JSquash: Source Code Analysis of Embedded Database Applications for Determining SQL Statements

Dietmar Seipel[1], Andreas M. Boehm[1], and Markus Fröhlich[1]

University of Würzburg, Department of Computer Science
Am Hubland, D–97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de, markusfroehlich1@gmx.de, ab@andiboehm.de

Abstract. In this paper, we analyse Java source code of embedded database applications by means of static code analysis. If the underlying database schema is changed due to refactoring or database tuning, then the SQL statements in the embedding Java program need to be adapted correspondingly. This should be done mostly automatically, since changing software manually is error–prone and time consuming.
For determining the SQL statements that access the database, we can either look at the database logfile, an audit file, or at the Java source code itself. Here, we show how to statically determine even the strings for dynamically built SQL statements directly from the Java source code. We do this without using a debugger or a virtual machine technique; instead, we trace the values of variables that contribute to a query string backwards to predict the values as precisely as possible.
We use PROLOG's declarative features and its backtracking mechanism for code analysis, refactoring, and tuning.

1 Introduction

During the software life–cycle, enterprise–class databases undergo a lot of changes in order to keep up with the ever–changing requirements. The growing space requirements and complexity of productive databases make the task of maintaining a good performance of the database query execution more and more complicated. The performance is highly dependent on the database schema design [14], and additionally, a complex database schema is more prone to design errors.

Increasing the performance and the manageability of a database usually requires *analysing* and *restructuring* the database schema and therefore affects the application code indirectly. The application code highly depends on the database schema, because the database queries are embedded in the source code. Usually, they are contained in string variables or, more dynamically, statements are generated step by step by using control structures and string concatenations.

Nearly any type of database schema modification implies the adaption of the queries embedded in the application code. Sometimes, the logic of the construction of the query strings has to be changed, too. Some efforts have been made to prescind the applications code from the data persistence in the database [11]. The employed methods require a configuration that tells the database service the mapping between database tables and

application objects. Moreover, the relationships between the tables must also be included in that configuration. Therefore, such approaches are also affected by database changes and need to be adjusted accordingly.

Recently, we have developed a standardized XML representation of SQL statements and database schemas named SQUASHML, that was introduced within the PROLOG based toolset Squash for *refactoring* and *tuning* relational database applications [1]. Squash is the first tool that supports the analysis and the refactoring of database applications with coordinated simultaneous modification of the SQL code and the database schema definition. Squash also detects inconsistencies and common flaws in SQL statements. It can determine an optimized configuration of indexes as well as of tables, and propose modifications of the database schema that result in an efficiency gain. The changes are applied automatically to both the schema and the SQL statements.

In this paper, we develop an extension named JSquash of Squash that analyses a given Java source code and presents all expressions in the code that influence the construction of the embedded SQL statements. Our approach estimates or even determines the values of variables in the source code of the databases application by static code analysis; this is used to predict the embedded database queries that can be generated by the application.

Related research on *source code analysis* has mainly focussed on static analysis providing information about security issues (secure code) [8, 3], model extraction [10], code smells [7], obvious errors [15, 13], code metrics [6], and flow analysis [5]. Recently published results were mainly about supporting the complex task of understanding the run–time behaviour of legacy and non–legacy software systems [18, 6]. An extensive amount of research has been dedicated to system analysis by extracting information generated during the run–time of a software system [6]. These approaches require the code being instrumented prior to execution by various techniques, such as wrapping, logging or extended virtual machines. The output of such an analysis is an execution trace containing information about the run–time behaviour of the system, such as call hierarchies of methods and object creation. Such traces of productive software systems are often very voluminous, and usually the values of interesting variables are missing.

The rest of the paper is organized as follows: Section 2 summarizes the basic concepts of managing source code with PROLOG in the JSquash repository. Section 3 describes some basic methods of static code analysis supported by JSquash, such as the calculation of metrics and the detection of code smells. Section 4 presents the strategies of JSquash used for recursively evaluating the control flow of the source code in order to determine values of variables at run–time for predicting embedded SQL statements. Section 5 shows how JSquash can visualise the creation of embedded SQL statements using HTML 4, CSS, and JavaScript technology, and how Squash can visualise join conditions of complex SELECT statements. Finally, Section 6 summarizes our work.

2 Management of Java Source Code with PROLOG

In JSquash, Java code is first parsed and transformed into an XML representation called JAML [9]. Subsequently, a source code *repository* is built from the XML representa-

tion, which suitably represents the constructs of the analysed programming language. It is essential to choose an adequate format for the repository, such that the analysis can handle the data efficiently. The analysis is implemented using the declarative logic programming system SWI–PROLOG [4, 19].

2.1 Representation of Java Source Code in XML

The XML representation JAML completely resembles the Java source code; it even conserves the layout of the source code. JAML enables standardised access to the source code. Generally, XML is a markup language for representing hierarchically structured data, which is often used for the exchange of data between different applications; as a particular XML language, JAML is a very good candidate for an intermediate representation during the creation of the repository out of the Java source code. It comes with a plugin that can be installed as an online update in the integrated development environment Eclipse, that performs the transformation from Java to XML on–the–fly while programming. For every type of expressions, i.e., variables, statements and control structures, there is a special XML element in JAML, which holds the hierarchical design of the program as well as detailed information about the represented Java component.

According to the W3C XML 1.1 recommendation, the terseness in XML markup is of minimal importance, and JAML increases the volume by a factor of about 50. To meet our requirements, the contained information has to be condensed; e.g., the single indenting white spaces are of no interest.

For example, the Java variable declaration `int a = 5`, that additionally includes an initialisation, is presented in JAML, as shown in the listing below; for clarity, the representation has been strongly simplified. The Java declaration is represented by a `variable-declaration-statement` element in JAML. The included `type` element sets the data type of the variable `a` to integer. The assignment is represented by the `variable-declaration` subelement: the identifier of the variable is given by the `identifier` element, and the literal expression is given by an `expression` element.

```
<variable-declaration-statement>
  <type kind="primitive-type">
    <primitive-type> <int>int</int> </primitive-type>
  </type>
  <whitespace/>
  <variable-declaration-list>
    <variable-declaration>
      <identifier>a</identifier> <whitespace/>
      <initializer>
        <equal>=</equal> <whitespace/>
        <expression>
          <literal-expression type-ref="int">
            <literal>
              <number-literal>5</number-literal>
```

```
            </literal>
          </literal-expression>
        </expression>
      </initializer>
    </variable-declaration>
  </variable-declaration-list>
</variable-declaration-statement>
```

For deriving the JSquash repository, the JAML data are represented in field notation and processed using the XML query, transformation, and update language FNQuery [16, 17]. The query part of the language resembles an extension of the well–known XML query language XQuery [2]; but FNQuery is implemented in and fully interleaved with PROLOG. The usual axes of XPath are provided for selection and modification of XML documents. Moreover, FNQuery embodies transformation features, which go beyond XSLT, and also update features.

2.2 The JSquash Repository

The JSquash repository stores the relevant elements of the Java code, which are extracted from the JAML representation, in the form of PROLOG facts. These facts represent information that at least consists of the type and a description of the location within the source code including the locations of the surrounding code blocks. Additionally, necessary parameters can be added, depending on the type.

The construct of a *path* reflects the hierarchy of the code nesting. A path starts with the file number of the source code; the further elements of the path are derived from the position attributes (`pos`) of the JAML representation.

The JSquash repository supports access to variables, objects, classes as well as method calls. E.g., a variable declaration is stored in the repository using a fact

```
jsquash_repository(
   Path:'variable-declaration', Type, Id, P:T).
```

where `Path` is the path of the statement, `Type` is the type and `Id` is the name of the variable, and `P:T` is an optional reference to the in–place initialisation.

For example, the representation of the following fragment of a Java source code, which starts at position `101` in the Java source file, in the JSquash repository will be explained in more detail.

```
1: int a = 5;
2: int b = 10;
3: int c = a + b;
4: int d = c;
```

Due to surrounding blocks and preceding statements, all paths have the common prefix "`0, 49, 3, 49, 94`", which we abbreviate by "`...`". Since `a` starts at position `105` and `5` starts at position `109`, the declaration `int a = 5` in line 1 is represented by the following two facts:

```
jsquash_repository(
   [..., 105]:'variable-declaration', int, a,
   [..., 105, 109]:'literal-expression').
jsquash_repository(
   [..., 105, 109]:'literal-expression', int, 5).
```

The type `variable-declaration` requires two parameters: the type of the variable and its name. In this example, the in–place initialisation with the value 5 is represented in the repository by the path of that expression as a reference, here `[..., 105, 109]`, together with the fact of the referenced expression. Such expressions are represented in JSquash by the type `literal-expression`.

The source code of line 2 is represented by two similar facts, where "`5`" is replaced by "`10`", "`105`" is replaced by "`119`", and "`109`" is replaced by "`123`".

Line 3 is more complex, because it contains the sum of `a` and `b` in the initialisation, a binary expression that comes in the repository as the type `binary-expression`.

```
jsquash_repository(
   [..., 134]:'variable-declaration', int, c,
   [..., 134, 138]:'binary-expression').
jsquash_repository(
   [..., 134, 138]:'binary-expression',
   [..., 134, 138, 138]:'variable-access-expression', +,
   [..., 134, 138, 142]:'variable-access-expression').
jsquash_repository(
   [..., 134, 138, 138]:'variable-access-expression', a).
jsquash_repository(
   [..., 134, 138, 142]:'variable-access-expression', b).
```

The reference of the declaration fact points to the description of the binary expression, which holds two references, one for the left and one for the right expression; in our case, both are accesses to variables.

Line 4 contains an access to a variable instead of a literal expression. This is represented in the repository by the type `variable-access-expression`, which works like `literal-expression`, but it has no qualifier for local variables – as in our example.

```
jsquash_repository(
   [..., 152]:'variable-declaration', int, d,
   [..., 152, 156]:'variable-access-expression').
jsquash_repository(
   [..., 152, 156]:'variable-access-expression', c).
```

The notation described above has been used for convenience to improve the readability of rules referring to the repository. For *efficiency* reasons, we store different facts with the predicate symbols `jsquash_repository_/3,4,5` in the repository; then we can make use of the index on the first argument to search for facts of a given type. The predicates from above are derived using the following simple rules:

```
jsquash_repository(P1:T1, Type, Id, P2:T2) :-
   jsquash_repository_(T1, Type, Id, P2:T2, P1).
jsquash_repository(P1:T1, P2:T2, Op, P3:T3) :-
   jsquash_repository_(T1, P2:T2, Op, P3:T3, P1).
jsquash_repository(P:T, Id) :-
   jsquash_repository_(T, Id, P).
```

Using the design of the repository described here, JSquash is able to work efficiently with application code of any complexity.

3 Static Code Analysis

Source code analysis comprises the manual, tooled or automated verification of source code regarding errors, coding conventions, programming style, test coverage, etc.

The tool JSquash supports some standard methods of static code analysis, such as the calculation of metrics and the detection of code smells. In the following, we will show some examples.

3.1 Local and Global Variables

The following rule determines the classes and identifiers of all local (if `Type` is given by `'variable-declaration'`) or global (if `Type` is `'field-declaration'`) variables within the source code, respectively:

```
variables_in_system(Type, Class, Id) :-
   jsquash_repository([N]:'java-class', _, Class),
   jsquash_repository([N|_]:Type, _, Id, _).
```

Note, that the repository facts for Java classes have a path consisting of only one number. Every variable in a Java class must have this number as the first element of its path.

3.2 Detection of Flaws and Code Smells

JSquash supports the detection of some types of code flaws, i.e., code sections that could be sources of potential errors.

The following rule determines the method signatures of all methods having more than one `return` statement:

```
methods_with_several_returns(
      Class, Method, Signature) :-
   jsquash_repository([N]:'java-class', _, Class),
   jsquash_repository([N|Ns]:'method-declaration',
      Method, Signature, _),
   findall( Path,
```

```
      ( jsquash_repository(
           Path:'return-expression', _, _),
        append([N|Ns], _, Path) ),
      Paths ),
   length(Paths, Length),
   Length > 1.
```

The method declarations within a Java class have the position of the class as the first element in their path. Similarly, the return expressions of a method have the path of the method as a prefix of their path.

The following rule detects all `if` conditions that always evaluate to `false`. It determines the sections of all such `if` statements within the analysed code. If all possible evaluations of an `if` condition are `false`, then the `if` statement is unnecessary.

```
unnecessary_if_statements(If_Path) :-
   jsquash_repository(
      If_Path:'if-statement', P:T, _, _),
   init_run_time(R),
   set_run_time(R, R2, [
      searched_expression@path=P,
      searched_expression@type=T]),
   forall( eval(R2:P:T, Value),
      Value = [boolean, false|_] ).
```

After initialising the run–time, which will be explained in the following section, with `init_run_time/1`, the path and the type of the searched expression are stored in the run–time using `set_run_time/3`.

All of these features make use of the basic code analysis that is supported and integrated in JSquash. This comprises the detection of the following facts: which code sections influence the values of variables, which methods are calling other methods (call dependencies), and which objects are created at which time by which other objects. The predicate `eval/2` for backward tracing the flow of control leading to the values of variables will be explained in the following section.

4 Backward Tracing of the Control Flow

While looking for actual values of variables, JSquash recursively calls the predicate `eval/2` until the evaluation reaches assignments having concrete literal expressions on their right hand side. As literal expressions need not to be evaluated, their values can immediately be used as arguments in complex expressions. After the recursion is rolled out completely, the calculated values of the partial expressions are returned until the value of the examined expression is determined.

For example, during the search for the current value of `d` in line 4 of the code fragment from Section 2.2, first the variable `c` in the right hand side of the assignment has to be evaluated. Therefore, JSquash detects the assignment in line 3, where `c` can be

```
public | |class| |Sender| {

    public void sendMessage() {

        String tCondColumns = { "a","b","c" };
        String tCondValues = { "1","2","3" };
        String tOrderByColumns = { "c","d" };

        String tSQL = "SELECT * FROM training";
        tSQL += " WHERE ";
        tSQL += tCondColumns[0] + " = " + tCondValues[0] + " AND ";
        tSQL += tCondColumns[1] + " = " + tCondValues[1] + " AND ";
        tSQL += tCondColumns[2] + " = " + tCondValues[2];

        tSQL += " ORDER BY ";
        tSQL += tOrderByColumns[0] + ", ";
        tSQL += tOrderByColumns[1];
        tSQL += " ASCENDING";

        Connection con = DBTools.getConnection();

        try {
            PreparedStatement st = con.prepareStatement(tSQL);
            st.execute();
        }catch {Exception  e)
            e.printStackTrace();
        }
    }

}
```

Fig. 1. A code example that uses arrays of strings to construct an SQL statement.

evaluated by determining the current values of `a` in line 1 and `b` in line 2 and by returning their sum. Another example using arrays of strings to construct an SQL statement is shown in Figure 1.

The current state of the analysed program plays an important role for `eval/2` , because it determines the control flow of the applications's calculations. The program state is the set of all variables and their current values that are needed for evaluation of the control structures involved in the calculation of the value of the examined variable.

4.1 Overview of the Evaluation Strategy

Evaluating a specific variable means accessing it at a specific point of run–time. As variables always receive their values strictly via assignment expressions, the analysis component has to find the last assignment before the examined access to the variable.

Therefore, the possible assignments have to be examined. To do so, a simulation of the control flow has been implemented in JSquash that considers control flow statements such as loops and `if-then-else` constructs. This machine recognizes, if a program block directly or indirectly influences the value of the examined variable.

The search for the current value of a variable has been implemented in the predicate `eval/2` . Given the reference to a variable, it returns the corresponding value, depending on the current state. On backtracking, it determines all further value assignments for the given variable. E.g., `if` statements may lead to alternative value assignments,

if the condition cannot be evaluated and the `then` or `else` part assigns a value to the examined variable. But, often there is only one possible value assignment.

`eval/2` is always called with the first argument of the form `R:P:T`. The *run–time* object `R` holds fundamental information for determining the values of variables. It is represented as an XML object in field notation, which holds the following information:

- the current position of the analysis in the source code,
- the analysed control structure,
- the state of the variables defining the current state and control flow,
- the currently examined variable, and
- the currently examined assignment.

The run–time is needed as a supporting data structure for analysing the lifetime behavior of loops and other constructs that have nested blocks, including recursive method calls. In these cases, it is used for holding the distinct states of the blocks passed during the analysis. The value history of the run–time is used extensively to reflect the construction and dependencies of each single value of a variable.

`P:T` *references* the currently searched expression. Based on the type of the referenced expressions, JSquash can decide which rule of `eval/2` has to be used for determining the actual value of the given expression. In the case of variable accesses, the `eval/2` rule – which is shown in Section 4.2 – determines the current value of the referenced variable at the time of the access.

For each type `T` of *control structure*, a rule has been developed that simulates its behaviour and functionality. These rules implement evaluation strategies that yield the current value for all type primitives. The handling of local and global variables (class fields) is implemented separately, since the evaluation strategies differ. In the following, we will show some examples; more complicated cases, such as the handling of Java loops, cannot be shown due to the inherent complexity of their evaluation.

4.2 Variable Access Expressions

While processing the Java code fragment of Section 2.2, JSquash has at first to resolve the identifier `d`. The JSquash repository fact

```
jsquash_repository(
   [..., 152, 156]:'variable-access-expression', c).
```

shows that the variable access expression at `[..., 152, 156]` refers to the variable `c`. Based on the repository fact

```
jsquash_repository(
   [..., 134]:'variable-declaration', int, c,
   [..., 134, 138]:'binary-expression').
```

the predicate `next/2` finds out that the most recent assignment defining `c` was the binary expression at `[..., 134, 138]`, cf. line 3 of the Java code fragment:

```
eval(R:P:T, Value) :-
   T = 'variable-access-expression',
   jsquash_repository(P:T, Id),
   set_run_time(R, R2, [
      @searched_id=Id,
      @search_mode=variable,
      @scope=T ]),
   next(R2:P:T, R3:P3:T3),
   eval(R3:P3:T3, V),
   handle_postfix(P3:T3, V, Value).
```

Similarly, the repository fact

```
jsquash_repository(
   [..., 105]:'variable-declaration', int, a,
   [..., 105, 109]:'literal-expression').
```

is used later during the computation to find out that the variable `a` is declared using a literal expression.

For finding the most recent assignment to the examined variables, the predicate `next/2` has to traverse the JSquash repository facts in inverse code order. This can be supported by further PROLOG facts in the repository, which link a statement to its preceding statement in the code order.

4.3 Binary Expressions

The repository fact

```
jsquash_repository([..., 134, 138]:'binary-expression',
   [..., 134, 138, 138]:'variable-access-expression', +,
   [..., 134, 138, 142]:'variable-access-expression').
```

shows that the binary expression for `c` refers to two variable access expressions. After evaluating them, the resulting values are combined using the binary operator (in our case `+`) indicated by the fact from the repository:

```
eval(R:P:T, Value) :-
   T = 'binary-expression',
   jsquash_repository(P:T, P1:T1, Op, P2:T2),
   eval(R:P1:T1, V1),
   eval(R:P2:T2, V2),
   apply(Op, [V1, V2, Value]).
```

If we evaluate the following code fragment, then both expressions within the binary expression in line 2 are evaluated w.r.t. the same runtime `R`, but with different references `P1:T1` and `P2:T2`, respectively:

```
1: int a = 5;
2: int c = a++ + a;
```

The call `eval(R:P1:T1, V1)` evaluates the left expression `a++` to `5`; only afterwards, the value of `a` is incremented to the new value `6`. The call `eval(R:P2:T2, V2)` for the right expression `a` re–evaluates the left expression, since the new value of `a` is relevant. Thus, the right expression `a` correctly evaluates to `6`, and finally, `c` evaluates to `11`, i.e., `5 + 6`.

4.4 Literal Expressions

The repository fact

```
jsquash_repository(
   [..., 105, 109]:'literal-expression', int, 5).
```

shows that the value of the literal expression at `[..., 105, 109]` is `5`. This type of expression is evaluated using the following rule:

```
eval(R:P:T, Value) :-
   T = 'literal-expression',
   jsquash_repository(P:T, _, Value).
```

5 Visualisation of Embedded SQL Statements

The tool JSquash can detect and analyse SQL statements embedded in the Java source code of database applications.

5.1 SQL Statements Embedded in the Source Code

For presenting the results of the analysis to the user, JSquash includes a generator component, that produces an HTML document containing the complete information about the detected SQL statements, including the full SQL code and the source code that contributes to each SQL statement. This HTML document comprises a fully functional *graphical user interface* (GUI) that can be opened and used with any Web browser. The GUI is implemented in HTML 4 using cascading style sheets (CSS) and JavaScript; the JavaScript helps building a dynamic GUI.

The expressions that contribute to the following generated SQL statement have been detected by JSquash and are automatically highlighted, see Figure 1. JSquash was also able to build the full SQL statement by only analyzing the source code:

```
SELECT * FROM training
WHERE a = 1 AND b = 2 AND c = 3
ORDER BY c, d ASCENDING
```

This statement – which is the second SQL statement in the GUI of Figure 2 – is visualised in Figure 3. The left side in of the GUI shown in Figure 3 displays all the class files of the source code that contribute to the detected SQL statements. Clicking on the

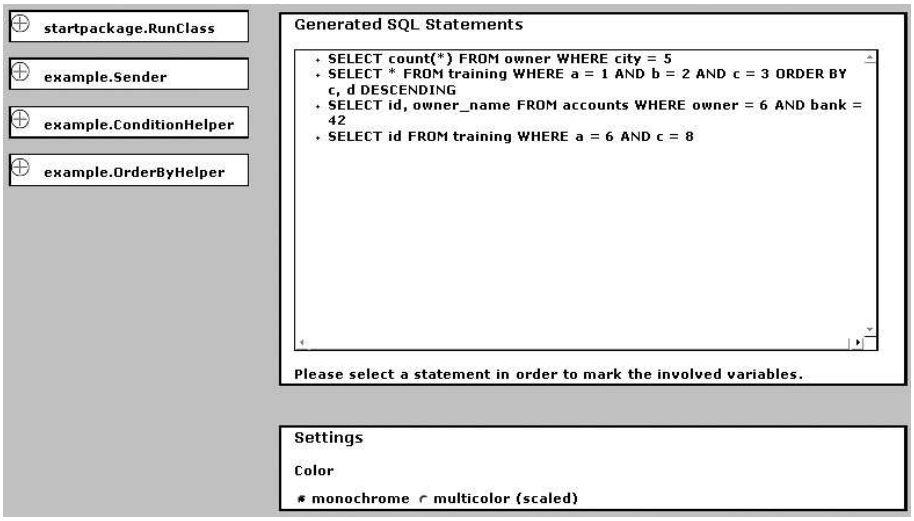

Fig. 2. The GUI of JSquash. No SQL statement is selected.

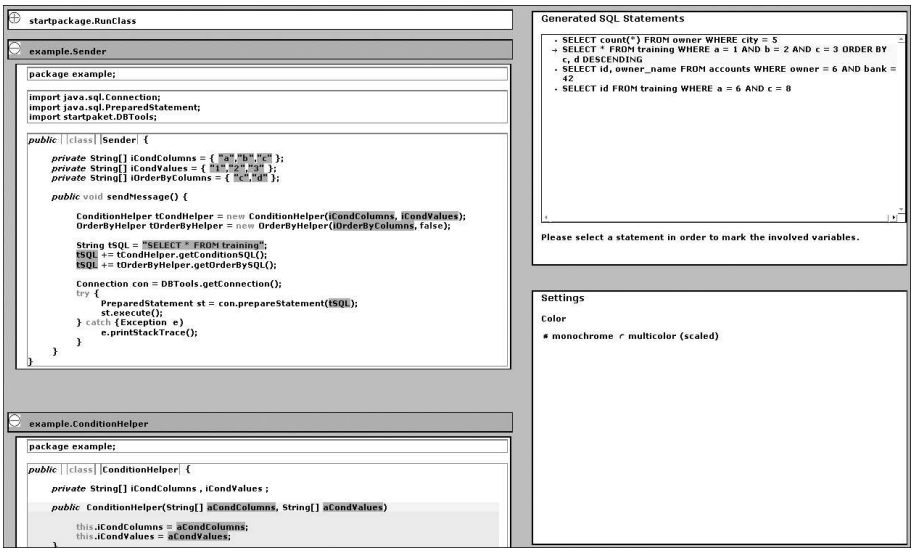

Fig. 3. All contributing values and variables of the selected second SQL statement are marked.

buttons at the left side of the class name opens (+) or closes (-) the source code, respectively. At the upper right side, all detected SQL statements are shown. Below is the block of settings, that allow for changing the highlighting colors.

If an SQL statement of the list is selected, then the corresponding code sections and expressions are automatically highlighted in the listings at the left side, cf. Figure 3. Thus, the user can easily analyse all code sections that contribute to the selected SQL

statement. This feature is implemented in JavaScript and CSS, making extensive use of the path information from the repository.

5.2 Representation and Visualisation of SQL Statements

Recently, we have developed the tool Squash for analysing, tuning and refactoring relational database applications [1]. It uses an extensible and flexible XML representation for SQL database schema definitions and queries called SQUASHML, that is designed for representing schema objects, as well as database queries and data modification statements.

The core of SQUASHML follows the SQL standard, but it also allows for system–specific constructs of different SQL dialects; for example, some definitions and storage parameters from the Oracle database management system have been integrated as optional XML elements.

The SQUASHML format allows for easily processing and evaluating the database schema information. Currently, supported schema objects include table and index definitions. Other existing XML representations of databases, such as SQL/XML, usually focus on representing the database contents, i.e., the table contents, and not the schema definition itself [12]. SQUASHML was developed specifically to map the database schema and queries, without the current contents of the database.

The SQL statements detected by JSquash are transformed to SQUASHML, and then the tool Squash can be used for the visualisation of the single statements. E.g., the following SELECT statement from a biological application joins 9 tables; the table names have been replaced by aliases A, ..., I:

```
SELECT * FROM A, B, C, D, E, F, G, H
WHERE A.ID_DICT_PEPTIDE IN (
   SELECT ID_PEPTIDE FROM I
   WHERE I.ID_SEARCH = 2025
   GROUP BY ID_PEPTIDE
   HAVING COUNT(ID_SEARCH_PEPTIDE) >=1 )
AND A.ID_SEARCH = 2025
AND c1 AND c2 AND A.FLAG_DELETED = 0
AND c3 AND c6 (+) AND c7 (+) AND c4 AND c5
AND E.LEN >= 6 AND A.NORMALIZED_SCORE >= 1.5
ORDER BY ...
```

The following 7 join conditions are used:

```
c1: A.ID_SEARCH_PEPTIDE = B.ID_SEARCH_PEPTIDE
c2: A.ID_SPECTRUM = G.ID_SPECTRUM
c3: A.ID_SEARCH_PEPTIDE = C.ID_PEPTIDE
c4: C.ID_SEQUENCE = D.ID_SEQUENCE
c5: A.ID_SEARCH = H.ID_SEARCH
c6: B.ID_PEPTIDE = E.ID_DICT_PEPTIDE
c7: B.ID_PEPTIDE_MOD = F.ID_DICT_PEPTIDE
```

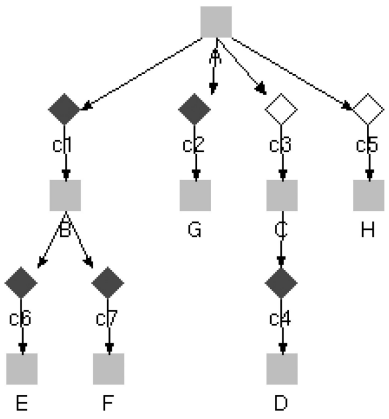

Fig. 4. Join Conditions in a Query

This query is parsed into the following SQUASHML element; we leave out some opening and closing tags, respectively:

```
<select>
  <subquery id="subquery_1">
    <select_list> <expr> <simple_expr>
      <object table_view="A" column="ID_SEARCH"/> ...
    <from> <table_reference> ...
      <simple_query_table_expression
        object="A" schema="USER"/> ...
    <where> ...
    <order_by> ...
</select>
```

The conditions in the WHERE part (e.g., the join condition `c1`) look like follows:

```
<condition>
  <simple_comparison_condition operator="=">
    <left_expr> <expr> <simple_expr>
      <object table_view="A" column="ID_SEARCH_PEPTIDE"/> ...
    <right_expr> ...
      <object table_view="B" column="ID_SEARCH_PEPTIDE"/> ...
</condition>
```

Squash provides a number of different visualization methods for the database schema and the queries. Complex select statements tend to include many tables in join operations. Therefore, Squash uses a graph representation for query visualization, cf. Figure 4. If a SELECT statement contains nested subqueries (like the statement shown above), then these queries can be included in the graphical output if desired.

6 Conclusions

We have shown, how to derive the content of application variables in Java programs using means of static code analysis. Our tool JSquash, which is implemented in PROLOG, predicts the values of variables as precisely as possible; obviously, some values cannot be discovered at compile–time, e.g., if a value was obtained through I/O operations.

Now, we are able to analyse *embedded* SQL *statements* of a given database application, either by analysing audit files of the database connection using the basic tool Squash [1], or by static source code analysis with the extended tool JSquash. The statements derived from the source code of the database application can be imported into Squash, which can then generate database modifications for improving the performance of the application.

Future work will be on developing methods that preserve the linkage between the detected single SQL statement fragments and their positions as well as each of their effects in the completed statement. This extension to SQUASHML will then allow for injecting the changes proposed by Squash into the original source code of the application, and it will help conducting the appropriate changes there.

Moreover, we will try to apply similar techniques of static code analysis to PROLOG programs with embedded SQL statements as well.

References

1. BOEHM, A. M., SEIPEL, D., SICKMANN, A., WETZKA, M.: *Squash: A Tool for Analyzing, Tuning and Refactoring Relational Database Applications.* Proc. 17th International Conference on Declarative Programming and Knowledge Management, INAP 2007, pp. 113–124
2. CHAMBERLIN, D.: *XQuery: a Query Language for XML.* Proc. ACM International Conference on Management of Data, SIGMOD 2003. ACM Press, 2003, pp. 682–682
3. CHESS, B., MCGRAW, G.: *Static Analysis for Security.* IEEE Security & Privacy 2(6). 2004, pp. 76–79
4. CLOCKSIN, W. F.; MELLISH, C. S.: *Programming in* PROLOG. 5th Edition, Springer, 2003
5. CORBETT, J. C.; DWYER, M. B.; HATCLIFF, J.; LAUBACH, S.; PASAREANU, C. S.; ZHENG, R. H.: *Bandera: Extracting Finite State Models From Java Source Code.* Proc. International Conference on Software Engineering, ICSE 2000, pp. 439–448
6. DUCASSE, S., LANZA, M., BERTULI, R.: *High–Level Polymetric Views of Condensed Run–Time Information.* Proc. 8th European Conference on Software Maintenance and Reengineering, CSMR 2004, pp. 309–318
7. VAN EMDEN, E.; MOONEN, L.: *Java Quality Assurance by Detecting Code Smells.* Proc. 9th Working Conference on Reverse Engineering, WCRE 2002. IEEE Computer Society, pp. 97–108
8. EVANS, D., LAROCHELLE, D.: *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Software 19(1). 2002, pp. 42–51
9. FISCHER, D.; LUSIARDI, J.: JAML*: XML Representation of Java Source Code.* Technical Report, University of Würzburg, Department of Computer Science. 2008
10. HOLZMANN, G. J.; SMITH, M. H.; *Extracting Verification Models by Extracting Verification Models.* Proc. Joint International Conference on Formal Description Techniques, FORTE 1999, and Protocol Specification, Testing, and Verification, PSTV 1999, Kluwer, pp. 481–497

11. JBOSS; RED HAT: *Hybernate.* https://www.hibernate.org/
12. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075–14:2003 Information Technology – Database Languages – SQL – Part 14:* XML *Related Specifications (SQL/XML).* 2003
13. MARINESCU, R.: *Detection Strategies: Metrics–Based Rules for Detecting Design Flaws.* Proc. 20th IEEE International Conference on Software Maintenance, ICSM 2004, pp. 350–359
14. RAMAKRISHNAN, R.; GEHRKE, J.: *Database Management Systems.* 3rd Edition, McGraw–Hill, 2003
15. REN, X.; SHAH, F.; TIP, F.; RYDER, B. G.; CHESLEY, O.: *Chianti: A Tool for Change Impact Analysis of Java Programs.* ACM SIGPLAN Notices 39(10). 2004, pp. 432–448
16. SEIPEL, D.: *Processing XML Documents in* PROLOG. Proc. 17th Workshop on Logic Programmierung, WLP 2002
17. SEIPEL, D.; BAUMEISTER, J.; HOPFNER, M.: *Declarative Querying and Visualizing Knowledge Bases in XML.* Proc. 15th International Conference on Declarative Programming and Knowledge Management, INAP 2004, pp. 140–151
18. SYSTÄ, T.; YU, P.; MÜLLER, H.: *Analyzing Java Software by Combining Metrics and Program Visualization.* Proc. 4th European Conference on Software Maintenance and Reengineering, CSMR 2000, IEEE Computer Society, pp. 199–208
19. WIELEMAKER, J.: *An Overview of the* SWI–PROLOG *Programming Environment.* Proc. 13th International Workshop on Logic Programming Environments, WLPE 2003, pp. 1–16
20. WIELEMAKER, J.: SWI–PROLOG. Version: 2007. http://www.swi-prolog.org/