Flexible Database Application Adaptation through wrappers

Author/Student: Aranda Granda, Francisco Javier.

Supervisor: Lloret Gazo, Jorge.

Master on System Engineering and Computer Science

Official Post-degree program on Computer Engineering

(Research work presented on September 2010)

Summary

Objectives

When a DB schema is modified, the programs accessing it often need to be modified as well. Frequently these modifications will be expensive and error-prone.

The general objectives of this research line are 1) to reach a better understanding of the problem, and of the existing solutions, and 2) to develop new solutions to the problem which might be advantageous.

Conclusions

After examining several approaches, I have decided to develop a new open solution intended to be so general that it can be used in any scenario. This solution is open-source (already published in source-forge) with the hope that other players/parts might benefit from it for the discussion of the problem and the accomplishment of further goals.

Unfortunately, some difficult study cases suggest that a fully automated algorithm which solves any case may not be achievable. Therefore, as a difference to other studied approaches I have preferred 1) to avoid the static code analysis and to make a recollection and analysis of the actual program execution traces, and 2) to leave aside (by the moment) the idea of performing a fully automated statement translation; this is, to rely on the practitioners to define the correct handling of the expected DML statements, and let a wrapper component to identify the requested statements and apply the handling configured for them.

One important aspect I have considered is that DB operations might be incorrectly translated if the context (previous and following operations, and relevant data dependencies) is ignored. Therefore those approaches that try to translate the operation in a one-by-one basis might be inadequate. In my implementation means are provided to identify these context-dependencies, and to take them into account in the adaptation. This scenario is illustrated by the selected study case.

The implemented adaptation mechanism is a JDBC wrapper which allows 1) to obtain very detailed information of the DB accesses and 2) to translate such operations by the means of matchers which identify each SQL statement case, and handlers which take care of doing the right operations over the actual schema.

The implementation includes a rudimentary process to analyze the compiled DB operations. This analysis produces a summary of the executed SQL statements, as well as the dependencies between them (which appear in multi-operation transactions), with this summary a base adaptation configuration is created, which can be easily checked and edited to produce the final configuration for the handler.

The current work presents a global vision of the problem, and shows in a practical way how to solve a non-trivial study case using the implemented mechanism.

Índice de Contenidos

1 Introducción....3
2 Adaption alternatives and related works....3
3 Study case description....6
4 Description of the developed wrapper....7
4.1 Data accesss operations translation....9
4.1.1 Literal translation....9
4.1.2 Regular expression based translation....10
4.1.3 Use of special handlers....10
4.1.4 Use of custom handlers....11
4.1.5 Use of state variables....12
4.2 The statements report....13
4.3 Obtainment of the DB access log....14
4.4 Limitations and further work....16
Bibliography....18

1 Introduction

Most frequently computer programs store and retrieve information into and from databases. We name *schema* to the structure definition of these stored data. This schema may experience changes along time. Because of these changes the programs may need to be modified in order to continue working correctly. We name *Database Program Evolution* to the process of adapting the program to such changes.

Therefore in our study case we have a relational DB containing a simplified Wiki. The DB schema is modified in such a way that the references made by the program to the tables and columns are no longer valid, so the program and the DB access operations must be modified in order to work properly.

The DB Program Evolution problem is not new. As early as 1982 Schneiderman and Thomas [6] proposed to define the DB change as a sequence of elemental steps, in function of which it could be determined the changes over the program's DB access operations. After this some others proposals have appeared, which we will describe further in this paper.

However, in some authors' opinion [3] these efforts show to be insufficient (*"The lack of support, in terms of methods and tools, in database maintenance and evolution is testified by the low number of scientific references that can lead to practical application"*). It would be questionable whether the number of scientific references is scarce or it is not. All the references given in section 2 talk about their experimental work and results on large and complex study cases. Unfortunately I have not been able to find any of those solutions made public (either via open source or commercial software) so I could verify it over my own scenarios. Maybe these would have been freely available if I had asked the authors for a copy (But why would they give it away to a complete stranger if they didn't release it wide open in the first place?).

I don't know the exact reasons why such tools (in case they ever had been built) are not available. I consider it would be useful to make an available open source solution. This solution would allow other researchers or developers who might be interested in the subject, avoiding to start it from scratch.

Specifically, the developed frameworks allows two things: 1) The collection of real information about the program DB accesses, obtained through its execution, and 2) the adaptation/translation of the program accesses (which wouldn't have been modified in the source code) to the DB (which schema would have already changed).

This paper is structured in the following way. After the present introduction, in which we have described the DB program evolution problem, in Section 2 we will describe briefly the alternatives proposed by other authors. After that we will introduce our solution, pointing in first place, in section 3, the study case upon we have worked, and describing in section 4 what does the implemented *wrapper* in order to deal with that study case, covering aspects such as the adaptation itself (4.1), and the gathering of the information required for such adaptation (4.2 & 4.3). Likewise, the limitations and the to-do's of the current approach are also discussed (Error: Reference source not found).

2 Adaption alternatives and related works

In this section I will discuss different existing options for tackling with the database

program evolution. Along with every option we will find references to the significant related works.

A relevant aspect for the undertaking of the program evolution is knowing the initial state of the system, that is, what is the DB schema definition, and what are the accesses performed by the program.

Hainaut et al ([7][3]) discuss about the possibility of undertaking a reverse engineering of the schema using the *DB-MAIN* software. The obtained conceptual schema would be useful for the following steps in the program evolution.

PRISM++ software developed by Curino et al [12] use a meta-schema where the pertinent management data about the actual schema is stored.

In the gathered bibliography the prevalent approach is using static code analyzers [3][5]. This analysis examines the data dependencies in the source code, and starting from the calls to the DB access API, tries to find out which SQL sentences were executed. It would be questionable if the static code analysis is preferable. In this work I have preferred a different approach, based on an execution analysis, where the information is gathered from the actual accesses performed by the running program.

For this execution information gathering several similar JDBC wrapper implementations can be found, such as *P6Spy*[11], *ASPY* [8] or *Elvyx* [9]. However, these tools seem to be oriented to different purposes such as detecting accesses/sentences which consume most resources, or produce deadlocks, but do not seem suitable for the software evolution scenario. For the DB program evolution we need thorough information which may allow us to determine relevant aspects such as the relation between executed statements (which are the statement sequences, and the data dependencies woven among statements).

A comprehensive information gathering should contain not only the executed statements, but other elements such as the stack trace of the point of execution where the statement is executed, the value of parametrized parameters (like the '?' in JDBC prepared statements), and the returned values. This gathering, however, may suppose a performance overhead on the system, and produce an excessive amount of output if not used with care.

In order to undertake the schema and program evolution, it is adequate to know not only about the system itself, but also about the nature of the modification to be made. This change may be defined as a sequence of elemental modifications upon the schema (like copying a table, renaming a column, etc). Both Hainaut et al [3] and Curino et al [1][2] define their own transformation/schema modification operation (SMO) sets. They also identify and explain concepts such as the reversibility of the change operations.

Following these transformational approaches, if a single DB access is modified in some way by the application of a single SMO, then a complex change could be performed by the accumulation of the modifications of the SMO's in the sequence that defines the complex schema change.

However, sometimes there are peculiar schema modifications that don't seem to have an associated SMO, or it happens to be complicated to determine the modification upon the program by a concrete SMO. I have considered that this approach can't come to an universal solution in a near future (specifically I don't see it adequate for the µWiki study case shown in this paper). Therefore I prefer to postulate that the practitioners (DB administrators and programmers) are who will have the ultimate responsibility in determining the correct program evolution for each case.

Curino et al [2], also obtain and use a mapping called DED (*Disjuntive Embedded Dependencies*), that indicates the correspondence between the data on the previous schema and on the subsequent schema following the change. The direct mapping would tell how to fill the data into the new schema from the data on the old schema, while the inverse mapping would help in the translation of

the old queries (which should obtain old schema data from the new schema data).

Hick et Hainaut [3] propose a "*syntetic approach*", according to which a tool would examine the schema versions preceding and succeeding the change, and would deduce the concrete modifications made. However there is a chance that the modifications would be ambiguous, so I consider this solution would neither be universal.

In relation to the program modification, the most obvious approach is to manually (relying on a programming expert) change the program code so it performs the DB access operations the way it's needed according to the new schema. Other known alternatives are 1)Use SQL Views to simulate the legacy schema on top of the actual schema, or 2) introducing in the program a wrapper on behalf of the DB access API, which would be responsible for translating the requested operations.

The SQL Views alternative is proposed both by Hick et Hainaut [3], and by Curino et al [2]. These SQL Views would simulate the tables and views from the legacy schema, using the data in the actual schema tables ans views. E.g. in the *PRISM* [12] demo an SQL script is generated which defines such views in function of the initial schema and the defined changes. However the use of views might not be an adequate solution in the case of data updating operations. This difficulty dealing with update operations is relieved to some degree with some techniques (handling the update operation as select operations) introduced in *PRISM++*[1].

The use of wrapper technologies, is proposed among others by Thiran et al [7]. It implies adapting the program introducing a patch which would make a middle component to proxy the data access API. A wrapper component would be fit for updating operations. However, proxing an API might be a little intrusive in the program, and the implementation and configuration of such a component might be complicated.

The use of wrapper or views are superficial solutions. These type of solutions might be adequate if there is a change in the way data is stored, but the nature of the data remains unchanged (e.g. if we make a 3NF normalization, or we rename a column). But in case that the change in the schema is given by a conceptual change in the data (e.g. the introduction of additional attributes, or the alteration in the cardinality of existing associations) it will be needed a deeper change in the program code so it adapts itself to the new nature of the data (unless these new attributes and changes might be ignored by the program).

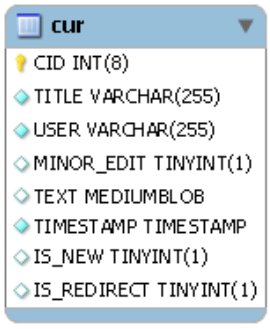
For example if in a PERSON's table we add a new column (e.g. PHONE), if this data must appear in the data input and output screens, then a superficial change in the data access would be insufficient. However, if such data might be ignored by the program (No need to input that phone or to show it), then it probably won't be even necessary to modify the data access code (no need to query the column, not even considering it on insert or update operations).

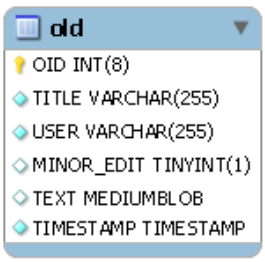
Karahasanovic [4] proposes and implements a visualization of the schema change's impact upon the program through a dependency graph of the classes, methods and properties of a Object Oriented Program. This might offer for the maintainers a better information than a simple listing of the files and line numbers where references to the affected tables and columns are found.

The DB program modification is a subject of interest of agile methodologies. Fowler et Saladage [10] describe a practical experience. They come to tell about a wrapper implementation that adapts in some cases programs to modified schemas, but they don't give further detail. It seems that the main concern under the agile methodologies is to manage the schemas as another element under the version control and the continuous integration, controlling the correct propagation of the DB changes into the different development/testing/production environments. This gets more complicated on an scenario in which different branches of development co-exist, or if changes might be needed to be compared or reverted.

3 Study case description

In this case we make use of the same example used by Curino et al [2] for showing PRISM capabilities. This example is based on the evolution of the MediaWiki schema between the versions named as 41 and 42 (that correspond to the revisions 6696 and 6710 under the actual MediaWiki SVN repository). The initial and final states of the changed schema are shown in figures 1 and 2.





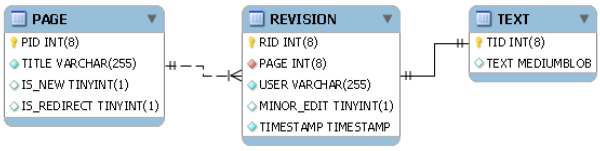
CUR

CID	TITLE	IS_NEW	TEXT	MINOR_EDIT
1	Mercury	1	First planet	0
3	Venus	0	Second planet	1

OLD

OID	TITLE	TEXT	MINOR_EDIT
2	Venus	Second	0

Figure 1: Schema and sample data v.41



REVISION

RID	PAGE	MINOR_EDIT
1	1	0
2	3	0
3	3	1

PAGE

PID	LATEST	TITLE	IS_NEW
1	1	Mercury	1
3	3	Venus	0

TEXT

TID	TEXT
1	First planet
2	Second
3	Second planet

Figure 2: Schema and sample data v.42

In this example we focus on the elemental information from a Wiki page (the page title, the edited text it contains, who edited it, and when it was posted), and on the revision history of the pages. As an example we use a minimalistic data set: We show two pages: "*Mercury*" and "*Venus*". *Mercury* only contains the current version. On the other hand, *Venus* has not only the current version, but a previous one in the *OLD* table.

In the initial DB schema we start with table CUR (current page versions) and table OLD (Old page versions). In the posterior schema we can find table PAGE (page information, shared by all the page's versions), table REVISION(specific information about a page's version), and table TEXT (for whatever reason –performance, indexing, etc– the page's texts are split into a separate table). The current version of a page can be obtained through the `PAGE.LATEST→REVISION.RID` reference (cardinality: 1 page, 1 current revision). The page associated with a revision can be obtained through the `REVISION.PAGE→PAGE.PID` reference (cardinality: N revisions, 1 page). A page's version's text can be obtained through the `REVISION.RID→TEXT.TID` reference (cardinality: 1 revision, 1 associated text).

In order to test the program adaptation, a sample minimalistic Java Wiki application (µWiki) has been implemented, that will be modified with its DB schema. The data access code has been implemented in the manner of a direct JDBC API use (controlling transaction, preparing statements, traversing the resultset rows and columns). Likewise, a DBUnit unit test has been implemented for the program's DB access module. This test would be useful not only for testing the correct working

of the data access, but also for providing a comprehensive use case set, upon which the wrapper can gather the required information for performing the adaptation.

The SQL statements executed by the program comprise a range of difficulty. Some of them can be translated without much problem, but some other should be examined with greater detail.

As a simple case we could find the query statement to request a current page:
`select CID, TITLE, USER, TEXT, TIMESTAMP from CUR where TITLE=?`
Which in version 42 would be executed as:
`select PAGE AS CID, TITLE, USER, TEXT, TIMESTAMP from REVTEXT r,PAGE p,TEXT t where r.RID=p.LATEST and r.RID=t.TID and p.TITLE=?`

We might notice that 1) the query turns into a *join* of several tables, following the previously described associations (`r.RID=p.LATEST`,`r.RID=t.TID`); 2) the symbol "?" is used in JDBC to identify a parameter in a PreparedStatement,; and 3) we rename a resultset's column (`PAGE AS CID`), as the name expected by the program and the actual column name differ.

On the other hand, the insertion/update operation implies greater difficulty. Let's examine the statements from version 41:

```
> insert into OLD (OID, TITLE, USER, TEXT, TIMESTAMP) select CID, TITLE, USER, TEXT, TIMESTAMP
 from CUR where TITLE=$title;
> delete from CUR where TITLE=$title;
> insert into CUR (TITLE, USER, TEXT, TIMESTAMP) values($title,$u,$tx,$tm);
```

Notice that 1) The modification is performed through moving the previously current page's row from table *CUR* into table *OLD*, and then inserting the new version into *CUR*. 2)We have represented the parameters as $parameter to make them more comprehensible (though in the Java source code, they won't be written in that way, but as "?").

In schema version 42 the modification will be executed as:

```
> insert into REVISION (USER, TIMESTAMP, PAGE) values($u,$tm, 0);
  (returning rid into $rid)
> insert into PAGE (LATEST, TITLE) values($rid, $title) on duplicate key update LATEST=$rid;
  (returning pid into $pid)
> insert into TEXT (TID, TEXT) values ($rid, $tx);
> update REVISION set PAGE=$pid where RID = $rid;
```

Notice that this modification is quite different: 1)The moving of the older version from CUR to OLD doesn't make sense any more, and the related statements (initial insert and delete) have no correspondence on version 42. 2) The insertion of a new version into CUR turns into several statements in order to insert/update en every table where the data has been distributed. 3) As the data is spread over different tables, and is related by keys, we should take care of keeping the coherence between these keys, specifically 4) we obtain the keys of inserted rows (`returning rid into $rid`), which we need in the following statements. We might also find that 5) there are operations (such as the `insert into PAGE`), where an update or an insert should be done depending on whether the row already exists or it does not.

This example of DB program evolution is adequate because 1) It illustrates an schema change that is difficult to specify using schema modification primitive operations, in which case it whould be preferrable to make a customized treatment, than a treatment generated by automatic rules, and 2) because although it implies drastic changes in the schema, it does not imply deeper changes in the program (such as in the user interfaces), so it can be resolved 'cleanly' with a superficial change in the data access layer.

4 Description of the developed wrapper

The developed solution consists of a JDBC API wrapper, which allows to perform to different tasks in the DB program adaptation: 1) The gathering of information about the access operations executed, and 2) the translation of the access operations requested by an unmodified program, so it

continues working over an updated schema. Although both tasks can be done in a cmpletely independent manner, it's desirable to search certain integration between them, as the information obtained about the access is of great importance for the latter translation.

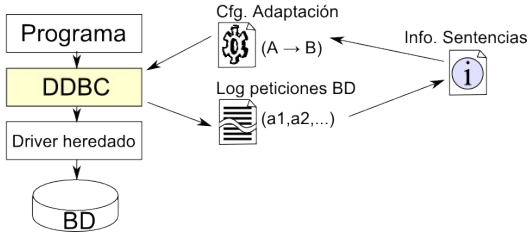


Figure 3: The DDBC wrapper allows to obtain the data and to adapt the DB access

In Figure 3 we can observe how the DDBC wrapper (*Decorator DB Connection*) mediates in the program DB requests (initially intended to be resolved by the inherited driver).

The data accesses A performed (a_1,a_2, …) should be recorded in the requests log in an environment where the schema change is not made yet. This information is useful for producing the adaptation specification (A→B) which tells how to translate the requests to previous schema A to the correspondent requests in later schema B.

In addition to the wrapper I have developed some utilities for summarizing the access log (which can be too verbose and redundant), and to analyze the statement sequence in order to find the data dependencies between them (these dependencies are relevant for some non-trivial translations). The obtained statements report is useful for guiding the developers into the sections of the program which should be checked and modified (and thus for making an estimation of the impact/cost of the modification).

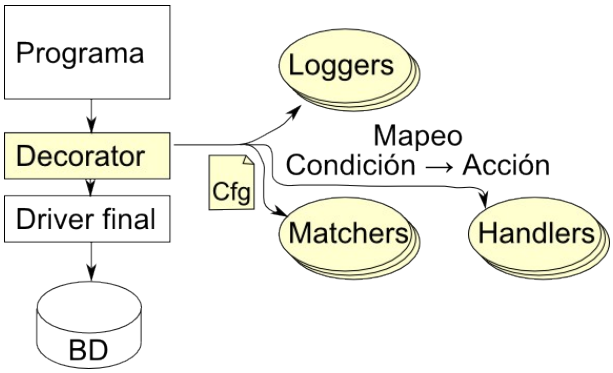


Figure 4: Wrapper components

In Figure 4 we can see how the *decorator* (a wrapper with the underlying component interface, capable of being chained with other similar decorators to provide accumulated functionality) has a behavior which is determined by its configuration, which indicates and configures as well its subordinate components (*loggers*, *matchers* and *handlers*). The user can not only select and configure a set of predefined components, but he also can (as may be foreseen) develop and use her own customized components for her own specific needs (if the predefined components should not be adequate). With this modularity we seek to give the wrapper all the flexibility and power to adapt to any technically feasible adaptation scenario.

The *logger* components are responsible for recording the data accesses made. These loggers may frequently be disabled for privacy and performance concerns (specially once the program has been adapted and/or it is in production stage). The adapter interpreter/translator uses the *matcher* components to identify which case of SQL statement is requested (E.g. one matcher would check if the statement fulfills one given regular expression), and then uses a *handler* component (associated to the case/matcher by the mapping rule) will do the treatment corresponding to the initial statement on the actual schema.

Both the study case application and the wrapper are written using the Java programming language. The details about the configuration and the modules, and about the adaptation process can be checked in the DDBC user manuals (in addenda), or in the source code.

The implementation has found some challenges. A “bare bone” JDBC implementation has around 700 methods as of Java SE 6.0, leading to over 100KB boilerplate source code, which was a little more complex than what an automatic generator for the delegate pattern would build (such as the

one available in the Eclipse IDE). Moreover, the generated methods have to be utterly customized by hand, and those customizations would be difficult to keep if the the generator did not provide a way to preserve them when they have to be regenerated. There have been some difficulties related with the fact that some complex JDBC interfaces extend others (*CallableStatements* extends *PreparedStatement*, which extends Statement), which makes difficult to implement an extendable base implementation (As *MyCustomPreparedStatement* wouldn't extend both *BasePreparedStatement*[for default *PS* method implementation] and *MyCustomStatement* [To avoid re-implementing my custom *Statement* methods], since there is no multiple inheritance). Another difficulty is the multiple ways to execute statements. This would lead to excessive code repetition unless we would channel all the requests into a unique point (the *HandlerMapper* class), capable of handling all the access “flavors”and parameters.

4.1 Data accesss operations translation

Although in the process of adaptation the preparation and use of the adaptation configuration come in last place, it's better to explain them in first place as they give meaning to the first steps, which may be more difficult to understand without them.

The translation of the SQL statements is made by an interpreter, which follows an *adaptation specification*. The *adaptation specification* contains a set of rules defined in the condition-action shape. The conditions are checked through the matchers. In case of a condition being true for a requested statement, then the access operation will be executed using the handler attached in the rule. Currently the conditions are checked sequentially, and once a condition is found to be true, the trailing ones are skipped (so first rules take precedence over the later). In case no condition is verified a default handler may be executed.

In a DB schema change, it is predictable that there are changes only in a part of the schema, so the majority of the applicable statements will remain unchanged. Of the changed SQL statements, it is expected that the queries (SELECT statements) will be fixed executing one equivalent query (even if data is split between tables, as joins and unions of tables can be used in the resulting statement). On the other hand update statements on our experience entail greater difficulty (as updates might be done amongst various tables, and might not be done, or done in a different manner depending on conditions).

Now we will go through a set of situations we may meet in the translation, and the resources we have to solve them. Either with the *matcher* and *handler* components currently included in DDBC (some based in literal translation, some other which translate using patterns, or those who will execute a statement unchanged, or just provoke a failure). We will also see about the development and use of custom *handlers*, and about the use of state variables for the cases in which the same statement may have different interpretations depending on the context.

4.1.1 Literal translation

In the easiest translation case, we may find that some frequently used statements are always literally the same. E.g. for obtaining a list of available wiki pages we may execute:

```
select TITLE from CUR;
```

which in schema version 42 we should execute as

```
select TITLE from PAGE;
```

For this type of translation we have a literal matcher that is activated when the statement is the one literally specified, and a literal *handler* which executed an specified statement in the place of the one initially requested by the program. Therefore, the previous sample translation could be represented by the following rule:

```
<mapping id="select title from cur">
  <matcher type="equals">select TITLE from CUR</matcher>
  <handler type="literal">select TITLE from PAGE</handler>
</mapping>
```

It can be pointed that many statements content variable values. If for example we had a "`where USER='user1'`" clause, then the literal identification wouldn't work if we had multiple possible values for *USER*. Nevertheless in JDBC it is possible (even recommended) to execute these as *PreparedStatement*s, identifying the variable values with the placeholder token (?), so the condition would be expressed as "`where USER=?`".This allows us to use statements which are literally the same even if the concrete values are different.

4.1.2 Regular expression based translation

In some cases we might find that the same statement has small variations, such as the value of a variable (which would be specified as a literal, not as an input["?"]), or little variations in the *resultset* column list, or in the conditions in the *WHERE* clause.

In those cases it would be convenient to identify the statements by the pattern they follow, and to be able to translate identifying the sections of the original statement which must be used verbatim in the executed statement.

For instance: The recurring queries which follow the pattern
`select CID, * from CUR where *`
would be executed in schema version 42 as queries with the pattern
`select PID as CID, * from PAGE P, REVISION R, TEXT T WHERE P.LATEST=R.RID AND R.RID=T.TID AND *`
There is further detail of this transformation in the similar example in section 3.

This rule of translation would be specified as:

```
<mapping id="select cur where ANY">
  <matcher type="regexMatcher">select CID, (.*) from CUR  where (.*)</matcher>
  <handler type="regexHandler" >
  <rx-rule>
    <rx-match>select CID, (.*) from CUR where (.*)</rx-match>
    <rx-subst>select PID AS CID, $1 from PAGE p, REVISION r, TEXT t where
 p.LATEST=r.RID and r.RID=t.TID and $2</rx-subst>
  </rx-rule>
</handler>
</mapping>
```

Notice how the wildcard "*" mentioned before is represented as the regular expression ".*" (any character [.] from 0 to N times); how "capturing blocks" are specified by parenthesis in the match expression, and how their number or order ($1, $2) is referred in the substitution expression.

As *DDBC* is implemented in the Java programming language, the regular expressions used will follow the syntax required by the standard Java regular expression package (*java.util.regex*).

4.1.3 Use of special handlers

I have developed a *handlers* that doesn't change anything, and another handler that signals an exception/failure to the program. It may appear shocking to interpose a component that doesn't do anything, or that makes a failure to intentionally happen, but it makes perfect sense under certain circumstances.

For example, most frequently most of the executable statements won't be affected by the concrete schema changes. For those statements we would like the requests to be passed through to the delegate driver without change (*passThrough* handler). We must also notice that the translator might

find statements that were not foreseen, or any other statement for which we can't guarantee that it will return the expected results and (more important) that it won't corrupt the stored data. For those cases it seems quite sensible to just throw an exception as if an incorrect operation was requested. Hopefully that exception will be logged and reviewed, and an adequate rule for the rejected statement will be added if applies.

That exception throwing *handler* (*fail*), might be useful as well if we have to rush a working configuration before we have worked out the handling for every predictable statement. We might cover first the easiest and more frequent operations (e.g. leaving the wiki application in a read-only state, which is better than leaving it in an inoperative state during the servicing time). That configuration might be updated as we complete and correct the handling of the statements.

As well as we define rules for given conditions, we should define a default rule (otherwise), for the statements which don't comply any of the given conditions. In the case of the lenient assumption that those statements should be given a chance just like they are requested we might represent it as:

```
<otherwise>
  <handler type="passThrough" unexpected="true" />
</otherwise>
```

otherwise, if we want a greater degree of security, then we should use the fail handler:

```
<otherwise>
  <handler type="fail">Unexpected statement</handler>
</otherwise>
```

4.1.4 Use of custom handlers

It is possible (particularly with the update operations) that the simple translation mechanisms described before may prove insufficient, so it might be needed to implement a custom handler. In our study case (see update sequence described in section 3), we find statements ("`insert into old (...) select ... from cur`" and "`delete from cur`") that are no longer needed, while others ("`insert into cur(...) values ...`") turn into multiple statements.

A custom handler must be an implementation of the *StatementHandler* interface. A good starting point which could be taken as a model would be the class *HandlerCmdEdit410in420* created for the study case.

In order to be used, the *handler* class must be declared in the adaptation configuration, being assigned a handler type name (in this case "*cmdEdit*").

```
<def type="cmdEdit" class="uwiki.evol.HandlerCmdEdit410in412" />
```

After that, the custom handler may be used in the mapping rules just like the predefined. It's not necessary to create a custom handler for each statement. In our case we have chosen to create a handler covering all the statements in the update operation (as there is dependence between the statements, it's reasonable to encapsulate that dependence inside a class).

It would be possible to parametrize the custom handlers (though in our study case we didn't have the need), just like the predefined ones (though in complex cases like the *regexHandler* it will require to implement a custom parser as well).

This would be an example of mapping rule using the custom handler:

```
<mapping id="delete cur where title">
  <matcher type="equals" fromState="cmdEdit-deleteCur">
    DELETE FROM cur WHERE title=?</matcher>
  <handler type="cmdEdit" toState="cmdEdit-insertCur"/>
</mapping>
```

Notice the attributes *fromState* y *toState*, which we will describe in the following point.

4.1.5 Use of state variables

It is possible that depending on the context a statement may be translated in different ways. I will try to illustrate it in spite that this doesn't strictly happen in the µWiki study case. Notice that in our case the statement "`delete from CUR where TITLE=?`" is part of a row relocation operation (first copy into target, then delete origin). Such relocation is no longer needed in version 42 (attempting to do so would result first in key duplication, and then in deleting non-redundant information). However, it would be feasible that under different circumstances the data deletion was actually required. To tell apart which treatment do we provide for the operation we would need state variables in order to track the context where such operation is requested.

In our case we find that immediately before the delete we have done an "`insert into OLD (<columns>) select <columns> from CUR where TITLE=?`". This fact (and the coincidence of the value in the "`where TITLE"` condition) allows us to tell apart whether we are moving the page data or we are really deleting it.

Furthermore, the meaning of an operation could depend not only on previous operations, but on following operations. If this was the case, then we would need to wait until the uncertainty is resolved before performing the due operations. E.g. notice that the "`insert into OLD`" statement would be interpreted as an actual insertion, but in our case, because of the following delete statement it would mean to change the page latest revision status from current into old.

In the implementation, for the sake of simplicity I have assumed that the co-dependent operations/statements are executed inside the same JDBC transaction (which would be quite recommended regardless of the adaption). It would be as well useful that for a given intent (e.g. updating a Wiki page) the component statements would be executed in a fixed order (e.g. first update in REVISION table, then in TEXT and finally in PAGE), so we don't have to worry about the possible permutations. There may be cases where the SQL transactions are not taken into account, or the statements are not executed in a fixed sequence (because of conditions, loops or parallelism). Then maybe we should work out a more sophisticated interpreter.

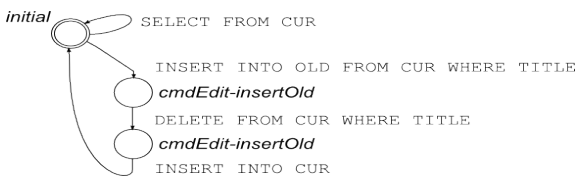


Figure 5: Transaction States for µWiki

Following these assumptions, we can draw a graph (like in Figure 5) showing the possible transaction status. The transaction status is determined by the statements requested since the current transaction started. There would be an "initial" state where there are no executed operations since the transaction was last committed, more states as a result for each statement class executable from the initial states, and more states reachable from those states by the execution of other statements until the transaction commit.

We might need as well some additional variables to store values we would need 1) for executing later statements, if such variable is not available in the triggering statement, or 2) for checking the data dependencies between statements (e.g. if the "`delete from cur statement`" would not refer to the same title than the "`insert into old statement`", they would not be co-dependent, and a different treatment would apply).

The information about the statement sequences, and about the data dependencies beneath can be obtained with the utilities I have developed to analyze the detailed logs.

4.2 The statements report

In order to elaborate the adaptation specification draft we need to know which SQL statements are executed in the application, discarding the repeated statements and other redundant information,

such as the value of parameters or results, which are not relevant.

It is also important to know which statements are grouped in the same SQL transaction, and the data dependencies that would exist among these statements. The data dependencies would be deduced from the coincidence in the value of the parameters and the results of the involved statements. It's convenient to compare the coincidences along several transaction in order to discard those coincidences which are casual (e.g. binary flags, redundant descriptions) and keep those which are systematic.

This data dependence analysis implies that in the initial data gathering we must obtain and store exhaustively the values of the parameters and the results related to each executed statement. This can be a huge amount of data. Therefore redundant and large dataset operations should be avoided in the data gathering phase described in section 4.3.

We should also be able to tell the programmers which points in the program should be checked for the specific schema changes. For that purpose we should record the code locations where the related statements are executed. In the Java language, thanks to the debugging information is easy to obtain that information (the class, the method and line number) through the stack traces.

The statement report can be generated with the *MainGenSr* Java application class, providing the filenames of the input (*test-log.xml* log file) and the output (*test-sr.xml* statements report)

```
java ddbc.tool.sr.MainGenSr test-log.xml test-sr.xml
```

This would be en excerpt of the generated statements report:

```
<statementReport>
  <statements>
    <statement stmtId="SR-stmtC0000">
      <normSql>select c.cid, c.title, c.user, c.minor_edit, c.text,
 c.timestamp, c.is_new, c.is_redirect from cur c order by c.title
 asc</normSql>
      <location>
        <f class="org.apache.commons.dbcp.DelegatingConnection"
m="prepareStatement" f="DelegatingConnection.java" l="248"/>
        <f class="uwiki2.WData" m="selectCurWhere" f="WData.java" l="83"/>
        <f class="uwiki2.WData" m="listCurrent" f="WData.java" l="44"/>
                        (more stack frames).................
      </location>
                        (possibly more locations)..........
    </statement>
                        (more statements)..................
  </statements>
  <sequences>
    <sequence seqId="SR-seqC0003">
      <stmtRefs>
        <stmtRef idRef="SR-stmtC0003"/>
        <stmtRef idRef="SR-stmtC0004"/>
        <stmtRef idRef="SR-stmtC0005"/>
      </stmtRefs>
      <dataDependencies>
        <group>s1p1,s2p1,s3p1</group>
 (possibly more sequence data dependencies).....................................
      </dataDependencies>
    </sequence>
                        (possibly more sequences)..........
  </sequences>
</statementReport>
```

In the fist place we can find the enumeration of the found statement classes. I call “statement class” to the abstraction of statements which do the same operations in the same tables and columns with the same options, regardless of the values used in each statement instance(E.g. There would be a

statement class for the “select * from cur where title=(value)”, which would contain all the similar statements such as “`select * from cur where title='Mars'`” and “`SELECT * FROM CUR WHERE TITLE ='Jupiter'`”). For each statement class we can find the location(s) where it is executed.

In the statement report we may also find information about the statement sequences where these classes may be involved (It may be important to know the related statements, and the possibility of finding the same statements in different sequences in order to ensure a correct translation/treatment.

We may find the data dependencies (systematic coincidences) which occur in the sequence. There are match groups, that contain the reference set to the elements which must share the same value. These references have the format `s<n>{p<n>|r<n>|k<n>}` telling the statement numer of order (*s*), and the type and number of order of the related element, where the element can be a parameter (*p*), a column in the *resultset* (*r*), or a returned key (*k*). So in our case `s1p1,s2p1,s3p1` tells that the page title parameter (p1) is equal in all three statements (s1,s2,s3).

4.3 Obtainment of the DB access log

In our case the purpouse of the obtained log is to adapt the program. Apart from the obvious need for the statement expressions, we mainly need to obtain the boundaries of the SQL transactions, and the value of the parameters and the results. These are needed to determine the dependencies between statements. It may be also useful to obtain the program locations where the DB API calls are made (in order to correlate with the source code, and to help applying the modifications).

Other known DDBC wrappers (which have a different purpose) focus on the operation execution time, which leads to potential problems such as performance bottlenecks and inter-locks. The measure of the data traffic may come handy as well. The main instance of such wrappers/loggers is *Elvyx* [9]. Other examples are *P6Spy* [11] (*Elvyx*'s predecesor), and *ASPY* [8] (an alleged improvement over *P6Spy*).

Though *DDBC* stores a superset of the information stored by these other wrappers, the intent is not to supplant them, as the large amount of data processed and stored may pose a problem in terms of performance and privacy.

The *DDBC* log obtention for the adaptation is expected to be a punctual task. It is required that all the SQL statements, and the different transactions the program may execute are registered. These statements and transactions should be executed repeadly with small variations to tell apart the casual data coincidences from the consistent ones (which would be interpreted as a data dependence within a statement sequence). Therefore, in order to obtain the needed information it is neccesary to execute a suficiently comprehensive test suite, which would be similar to what we would use in program/unit testing.

Though such test suite could be "manually" executed, it would be advisable to provide a unit test. In out case our test is based on *DBUnit* (a *JUnit* extension that allows the initialization, verification and cleaning of the tested schema).

The obtained log is stored in a local file that would look like this:

```
<custom conn="CONN0000" src="CONN0000" tag="CONN_new"
          start="20100507.115324.453" d="0">
  <f class="ddbc.cpn2.impl.CpConnection" m="&lt;init&gt;"
          f="CpConnection.java" l="48"/>
                    (more stack frames).................
</custom>
<call conn="CONN0000" src="CONN0000" m="CONN_setAutoCommitB"
          start="20100507.115324.468" d="16">
  <p>true</p>
</call>
<call conn="CONN0000" src="CONN0000" m="CONN_prepareStatementS"
          start="20100507.115324.484" d="94">
  <f class="org.apache.commons.dbcp.DelegatingConnection" m="prepareStatement"
          f="DelegatingConnection.java" l="248"/>
                    (more stack frames).................
  <p>SELECT c.cid, c.title, c.user, c.minor_edit, c.text, c.timestamp,
 c.is_new, c.is_redirect FROM cur c  ORDER BY c.title ASC</p>
  <r>CpPreparedStatement:PRST0000</r>
</call>
<call conn="CONN0000" src="PRST0000" m="PRST_executeQuery"
 start="20100507.115324.578" d="0">
  <r>CpResultSet:RSST0000</r>
</call>
                    (remaining lines ...)..............
```

The log is in XML format (excluding the header and the root element, which are not written). The main elements are:
`<call>` (which represents the calls to the JDBC API) and
`<custom>` (other elements that not correspond to a method in the JDBC API).
These main elements can contain the inner elements
`<f>` (invocation *stack frame*),
`<p>` (invocation parameter) and
`<r>` (return value of the invocation).

In the shown lines can be seen how a connection is created (with the identifier *CONN0000*), a statement is prepared (returning a *CpPreparedStatement* with identifier *PRST0000*), and this statement is used to execute a query.

An interesting aspect is the representation of some values as a *hash*/*digest* of these. For example:

```
<call conn="CONN0000" src="RSST0000" m="RSST_getBytesS" start="20100507.115324.593" d="0">
  <p>text</p>
  <r>96edae55495cc8e82d9b1f090360fcb9de01649eb113d8214780b0a6b002bdcd:    14:Primer planeta</r>
</call>
```

Watch how a binary data has been converted into a string with its cryptographic *hash*(*SHA1*), along with the data length and a representation of the first bytes. This representation is useful is useful for values that may be too cumbersome to write on the log(in the order of KB's and above). It can be used as well to mask confidential data (although in fist place these data shouldn't be available in the testing environment where the log should be obtained).

4.4 Further work and limitations

The implemented wrapper provides no only a non-intrusive program adaptation mechanism for a wide range of schema evolution scenarios, but it also has been made flexible to be easily modified and integrated (component based, implementation of standard interfaces), so it has some potential of leading for further improvements.

At the moment the actual translation has be done “by hand”, but probably there may be cases where

the statements would be automatically translated (as done for example in PRISM [12]), at least to some extent (as an example of difficulty of such task, it is mentioned in section 4.1.5 that the very same statement could find different translations in different contexts). In any case there is freedom for any part to implement and use the handler components they want.

This work has some pending issues and limitations, which should be mentioned in order to assess whether this solution is adequate for your problem (maybe one of the limitations is one of the requirements), and to provide some clue for anyone who might want to make some improvement or give some advice.

- **Improve** the **support** for statements where the **variables** appear as **literales** instead of JDBC parameters. It would be possible that two statements, being structurally identical, may be identified as different because of having different literal values within. Furthermore these values wouldn't be taken into account for the identification of data dependencies. It is necessary to change the process of elaboration of the statements report to solve these problems.
- Store and reproduce the **previous schema metadata** . Sometimes the behaviour of the data access is influenced by the schema metadata(definition of tables, columns, etc). When the schema is changed this metadata changes, and thus the dynamics of the program may be altered. This may cause problems, which we should avoid by “handling” the accesses to the metadata in order to return old metadata (which the program needs to work as expected) instead of the current one.
- To use some kind of **schema change specification language**. It can be based on the SMO's by Curino et al [2]. It may be interesting as well the concept of data mapping between schema versions(Curino DED's).
    - Such specifications are required to do any kind of automatic translation. Just the same as in language translation you don't only need the source expression, but you also need to know the target language. Translations of not-too-difficult statements for not-too-difficult scenarios is feasible and probably good enough. But precaution should be taken regarding implementing or “buying” an universal solution, as it seems to be a major technical challenge.
    - The knowledge of which tables and columns are affected by the schema change would allow to easily tell apart the program' statements potentially affected (those which refer to the changed elements), from those who aren't. A further analysis might provide an impact estimate (how much effort to deal with the change) for the schema evolution. This information might be useful for some early decision making (What does this schema change imply? Can we afford it? Alternatives?). This form of impact analysis was proposed by Maule et al. [5].
- To improve the repository of *matchers* and *handlers*. Adapting the program should be as easy as possible. To avoid as much as possible the need of developing tailor-made *handlers* .
- To develop some tool that allows (using the obtained access log), to reproduce the behavior in order to make unit tests which would be useful in the case of any modification in the system, in order to check that the requests still work, with the replies following the expected patterns.
- To test and to enhance the framework to escalate to big applications: The sequential matching mechanism is simple and adequate for reduced programs, but might be a bad solution if over a hundred matchers would be checked for each statement. Some filters based on elements such as the type of request (select, update, etc), and the table names should be implemented.

Currently I don't have any roadmap of when I'm going to what and how. If you have any relevant interest in any area please feel free to contact the author (fjarandag at gmail).

Bibliography

[1] Carlo Curino, Hyun Jin Moon, Alin Deutsch, Carlo Zaniolo, *Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++*, 2010,PVLDB 4,2 p117-128
[2] C. Curino, H. Moon, C. Zaniolo, *Graceful Database Schema Evolution: the PRISM Workbench*, 2008,PVLDB 1,1 p761-772
[3] J.M. Hick, J.M. Hainaut, *Database Application Evolution. A transformational Approach*, 2006,DKE 59,3 p534-558
[4] Amila Karahasanovic, *Identifying impacts of database schema changes on applications*, 2001,in: Proc of the 8th Doctoral Consortium at theCAiSE*01. Freie Universität Berlin, Interlaken, Switzerland, 2001, pp. 93–104.
[5] A.Maule, W. Emmerich, D.S.Rosenblum, *Impact Analysis of Database Schema Changes*, 2008,ISCE 2008
[6] B.Schneiderman, G.Thomas, *An Architecture for Automatic Relational Database System Conversion*, 1982,ACM Transactions on Database Systems, Vol. 7, No. 2, June 1982, Pages 235-257
[7] Ph. Thiran, J.L. Hainaut, *Wrapper Development for Legacy Data Reuse*, 2001,WCRE 2001, pages 198-207
[8] A. Torrentí-Román, L. Pascual-Miret, L. Irún-Briz, S. Beyer, F. D. Muñoz-Escoí, *Aspy An Access-Logging Tool for JDBC Applications*, 2007,Univ.Politécnica Valencia TR-ITI-ITE-07/24
[9] (various project commiters), *Elvyx*, http://elvyx.sourceforge.net/
[10] Martin Fowler, Pramod Saladage, *Evolutionary Database Design*, http://www.martinfowler.com/articles/evodb.html
[11] Andy Martin, Jeff Goke, Alan Arvesen, Frank Quatro, *P6Spy*, http://www.p6spy.com/
[12] C.Curino, H.J.Moon, C.Zaniolo, *Prism*, http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism