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 static program analysis techniques for identifying the impact of relational database schema changes upon object-oriented applications. We use dataflow analysis to extract all possible database interactions that an application may make. We then use this information to predict the effects of schema change. We evaluate our approach with a case-study of a commercially available content management system, where we investigated 62 versions of between 70k-127k LoC and a schema size of up to 101 tables and 568 stored procedures. We demonstrate that the program analysis must be more precise, in terms of context-sensitivity than related work. However, increasing the precision of this analysis increases the computational cost. We use program slicing to reduce the size of the program that needs to be analysed. Using this approach, we are able to analyse the case study in under 2 minutes on a standard desktop machine, with no false negatives and a low level of false positives.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; H.2.1 [**Information Systems**]: Logical Design

General Terms

Languages, Verification

1. INTRODUCTION

Databases are widely used to store, query and update large data sets. Database management systems (DBMSs) are designed to address the complex issues that arise when providing persistence of data, such as concurrent access and efficient execution of complex queries over large datasets. It is often cost-effective to use general purpose DBMSs rather than application specific solutions and for this reason, many modern software applications rely on DBMSs.

A *database schema* defines the data structure of a database. If the system requirements change, the database schema may require changes, most commonly requiring additional information and relationships to be stored [27]. As more information is added, and as the schema becomes more complex the level of coupling and dependency between application and database increases. This level of coupling may become problematic when the database schema requires changes.

The effects of database schema change upon applications is currently often estimated manually by application experts [2]. Assessing these effects manually is both fragile and difficult [17], and can be frequently incorrect [20]. Moreover impact analysis from code inspections can be prohibitively expensive [22]. Therefore, in this paper we address the problem of assessing the effects of database schema change in a more reliable and cost-effective way.

Relational DBMSs are currently the most popular type of DBMS [6]. At the same time applications that create, update and query data in such enterprise applications are often written using object-oriented programming languages, such as C++, Java and C#. This results in the so-called impedance mismatch problem, which denotes the significant conceptual gap between object-oriented applications and relational databases. This mismatch complicates impact analyses because many of the techniques used to help overcome the mismatch complicate program analyses.

We present an approach for predicting the impact of relational database schema changes upon object-oriented applications. The main contribution of this paper is the presentation of techniques to extract dependency relationships between applications and database schemas, which are suitably precise for the purpose of impact analysis but still computationally feasible. The precision of our analysis comes from using a recognised k-CFA context-sensitivity [26], but with a higher value of k than used in previous related work. We reduce the computational cost by reducing the size of the original program through program slicing. Our implementation of both program slicing and dataflow analysis is built using Microsoft's Phoenix framework [21]. We then use the results of the dataflow analysis to reason about dependency relationships, and we use CrocoPat [4] for the final impact calculation. We evaluate our approach using a commercial content management system as a case study. We have considered a version history of two years, which had 62 different versions of the schema with corresponding application changes. We have used our implementation to analyse the three versions with the most significant changes. The version we present in this paper has 78 kLOC of C# with 417 ADO.NET invocations each of which may perform multiple queries or call multiple stored procedures. The case study shows that the precision of the dataflow analyses in related work are insufficient and also, that our combination of program slicing with a precise k-CFA

*The research described in this paper is partially supported by Microsoft Research UK under MRL Contract 2005-054.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

dataflow analysis, produces accurate and precise results in under 2 minutes on a standard desktop machine.

The paper is further structured as follows. In Section 2, we give a motivating example and describe the common data access practices for which such analyses are significant. We describe our approach n Section 3 and its implementation in Section 4. In Section 5, we present the results of the case study. We describe related work in Section 6 and we discuss the main findings in Section 7. We conclude the paper in Section 8.

2. MOTIVATING EXAMPLE

Consider a group of scientists, who store experiment 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 Experiments.Name,                    // Q1
       Experiments.ExperimentId
FROM Experiments
WHERE Experiments.Date=?

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

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

SELECT Readings.ReadingId, Readings.Data // 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 the `Experiments.Name` field has not been used and instead that `Experiments.ExperimentId` field was used to give each experiment a unique name. The DBA therefore decides that the `Experiments.Name` column 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 Impact of Schema Change

The schema changes will impact 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 a 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`

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 affected. 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 experiment readings were inserted long after they were read, this value could be wrong. This may or may not be the desired behaviour. To this end, we 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 caused by the addition of a column that may need to be used in a query. For example, `Change 1` adds the `Readings.Date`. 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 new data may need to be returned. `Q4` will execute without error 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 focus of our work is not the reconciliation of the impacts themselves, but rather the difficulty of discovering and predicting them.

2.2 The Difficulty of Schema Change

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

Before the schema change is made: The DBA has to estimate the benefits of each change against the cost of reconciling the existing applications 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 deprecating and versioning schemas [2].

After a change is made: Developers need to locate all affected areas of their application and reconcile them with the changed

schema. Again, without knowing impacted locations, the schema change process may introduce application defects.

If we consider the information required by the DBA and application developers, at these two critical points, we observe that they need the same information, i.e. the location of every statement in the application that is impacted by the change. We aim to provide this information through automated change impact analysis.

2.3 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.

In Figure 1 we show an example of how database access logic can be implemented. The figure shows how `Q1` from our example scenario, might be executed in practice.

```
00 public class Experiment{
01  private int expId;
02  private string name;
04
05  public Experiment(DBRecord rec){
06   expId=rec.GetColInt("Experiments.ExperimentId");
07   name =rec.GetColString("Experiments.Name");
08   }
09
10  public static IEnumerable<Experiment> Q1(DateTime d){
11   DBParams dbParams = new DBParams();
12   DBRecordSet queryResult;
13   List<Experiment> exps = new List<Experiment>();
14
15   dbParams.Add("@ExpDate", d);
16
17   queryResult = QueryRunner.Run(
18    "SELECT Experiments.Name,Experiments.ExperimentId"+
19    " FROM Experiments"+
20    " WHERE Experiments.Date={@ExpDate}",
21    dbParams);
22
23   while (queryResult.MoveNext()){
24    exps.Add(new Experiment(queryResult.Record));
25   }
26
27   return exps;
28  }
29 }
30
31 public class QueryRunner{
32  public static DBRecordSet Run(string query, DBParams p){
33   // pre execution functionality ...
34   DBRecordSet result = Exec(query, p);
35   // post execution functionality ...
36   return result;
37  }
38
39  private static DBRecordSet Exec(string query, DBParams p){
40   return DBConnection.Exec(query, p); // SLICE ME
41  }
42 }
```

Figure 1: C# data architecture for Q1

The entry point in this example is the `Q1` method on Line 10, which returns a collection of `Experiment` objects that match the supplied parameter. This method creates an SQL query as a string, and supplies the missing parameter in the form of a `DBParam` object[1]. The query is used in a call to the `QueryRunner.Run` method on Line 17. Lines 33 and 35 are place holders for what, in practice, might be code to manage connections, transactions or other details that might be required for each query. Line 34 calls the `Exec` method, which finally executes the query against the database on Line 40 and returns the results. On Lines 23-25 the results are stored as a new `Experiment` object for each returned record. Each `Experiment` object is populated by extracting values from the supplied `DBRecord` in the constructor of the `Experiment` class. Please note that we have omitted many details, such as checking the number of returned rows or catching database exceptions. However this example gives a taste of the types of pattern that may occur in real applications.

In this example, all types belonging to the persistence API begin with DB (i.e. `DBConnection`, `DBRecord`, `DBRecordSet` and `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 3.

In Figure 1, the database query is executed in the `QueryRunner` class. If we were to add a new class for the `Readings` 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 common practice. In fact, many recommended architectural patterns for data access have such layers of abstraction in order to create loosely coupled and maintainable programs. Such patterns and the reasons for using them, are well described, examples of which can be found in Chapters 3 and 10-13 by Fowler [10]. Several other similar references also exist, which discuss architectural patterns, and the reasons for their use [11]. We note that such patterns and architectures are in widespread use, and therefore, that any analysis techniques that we develop, should be able to cope with the complexity caused by such layers of indirection and abstraction.

3. APPROACH

Previous work on extracting queries from modern OO languages has been based on *program analysis*. Program analyses are compile-time techniques for approximating the run-time properties of programs. String analysis is a particular form of program analysis where the possible runtime values of string variables are predicted for selected locations in the program.

An example of this is the approach taken by Gould et al [12], in which they use string analysis to predict the values of strings passed to the Java JDBC library methods, in order to check that the queries are type safe with respect to the database schema. The string analysis used was the JSA application created by Christensen et al. [8]. Whilst this string analysis is suitable for many such applications, we have found that it was not precise enough for the purpose of schema change impact analysis. We now describe why impact analysis requires greater precision.

3.1 Requirement for Context-Sensitivity

Context-sensitivity is a measure that specifies how precisely the calling context of procedures are represented in dataflow analyses [25]. *k-CFA* analyses are where all or some of the propagated data in the dataflow analysis include, in their definition, a call string that represents the last *k* calling call-sites [14].

The approach of Christensen et al. is context-insensitive which, whilst being computationally less expensive than the analysis we propose, causes a loss of precision which makes impact analysis difficult, as we describe next.

The data access architectures discussed in Section 2 encourage data access logic to be consolidated in a small number of methods.

[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.

Often, these methods take queries as parameters, and simply execute any query that is supplied. This separation of query definitions and executions, across multiple method calls can cause dataflow analyses to over-approximate the possible queries at query execution sites. This can then lead to an over-approximation of which query results are associated with which query definitions and executions. We need to avoid this over-approximation, as it may lead to a large number of false positives when performing impact analysis. We provide an example of this below.

3.2 Required Precision of Context-Sensitivity

For our example code in Figure 1 a 1-CFA analysis would be sufficient to unambiguously associate the query with it's execution and the use of it's results. However, suppose we added a new class for the `Readings` table. This would result in a class like Figure 2.

```
00 public class Reading {
02   private int readingId;
03   private Byte[] data;
04
05   public Reading(DBRecord rec){
06     readingId = rec.GetColInt("Readings.ReadingId");
07     data = rec.GetColBinary("Readings.Data");
08   }
09
10   public static IEnumerable<Reading> Q4(int expId){
11     DBParams dbParams = new DBParams();
12     DBRecordSet queryResult;
13     List<Reading> rdgs = new List<Reading>();
14
15     dbParams.Add("@ExpId", expId);
16
17     queryResult = QueryRunner.Run(
18       "SELECT Readings.ReadingId, Readings.Data " +
19       "FROM Readings " +
20       "WHERE Readings.ExperimentId={@ExpId}",
21       dbParams);
22
23     while (queryResult.MoveNext()){
24          rdgs.Add(new Reading(queryResult.Record));
25     }
26
27     return rdgs;
28   }
29 }
```

Figure 2: C# data architecture for Q4

This class would also run its queries through the `QueryRunner.Run` method. On Line 40 of Figure 1 a dataflow analysis would estimate the possible runtime values of the `query` parameter. In practice this involves giving the variables in the program, unique identifiers to associate with possible runtime values, calculated by the dataflow analysis. For example, a context-insensitive analysis would use an identifier such as `query` to represent the `query` parameter. This identifier would be the same for each invocation of the method, therefore the analysis would approximate the values of `query` as a combination of the actual parameters provided at the call-sites on Lines 17 of the `Experiment` and `Reading` classes. The analysis would then use this over-approximation of the `query` parameter for every method invocation. This then means that in the constructors of `Experiments` and `Readings`, we would be unable to determine which query resulted in the given `DBRecord` object. This could cause a number of false positives in the later analysis stages.

In contrast, a 2-CFA analysis would create separate identifiers, including context information, such as `[Experiment.cs:17][QueryRunner:34]query` and `[Reading:17][QueryRunner:34]query`. These identifiers are prefixed with a string of the last 2 call-sites, where call-sites are represented by the class name and line number. This allows the dataflow analysis to distinguish between different values of the variables belonging to separate calling contexts. A 2-CFA analysis of our example would be able to unambiguously associate all query uses with the correct definition and execution sites.

A 1-CFA analysis would be able to distinctly recognise the calling context from the previous call site only. This means that at the call to `DBConnection.Exec` we would approximate the query as being either value, as the identifiers would both be `[QueryRunner:34]query`. Again, this would cause the same problems as a context-insensitive analysis in that ambiguous query executions result in the inability to correctly associated query definitions with executions and use of results.

The example we present here is very simple, and the types of architectural patterns used in practice may be far more complex. We therefore argue that 1-CFA is not precise enough for impact analysis, because these conservative approximations can lead to the amount of impacts being overestimated to the point where the analysis is no longer useful. We also note that this requirement for precision changes the trade-offs involved in the selection of program analyses. Increased precision comes at a cost, therefore we are not claiming any improvement over the techniques of previous work on string analysis [8, 12], we are simply identifying that cost-benefit ratios in the case of impact analysis are different, and therefore call for different techniques.

3.3 Approach Overview

We have motivated the need for a precise k-CFA analysis. However, the reason why higher values of k are not routinely used, is that as k increases, k-CFA analysis becomes computationally expensive, especially for large programs. In fact, k-CFA analyses where $k > 0$ have exponential complexity with respects to program size [14]. This presents us with the problem of how to increase the precision of the analysis sufficiently, whilst keeping the computational cost feasible. The way we address this problem, is to reduce the cost of the k-CFA analysis, by only analysing those parts of the program that may interact with the database.

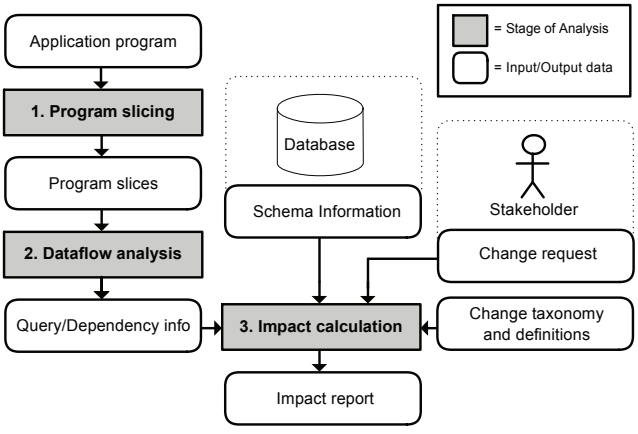

Figure 3: Approach Overview

Our approach can be conceptually divided into three stages, as shown in Figure 3. The first stage isolates the subset of the program that interacts with the database. We then perform a dataflow analysis on this subset in order to find possible queries and schema dependent code. The analysis results are then used to perform the final impact calculation, which is guided by one of the stake-holders as defined in Section 2. We describe these stages next.

3.4 Program Slicing

A program slice contains 'the parts of a program that (potentially) affect the values computed at some point of interest' [28]. By taking a series of program slices where the point of interest[2] is a database call, we can extract a subset of the source application that can affect, or be affected by, these database calls. This subset will potentially be much smaller than the original source program. Thus by running the k-CFA dataflow analysis only over this subset, we can potentially reduce the cost of our approach, whilst maintaining the required level of precision. We will discuss the size of the reduction we have achieved in practice in Section 5.

There are alternatives to program slicing for reducing the cost of the dataflow analysis, as we shall discuss in Section 6. However, program slicing serves as a good illustration of our approach as it is well understood and algorithms exist for many different programming languages and language paradigms. We note that program slicing is by no-means prescriptive and that techniques based on dependency information, similar to program slicing, may provide greater reductions in the size of the program, and we identify this as an important area for future research.

We do not describe slicing algorithms in detail in this paper. We refer interested readers Frank Tip's survey of slicing techniques [28]. We base our prototype implementation on the slicing algorithm proposed by Liang and Harrold [19].

The slicing output will be a subset of the original program. This gives us enough information to simply assemble a subset of the source program, on which we can perform the dataflow analysis.

3.5 Dataflow Analysis

Dataflow analysis is a program analysis technique for computing a possible set of runtime properties that may occur at a given point in a program. In our case we wish to predict the values of any queries that may be executed against the database. In our example scenario in Figure 1 we wish to know the values of the parameters in the call to `DBConnection.Exec` on Line 40.

We base our dataflow analysis on the string analysis in [7]. We do not repeat the description of this algorithm here, and refer interested readers to the paper for more details. Choi et al. describe how their analysis can be applied to languages with strings, heap types and how to handle primitive types, e.g. integers.

While Choi et al.'s algorithm provides a good starting point we need to modify it to render it suitable for the calculation of possible query values in database applications. We perform two key modifications. Choi et al.'s algorithm is 1-CFA and we have argued above that we need k-CFA where k is greater than 1. Secondly Choi et al.'s algorithm only works on strings, whilst we also need to calculate the value of other types which may also represent queries. We use the term *query types* to denote all query representing types, and types that may be involved in the execution and use of database calls. For example, the query types in Figure 1 are `DBConnection`, `DBRecord`, `DBRecordSet`, `DBParams` and the `string` type, because each of these types can either represent a query, or can be directly used to execute or represent the results of a query.

[2]A point of interest is referred to as a *slicing criterion* in the program slicing literature.

3.5.1 Increasing Context-Sensitivity

In order to increase Choi et al's algorithm from 1-CFA to k-CFA we use the techniques described in [14]. This involves modifying the property space of the dataflow analysis, so that abstract variables and abstract heap locations can be distinguished between different calling contexts. This involves extending the identifiers to include a string of the last k call sites, similar to the approach illustrated in Section 3.2. This also requires the altering of transfer functions for calls and call returns, so that formal parameters are identified with call strings as necessary.

3.5.2 Adding Query Types to String Analysis

In order to handle query representing types, we treat all query representing types as strings, or collections of strings. In fact, any modifying methods on these objects can simply be aliases of defined string manipulations such as replace, concat, substring etc. Therefore we simply add all query representing types and operations as aliases of the existing string types and methods.

For query types that do not directly represent queries, such as the `DBRecord`, we alias a generic heap type. Because all of these query types are heap types, we wish to assign them a unique abstract location identifier based on the program location and context where the object was instantiated. Calculating the dataflow of these heap types allows us to associate each individual query type with its uses or redefinitions.

By aliasing the functionality specified for string types in the string analysis we can reuse the formal specification of the analysis. This allows us to maintain the guarantee of termination and other useful properties of the original string analysis.

To conduct the query analysis we perform a standard fixed point iteration of the graph provided from the previous slicing stage. The result of this process, is a dataflow graph where all query representing types have an estimated set of possible runtime values, and all other query type objects (e.g. returned result sets) are associated with unique identifiers based on where they may be instantiated. We now describe how we use this information.

3.5.3 Extracting Dataflow Information

Figure 4 shows an example of the output produced by the dataflow analysis. We store these analysis results using the RSF file format [30]. This example shows the output of the query analysis for the example code in Figure 1. The first line indicates that the execution with the identifier `Exec1` is associated to a code location by the relationship `ExecutedAtLine`. Intuitively, we use this to denote that an execution with the id `Exec1` is executed at Line 40. The next line similarly notes one of the possible queries that might be executed by this query[3].

We extract the information presented here by looking over the dataflow graph for all interesting methods. We define an interesting method as any method which may use or modify a query type in a way that may be significant to the impact calculation that we describe later. This typically includes most of the methods in the persistence libraries that use these query types.

For any method which may execute queries, we store the predicted values of the query, and any auxiliary query types that it may use. For example, in Figure 1 we would find the `DBConnection.Exec` method on Line 40 and we would output the possible values of the query parameter at this point. We also output the location identifiers of any objects used as parameters, such as parameter collections, and also any returned objects. This is shown

[3]The ellipsis in this line denotes that the text has been truncated for display, in the actual output this would contain the full query.

```
ExecutedAtLine Exec1 Example.cs:40
ExecutesQuery  Exec1 "SELECT Experiments.Name, ...
Executes       Exec1 Loc2
ReturnsResult  Exec1 Loc3
UsesParams     Exec1 Loc1

ReadsResultObject Read1 Loc3
ReadsColumn       Read1 Experiments.ExperimentId
ReadAtLine        Read1 Example.cs:06

ReadsResultObject Read2 Loc3
ReadsColumn       Read2 Experiments.Name
ReadAtLine        Read2 Example.cs:07

PrmAddName        PrmAdd1 @ExpDate
PrmAddType        PrmAdd1 DateTime
PrmAddLine        PrmAdd1 Example.cs:15
PrmAddTo          PrmAdd1 Loc1
```

Figure 4: RSF query analysis output

on the first five lines the Figure 4 where `Loc2` is the unique location identifier of the object which we use to uniquely identify the query object.

For methods which do not execute a query, but may still provide valuable impact information, we store any information we can. For our example in Figure 1, on Lines 06 and 07, we can see that a `DBRecord` object is accessed. `DBRecord` is an interesting type involved in a query, therefore we will have included it in our query analysis. The parameter `rec` will be associated with a location resulting from the execution of a query, and because we are using a k-CFA analysis, where *k* is greater than 1, it will be associated with only one query execution. This is shown in the second block of text in Figure 4. The output shows the line number, and uniquely assigned identifier `Read1` used to identify this site. `Loc3` is the object that is the returned result of the query, indicated by the `ReturnsResult` relationship. We see that `Loc3` is also the result object that is read, shown by the `ReadsResultObject` relationship, therefore `Loc3` associates the read site `Read1`, as a read of the results of execution `Exec1`.

This process of adding query types is simple to implement for many different persistence libraries or technologies, and can be extended almost arbitrarily. We omit the details of the libraries for which we have implemented query types. However, in future versions of our prototype, we expect to be able to release a reference implementation, including the ADO.NET libraries.

Collecting all of this information may not always be possible. Reducing the source program to a subset allows us to run a more precise dataflow analysis than previous approaches. Even so, there are many cases where we could lose accuracy, as is typical for static program analysis. However, our intuition is that even if the dataflow analysis is conservative, resulting in several queries that will not occur in practice, by collecting any available information we can still provide enough information to do useful impact calculation. In Section 7 we discuss how useful our analysis was in practice, and where accuracy was lost.

3.6 Impact Calculation

Once we have performed the dataflow analysis, we use the gathered information to predict the possible effects of database schema change. We call this process *impact calculation*.

A good description of the types of impacts that may arise from a given schema change, are described in [2]. This book describes a catalogue of possible changes that can occur in relational database schemas, including 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. As discussed above, we extract information from the dataflow analysis and store it using the RSF file format, as shown in Figure 4. We then use a relational language to reason about the query data. These relational programs are relatively short. We create one program for each impact type we would like to analyse.

```
AffectedReads(x)  :=  ReadsDroppedColumn(x, $1);

FOR AffectedRead IN AffectedReads(x) {
 AffectedReadLines(x):=ReadAtLine(AffectedRead, x);
 PRINT "Read of dropped column at lines:", ENDL;
 PRINT [" "] AffectedReadLines(x);
 PRINT " Returned from queries executed at:", ENDL;

 AffectedQueryLocations(x)
   :=ReadsResultObject(AffectedRead,x);

 AffectedExecutions(x)
   :=EX(y, AffectedQueryLocations(y) &
           ReturnsResult(x, y));

 AffectedExecLines(x)
   :=EX(y, AffectedExecutions(y) &
           ExecutedAtLine(y, x));

 PRINT [" "] AffectedExecLines(x);
}
```

Figure 5: Example RML program

To execute our impact calculation programs we use CrocoPat [4]. This tool allows efficient execution of relational programs against arbitrary relational data. The tool uses the RSF, and RML file formats for specifying input data and relational programs respectively. We have chosen to use CrocoPat for several reasons. It allows us to easily change and alter the input data and relational programs. We can express powerful relational programs in relational manipulation language (RML). RML is a powerful language based on first-order predicate calculus, which gives us the power to perform complex reasoning over the information we have extracted. Finally, due to its BDD based implementation, CrocoPat is more efficient than other relational language processors.

Figure 5 shows an example RML program. The program takes our sample RSF file, and an arbitrary string column name as input. Change 3, in our example in Section 2 was the dropping of column `Experiments.Name`, so for this change the parameter would be the string `Experiments.Name`. In this example we search for any accesses of `DBRecord` for the column name that has been removed.

The first line of the program creates a set called `AffectedReads`. This set consists of all query reads from `DBRecord` where the specified column name is used; the parameter is referenced by the variable `$1`. We then iterate over each of these affected reads, and print out the location of the read, and the location of the query that returned the result set.

Analysing our example with this RML program results in:

```
Read of dropped column at the following lines:
  Example.cs:07
    Returned from queries executed at:
      Example.cs:40
```

This catches an instance of `err2` from our example in Section 2. This example just shows one very short RML program for illustrative purposes. In practice, 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. This is a very flexible approach, where the set of programs can easily be extended to include arbitrary warnings or error messages, allowing us to tailor our approach to various DBMS features such as views, triggers or constraints. It also allows us to customise the impact calculation for different persistence libraries or technologies. However, it should be noted that we expect to integrate a library of impact calculations programs into our implementation, for all common changes, such as the catalogue of changes provided by Ambler and Sadalage [2]. This would allow users to easily run impact calculation for most common changes, as well as being able to run arbitrary impact calculation of their own as required.

4. IMPLEMENTATION

We have developed a prototype system that we call SUITE (Schema Update Impact Tool Environment). The architecture of SUITE is shown in Figure 6. It shows SUITE's key constituent components and the possible interactions between them.

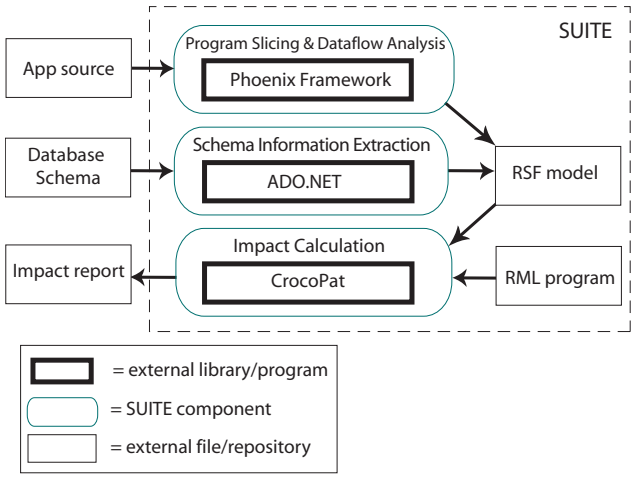

Figure 6: Tool Architecture

As described above, we use the RSF file format [30] to store the results of the dataflow analysis, we write impact calculation programs in RML, and we use the CrocoPat [4] tool to execute these programs. 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 have implemented a slicing algorithm and dataflow analysis on top of the Microsoft Phoenix framework [21], and extract the schema information using ADO.NET. We note that Phoenix provided considerable support for analysis of compiled .NET binaries, and the conversion to a static single assignment (SSA) form, upon which we build the implementation of the program slicing and dataflow analysis. The total size of the SUITE implementation to date is 19 KLOC written in C#.

5. EVALUATION

In order to evaluate the feasibility of our technique, we have conducted an in-depth case study. Our subject application is the irPublish™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 [11] and Fowler [10]. It is also important to note that irPublish has been developed using established software engineering practices such as 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 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, each having detailed SQL DDL 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 change made the eight modifications 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 LoC, across 3 compiled binaries, with 417 ADO.NET invocations sites, each of which could execute multiple possible queries. The schema version consisted of 88 tables with a total of 536 columns and 476 stored procedures.

Measured over three timed executions, the source code program analysis took an average of 1 minute 18 seconds, whilst impact calculation took an average of 21 seconds, this gives an average execution time of 1 minute 39 seconds. This was executed on a 2.13Ghz Intel Pentium processor, with 1.5GB of RAM. The program slicing resulted in a reduction of the program from 191173 instructions to 70050 instructions; this is a reduction to 37% of the original program size. The value of k=2 was used for the dataflow analysis, which was the minimum value required to catch all impacts for the case study. However, there were places in the application where the value of k would need to be up to 7.

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	True positives	False positives	Identified
ChangeSc1	5 warns	2	3	OC2, OC3
ChangeSc2	4 warns	0	4	-
ChangeSc3	4 warns	0	4	-
ChangeSc4	4 warns	0	4	-
ChangeSp1	1 err	1	0	OC1
ChangeSp2	1 warns	1	0	OC4
ChangeSp3	1 warn	0	1	-
ChangeSp4	none	0	0	-

We consider a predicted impact to be a true positive if the error or warning that we highlight requires altering as suggested. We consider a false positive to be a warning or error that was identified but not acted upon. A false negative would be a change that was made, but not predicted. A true negative would be correctly identifying a location as being unaffected, or requiring no alteration.

We omit false negatives from the table for the sake of brevity, but note that there were no false negatives. We omit true negatives from the table as they are difficult to calculate by hand. In the fourth column we note which observed changes the true positives apply to.

For ChangeSc2, ChangeSc3 and ChangeSc4 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 true positives.

We note that the case study only shows false positive warnings and no false positive errors. We consider our analysis to be more accurate for predicting errors than warnings.

It is interesting to note that all changes were indicated by at least one predicted warning or error message created by SUITE, as there were no false negatives. 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 is an example of a true negative.

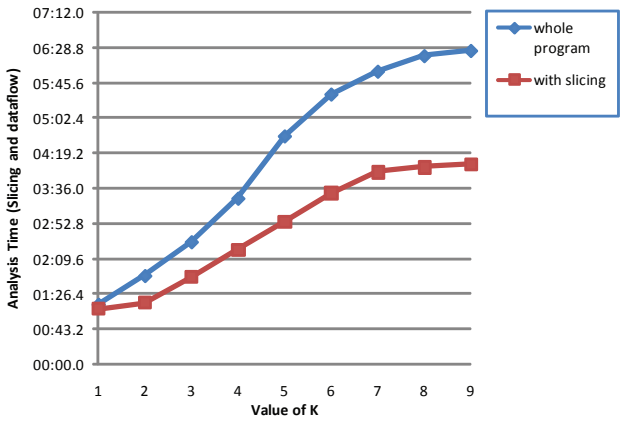

Figure 7: Analysis times for values of k

The graph in Figure 7 shows the execution times of the analysis as the context-sensitivity is increased. We can clearly see that in our case study, our approach provides a speedup over analysis of the whole program.

This graph shows that for our particular case study, the use of slicing and dataflow decreases the overall cost of the analysis. The cost of the dataflow analysis dominates the cost of the analysis. For the points plotted on this graph, the cost of slicing was between 56% for k=1 and 16% for k=9. This shows that our combined slicing and dataflow approach, provides an overall reduction in the execution time of the analysis because the time to conduct the program slicing, is less than the time saved by running the dataflow analysis over the reduced subset of the program. It is also interesting to note that above k=5, the gradient decreases. This is because many dataflow paths in the program have already reached a fixed point. Paths with very deep call depth or recursion, continue to increase the cost above k=5 but their effect on the cost of the analysis has less effect as k is increased.

Slicing may also be expected to be more scalable than the dataflow analysis, as the dominant cost of many program slicing algorithms is the computation of summary edges, which has been shown to have polynomial complexity with respect to program size [24], whilst context-sensitive dataflow analysis has exponential complexity. We aim to study the scalability of our approach in future work, but omit results here as our current prototype is not mature enough to be able to draw reliable results for large programs.

There may be substantial opportunities to improve the speed of the slicing algorithms we are using, as shown by the timing statistics reported by Binkley and Harmann [5]. However, we predict that such improvements may not be possible for the dataflow analysis. This is because the dataflow analysis we describe is not representable as an efficient bit-vector problem [1], therefore to provide a significant improvement may require creating an efficient data structure for the representation of the dataflow information. This may be even more difficult if we were to include language features such as by-reference parameter passing that will increase the complexity of the dataflow property space. At present, we are unaware of any ways to provide a more efficient data structure, and therefore expect the dataflow analysis to remain the dominant cost in our approach.

All these observations imply that our results could potentially be useful in a real development environment, however we cannot conclude this solely from our current evaluation. We claim that we have shown our approach to be both feasible and promising, and we shall conduct further evaluation of the usefulness, accuracy and scalability of this approach in future work.

6. RELATED WORK

There is a great deal of work related to software change impact analysis [3, 23, 17]. However, we are only aware of one similar project that focussed on impact analysis of database schemas [16]. 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 different problem, however we take inspiration from the approach 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 the string analysis of Christensen et al. and secondly, the dynamic query type checker of Gould et al. [12],

both of which we discussed in Section 3. We also discussed the string analysis of Choi et al. [7], the basis of our own query analysis.

Although we have only discussed k-CFA as a way of implementing a context-sensitive dataflow analysis, there are alternatives [18]. Such alternatives need to make sure that they also provide the ability to associate query definitions, executions and the use of query results across multiple method calls. We suspect that such alternatives will prove to be very similar to k-CFA in many respects, and we identify this as an area for future work.

We argued that a context-sensitive dataflow analysis is expensive, and therefore we have used program slicing to reduce the amount of dataflow analysis. There are other options to reduce the cost of k-CFA analysis such as demand driven interprocedural dataflow analysis [13]. However, such techniques are comparable to slicing, producing very similar results. We chose slicing as it is a well-understood concept that can be applied to many different programming languages. However, as discussed slicing is not prescriptive, and we plan to investigate other forms of dependency analysis in future work.

Some related work has been made in analysing transparent persistence [29] using program analysis, although the focus here is on providing optimisation of queries. This work bears similarity to work on extracting queries from legacy applications [9], however 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 embedded persistence technologies could be incorporated into our approach in the future, and insights into formalisation of such techniques.

There has also been some related work produced by the testing community [15]. 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.

The database community has the concept of *schema evolution* which we argue is orthogonal to our approach. We are not aware of any work from this community which directly measures the impact of a schema change upon applications, using techniques such as dependency analysis. However, we expect that our techniques can be used to help inform schema evolution approaches, and we fully expect schema evolution techniques to be just as applicable for managing change as before.

Ultimately the approach outlined in this paper does not aim to replace any existing techniques for database change, but only help to extract tacit information to aid these techniques.

7. 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. However, as illustrated by our motivating example, and confirmed by our case study, we have found that these existing techniques require increased precision in order to be useful for our purposes, which comes at an additional computational cost. We claim that this cost can be reduced by using a dependency analysis such as program slicing.

7.1 Context-Sensitivity

Our example scenario in Section 2 presented an example architecture for database access components. This example identified a requirement for context-sensitive program analysis. This was because the architectural patterns used, result in code where query definitions, query executions and the use of queries results are spread across multiple 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. 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.

We claim that this increase in context-sensitivity can be achieved at a reasonable cost, by using program slicing or similar dependency analysis to create a subset of the program, and only analysing this reduced subset. Our case study shows that this approach seems promising, and can be applied to real applications.

7.2 Threats to Validity

The major threat to validity of this work is the maturity and correctness of our prototype implementation. We see no problems with the presentation of our general approach of dataflow analysis and impact calculation. However, there are some inaccuracies in the implementation of our tool, notably the program slicing algorithms, e.g. the handling of instance variables and static variables, as well as some object oriented features such as inheritance. We are not aware of any work we could have used to corroborate our results, other than the work on slice sizes in C and C++ programs [5].

We present this work despite these drawbacks, as manual inspection and testing indicate that our results are valid, as problems caused by slicing inaccuracies occur rarely in our current case study. However, in a more mature implementation slice sizes and cost could increase. We would still expect the dataflow analysis to be the dominant cost of this approach, as discussed in Section 5. Therefore, even a moderate reduction in program size produced by program slicing or similar analysis, may still provide an increase in performance of the analysis as a whole. We also plan to develop techniques that perform much better than our slicing approach, in both reduction size and computational cost, making our approach still valid, and worthy of continued investigation.

7.3 Schema Change

Another interesting result, is that the types of schema change that occurred, agrees with the predictions of a study by Dag Sjoberg [27]. 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? 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 analysing the impact of database schema changes upon object-oriented applications. We found that current string analysis techniques, whilst useful in other areas, require an increase in context-sensitivity in order to be useful for impact analysis. This increase in precision could have a vastly increased computational cost. To counter this, we reduce the parts of

the program to which the analysis is applied with program slicing. We have demonstrated the feasibility of this approach by applying our analysis to a case study of significant size.

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

In future research we hope to investigate alternatives to program slicing for reducing the cost of the dataflow analysis, and how the availability of impact analysis techniques may affect the development of database applications.

Acknowledgements

We thank Microsoft Research UK for the generous funding of this research, Lori Clarke for discussions on dataflow analysis, Barbara Ryder for discussion of context sensitive analysis, Luca Cardelli and Gavin Bierman for discussion of LINQ, James Higgs and the Interesource team for the provision of our case study, Andy Ayers, Jim Hogg and John Larc for help with the Microsoft Phoenix Framework, 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 discussion on program slicing, Alexander Egyed for discussion of traceability and the anonymous reviewers for their comments.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilers – Principles, Techniques and Tools*. Addison Wesley, 1986.

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

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

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

[5] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM TOSEM*, 16(2):8, 2007.

[6] F. Biscotti, C. Graham, and D. Sommer. Market Share: Database Management Systems Software, EMEA, 2005. Technical report, Gartner, June 2006.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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. IEEE Computer Society, 2004.

[13] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 20(4):104–115, 1995.

[14] S. Jagannathan and S. 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.

[15] 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.

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

[17] J. Law and G. Rothermel. Whole program Path-Based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[18] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *15th Int. Conference on Compiler Construction*, volume 3923 of *LNCS*, pages 47–64, Vienna, 2006. Springer.

[19] D. Liang and M. Harrold. Slicing objects using system dependence graphs. *icsm*, 00:358, 1998.

[20] M. Lindval and K. Sandahl. How well do experienced software developers predict software change? *Journal of Systems and Software*, 43:19–27(9), October 1998.

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

[22] S. L. Pfleeger. *Software engineering: theory and practice*. Prentice-Hall, Inc., 1998.

[23] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. of the* 19^{th} *Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*, pages 432–448. ACM Press, 2004.

[24] T. Reps, S. Horwitz, M. Sagiv, and G. 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.

[25] B. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. *Lecture Notes of Computer Science*, 137:126–137, 2003.

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

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

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

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

[30] 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.