UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note
RN/06/13

Impact Analysis of Relational Schema Changes on Native Language Queries

April 2006

Andy Maule

Wolfgang Emmerich

David S. Rosenblum

Abstract

We present a technique for analyzing the impact that relational schema changes have on applications that use object-relational mappings and native language queries. We present a meta model that identifies the data that needs to be obtained using static analysis from database, object-relational mapping and object-oriented application programs. We discuss a number of static analysis algorithms that we use to extract potentially relevant data from these sources. We show how to specify schema impact using the Object Constraint Language. We discuss an implementation of these techniques in the SUITE environment. SUITE relies on the Soot framework for static analysis and CrocoPat for efficient execution of relational programs. We evaluate both quality and performance of the SUITE environment using versions of a database schema and an application from the bio-informatics domain.

Impact Analysis of Relational Schema Changes on Native Language Queries*

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 present a technique for analyzing the impact that relational schema changes have on applications that use object-relational mappings and native language queries. We present a meta model that identifies the data that needs to be obtained using static analysis from database, object-relational mapping and object-oriented application programs. We discuss a number of static analysis algorithms that we use to extract potentially relevant data from these sources. We show how to specify schema impact using the Object Constraint Language. We discuss an implementation of these techniques in the SUITE environment. SUITE relies on the Soot framework for static analysis and CrocoPat for efficient execution of relational programs. We evaluate both quality and performance of the SUITE environment using versions of a database schema and an application from the bio-informatics domain.

1. INTRODUCTION

Although commercial implementations of object-oriented and deductive data models are available, the vast majority of databases in practical use are relational. As a result, software engineers need to bridge the impedance mismatch between applications that are now mostly written in object-oriented programming languages and relational database schemas. Software engineers have been using *call level interfaces* (CLIs), such as Java Data Base Connectivity (JDBC) or Open Data Base Connectivity (ODBC), to embed queries and updates into application programs. Using these approaches, queries are either statically or dynamically composed as strings in the programming language and are then passed to the call level interface, which interprets and executes the queries and updates. CLIs cause a number of problems, which have been studied extensively in related work [5, 13, 20, 7]. Queries are only compiled by the CLI at run-time, and defective queries are not detected prior to execution of the program.

Preventing these problems using static analysis techniques has been addressed in recent work [9]. The programming language research community has pursued different approaches to solve the problem. Firstly, *object-relational mappings* (ORMs) have been developed and are becoming increasingly popular. These mappings specify the binding between relational databases and object-oriented programs, with examples being Hibernate [13], Java Data Objects (JDO) [20] or container managed persistence in the last Enterprise Java Bean Specification [7]. These ORMs support the generation of data access layers where an object model is manipulated that serves as an object-oriented representation of items of interest in the database schema. This object model is manipulated by means of the ORM API, which in turn creates the required queries to be run against the database. The object model used by the ORM is often derived automatically from relational schemas or external configurations so that should the schema change the data access layer can be simply regenerated.

There is still a need for developers to be able to execute arbitrary queries against the ORM. Many ORMs have some form of string based query language by which these queries can be specified, such as JDOQL for JDO, and HQL for Hibernate. Because these queries are specified as embedded strings, they suffer from many of the same problems identified for CLIs. *Native language queries* [6] support the dynamic definition and execution of statically typed queries with ORMs in terms of programming language primitives. One such native language query mechanism was developed for C-ω by Bierman et al. [3]. This work heavily influenced the next version of Microsoft's .Net Framework, which is scheduled to include the native language query technology LINQ [16] that will allow support for native query language capabilities for C# and VB.NET. An open source project referred to as *Plain Old Java Queries* (POJQ), which was strongly influenced by [5], provides a comparable mechanism for Java and JDO. Given the attention the Microsoft and Sun are paying to ORMs and Microsoft's inclusion of native language queries we postulate that they will gain increased use in years to come.

We address a follow-on question, which is motivated by the use of databases as an integration mechanism. In enterprise settings, a large number of applications are written against the same database. In these settings, the database and its schema are often under the control of a different

*This research is partially funded by Microsoft Research UK under MRL Contract 2005-054. David Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

team from those that build the applications. Likewise in scientific computing, for example in bio-informatics genome and proteomics databases are compiled by institutions, such as the Sanger Centre or the European Bioinformatics Institute (EBI), and a multitude of scientists across the world develop applications against these schemas. In these settings, definition and use of database schemas span organisational boundaries. Moreover, databases and their schemas are long-lived and need to evolve to meet new requirements or record novel scientific discoveries. In such settings many engineering practices for dealing with inconsistencies become unfeasible when development teams have little or no influence over each other's methods. The question then arises as to what impact schema changes have on applications, and how these changes can be dealt with.

The main contribution of this paper is the presentation of a method for determining the impact of schema changes on applications that use ORMs and native language queries. We discuss how we have implemented the method using off-the-shelf static analysis and program comprehension components. We use the version histories of the schema and an application of the Ensembl [11] genome database in order to evaluate our approach. For a number of changes to the Ensembl schema we use our method and its implementation in SUITE (Schema Update Impact Tool Environment) to predict change impact and we compare our predictions against the application's version history.

This paper is further structured as follows. Section 2 presents an example that we use to further motivate and illustrate our approach. In Section 3, we present our approach to predicting the impact of schema changes in detail. We then show in Section 4 how we have used the Soot framework [22], RSF [23] and CrocoPat [2], a BDD-based efficient relational program interpreter, to implement our approach. We present the results of applying SUITE to the Ensembl schema and the BioJava application in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2. A MOTIVATING EXAMPLE

We now present a running example that we use to illustrate our approach to schema impact analysis. Consider a group of scientists who are conducting experiments and are storing the resulting data in a database that has the schema shown below. 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 dynamically composed queries and updates. In these queries and updates '?' represents parameters supplied at runtime.

```
// Query Q1
SELECT *
FROM Experiments
WHERE Experiments.Date=?

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

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

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). This leads to the following new version of the schema.

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.

At this point the DBA has no information about the impact these changes might have. As there are often different ways a schema can be changed to have the same desired effect, the DBA would need to perform change impact analysis to decide which changes are preferable. It is quite reasonable to assume that the DBA has access to the code of applications, even if they are written by a different organisation. So the DBA would need an analysis tool that tells them which queries in the application source are affected by any changes they would like to perform.

Once the schema change has been made, application developers must reconcile their application with the updated schema. This is done by locating all queries that may be affected, and correcting them and their uses accordingly. In our example all three queries are affected and the schema changes result in the following six problems:

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`

If queries are embedded as strings in JDBC or ODBC calls, the developer does not have automated assistance for changing the queries and updates. In this paper, we assume the developer is using statically typed queries in POJQ and JDO, which enables the Java type checker to type check queries. The example queries are given Figure 1.

```
01 // POJQ predicate definition
02 class Q1Pred extends Predicate<Experiments>{
03   private Date date;
04
05   public Q1Pred(Date date){
06     this.date = date;
07   }
08
09   public boolean match(Experiments exp){
10      return exp.getDate().equals(date);
11   }
12 }
13
14 // Query Q1
15 Predicate<Experiments> pred=new Q1Pred(aDate);
16 Query query = JDOQLHelper.newQuery(pm, pred);
17 Collection results=(Collection)query.execute();
18
19 // Update Q2
20 Experiments exp = new Experiments();
21 exp.setExperiment_id("EXP001");
22 exp.setDate(todaysDate);
23 exp.setName("Experiment 1");
24 exp.setDescription("Desc...");
25 pm.makePersistent(exp);
26
27 // Update Q3
28 Readings reading  = new Readings();
29 reading.setReading_id(1);
30 reading.setExperiment_id("EXP001");
31 reading.setData(byteData);
32 pm.makePersistent(reading);
```

Figure 1: POJQ Queries and JDO Updates

Type checking the POJQ queries using the Java compiler will detect problems err1–err4. As inserts are handled by JDO and JDO is unaware of the additional constraints imposed on the schema, errors err5 and err6 will go undetected by the type checker. The static analysis approach of Gould et al. [9] does not address inserts and updates, so it would only detect err1 and err2. If Gould et al. had implemented their static analysis approach for inserts and updates their technique could detect err3 and err4. However, it is unclear how their approach would be extended to cover constraints.

`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 where to 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. To help avoid this kind of situation, it would be helpful if the application developer was advised of any changes to the schema that could have this kind of subtle underlying effect.

Errors err1–err6 only highlight some of the many possible impacts that can occur as the result of a schema change, and those simple error descriptions are missing a large amount of information that the developer must infer themselves. For example, the developer can make a simple name change in the ORM to solve errors err1 and err3. If the developer knew the semantics of the schema change that caused this error, and they were given some advice on remedial action, they could make the required alterations far more quickly.

Moreover, Change 1 adds the `Readings.Date` field. This will not affect the validity of Q1, 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. Q1 will run without any errors occurring but will not return all of the required data. Alerting the application developer to any queries in the application that have new closely associated schema elements may be useful.

Given the above example and shortcomings of static type checking techniques, the goal of this paper is to present a change impact analysis technique for database schema changes that retrieves information to better inform the change process. Firstly, we want to support the DBA to predict the effect of schema changes in order to better inform the choice between alternative changes. Secondly, we inform the application developer of all queries and updates that will be invalidated by a schema change. Thirdly, we will give detailed diagnosis for each of the affected queries and suggest remedial actions. Fourthly, we will inform the application developer about situations that affect the underlying query semantics without affecting their syntactic and static semantic validity. Finally, we will inform the application developer about new schema elements that might have to be added to existing queries.

3. APPROACH

Each different database vendor supports a slightly different dialect of the SQL data definition language (DDL) and data constraint languages (DCL) for specifying relational database schemas. Likewise, the main ORM frameworks, such as Hibernate, JDO and TopLink all take a slightly different approach to defining mappings between object-oriented programming languages and relational schemas. Finally, native language query frameworks, such as POJQ and LINQ differ slightly in the way they define queries. Unfortunately each of those differences will influence the way schema impact analyses are performed. It will therefore not be generally possible to devise a one-size-fits-all approach for the impact analysis of schema change to all different databases, ORM frameworks and native query languages.

Instead, it will be necessary to adapt the impact analysis approach to the particular combination of DDL and DCL dialects, object-relational persistence and native language query mechanism employed. To cater for such adaptation, we use MOF-based meta modelling to define schema impact analyses. We illustrate this approach for MySQL, JDO and POJQ using the most commonly found concepts in DDLs, ORMs and native language queries.

We perform our schema impact analysis in two passes. The first pass extracts all relevant data from the database schema, from the object relational mapping and from the application program with embedded native language queries. The second pass then performs a detailed analysis of these data in order to calculate the precise impact of changes.

3.1 Entities/Relationships for Impact Analysis

Figure 2 shows the MOF meta model of entities and their relationships that we extract during the first pass of the analysis. The model serves two purposes. Firstly, it specifies the data we need to extract from database schemas, ORMs and native language queries included in database applications. Secondly, it serves as the data declaration for the OCL definitions that we use to precisely specify the impact analyses.

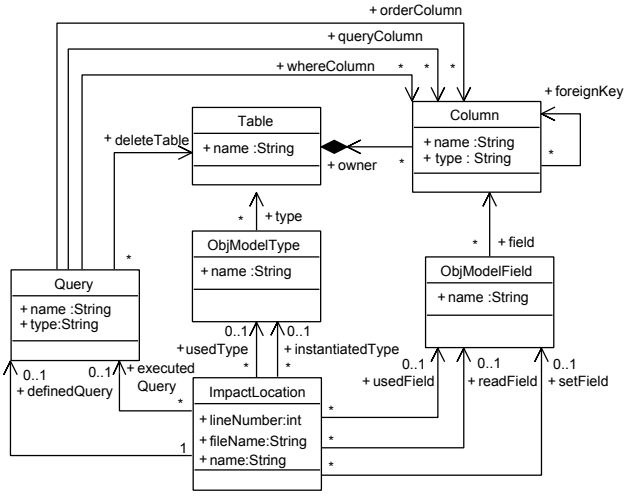

Figure 2: MOF Model for Impact Analysis

For the research reported in this paper, we have considered the most common kinds of queries encountered in database applications, namely *select*, *update*, *insert* and *delete* queries. Therefore the meta model only incorporates the entities and relationships needed to assess the impact of schema changes on these kinds of queries.

The tables and columns defined in a schema are represented by classes `Table` and `Column`, respectively. `Table` objects consist of multiple `Column` objects. Each `Column` object has an attribute `name` that stores the column's name and an attribute `type` that stores the column's type. These are the only concepts from the schema that are presently required by our model.

The queries that are executed in the application are represented in our model by `Query` objects. Each `Query` object has a `type` attribute; its value records what sort of query it is and is either ``SELECT'', ``INSERT'', ``UPDATE'' or ``DELETE''. Queries reference schema columns in three different ways that are represented in our model as follows. Firstly, the `queryColumn` role represents columns used in the body of a query, such as the columns to be returned by a `SELECT` query or the columns to be updated by an `INSERT` query. Secondly, the `whereColumn` role represents columns in the query's `WHERE` clause. Thirdly, the `orderColumn` role represents columns present in the query's `ORDER BY` clause. Similarly, `DELETE` queries can reference a table from which to delete data, and this relationship is specified by the `deleteTable` role.

Objects of class `ImpactLocation` represent locations in the source code that could possibly be affected by a schema change. Impact location objects are present for all locations in the source code where a query is executed as well as the single location where a query is defined. However, definition and execution locations of queries are not the only impact locations we consider. We are considering applications that use some form of ORM, and we are interested in identifying any points in the application where the object model representation of the schema is used. We therefore also store any location that signals instantiation or use of an object representation of a table (represented by `ObjModelType`). The `ObjectModelType` itself is associated with a single `Table` object. This association represents the relationship between tables and classes defined by the object-relational mapping. Finally, we also store the impact locations of any uses of the fields of the object model types that represent columns in the schema. Any `ImpactLocation` that reads, sets or otherwise uses attributes/fields will have respective associations with the `ObjectModelField` in question.

The model described here is a meta model capable of describing all database queries that can be executed by the application, and all elements of a schema and the object-relational mapping that we are interested in analysing. The model defines relationships between all these entities, and provides a means by which we can determine change impact. In effect, we use this model to trace dependencies of statements in the source code via the ORM to declarations in the schema. We shall next describe the static analysis algorithms that create instances of the meta model.

3.2 Static Analysis Algorithms

We must obtain data to instantiate the meta model from three different sources: the database schema, the object relational mapping and the Java application that has embedded native language queries. The application analysis depends on the ORM analysis and that, in turn, depends upon the schema analysis. We discuss the algorithms in the order in which they are applied to populate the meta model.

Algorithm 3.1 *Schema Analysis*

foreach *table stbl* **in** *schema* **do**
  tbl = **new** `Table`*(stbl's name)*
  foreach *column scol* **in** *stbl's columns* **do**
    col = *new* `Column`*(scol's name, scol's type)*
    create `owner` *link between col and tbl*
  enddo
  foreach *relationship rlt* **in** *schema* **do**
    pk = `Column` *that matches rlt's primary key*
    fk = `Column` *that matches rlt's foreign key*
    create `foreignKey` *link between pk and fk*
  enddo
enddo

Algorithm 3.1 defines the static analysis of the schema. It determines creation of instances of entities `Table` and `Column` based on traversal through the schema. It creates a `Table` object for each table in the schema and a `Column` object for each column of any table. Once those objects have been determined the algorithm creates links between `Column` objects to reflect foreign key constraints.

Algorithm 3.2 *Object-Relational Mapping Analysis*

```
foreach class cls in ORM mapping do
    omt = new ObjModelType(cls's name)
    tbl = Table object that matches cls's table
    create type link between omt and tbl
    foreach field fld in cls's field do
        omf = new ObjModelField(fld's name)
        col = Column objects that matches fld's column
        create field link between omf and col
    enddo
enddo
```

Algorithm 3.2 determines the creation of instances of `ObjModelType` and `ObjModelField` based on the declarations given in the JDO object-relational mapping. The algorithm iterates over all classes identified in the mapping and creates an instance of `ObjModelType` for each of them. It then creates a link between the `ObjModelType` and the `Table` object that represents the table in the schema that is represented by the class. The algorithm then does the same for each field of the class and links it to the `Column` object that the field represents in the object-relational mapping.

Algorithm 3.3 *Java Analysis*

```
foreach statement stmt in application do
    if (stmt instantiates any ObjModelType o) then
        i=new ImpactLocation(file name, line no)
        create instantiatedType link between i and o
    elsif (stmt calls method of any ObjModelType o) then
        i=new ImpactLocation(file name, line no)
        if (method reads any ObjModelField o) then
            create readField link between i and o
        elsif (method sets an ObjModelField) then
            create setField link between i and o
        else
            create usedType link between i and o
        endif
    elsif (stmt instantiates a subclass of Predicate) then
        i=new ImpactLocation(file name, line no)
        q=new Query(''SELECT'')
        determine semantics of query q
        create definedQuery link between i to q
    elsif (stmt calls a JDO method) then
        i=new ImpactLocation(file name, line no)
        arg=argument supplied to the JDO method
        if (stmt calls an update object method) then
            q=new Query(''UPDATE'')
            find all places where arg was updated
            Create definedQuery link between q and i
            create executedQuery link between i and q
        elsif (stmt calls a delete method) then
            q=new Query(''DELETE'')
            deltbl=find table associated with arg
            create deleteTable link between q and i
            create executedQuery link between q and i
            create definedQuery link between q and i
        elsif (stmt calls an insert object method) then
            q=new Query(''INSERT'')
            find all places where arg was set
            update q with any columns that will be inserted
            create executedQuery link between q and i
            create definedQuery link between q and i
        endif
    elsif (stmt uses any ObjModelType o) then
        i=new ImpactLocation(file name, line no)
        create usedType link between i and o
    elsif (stmt uses any ObjModelField f) then
        i=new ImpactLocation(file name, line no)
        create usedField link between i and f
    endif
enddo
```

The most complex static analysis is described in Algorithm 3.3 and is performed in order to extract the information about creation and execution of queries, and the use of query results. The algorithm, at least conceptually, examines every statement of the application[1]. The algorithm then investigates whether the statement performs any instantiation, invocations or field accesses of any class included in the object relational mapping, which would be represented as an object of `ObjectModelType` or `ObjectModelField`. If it does, the algorithm records this by creating a new `ImpactLocation` object and creating respective links to the objects representing the ORM entities. Next the algorithm considers whether the statement performs a native language query. If that is the case, it records the impact location, creates a new `Query` object, sets the type of that query to `Select` and creates a `definedQuery` link between query and impact location objects. If the statement calls the JDO library to perform an insertion, update or deletion operation, the algorithm also creates both an `ImpactLocation` and a `Query` object, determines the query type according to the JDO operation called and records the relationships between `ImpactLocation`, `Query`, `ObjectModelType` and `ObjectModelField` accordingly.

The complexity of Algorithm 3.1 is $O(t \times c)$ with t being the number of tables and c being the maximum number of columns per table. When using our example scenario Algorithm 3.2 has exactly the same complexity, as our object-relational mappings have exactly one class per table and one field per column. In practice the amount of classes and fields specified by an ORM varies, but is rarely in excess of one class per table and one field per column. The worst case complexity of Algorithm 3.3 is $O(s^2)$ with s being the total number of program statements. Once all three algorithms have completed, we have all the raw data needed to perform change impact analyses.

3.3 Impact Calculation

The delta between consecutive versions of a database schema can be expressed as a sequence of *elementary changes*, with each elementary change produced by some primitive database operation. In practice, DBAs frequently use elementary changes from a catalogue of some 70 well understood database refactoring patterns presented in [1]. These patterns include elementary changes such as add column, rename column, drop table and so forth.

The approach we take in this paper is to determine the impact that each of these elementary changes has on application programs that have embedded native language queries. For each elementary change, we define an OCL function that calculates the set of `ImpactLocation` objects based on the other meta model entities and their relationships.

[1]In practice our implementation performs a number of optimizations that we omit here for clarity of exposure.

We now describe those OCL functions that will identify the impact in our motivating example. The examples were created and verified using the UCL-MDA tools described in [21]. Each of these functions takes as parameter the set of all `ImpactLocation` objects and returns the subset that will be affected by the change. These OCL functions illustrate the finding of impacts in `SELECT` and `INSERT` queries only; these are the only types of query considered by our example, and we have omitted the functionality required by other query types for brevity. The functions are classified into those that calculate *warnings* about locations and those that calculate locations that will cause *errors* following the change.

For Change 1 in our example scenario we are adding the `Readings.Date` column to the schema. Because this column is new, none of the existing `SELECT` queries in the application will directly reference it. Despite there being no inconsistent column references in the code, there may be places where this column needs to be added to the application. The application developer needs to check all queries that reference the table to which the column is being added. Each of these possibly affected queries needs to be checked to make sure that the newly inserted column is added to the application wherever required.

The OCL function requires a parameter to specify the table to which the column is being added. The function will return all `ImpactLocations` having `SELECT` queries where any of the used columns belong to the supplied table parameter. The function also returns any impact locations where any object model that maps to the table parameter is used or instantiated. This function calculates code locations where the new column may need to be referenced that will be included in a warning.

```
findAddRequiredColumnWarningLocations
  (locations:Set(ImpactLocation), table:Table):
    Set(ImpactLocation) = {
 locations->select(location |
   location.allQueries()->select(q | q.type="SELECT"
   ).allColumns().owner->includes(table) or
   location.allModelTypes().table->includes(table)
 )
}
```

The OCL functions use a number of auxiliary functions whose definition we omit for brevity. Functions `allQueries`, `allFields` and `allModelTypes` allow a shorthand for fetching all objects of a particular type that are associated with a specific `ImpactLocation` object on which they are called. Function `allQueries` returns all queries that are defined or executed for a given impact site. Using `allFields` we can find all `ObjModelField` objects that are used, set or read and the associated impact sites. Finally, `allModelTypes` returns all `ObjModelType` objects that are used or instantiated at the given `ImpactLocation`.

For the purposes of our example, the column being added in Change 1 will have no default value and will not allow null values. This means that although no `SELECT` queries will be affected by the change, there are `INSERT` queries that will be affected. The following OCL function finds all `INSERT` queries that insert values into the table having the new column. As the new column is mandatory any `INSERT` queries for this table may cause errors in the application, since the required column will be missing from the query. These sites are error sites, and in our example this function will predict errors err5 and err6.

```
findAddRequiredColumnErrorLocations
  (locations:Set(ImpactLocation), table:Table):
    Set(ImpactLocation) = {
  locations->select(location |
    location.allQueries()->select(q | q.type="INSERT"
    ).queryColumn.owner->includes(table)
  )
}
```

We next consider Change 2, which renames the column `Experiments.Date`. The OCL function requires as argument the column that is to be renamed. The impact locations affected will contain the definition or execution of any query that references the column or any location in the source code that uses an object model field that maps to the changed column. This function will predict err1 and err3.

```
findRenameErrorLocations
  (locations:Set(ImpactLocation), renamed:Column):
    Set(ImpactLocation) = {
  locations->select(location |
    location.allQueries().allColumns()
      ->includes(renamed) or
    location.allFields().column->includes(renamed)
  )
}
```

Finally, we consider Change 3 from our example scenario in which column `Experiments.Name` is dropped. The effects of this schema change are that any query that references the dropped column will become invalid. Also, any locations in the application that reference the object model fields that refer to the dropped column may also be invalid and need to be removed. Therefore, this function operates by returning the definition or execution of any query that references the dropped column, and any location where an object model field that maps to the column is used. In our example this will predict err2 and err4.

```
DropColumnErrorLoc
  (locations:Set(ImpactLocation), dropped:Column):
    Set(ImpactLocation) = {
  locations->select(location |
    location.allQueries().allColumns()
       ->includes(dropped) or
    location.allFields().column->includes(dropped)
  )
}
```

We now have far more information at our disposal than can be obtained by the previously mentioned static typing techniques. For each change, we know the semantics of the change primitive as well as what will be affected. This allows us to infer a great deal of information that may be useful to the developer. Consider, for example, err1 and err2; we now know that err1 is caused by a renaming change (Change 2) and that err2 is caused by a drop column change (Change 3). Given this information we can now postulate that to alter the code to correct err1 will be relatively simple because the object-relational mapping can simply be amended, whilst the application logic can remain largely unchanged. Handling err2, however, will require the reference to the missing column to be removed, possibly also requiring the removal of its associated logic and any other changes to the source code that may be required to accommodate this. This information could be of great benefit to both the application developer and the DBA, as knowing the effort required to reconcile an impact site can help the DBA choose between

alternative schema changes. This outlined meta model approach now puts us in a position whereby we can provide a great deal of useful information about the impact of a schema change.

4. IMPLEMENTATION

We have implemented the Schema Update Impact Tool Environment (SUITE) to establish the feasibility of the analysis technique described above. SUITE focuses on analysing the impact of schema changes on queries in Java programs that have been written using the Plain Old Java Queries (POJQ) and the JDO ORM libraries.

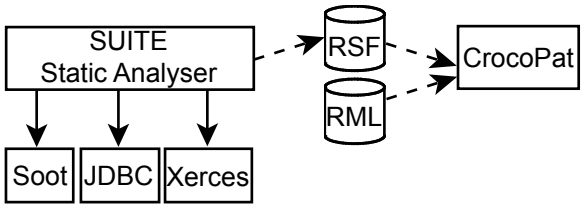

Figure 3: Tool Architecture

Figure 3 presents an architectural overview of SUITE with rectangles representing components, database icons representing files, solid arrows denoting invocations and dashed arrows indicating data flow. The environment uses four external components. We use JDBC to extract schema information from MySQL as indicated in Algorithm 3.1. The JDO ORM specifies object relational mappings in an XML file and our environment uses Apache's Xerces parser to implement Algorithm 3.2. We use Soot to perform the static analysis described in Algorithm 3.3. The SUITE Static Analyzer then serialises the instances of the meta model shown in Figure 2 that were created by the three algorithms into a file. The file uses the Rigi Structured Format (RSF) [23]. This RSF file is read alongside a relational program written in the Relation Manipulation Language (RML) [2] that can be executed by CrocoPat, a tool for simple and efficient relational computation. The RML relational programs are direct implementations of the OCL functions discussed in Section 3.3. We could have used the OCL interpreter included in the UCL-MDA tool suite in order to directly execute the OCL functions. However, execution by the BDD-based CrocoPat implementation is by several orders of magnitude faster than OCL interpretation.

4.1 Static Analysis of Dependency Data

Obtaining the schema meta data is trivially implemented for most modern DBMSs. Most database APIs include some facility for iterating over this meta data. We use a JDBC driver to analyse the schema of MySQL databases for the implementation of Algorithm 3.1.

The implementation of Algorithm 3.2 is equally trivial. We use Xerces to parse the XML file of the JDO ORM. We use an XPath expression [4] to locate all classes and then use another XPath expression to identify all fields of that class. This leaves us with the remaining problem of how to efficiently extract all queries and possible impact points from the application source code.

In order to implement Algorithm 3.3, we needed a tool which allows us to efficiently reason about the structure and behaviour of Java programs. We use Soot [22] which provides such a reasoning facility based on byte code and data flow analysis for Java. Soot provides us with the primitives for a more or less direct implementation of Algorithm 3.3. Firstly, Soot allows us to iterate through all the application code, identifying any statements of interest, such as all places where a specific method is called, or all locations where a class is instantiated. Secondly, it allows us to implement data flow analyses by which we deduce more complicated results such as all locations where a specific instance of a Predicate is converted into JDO Query object. Soot already provides much of the functionality required for data flow analysis such as calculating possible call graphs and providing basic implementation of backwards and forwards data flow analyses which can be extended and customised. Soot provides a rich API by which we can easily analyse the syntax and structure of each statement in our application. This API has proved rich enough to provide all the operations required by Algorithm 3.3.

To illustrate the use of Soot, let us reconsider Algorithm 3.3. Some parts of the analysis are only specified vaguely in pseudo code, for example when we need to determine the query semantics defined by a POJQ Predicate object. In order to find out the semantics of a query defined by a POJQ Predicate, we use Soot to analyse the definition of the Predicate class. Consider the example code in Figure 1. The definition of `Q1Pred` in Line 02 uses Java generics, specifying the `Experiments` class as an extent. The `Experiments` class is specified by our ORM as mapping to the `Experiments` table in the schema. Specifying this class as an extent defines the query as `SELECT * FROM Experiments`. When we analyse the definition of the Predicate subclass using Soot, we can find this information, and programmatically determine the type of the extent class used by the predicate. Thus we can determine from which table the query is selecting. The `WHERE` clause of the SQL query is specified in the `match` method, and we can see in Line 10 that in this case we are matching based on the `Experiments.Date` field. Using Soot we can locate instances of `ObjectModelField` that are read within this method, giving us a clear indication of exactly what elements are present in the `WHERE` clause.

4.2 Analysis using CrocoPat

In order to evaluate instances of our meta model we considered several approaches, but eventually decided upon using CrocoPat [2]. As mentioned previously, the main reason for choosing to use CrocoPat was that it provides far better performance than any of the other techniques we considered.

We present CrocoPat with an instance of our meta model specified in the RSF file format. Serialising meta model instances to the RSF file format is trivial, because RSF is sufficiently simple, and data represented in our model fits nicely into RSF's relational paradigm. Figure 4 shows an excerpt of the RSF file that was produced from the analysis of our example scenario.

CrocoPat reads RSF line by line. Each line of the file represents a triple, with each value being separated by white space. The first value represents a relationship, the second and third values represents nodes which are associated by this relationship. For example the first line of this RSF fragment, shows that the `Exp-main-25` node is related to the node `INSERT` by the relationship `Query_AttributeTypeHasValue`. This shows that the attribute

```
Query_AttributeHasValue         Exp-main-25  INSERT
Query_DefinedAt_ImpactLocation Exp-main-25  sample.Main:25
Query_HasQueryColumn_Column     Exp-main-25  experiments.experiment_id
Query_HasQueryColumn_Column     Exp-main-25  experiments.date
Query_HasQueryColumn_Column     Exp-main-25  experiments.name
Query_HasQueryColumn_Column     Exp-main-25  experiments.description
```

Figure 4: RSF sample output from static analysis

`type` of query 'Exp-main-25' is of value `INSERT`. The rest of the sample shows the definition impact location where the query is executed and all the columns that belong to the query. Instances of the meta model of Section 3 are presented in this format to CrocoPat. Note, that we use a slightly different naming convention for the relationship defined on the first line. This is because the first line shows that INSERT is an attribute value of Exp-main-25 in the meta model, whilst the remaining lines represent associations.

We use CrocoPat to evaluate RML programs against the supplied RSF instance of our meta model. We create RML programs for each elementary schema change; each program having the semantics specified by the OCL functions illustrated above. The RML language provided by CrocoPat is sufficiently expressive to be able to evaluate all changes that we have so far considered.

```
AffectedQueries(x):=Query_HasQueryColumn_Column(x,$1) |
                    Query_HasWhereColumn_Column(x,$1) |
                    Query_HasOrderColumn_Column(x,$1);
AffectedQueryDefinitions(x):=EX(y,AffectedQueries(y) &
                Query_DefinedAt_ImpactLocation(y, x));
AffectedQueryExecutions(x) := EX(y, AffectedQueries(y) &
                Query_ExecutedAt_ImpactLocation(y, x));
AffectedFields(x) := Column_MapsTo_Field($1, x);
AffectedFieldUses(x) := EX(y, AffectedFields(y) &
                (Field_UsedAt_ImpactLocation(y, x) |
                 Field_ReadAt_ImpactLocation(y, x) |
                 Field_SetAt_ImpactLocation(y, x)));
PRINT ["IMPACT:Query definition at "]
                AffectedQueryDefinitions(x);
PRINT ["IMPACT: Query execution at "]
                AffectedQueryExecutions(x);
PRINT ["IMPACT: Object model field used at "]
                AffectedFieldUses(x);
```

Figure 5: Calculation of Column Renaming Impact

Figure 5 shows an example RML program. This program implements the OCL function illustrated in Section 3.3 that locates all impact locations that will be affected by a column renaming. The unique name for the column to be renamed is passed as a parameter into CrocoPat, and is represented in the application by `$1`. RML uses Prolog like facts and the relationships defined in the RSF input to locate all affected impact locations. RML is described in more detail in [2].

4.3 Performing the analysis

SUITE allows us to do the following. Firstly we populate an instance of our meta model for a given schema and application, using the static analysis techniques outlined previously. Secondly we specify an arbitrary series of elementary changes that we wish to make to the schema. Each change we specify has an associated CrocoPat RML program. The tool then executes the impact analysis by using CrocoPat to execute the RML against the instances of the meta model that we have extracted.

The results of the analysis are exported to a report. For each elementary change, this report lists the following:

1. the predicted severity of any affected impact locations,
2. advice on how impact locations should be checked for errors, or how errors should be remedied,
3. the probability that the impact locations will actually be affected, and
4. the list of impact locations that have been identified.

Items 1-3 are fixed for each change primitive, and included as static text to help the reader of the report understand the results and place the predicted impacts in the correct context. Only item 4 changes for each instance of the meta model and is calculated by the RML programs. This report is the final output of our prototype tool. Figure 6 shows an example of the report that could be produced for our example scenario.

```
CHANGE:   Drop a column
Entity:   experiments.name
ERROR(S): There will be errors following this update
SEVERITY: Severe - requires the removal of all reference
          to this column at these impact sites.
IMPACT:   Query definition at sampleApp.main():15
IMPACT:   Query definition at sampleApp.main():25
IMPACT:   Query execution at sampleApp.main():17
IMPACT:   Query execution at sampleApp.main():25
IMPACT:   Object model field used at sampleApp.main():23
```

Figure 6: Impact Diagnosis provided by SUITE

This section of the output report shows all code locations that are affected by Change 3. The report indicates the line numbers in our example source code that are affected, and provides some simple information about the types of impacts. Lines 15 and and 17 identify the definition and execution of Q1. This query selects all columns from the experiments table, implicitly selecting `Experiments.Name`, therefore it is marked as an impact location. The remaining impact locations all relate to Q2, where `Experiments.Name` is explicitly referenced.

5. EVALUATION

In order to evaluate this work, we are interested in understanding whether SUITE scales to the size of databases that occur in practice. We also want to see whether the impact analysis results that SUITE produces assist DBAs and application programmers with their tasks during the evolution of a schema.

We use an experimental method to perform this evaluation. More specifically, we choose Ensembl [11] as a case study. Ensembl is a database that was developed by the EBI and stores a number of genomes, including the Human Genome. Ensembl is a good case study because it is large; the Human Genome part of the database has a volume of 7 GByte of data alone. Ensemble is available open source. As can be seen from Table 1, the schema has up to 69 tables and a total of up to 386 columns in the most complex version. Thus the Ensembl schema is more than four times as complex than the largest schema used in the evaluation of related work [9][2]. There is a full version history available for the Ensembl schema. Moreover, a number of different applications have been developed that query the Ensembl database, some of which are also available in open source form.

One of these applications is BioJava [17] developed at the Sanger Centre. BioJava is a Java tool for processing biological data that, amongst other functionality, provides an API to access the Ensembl database. We chose to use BioJava in our evaluation because it was developed outside the EBI, it was written in Java, which means that SUITE can analyze it, and there is a version history available for BioJava, which is correlated with the Ensembl schema history. BioJava has a data access layer package that uses the core schema of Ensembl. In the versions we have considered this package has 3836-5772 lines of Java code. The Java code includes up to 39 JDBC queries that we have translated into POJQ queries in each relevant version. These queries use up to 31 of the 69 tables of the core Ensembl schema, which is a sufficiently large subset for us to see impact of schema changes.

In order evaluate our approach, we have examined the schema changes in the Ensembl CVS repository. Generally, the EBI team are very defensive with respect to schema changes and many versions of Ensembl do not change the schema, in part because the impact of schema updates is so difficult to predict manually. However, Ensembl versions 31, 34 and 38 each make schema updates that have an impact on BioJava. There were other schema updates that did not have any impact on BioJava and we omit their discussion for reasons of brevity. We have studied these updates and broken them down into the elementary changes shown in Table 1 that SUITE can analyze. The table shows for each type of elementary change how many of these changes were performed in the transition between respective versions.

	Versions		
	30-31	33-34	37-38
Number of Tables	63	67	69
Number of Columns	342	371	386
Java LOC	3836	4964	5772
RenameColumn	1	2	1
AddColumn	2		2
DropTable	1	1	3
AddForeignKey	4	3	4
ExpandColumnValues		1	
ChangeColumnType			8
Total	8	7	18

Table 1: Ensembl Schema Updates from Version 30

[2]The largest schema evaluated in this paper has 82 columns.

In order to evaluate the quality of our schema impact analysis results, we can firstly compare the results with changes that the application programmers of BioJava actually performed in response to schema changes of Ensembl. A summary of the results is presented in Table 2. The results here show the number of impacted locations that were (P)redicted and (O)bserved in the successor version of BioJava. The name of those elementary changes that are likely to cause errors is given in italics, while names of changes that cause warnings are shown in normal font. We have been able to predict all changes that were made by the BioJava application programmers in the first two versions, and made predictions for the third version that have yet to be verified; We also did not have false positive error predictions in the first two versions. We will verify the third set of results when the next version of BioJava is released. For the prediction of the first Ensembl schema change (Version 30 to Version 31), SUITE predicted three locations in the BioJava application to be impacted as a result of the rename column change and one as the result of the drop table change. An example of a prediction where we issued a warning is AddColumn. In this particular example, whether or not the locations are to be changed depends whether the application is required to use the data that is now available in the new column. In the case of the AddColumn change primitive we can see that only one of the predicted impact locations is actually an observed impact in the BioJava case study. It is important to note that this does not invalidate our analysis. This proves that giving warnings is useful to application programmers as impact locations need to be considered; otherwise the one change that was made, could have been missed.

Likewise for the second schema change (Version 33-34), SUITE correctly predicted six impacted locations in the Java code as error locations as a result of each of the rename column changes. This time none of the warnings actually required changes. Far more locations were predicted for the third change, but in order to verify these predictions we must wait for the next release of BioJava which will be updated for version 38 of Ensembl.

	Versions					
	30-31		33-34		37-38	
	P	O	P	O	P	O
RenameColumn	3	3	6	6	-	/
AddColumn	8	1	4	0	4	/
DropTable	1	1	0	0	0	/
AddForeignKey	10	0	5	0	10	/
ExpandColumnValues	-	-	1	0	0	/
ChangeColumnType	-	-	-	-	9	/

Table 2: Impacted Locations found by SUITE

A DBA will most likely be interested in those elementary changes that might cause errors in applications. From this point of view, our tool predicts that the drop table changes leading to Versions 34 and 38 that would cause errors if the dropped tables were used will have no impact on the BioJava application at all. For Version 34 this is validated by the absence of any changes to BioJava as a consequence of the dropped tables. Using our tool, a DBA could obtain access to all applications that use their schema, via some application source code repository such as source forge or a web

enabled CVS repository, and prior to conducting a change calculate the impact the change might have on all known applications. This would greatly increase the information available to the DBA, informing the evolution of databases and enabling more reasoned decisions about whether or not to include an elementary change in a schema update.

We also performed a quantitative performance evaluation. Our evaluation metrics includes the time required for the execution of Algorithms 3.1-3.3 and the time to serialize the extracted instances of our static analysis meta model. We then want to know for each of the elementary change above how long it takes CrocoPat to execute the RML program to predict the impact that that elementary change has. The summary of these quantitative measurements is shown in Table 3. The times are given in milliseconds and are averages of three executions, which were measured on a Pentium 2.13GHz PC with 1.5GByte Memory and the Sun Java 1.5 run-time environment. The standard deviation of these measurements was negligible.

	Versions		
	30-31	33-34	37-38
Schema extraction [ms]	724	750	766
ORM extraction [ms]	359	380	390
Java extraction [ms]	21579	22980	23225
Number of RSF entries	1764	1914	1970
RenameColumn [ms]	641		682
AddColumn [ms]	651		697
DropTable [ms]	646	672	705
AddForeignKeyColumn [ms]	643	674	690
ExpandColumnValues [ms]		709	
ChangeColumnType [ms]			698
Average RML function [ms]	645	676	697

Table 3: Performance of SUITE impact analysis

We can note that there is a moderate startup cost of between 22 and 24 seconds in order to extract the RSF entries required for the impact analysis. This startup cost, however, would only need to be paid once and from then on elementary changes can be analyzed incrementally. It is also encouraging to see that the increase between the 3.9 kLOC version and the 5.8 kLOC version is less than 2 seconds, which we find encouraging. We also note that the averages of individual RML calculating the impact for elementary schema changes are more or less all between 640 and 710 milliseconds. The averages raise between the versions, which is attributable to the increased number of RSF entries that need to be interpreted.

We then measured the time it takes CrocoPat to to execute a simple "hello world" program against the RSF file of Version 33 and found that this requires 657 milliseconds. We can therefore conclude that 95% of the time is required to start CrocoPat and parse our RSF files. In future work, we will investigate various optimizations, including more efficient data extraction with Soot and making CrocoPat memory resident so that it avoids startup costs between different RML program executions. This will render the technique fast enough for use in interactive development environments.

6. RELATED WORK

There is little existing research that is directly applicable to the situations we are considering in which the impacts of schema changes must be identified in the presence of object-relational mappings and native language queries. What research does exist has a variety of limitations. In particular, impact analysis for object-oriented databases [12] avoids the problems caused by the impedance mismatch between object-oriented languages and relational databases and as a result has little relevance to the vast majority of databases, which are relational. The same applies to work on schema updates in object-oriented databases [8] that does not consider the impact of schema changes outside the DBMS.

More directly related is the work of Gould et al [9] who have devised a technique for static analysis of the syntactic and static semantic correctness of dynamically composed JDBC queries. We assume that queries will increasingly be statically typed using native language queries. Gould's work becomes obsolete to some extent for those queries formulated in POJQ and JDO or LINQ because syntactic and static semantic errors are detected by the Java or C# type systems. We instead focus on providing guidance to application developers on how to react to database schema changes.

We also consider that the static analysis approach could be abandoned altogether, and instead a dynamic approach to extracting information from the application could be adopted. This would be similar to the approach taken by the Chianti project [18] whereby a suite of tests is run against the application to obtain a runtime call graph.

The results of our analysis could also be of use to other research. The Chianti project for instance takes as input a set of changes to a Java application and performs program slicing as well as providing impact analysis of affected regression tests. Our impact locations could be used as an input to this process, allowing us to obtain program slicing information from the point of impact, back into the application code. Program slicing and regression test selection are two areas of research that would provide interesting uses for our impact analysis technique.

The implementation of our data flow analysis used in the extraction of query information from the application has some limitations. We do not take into account points-to analysis or inter-procedural data flow analysis. There is a large body of work focusing on such techniques that could be used to improve our implementation, including [14, 15, 19]. However, we consider our implementation to be an investigation into the feasibility and validity of our approach, and thus it is not a model application for how the analysis should be conducted.

Finally, we have made little effort to investigate how the developer would use the information produced by the impact analysis. There has been some research into visualising impact information by Karahasanovic [12]. This would be a useful starting point for future investigation, although our analysis technique provides rather different information than is dealt with by this paper, and would need to be altered accordingly.

7. FURTHER WORK AND CONCLUSIONS

There are a number of directions in which we intend to develop this work further. It will take a while before strongly typed native language queries will be widely accepted. In the meantime, application programmers will continue to use CLIs or maintain legacy code that uses JDBC or ODBC. We intend to experiment with the string analysis used in [9] in

order to extract the instances of our meta model from dynamically composed JDBC queries. There are a large number of improvements that can be made to SUITE. Firstly, we have already sketched some obvious performance enhancements. Secondly, it will be desirable to integrate SUITE as a plug-in into the Eclipse platform so that the capabilities of Eclipse are used to navigate to impact locations. Thirdly, SUITE could be extended to more advanced database schema concepts, such as views, triggers and stored procedures. Finally, we would be interested to apply our techniques to *data definition queries*, which are reflective queries that modify a schema. Such queries in theory can be handled using our approach, but the extent to which they can be reasonably handled will require additional investigation.

The evaluation of SUITE based on the history of a large-scale database has shown the potential benefits of our approach for analyzing the impact that database schema changes have on programs that use object-relational mappings and native language queries. The construction of SUITE was relatively straightforward. It was considerably simplified by the fact that queries and object-relational mappings are statically typed and that there are powerful components available with which we were able to build our static analysis tool. Using our approach DBAs will be able to assess the extent with which their proposed changes will affect programs that are written against their schemas, and have information about the effort required to reconcile these changes. Likewise application programmers obtain detailed guidance on what parts of their program have to be changed, or checked in response to the schema changes. We hope that our work has provided the foundation for the construction of efficient impact analysis tools that in support of a more agile evolution of database schemas.

Acknowledgements

We are grateful to Gavin Bierman and Luca Cardelli at Microsoft Research for convincing us of the industrial significance of native language queries with static typing. We are also indebted to Helen Parkinson at the EBI for helping us get started with Ensembl. We are grateful to James Skene for providing the UCL-MDA tools and assisting us with their use for checking and interpreting OCL.

8. REFERENCES

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

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

[3] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in C-ω. In A. Black, editor, *Proc. of the* 19^{th} *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.

[4] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium, November 1999.

[5] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106, New York, NY, USA, 2005. ACM Press.

[6] W. R. Cook and C. Rosenberger. Native Queries for Persistent Objects, A Design White Paper. *Dr. Dobbs Journal*, February 2006.

[7] L. DeMichiel and M. Keith. Enterprise Java Beans Version 3.0, Java Persistence API. JSR 220, Sun Microsystems, 4140 Network Circle, Santa Clara, CA 95054, December 2005.

[8] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database System. In *Proc. of the* 21^{th} *Int. Conference on Very Large Databases, Zürich, Switzerland*, pages 170–181, 1995.

[9] 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.

[10] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM Press.

[11] T. Hubbard et al. The Ensembl genome database project. *Nucleic Acids Research*, 30(1):38–41, 2002.

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

[13] G. King and C. Bauer. *Hibernate in Action: Practical Object/Relational Mapping*. Manning, October 2004.

[14] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):199–215, 1999.

[15] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79, New York, NY, USA, 2001. ACM Press.

[16] 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.

[17] M. Pocock, T. Down, and T. Hubbard. BioJava: open source components for bioinformatics. *ACM SIGBIO Newsletter*, 20(2):10–12, August 2000.

[18] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, 2004. ACM Press.

[19] 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.

[20] C. Russell. Java Data Objects. JSR 12, Sun Microsystems, 4140 Network Circle, Santa Clara, CA 95054, May 2003.

[21] J. Skene and W. Emmerich. Specifications, not Meta-Models. In *Proc. ICSE 2006 Workshop on Global integrated Model Mannagement (GaMMa 2006)*, 2006.

[22] 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.

[23] 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.