
QueryRunnerExecProcedure
Param:
DBRecord rec
LdStr:
"ReadingId"
Known Semantics:
DBRecord.GetItem
LdStr:
"ReadingId"
Known Semantics:
DBRecord.GetItem
Call:
Reading..ctor
= Dataflow
= Original intraprocedural edge (illustration only)
= Next node requiring interprocedural evaluation
= Inserted contextualised copies from Reading..ctor
= Node inserted from original intraprocedural graph

Figure 3: Building an interprocedural program slice

We illustrate an example of construction of an interprocedural program slice graph in Figure 3. This figure shows the first stages of this process for the `Readings.GetReading` graph. In this case the last interprocedural child of an interesting call, is the call to `Reading..ctor`. The traversal is started for this graph, and the call node is copied into the interprocedural slice, shown at the base of the diagram. Because, this call does not have a return type that is interesting, no nodes are inserted below this call site. However, this call does supply a parameter with an interesting data type, and therefore we insert a contextualised copy of nodes from the constructor method of the `Reading` class as shown. At this point we show how the intraprocedural nodes are added instead of adding an edge to the call to `QueryRunner.ExecProcedure` method. When we reach the `Param` node, the analysis returns to the analysing the original intraprocedural graph, adding the next node of the original call. In this case this is the node representing a call to the `QueryRunner.Execute` method. This node is then the next node that requires evaluation, and this is the current state in which we see the diagram. The building of the slice will continue from this highlighted node, until all path have been exhausted.

[3]Although in practice this is limited by the memory constraints of our implementation, and the size of the stack datastructure we use to store the context and return site information.

It is also worth illustrating that, in our example graph, we have replaced calls to methods for the DBRecord and DBParam classes, with special known semantics nodes. These nodes have the effect of noting the state of the program when they are evaluated in the final stage of our analysis, as described below.

So far we have avoided the discussion of the problems caused by resolving correct methods in the presence of virtual dispatch in object oriented languages. We propose the solution of using an interprocedural points-to analysis [20] to determine the possible types that a call site may point to. The details of how such points-to information can be incorporated into program slicing, is described by Larson et. al. [16].

The final stage of this program analysis involves interpreting these graphs to give an approximation of the possible queries that may result at a given hotspot. For each root node of the graph we create an object that represents the program state. This program state is then passed to each successive node in the graph. Each type of node has a semantics, so that given a program state, it will alter the state accordingly, allowing us to simulate an approximate semantics for the application. For example, the LdStr nodes shown in Figure 2 will insert a new string variable into the state.

It is important to note that the semantics of these nodes allow for approximations, and if a variable has several possible values at a given point, the semantics operate on all possible values, sometimes producing numerous permutations. Also, when two or more branches in the graph merge, we merge the program states of all incoming edges before continuing to the next child node. This ensures that for call sites with multiple parameters, we can approximate the correct effect.

When we encounter a hotspot, we examine the path that has been taken to reach this hotspot, and store it alongside the possible queries that can be executed here. We similarly make note when we find the use of a specific field of a query result, or the use of an SQL parameter for a stored procedure.

At this point we have extracted all the required information from the source code of the application. We output the results of this analysis, and use it to perform impact analysis, as described next. If required, this information can also be supplemented with information extracted from the schema. Extracting such data is a trivial programming exercise, as all major DBMSs provided a facility to examine meta data. Therefore, we do not describe this process in any detail, other than to note that this information can be readily obtained.

3.2 Impact calculation

Once we have extracted the possible queries from an application, we use this information to make predictions about the possible effects of schema change, we call this process *impact calculation.*

A good description of the impacts that may arise from a given schema change, are described in Database Refactoring [1]. This book describes a catalogue of possible changes that can occur in relational database schemas, and detailed descriptions about their effects. Schema changes can have a wide range of effects and this requires a flexible method of interrogating the extracted query information. To do this, we store the extracted data as a set of simple binary relations and use a relational language to reason about the query data. These relational programs are relatively short and we create one program for each type of impact we would like to analyze.

Most of the time, we require only simple relational programs, which can search the possible query strings for a regular expression term, returning a set of code locations that might be affected. However, the flexibility to write arbitrarily complex query programs, allows us to provide impact calculation even for subtle impacts, such as those described in Section 2.

```
Q_exec_TXT    Q1  "sp_StoredProcedure1"
Q_execAt_LOC  Q1  "QueryExecute.cs:123"
```

Figure 4: Example RSF data

We store this extracted data using the RSF file format [28] as shown in Figure 4. This example shows two very simple binary relations. The first line indicates that Q1 is linked to the value sp_StoredProcedure1 by the relationship Q_exec_TXT. This means that a query, denoted by the id Q1 executes the following query text, in this case, simply the name of a stored procedure. The second line similarly notes the same query, Q1, is executed in the QueryExecute.cs file at line number 123. This is a very simple example of the results that we can store, but this representation is useful as it can be easily extended to include arbitrary relationships, and can be efficiently analysed as described next.

```
AffectedQueries(x)  :=  Q_exec_TXT(x, $1);
AffectedExecs(x)    :=  EX(y, AffectedQueries(y) &
                              Q_execAt_LOC(y, x));

PRINT ["Exec found at: "] AffectedExecs(x);
```

Figure 5: Example RML program

Figure 5 shows an example relational program. The program takes our sample RSF file as input. The first line of the program creates a set called AffectedQueries. This set consists of all query identifiers where the text of the query matches a given parameter $1. We then find all affected executions of this query by executing the second line, where we select all execution locations where the query id is found in the set of affected queries. The final line, simply prints a list of all affected executions locations. In our example, we would supply the RSF from Figure 4 and the parameter 'sp_StoredProcedure1' as inputs. This would result in the program printing out the line "Exec found at: QueryExecute.cs:123".

Using similar programs we create more complex queries. In fact for every warning, or error that can arise, we write an RML program that takes the required parameters. For every proposed schema change, we run all the applicable impact calculation programs against the extracted data. This results in the prediction of all potential impact sites occurring in the application.

4. IMPLEMENTATION

During the course of our research we have developed a prototype system that we call SUITE (Schema Update Impact Tool Environment). The architecture of SUITE is shown in Figure 6. This diagram shows SUITE's key constituent

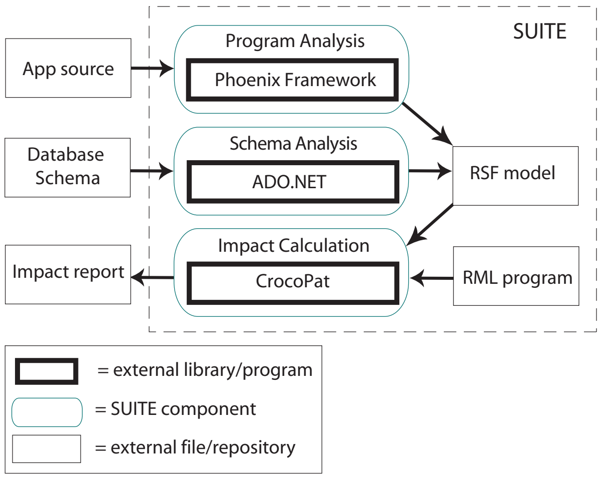


Figure 6: Tool Architecture

components and the dataflow between them during impact analysis.

As described in the previous section, we use a file format called RSF [28] to store the results of the program and schema analysis. For each possible change we have encountered, we have created programs to query the RSF model and perform impact calculation. We write these program in RML, and use a tool called CrocoPat [4] to execute these RML programs against the RSF model. The eventual output of this process is a text-based impact report.

We are currently targetting only C# applications that use SQL Server databases. We obtain an SSA representation of compiled .NET binaries by using the Phoenix framework [19], on top of which we implement the remainder of our program analysis.

We envisage extending SUITE for many different languages, persistence technologies and DBMSs in the future. In fact we have already started investigating the use of the Soot framework [26] for applying our technique to Java, as the Soot framework provides the same functionality for which we currently use Phoenix. This will allow SUITE to be useful in truly heterogeneous database applications, which we believe is an important feature that our proposed technique will offer.

Currently our implementation is still at the functional prototype stage and, as such, has some limitations. We currently do not implement the points-to analysis required for resolving calls in the presence of virtual dispatch, which we intend to include in future versions of our implementation. Also, we currently have accuracy issues regarding the processing of loops, recursion and parameters passed by reference. These problems have all been solved in related work, and we are incorporating these solutions into our prototype. However, these flaws restrict us from making claims that our analysis is safely conservative. We acknowledge that this is an important consideration for such a technique, but we defer this to future work.

5. RELATED WORK

There is a great deal of work related to software change impact analysis [2]. However, we are only aware of one similar application related to database applications [15]. This earlier work focuses on object-oriented databases whereas we consider relational databases. We argue that the object-relational impedance mismatch makes this a significantly harder problem, however we take inspiration from the approaches defined here, especially the work on visualisation of results, which we do not currently address.

There have been a number of recent works investigating program analysis techniques for extracting database queries from applications. Amongst these were two important papers which initially inspired our work. Firstly Christensen et al. described a string analysis technique [6] for statically predicting the values of string variables in Java programs. This work was then built upon by Gould et. al. in order to perform static type checking of dynamic string based queries for JDBC [13]. However, Christensen's program analysis was based on a context-insensitive algorithm. This means that for any data access architectures that use publicly accessible methods for query execution, the range of possible queries will become unknown. As shown by our case study, we found that this was not accurate enough to perform useful impact analysis, as too many of the queries became unknown. For example, consider the code sample shown in Fig 1, on Line 20 a query is executed from the string parameter `pName`. This method is public, and in a context insensitive analysis we would have to consider every context in which this method can be called, which is effectively infinite and would result in an unknown query value.

An improvement in precision of string analysis was introduced by Choi et. al. [5]. This analysis technique improves upon previous string analyses by supplying a context sensitive (specifically 1-CFA) analysis, and expanding the scope of the analysis to include some heap and string types, such as the Java StringBuffer. We acknowledge that this approach grealty improves the accuracy of string analysis, however, as we describe in Section 7, the precision of the analysis needs to be improved further still, in order to be useful for impact analysis. This technique was based on abstract interpretation [8], which is an alternative approach to our data flow analysis and program slicing. It is not clear how this abstract interpretation based approach would be affected by increasing the level of context sensitivity, and the scope of the analysis. However, we are interested in investigating this approach further, as it may offer potential performance improvements and a path to formalize our analysis.

Some related work has been in analysing transparent persistence [27] using program analysis, although the focus here is on providing optimisation of queries. This work is based on abstract interpretation [8] and bares similarity to other work on extracting queries from legacy applications [7], however both of these techniques are tailored to languages where queries are effectively embedded. This ignores many of the problems we have dealt with in this paper, but may provide an insight into how other persistence technologies, such as transparent persistence could be incorporated into our approach in the future.

There has also been some related work produced by the testing community [14]. However, the program analysis used in this research, suffers from the same precision problems as the other program analysis techniques mentioned. We also note that database oriented testing in general, does not eliminate the requirement for change impact analysis, and we consider such work to be orthogonal.

6. EVALUATION

In order to evaluate the feasibility of our proposed analysis, we have conducted an in-depth case study as described by Yin [29]. Our subject application is the irPublish^{TM}content management system (CMS), produced by Interesource Ltd. This application has been in development for five years and is 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. irPublish itself consists of many different components, of which, we chose to analyse the core irPublish client project. This 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 instances of design and architectural patterns proposed by Gamma et. al [12], Fowler [11] and Microsoft's and Sun's recommended architectural practice [18, 24]. It is also important to note that irPublish has been developed using established software engineering practices such as unit testing, source code revision history, 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 recommended real world practice, and therefore, our findings will also apply to many other similar applications.

We conducted a historical case study, based upon the version history of the irPublish client project. We examined the source code repository of changes going back two years. This gave us 62 separate schema versions to examine, each version having detailed SQL scripts describing the individual changes that were made. Of these possible schema versions we chose three interesting versions that had multiple complex schema changes, requiring significant changes to the application source code. We analysed these three changes in detail, comparing the actual changes that were made to the source code with the information obtained using our analysis technique.

We shall now illustrate the interesting results from one of these studies in detail. The remaining two versions indicated very similar results, and are not included here for the sake of brevity.

The schema version change consisted of eight individual changes shown below.

`ChangeSc1`	Added a column to a table
`ChangeSc2`	Added 3 columns to a table
`ChangeSc3`	Altered data type of a column
`ChangeSc4`	Added a new constraint to column
`ChangeSp1`	Added 3 new parameters to a stored proc.
`ChangeSp2`	Added new return columns to a stored proc.
`ChangeSp3`	Added new return columns to a stored proc.
`ChangeSp4`	Added a new parameter in a stored proc.

The actual source code analysed for this version was 78,133 lines of code, across 3 compiled binaries. The schema version consisted of 88 tables with a total of 536 columns and 476 stored procedures. The analysis resulted in the extraction of 896 possible queries.

Measured over three timed executions, the source code program analysis took an average of 62 seconds, whilst the impact calculation took an average of 58 seconds. This was executed on a 2.13Ghz Intel Pentium processor, with 1.5GB of RAM.

We now describe the observed source code changes, giving each observed change an identifier.

	Desc.	**Cause**
`OC1`	Added new parameters	`ChangeSp1`
`OC2`	Dynamic sql UPDATE, added column	`ChangeSc1`
`OC3`	Dynamic sql UPDATE, added column	`ChangeSc1`
`OC4`	New fields read from query results	`ChangeSp2`

There were four change sites where source code was changed as a direct consequence of the schema changes. In each case we indicate a description of what was changed and which schema change was responsible. In the following table we compare the predicted changes with the observed changes.

Change	**Predicted**	**Observed**	**Identified**
`ChangeSc1`	7 warns	4 warns	`OC2`, `OC3`
`ChangeSc2`	5 warns	none	-
`ChangeSc3`	5 warns	none	-
`ChangeSc4`	5 warns	none	-
`ChangeSp1`	1 err	1 err	`OC1`
`ChangeSp2`	3 warns	1 warn	`OC4`
`ChangeSp3`	1 warn	none	-
`ChangeSp4`	none	none	-

We consider an observed change to be successfully predicted if the location that we highlight requires altering as suggested. In the third column we note which observed changes the warnings or errors apply to. In some cases we found multiple warnings or errors that correctly identify one observed change, this is simply because we create an error or warning for each possible execution path, and that the observed change may been present on more than one of these paths.

For `ChangeSc2`, `ChangeSc3` and `ChangeSc2` we see predicted warnings, but no observed changes. This is because many warnings are often false positives. In the case of these three changes, tables have been altered, but these alterations will not directly cause any errors. The warnings highlight the places where the table is accessed by a query, and warn the user that the table alteration has been made and may require action. In many cases no action is required, but in some cases, like for `ChangeSc1`, some of the warnings are acted upon whilst other are not.

It is interesting to note that all changes were indicated by at least one predicted warning or error message created by SUITE. It is also worth noting, that SUITE did not predict any impact for `ChangeSp4`. This is in-line with our expectations because the stored procedure `ChangeSp4` is never called by the application. This an interesting result that shows that our tool is also useful for testing the absence of impact of changes.

We consider the total evaluation time of 2 minutes to be very encouraging for 75 thousand lines of code, especially as our tool is an early prototype with a lot of scope for optimisation.

There are threats to the validity of our evaluation, mainly due to the maturity and accuracy of SUITE. There are gaps in our implementation, however we have identified that most of these can be rectified using proven techniques that have been shown to be computationally viable.

All these observations imply that our results could potentially be useful in a real development environment, however we cannot conclude this solely from a historical case study. We would need to conduct a different study before we being able to make any claims about the accuracy level being useful, or indeed whether the level of false positives is acceptable. Instead, we argue that we have shown our approach to be both feasible and promising, and we leave the evaluation of the usefulness of this approach for future research.

7. SURPRISING RESULTS

The major interesting result of our work is that program analysis of database queries, may not be as simple as it would initially seem. The problem of string analysis for queries has been admirably studied in related work, as discussed in Section 5. However, as illustrated by our motivating example, and confirmed by our case study, we have found that these existing analysis techniques require significant modification to be useful for our purposes. We argue that real world applications cause problems for program analysis which are not addressed by the related work that we have discussed. In particular, the precision of the program analysis needs to be extended in two key areas; precision of context sensitivity and analysis scope.

7.1 Context sensitivity

Our example scenario in Section 2 presented an example architecture for database access components. This example, although trivial, identified a requirement for context sensitive program analysis. This was because the architectural patterns employed result in code where definition and use of queries are spread across numerous different method calls.

In our case study we confirmed this requirement. However, perhaps surprisingly, we noted that the standard 1-CFA analysis was not sufficient for extracting useful results. Our implementation uses a k-CFA algorithm, and whilst this high level of context sensitivity may not always be suitable, we have shown that by limiting the analysis using program slicing techniques, we can compute useful results in reasonable time. For other applications, the exact level of context sensitivity required will vary, dependent upon the number of nested calls present in the architecture of the application. However, it is clear that in many real world architectures, especially where similar architectural patterns are used, a high level of context sensitivity will be required in order to conduct useful impact analysis.

7.2 Analysis scope

Many of the previously mentioned related works, constrain the scope of their analyses to strings. Our example scenario in Section 2, presents the motivation for wanting to extend the scope of the analysis significantly beyond strings. In practice, database interactions require not only the analysis of strings, but the analysis of many data type found in persistence and other libraries.

Our case study confirms that this scope extension is indeed required. However, the extent to which the scope must be widened, and the increased precision of analysis required, is surprising. We identify several problems that our implementation had to address, which require an increase in the scope and complexity of the program analysis.

The first extension we require, is the use of mutable types. In languages such as Java and C#, strings are immutable, therefore cannot be changed once they are created. Alterations occur by creating an entirely new string object. With such immutable objects, side-effects of methods do not always have to be considered. This makes the search space of the program analysis much smaller. If we extend the analysis to other types, that are mutable, we increase the complexity of the analysis. Choi et. al. [5] take into account the state of some mutable type, such as the java StringBuffer object. However, we require the use many other heap types, increasing the cost and complexity of the analysis.

As in Kyung goo-Doh's work, important types for which source code is not available must be given a semantics. Whilst they formally define semantics for strings and StringBuffer types, we informally define semantics for a much larger set of types. For example we consider types from the ADO.NET libraries, custom persistence libraries used in our case study and several types from the .NET class libraries.

It is important to note that this extension means that we do not only consider SQL queries, but also calls to stored procedures and other interactions with the database. The extent of different features that an application uses can vary according to practice, but it is very common to use many of the more advances features provided by the DBMS [1].

7.3 Schema change

Another interesting result that we take from this case study, is that the types of schema change that occurred, agrees with the predictions of a study by Dag Sjoberg [23]. This study indicates that the types of schema change that are most likely are additions and deletions. We do not include the data from our case study that confirms this, however this observation brings about an important question for future research. Given that breaking changes are not popular, why are they not popular? Are schema changes avoided because they are difficult, or are they simply not often required? Our case study would seem to circumstantially indicate that changes are avoided because they are difficult to trace and manage. In order to answer these questions, we would need to conduct an experiment to see whether the presence of good impact analysis does in fact make schema change easier, and whether this would increase the frequency of making schema changes which may cause errors.

8. CONCLUSIONS

We have investigated the problem of using program analysis for providing change impact analysis of database schema changes. We found that current string analysis techniques, whilst useful in other areas, require significant extension to be useful for impact analysis. We have motivated, and provided details of how such extensions can be made. Although these extensions increase the complexity and cost of the analysis, we have found that useful results can be extracted with reasonable cost. We demonstrate this, by applying our analysis to a representative case study of significant size.

Our case study uses many recommended and widely accepted architectural patterns and software engineering practices. We argue that because many similar architectures and techniques are in widespread use, that our results are generaliseable to other similar enterprise applications.

We identify several areas for future research including; the use of other program analysis approaches such as abstract interpretation, increases in accuracy and performance for the SUITE prototype, the guarantee of safely conservative solutions, the use of dynamic analysis where conservative assumptions become expensive or inaccurate, and finally the investigation of how the availability of impact analysis techniques may affect the development of database applications.

Acknowledgments

We are greatful for the guidance we received from Barbara Ryder on the relationship between call graph construction and k-CFA dataflow analysis. We thank Luca Cardelli and Gavin Bierman for enlightening hints on the directions of language integrated native queries in C# and James Higgs and the Interesource team for the provision of our case study. We thank Andy Ayers, Jim Hogg and John Larc for technical help with the Microsoft Phoenix Framework.

9. REFERENCES

[1] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison Wesley, Apr. 2006.

[2] R. Arnold and S. Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[3] M. P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proc. of the* 1^{st} *International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan*, pages 223–240. North-Holland, 1990.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE TSE*, 31(2):137–149, Feb. 2005.

[5] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In N. Kobayashi, editor, *APLAS*, volume 4279 of *LNCS*, pages 374–388. Springer, 2006.

[6] A. S. Christensen, A. Møller, and M. 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.

[7] Y. Cohen and Y. A. Feldman. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM TOSEM*, 12(3):285–316, 2003.

[8] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, Oct 1991.

[10] D. S. Fabrizio Biscotti, Colleen Graham. Market Share: Database Management Systems Software, EMEA, 2005. Technical report, Gartner, June 2006.

[11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Addison Wesley, 1995.

[13] C. Gould, Z. Su, and P. 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.

[14] G. M. Kapfhammer and M. L. 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.

[15] A. Karahasanovic. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD thesis, Dept. of Informatics, University of Oslo, 2002.

[16] L. Larsen and M. J. 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.

[17] 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.

[18] Microsoft Patterns and Practices Developer Center. http://msdn.microsoft.com/practices/, 2007.

[19] Phoenix Framework. http://research.microsoft.com/phoenix, 2007.

[20] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55, New York, NY, USA, 2001. ACM Press.

[21] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2003. IEEE Computer Society.

[22] O. G. Shivers. *Control-flow analysis of higher-order languages or taming lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[23] D. I. Sjoberg. Quantifying schema evolution. *IST*, 35(1):35–44, 1993.

[24] J2EE Patterns. http://java.sun.com/blueprints/patterns/, 2007.

[25] F. Tip. A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, Amsterdam, Netherlands, 1994.

[26] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In D. A. Watt, editor, *Proc. of the* 9^{th} *Int. Conference on Compiler Construction, Berlin*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2000.

[27] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. *SIGPLAN Not.*, 42(1):199–210, 2007.

[28] K. 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.

[29] R. K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, Inc, February 1989.