SyQL: Querying Software Process Data Through an Object-Oriented Metamodel

Mirco Bianco[1]

[1] Faculty of Computer Science, Free University of Bolzano-Bozen, Italy
{Mirco.Bianco}@unibz.it

Abstract. The effective usage of the automatically collected software process data may be challenging. By using general purpose query languages like SQL, retrieval of software process information introduces a cost and competence barrier, which limits the adoption of Automated In-process Software Engineering Measurement and Analysis (AISEMA) systems, since it requires a considerable effort for query writing as well as a deep understanding of process data collection and representation. In this paper, we describe the implementation of a query language, SyQL, mainly but not exclusively designed for software process data. SyQL significantly reduces the competence barrier by providing a query interface aimed to software process data.

Keywords: query language, object-oriented, relational databases, fuzzy logic.

1 Introduction

One of the main goals of Software Engineering is the improvement of the efficiency and effectiveness of the software development process. Although many different approaches have been proposed throughout the years, process monitoring is still more an art than a science. Even when apparently quantitative parameters like defect rates are involved, human factors play a large role: developers are often under pressure to meet strict deadlines, and programmers under stress are known to make many mistakes [7].

Being able to continuously monitor the software process is of paramount importance to keep software quality under control, well as to carry out activities like software process auditing and certification, which are also very important today. For this purpose, Automated In-process Software Engineering Measurement and Analysis (AISEMA) systems [5] have been invented. These systems usually work on a relational database, which is generally very complex, and requires a specific knowledge for retrieving useful information. Therefore, using a general purpose relational data model and query language for storing and retrieving software process data requires a considerable query writing effort as well as an uncommonly deep understanding of process data collection and representation issues. For this reason, developing the data layer of software process monitors has traditionally been considered a labour- and knowledge-intensive activity [2].

The main contributions of this paper are the following: we describe a query language called System Query Language (SyQL), aimed at querying data collected from the

software development process but not limited to this domain, next we show how to use this language for retrieving information from a generic AISEMA database, and finally we show the benefits provided by SyQL through two controlled experiments.
SyQL is an object-oriented data manipulation language (DML) that uses a data model consisting of classes and methods for providing a clear view of the software development process to the different software process stakeholders (hereinafter collectively called *users*), including project managers. By using SyQL, it is possible to create a schema suited for a specific application (in this case the software engineering domain) consisting of new, user-defined classes and methods, which are designed and implemented by using well-known object-oriented techniques.
We claim that SyQL has significant advantages compared to general-purpose query languages like SQL. Specifically, SyQL helps the software process stakeholders to exploit the experience of software engineers who know process and product metrics well, even when those engineers are not available at query writing time. This problem is well recognized in the literature [4] and is one of the well-known barriers that limit the adoption of the AISEMA systems [5].
From a pair of controlled experiments, we observed that a user writes queries that are shorter and contains fewer errors by using SyQL than by using SQL or LINQ [11]. In addition to that, the measurements show that learning SyQL takes less time than LINQ. These results can be explained by looking at SyQL data model, which allows to hide the complexity inside the methods.
We can summarize the philosophy of SyQL in the following sentence: give more power than is offered by the DBMS in a smart way. More in detail, the SyQL data model distinguishes between internal and external methods.
Therefore, the problem faced by SyQL is the following: add new functionality to the query language in SQL while using the SQL engine's performance as much as possible.
In the current SyQL implementation, we have tried to solve this problem by doing a partial translation of the SyQL queries into SQL ones.

We proceed as follows: in Section 2, we introduce SyQL and its main aspects. Section 3 describes how a SyQL query is processed. Section 4 presents the abstraction layer that we have created for mapping the data from a relational database of an AISEMA system into an Object Oriented environment. Section 5 discusses the results coming out from the empirical validation of SyQL. Finally, section 6 draws the conclusion of our work and shows future directions.

2 SyQL

In this section, we introduce SyQL using examples and define the syntax.

2.1 SyQL By Way of Example

As a running example, consider a software company *FancyComp* that develops software *ERP*.
In this company, AISEMA software is installed, which is continuously and automatically collecting product and process metrics coming out from the

development of *ERP*. In addition to that, there are at least two developers involved in the development process of the software: John Doe from U.S.A. and Mario Rossi from Italy.
Since these two developers are working remotely in two distinct countries, we can assume that they never have had pair-programming sessions. And for this reason, a project manager (hereinafter *PM*) may want to look at classes of the *ERP* code that can be a source of problems. To have a preliminary list of these classes, the *PM* can use the data stored in the AISEMA relational database. However, since this database contains more than one hundred tables, this can be a hard even for a *PM*, who may be a person skilled in the art of relational databases. To overcome this issue, we provide him/her the following schema based on methods, object types, and values, which can be used through SyQL query language:

```
• Class {  getEffort(start, end) ClassEffort,
           hasBeenModified(start, end) boolean,
           getFullName() string, … }
• ClassEffort {  isSpentBy(string) boolean,
                 getValue() int, … }
• ClosedBug { getDescription() string,
              getModifiedClasses() Class[], … }
• Method { getDefiningClass() Class, … }
```

It is important to say that object types contain concepts, and concepts are object types that are responsible for the mapping between the schema defined upon and the AISEMA relational tables. In the defined schema, the object types *Class, ClosedBug,* and *Method* are concepts, whereas the *ClassEffort* type is only an object type, which makes possible the manipulation of measures that require a certain context to be correctly interpreted just like the effort.
An instance of concept *Class* or *Method* represents a class or a method of the *ERP* project, and an instance of object type *ClosedBug* represents a bug reported in the *FancyComp* bug tracking system that is now resolved.
The definition of the methods of the object type is provided in the following description.
By using the data-model defined above, the *PM* can write the following query:

```
[1]   FROM   Class c
[2]   WHERE  c.getEffort(TODAY - 1'week',TODAY).isSpentBy("Mario Rossi")
[3]    AND   c.getEffort(TODAY - 1'week',TODAY).isSpentBy("John Doe")
[4]    AND   c.hasBeenModified(TODAY - 1 'week', TODAY)
[5]   SELECT   c.getFullName();
```

Listing 1: SyQL query that selects cross-modified classes.

The query shown in Listing 1 returns a result that is a list of strings containing the class names. In line 1, the range variable *c* is declared as a set of instances of concept *Class* representing the classes constituting the *ERP* software. In lines 2-3, the range variable *c* is used for getting an instance of object type *ClassEffort* for each class through the *getEffort(..., ...)* method by specifying the last week time interval. The *ClassEffort* instance contains the data about the effort spent on a specific class in the last week, so that it is possible to retrieve the name of the developer who had spent

this effort by using the method *isSpentBy(String)*. In line 4, it is specified a condition for checking if the *ERP* class has been modified in the last week. Finally, in line 5, the *getFullName()* method is invoked by printing out all the names of the classes that satisfy the criteria specified in lines 2-4 described above.
By looking at the results, the *PM* can see the classes that may present quality issues coming from an excessive manipulation by the developers. Therefore, these classes can be good candidates for a code review. However, the AISEMA database can do more for supporting the *PM* in this kind of tasks. By using the concept *ClosedBug*, the *PM* can join those results with historical bug data of the *ERP* project. To that end, Listing 2 shows a SyQL query that returns a subset of the *Class* of the previous query by joining those instances to *ClosedBug* instances.

```
[1]   FROM  Class c, ClosedBug cb
[2]   WHERE cb.getModifiedClasses().contains(c)
[3]     AND c.getEffort(TODAY - 1'week',TODAY) IS HIGH
[4]     AND c.getEffort(TODAY - 1'week',TODAY).isSpentBy("Mario Rossi")
[5]     AND c.getEffort(TODAY - 1'week',TODAY).isSpentBy("John Doe")
[6]     AND c.hasBeenModified(cb.getCloseTimeDate(), TODAY)
[7]   SELECT   c.getFullName(), cb.getDescription();
```

Listing 2: SyQL query that selects the fixed classes that have received a high quantity of effort in the last week by both developers.

In addition to the query shown in Listing 1, the query in Listing 2 specifies one more range variable (*cb*) of *ClosedBug* concept in line 1, two more conditions in line 2 and 3, and one more method call in line 7.
In line 2, we specify a join condition between the two range variables *c,cb* by gathering the data collected from the *ERP* bug tracking system. The *getModifiedClasses()* method returns an array of *Class* instances representing the classes involved in the bug fixing process. In line 3, a fuzzy condition [16] has been used; this condition evaluates the *ClassEffort* instance returned by the *getEffort(..., ...)* method by computing the truth value of the *High* effort fuzzy membership function [16] related to classes. This membership function takes as parameters the absolute value of effort (minute*men), the duration of the interval specified from the arguments, and the number of developers involved in the effort. Finally, in line 7, the description of the bug is printed out by invoking the *getDescription()* method together with the class name.
In the next subsection, the SyQL syntax is described by looking to another query, which can be useful for tracing the daily activity of the developers by helping the *PM* to keep the *ERP* code-base under control.

2.2 Language Syntax

The SyQL query shown below (Listing 3) returns a collection of class names, the related effort spent by the developers since yesterday, and the number of methods for each class.

```
[1]   FROM Class c, Method m
[2]   WHERE    c.getFullName() = m.getDefiningClass()
[3]     AND c.getEffort(YESTERDAY) IS High
[4]   SELECT   c.getFullName(),
[5]            c.getEffort(YESTERDAY, TODAY),
[6]            COUNT(m)
[7]   GROUP BY c.getFullName(), c.getEffort(YESTERDAY, TODAY);
```

Listing 3: Sample SyQL Query.

This query uses two concepts: *Class* and *Method*. A SyQL concept is a self-defining entity that can expose some methods, which can be used in SyQL queries. Each concept is characterized by comprehending a defining SQL query. The relation defined by this SQL query contains the records from which the concept instances are created.

In a SyQL query, a concept is the minimum part of the query, i.e. a user can query some data if and only if these data are part of a concept.

In the current implementation, a concept is a Java class comprehending all the method definitions that can be called during the execution of a SyQL query.

Moreover, the query above also shows the most important elements of the language that we are going to describe.

First of all, we underline that the lexical structure of SyQL is similar to SQL, but there are some differences: firstly, the *Select-Clause* is after the *Where-Clause*; secondly, SyQL works directly on objects, and nested method calls are possible (see line 3 of Listing 6 on page 12).

In the first row of a generic SyQL query, the user must specify the *From-Clause*, which shall contain one or more *From-Element*(s). Each of them is composed by two literals: the former identifies the concept type, whereas the latter declares the concept name (like in SQL), and is also called range variable.

In the second and third rows of the query shown in Listing 3, we introduce the *Where-Clause*. In the example, there are two conditions: an equality-join condition and a fuzzy condition.

The fuzzy condition (line 3 of Listing 3) evaluates the effort spent yesterday by the developers according to a predefined membership function. In the fourth, fifth, and sixth rows, the *Select-Clause* is shown. This is a non-empty collection of comma-separated *Method-Call*s and/or *Aggregation-Functions*.

In the last row, we declare the *Group-By-Clause*, which has the same meaning as the SQL *Group By*.

As already mentioned before, if we compare the syntax of an SQL query with the corresponding one in SyQL, the most evident difference is the order of *Select-Clause*, *From-Clause*, and *Where-Clause*. This is necessary because it enables us to perform code auto-completion both in the *Select-Clause*, and in the *Group-By-Clause*. This choice has been adopted (for the same reason and with some minor differences) also by two other query languages: .QL [12], and LINQ [11]. For the same reason, the *Group-By-Clause* has been kept after the *Select-Clause*. Since the "auto-completion/code completion" is a very important feature. Robbes *et al.* 2008 [14] and Murphy *et al.* 2006 [13] show its importance in a production environment[1].

[1] Murphy *et al.* [13] monitored 41 developers who use Eclipse as IDE. They found that the 6.7% of the number of executed commands are for auto-completion. The auto-

Sometimes it is impossible to answer a particular question in a single query, so it is necessary to write more than one query. As in SQL, SyQL allows the user to make use of set operators like *UNION*, *EXCEPT*, and *INTERSECT*. These operators are very important, because they enable the user to perform basic set operations. Moreover, since they are already present in SQL, we have kept the same meaning for them, in order to provide users with a smoother learning curve.
The same happens for the expressions. To avoid learning problems, the structure of expressions is similar to the corresponding SQL expressions, except for the *Fuzzy-Expression*(s).
The *Fuzzy-Expression*(s) allow the user to express filtering conditions by using the fuzzy equal operator and linguistic variables. We want to stress that it is possible to define new linguistic variables during the entire life cycle of SyQL.
To avoid capitalization problems during the query writing process, the terminal symbols of SyQL are case insensitive (e.g. '**from**', '**From**', '**fRom**', and '**FROM**' are all matched as <*FROM*> token). This makes it easier to write queries.

3 Query Processing

In this subsection, we are going to show how a SyQL query is processed. A detailed description of the techniques adopted is given, and examples are provided.

3.1 Query Engine

The SyQL query engine sits on top of the AISEMA DBMS, and runs on a Java Virtual Machine (JVM). This architecture makes possible to implement a query engine without developing a sophisticated query planner/executor and to develop methods that cannot be evaluated inside the database. Indeed, we say a method is *external* if can be evaluated both inside the database and inside the JVM, and a method is *internal* if can be evaluated only inside the JVM.
The main idea at the base of this implementation was to push as many conditions as possible into the query engine of the underlying DBMS. In this way, the implementation of a query planner and executor becomes very simple and compact. Now we are going to describe the basic steps performed by the SyQL query engine to map the condition from the SyQL query to the relational DBMS.
The main step for converting the conditions present in the *Where-Clause* is the conversion of the *SyQLExpression* in Conjunctive Normal Form (CNF) [9] by using Double negative elimination, distributive property of AND, and De Morgan's laws.
The CNF notation is very helpful, because a clause (conjuncts) of the CNF formula can be processed by the underlying DBMS only if all the conditions (literals) inside the clause are evaluated as external, otherwise the clause must be evaluated by the SyQL query engine. A condition is evaluated as an external one if all the predicates (the method calls) of the condition are external, otherwise a condition is evaluated internally. If all the conditions of a query are evaluated as external, we have a complete translation of the SyQL query into a SQL query with no specific engine processing.

completion came sixth among the top command executed.

For this purpose, all the methods of a concept class that can appear in the SyQL query are annotated in two different ways: external or internal (see Listing 4 on page 8). We say a method is "external" if the returned value is present in one column of the defining SQL query, otherwise, it is "internal". If a method is "external", it is necessary to specify in the annotation the field name that identifies the column of the defining SQL query.

3.2 Query Data Flow

In Fig. 1, the query execution workflow is shown. After the user has sent the query to the SyQL engine, the query text is parsed by the SyQL parser, which is an LL(k) stack automaton [1].

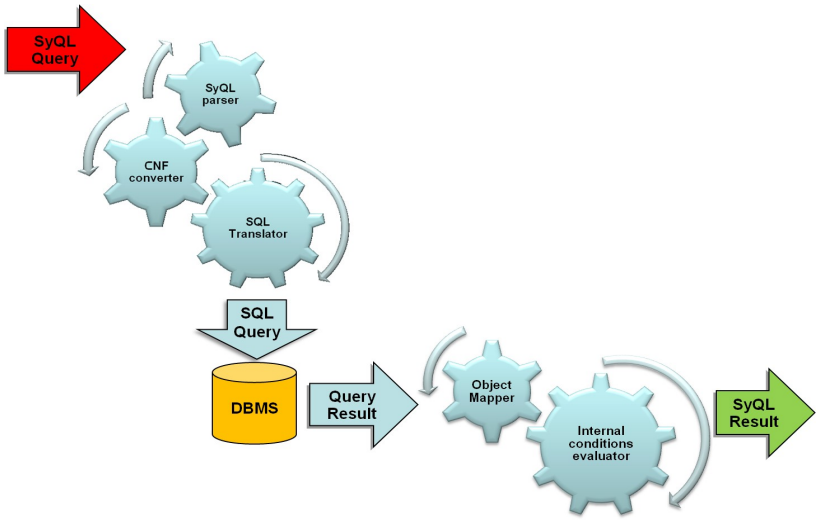

Fig. 1. The SyQL Query Workflow.

Next, the parse tree is processed by the *ReflectionVisitor* (not shown) for mapping range variable declarations to concept implementations. Then, the conditions are converted into CNF format (CNF converter). After that, the *SQLConverterVisitor* (SQL translator) visits the AST, so that the SyQL query is translated in SQL. This SQL query is executed against the SQL underlying database, hence the returning result set is a super set of the final one. The query engine fetches these records into the Java runtime by using JDBC driver. The *InternalConditionEvaluator* evaluates and discards the records that do not satisfy the internal CNF clauses of the *Where-Clause* expression, and if there is at least one fuzzy condition it also computes the truth value by applying Zadeh's rules [16]. Finally, the *FinalExpressionEvaluator* (not shown) visits the elements in the *Select-Clause,* and it computes the final query results. If aggregate functions are specified, the engine applies these operators to the groups defined in the *Group-By-Clause* (as in SQL). The grouping is performed by following the order of the elements specified in the *Group-By-Clause.* All the elements present in the *Group-By-Clause* must be present in the *Select-Clause*. We

want to stress that by design of the SyQL engine, the query result coming out from the execution of a SyQL query is a collection of Java objects, which can be simply printed out or easily reused inside the Java runtime environment.

3.3 SQL Translation

In this subsection, we show in practice how a SyQL query is translated into SQL. The definitions of the concepts involved in this example are provided in Listing 4, and the example of query conversion is provided in Listing 5.
In this query, the *PM* wants to retrieve all the classes of the "*MyNamespace"* namespace together with the effort data of the last two days, wherein the classes have "Facade" as a name prefix or receive high effort in the last two days by the developers.
In a nutshell, the *SyQLExpression* (lines 02-04) of the original version of the query is transformed into an equivalent CNF expression (lines 02-08 of the CNF version). Finally, the formula converter performs the SQL translation converting only the external CNF clauses. In the CNF version, the clauses at lines 04-05 and 08 are considered external, since they contain only external conditions (see the method annotations in Listing 4), therefore they are converted in SQL and executed against the AISEMA DBMS. The remaining conditions are evaluated by the internal condition evaluator. By using the CNF, the number of the conditions grows up, but it is not a limitation for the performance, because we use a caching mechanism for evaluating the conditions.

```
[1]   public class Class extends AbstractConcept {
[2]   ...
[3]      @ExternalCondition(columnName="class")
[4]      public String getClassName() {
[5]         return this.className;
[6]      }
[7]      @ExternalCondition(columnName="namespace")
[8]      public String getClassNamespace() {
[9]         return this.classNamespace;
[10]     }
[11]     @InternalCondition(cost = 20)
[12]     public ClassEffort getEffort(
[13]           SimpleDateTime begin,
[14]           SimpleDateTime end) {
[15]        ...
[16]     }
[17]     ...
[18]  }
```

Listing 4: Definition of concepts Class and Method (partial).

SyQL Original Query:

```
[1]   FROM     Class c
[2]   WHERE (  c.getClassName() LIKE 'Facade%' OR
[3]            c.getEffort(YESTERDAY, TODAY) IS High)
[4]         AND c.getClassNamespace () = 'MyNamespace'
[5]   SELECT   c.getFullName(),
[6]            c.getEffort(YESTERDAY, TODAY);
[7]
```

SyQL CNF version:

```
[1]   FROM  Class c
[2]   WHERE (  c.getClassName() LIKE 'Facade%' OR
[3]            c.getEffort(YESTERDAY, TODAY) IS High)
[4]         AND ( c.getClassName() LIKE 'Facade% OR
[5]               c.getClassNamespace () = 'MyNamespace')
[6]         AND ( c.getClassNamespace() = 'MyNamespace' OR
[7]               c.getEffort(YESTERDAY, TODAY) IS High)
[8]         AND   c.getClassNamespace() = 'MyNamespace'
[9]   SELECT   c.getFullName(),
[10]           c.getEffort(YESTERDAY, TODAY);
```

Translation to SQL:

```
[1]   SELECT *
[2]   FROM  (  SELECT DISTINCT e.name_namespace AS c_namespace,
[3]                           e.name_class AS c_class
[4]            FROM entity AS e
[5]            WHERE e.name_class IS NOT NULL AND
[6]                  e.name_method IS NULL
[7]         ) AS c
[8]   WHERE (  c.c_class LIKE 'Facade% OR
[9]            c.c_namespace = 'MyNamespace') AND
[10]           c.c_namespace = 'MyNamespace'
```

Listing 5: Example of query translation to CNF and then to SQL.

By looking at the SQL translation in Listing 5, it is possible to see that the SQL From-clause contains the defining SQL query of the *Class* concept, and the SQL Where-clause contains the translation of the external clauses. We say every occurrence of concept is translated into SQL by appending in the from-clause of the translated SQL query its defining SQL query as an inline view, and every external clause is translated into SQL by appending in the where-clause of the translated SQL query its SQL conversion performed by looking at the method annotations.
By looking at the defining SQL query of concept *Class*, it is possible to see that the data necessary to instantiate a *Class* object are fetched from *entity* table of the AISEMA DBMS. This table contains also the data for instantiating *Method* objects.
The *getEffort(..., ...)* method uses the data fetched from *entity* table to retrieve and aggregate the effort data from another table of the AISEMA database called *entity_property,* which has a column of *datatime* type.

4 Data Mapping

In this section, we describe how the data mapping for the already available SyQL concepts related to the Software Process Management works so far. For each concept, we show relevant methods, and for each of them, we provide a description.
The key concepts are the following: *Class, Method, TestMethod, Bug, ClosedBug,* and *User*. We consider these concepts the key ones, since the AISEMA system has been designed and implemented around them.
In the next subsections, we discuss how we have connected them by methods so that the user can combine these information by join-queries.

4.1 Class

The C*lass* concept allows the user to manipulate the code classes recognized by generic source code metrics extractors, like the ones presented by Scotto et al. [15]. *Class* provides methods to retrieve the following information: unique class identifier, software metrics, and effort metrics. The methods for retrieving effort data are the most important ones, since they allow to map a specific class to the users who have spent effort on it.
The effort is computed by using the following formula:

$$Effort_C(t_{start}, t_{end}) = \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} \sum_{m \in Methods_C} EffortEvent_{m,i}(t) + \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} EffortEvent_C(t)$$

$N :=$ Number of developers
$C :=$ Specific class
$t_{start} :=$ Beginning of the observation interval
$t_{end} :=$ End of the observation interval
$Methods_C =$ Set of methods defined into the scope of class C
$EffortEvent_{m,i}(t) =$ Quantity of effort (in seconds) spent in the timeframe t on the method m by the developer i-th
$EffortEvent_C(t) =$ Quantity of effort (in seconds) spent in the timeframe t inside the general scope of the class C

Formula 1: Effort definition for classes.

To make easier effort retrieval, we have implemented three different methods:

- *getEffort()*: it returns the effort spent on the class from its creation to now ($t_{start} = 0$, $t_{end} =$ now);
- *getEffort(SimpleDateTime endTime)*: it returns the effort spent on the class from its creation to the specified temporal parameter; ($t_{start} = 0$, $t_{end} =$ *endTime*);
- *getEffort(SimpleDateTime startTime, SimpleDateTime endTime)*: it returns the effort spent on the class within the specified interval ($t_{start} =$ *startTime*, $t_{end} =$ *endTime*).

With these methods, we can easily navigate along the temporal line by showing software metrics and effort evolution.
Another set of important methods are ones that retrieve the software metrics like the CK metrics [6], and the lines of code (LOC). For each type of software metric extracted by the code analyzer, the class provides two distinct methods: the former without parameters (e.g. *getLOC()*) that retrieves the last metric values, the latter with a temporal parameter that retrieves the value of the metrics at the specified time (e.g. *getLOC(SimpleDateTime)*). We also implemented a specific method (*getDeltaLOC(SimpleDateTime startTime, SimpleDateTime endTime)*) that returns the variation of lines of codes within a specified interval.
The *getFullName()* method concatenates the namespace name of the class if any with the class name.

Finally, we have defined the methods *hasBeenModified(SimpleDateTime)* to manage class modifications. This method returns true if the class has been modified during the day specified as argument, otherwise it returns false. In addition to that, a two-parameters version of *hasBeenModified* has been developed for checking time intervals larger than a day.
All the methods described upon are internal except the *getFullName()* method.

4.2 Method and TestMethod

The concepts *Method* and *TestMethod* allow the user to manipulate the methods of the classes recognized by the Source Code Metrics Extractor. These concepts provide methods to retrieve the same information of the concept *Class:* there are two methods for each software metric, and three methods for the effort. Like in the *Class* concept, the most interesting methods are the ones for retrieving effort data, which are finer grained than the ones defined in the *Class* concept, since a method is usually contained in a class.
The semantic of the effort methods (*getEffort(...)*) is the same but the effort is defined in a different way. In the following, we provide the effort definition for the *Method* and *TestMethod* concepts:

$$Effort_M(t_{start}, t_{end}) = \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} EffortEvent_{M,i}(t)$$

Where
$N :=$ Number of developers
$M :=$ Specific method
$t_{start} :=$ Beginning of the observation interval
$t_{end} :=$ End of the observation interval
$EffortEvent_{C,i}(t) =$ Quantity of effort (in seconds) spent in the timeframe t on the method M by the developer i-th

Formula 2: Effort definition for methods.

Anyway, the metrics available here are different from *Class* metrics. The collected software metrics for methods are the following: Lines Of Code (LOC), McCabe Cyclomatic Complexity (CC) [10], Number of input of the procedure (Fan-in), Number of output of the procedure (Fan-out), Halstead Volume [8], Number of Invocations for a method (NMI), and Number of Method Calls for a method (NMC).
In Java and C# code, the distinction between *Method* and *TestMethod* is done by reading the code annotations; the source code analyzer of the AISEMA system reads the annotations from the code, and it stores them into the appropriate tables of the data warehouse.
The *TestMethod* class allows the user to see which methods are tested by using the method *getTestedMethods()*. This method returns a *Vector* that contains all the methods invoked by the current *TestMethod* instance. These two concepts are very useful to check the quality of the code.

```
[1]   FROM Method m, TestMethod t
[2]   WHERE    NOT m.isTestMethod() AND m.isTested() AND
[3]            t.getTestedMethods().contains(m)
[4]   SELECT m, m.getCC(), COUNT(t)
[5]   GROUP BY m, m.getCC();
```

Listing 6: SyQL query that returns for each method the Cyclomatic Complexity and the number of test methods.

By using the query shown in Listing 6, the *PM* can assess the current status of testing of *ERP* project. For each tested method, the SyQL engine retrieves the McCabe Cyclomatic Complexity and the number of tests. This can be useful to identify a part of code that may not meet quality requirements.
Finally, as well as for *Class*, we have defined the methods *hasBeenModified(...)*.
Like in Class, all the methods described upon are internal except the naming ones.

4.3 Bug and ClosedBug

The user can query data coming from the bug tracking system by using these two concept classes.
The concept *Bug* exposes external methods that return typical bug information: the unique identifier, the opening time and date, the current state of bug (unassigned, assigned, resolved, invalid, etc.).
The concept *ClosedBug* adds the information coming from the bug fixing process, which are the resolver name/surname, the closing time and date, and the modified classes/methods. This last feature is the most important one, because it allows to join data coming from two different sources, i.e. from the source code and from the bug tracking system, and it is implemented as an internal method.

4.4 User

The concept *User* represents the entire set of users who are present into the AISEMA system. For each of them, the system collects automatically the effort spent on different artifacts: documents, web-pages, source code files, methods, etc. The effort tracking is the most interesting feature of this concept, since allows to map a specific user to specific *Method*(s) and *Class*(es). The actual implementation of this concept provides two important internal methods:

- *getEffort(SimpleDateTime start, SimpleDateTime end)* returns the value of the effort spent by the current user on code artifacts in the specified time interval;
- *getEffortMethod(SimpleDateTime start, SimpleDateTime end)* returns a *Vector* that contains the name of the artifacts, on which the current user has spent his/her effort in the specified time interval.

The other methods return the user's unique identifier and the user's surname, and they are external.

5 Validation

In this section, we briefly describe the two controlled experiments performed by us [3] for validating the benefits coming from the adoption of SyQL as a query language. After that, we present the main outcomes of these experiments, which were designed as a pair to compare the two couples of languages (SyQL Vs SQL and SyQL Vs LINQ [11]) preventing interference coming from users taking part to both SQL (or LINQ) and SyQL query design.

5.1 Experimental Setup

The two experiments had a similar design, and the differences are evidenced in the following description. In both, we wanted to compare the effort (minute*men) and the size of the code (LOC) necessary to complete a predefined set of process data extraction tasks by using both SyQL and SQL (or LINQ), taking into account the correctness of the resulting queries. The experiments variables are the following:

- *dependent variables*: task execution time (minute men), query length (LOC), query correctness (Boolean variable);
- *controlled variables*: two sets of tasks (on two distinct databases);
- *independent variables*: query languages (SyQL/SQL or SyQL/LINQ).

The tasks' execution times and the queries' LOC are two directly measured dependent variables. Tasks' execution time can be easily measured by collecting for each task the start and the end times. Lines of Code (LOC) can be counted offline (i.e. after the end of the experiment) by using a common predefined criterion for both languages.
During the experiments, the queries' texts and start/end times were both collected. After collecting all these data, we were able to compute the indirect dependent variables (one per query). Each of them is Boolean, and its value is true if and only if the corresponding query is correct. Correctness was evaluated by manually checking the queries' texts; a query was considered correct if and only if it completely fulfilled the requirements of the corresponding task. It is important to remark that we did not look only at the result-set produced, because it is not an error-proof method: a query can be wrong even producing the same result-set of a correct one.
As subjects for the first experiment, we took twenty-one (21) undergraduate students, attending an undergraduate course in Software Engineering Project (SEP) management. For the second experiment, we took nine (9) graduate students, attending a graduate course in Service Oriented Architecture (SOA).
During both experiments, the students were split into two groups, called “control” and “experimental”.
Each experiment consisted of two sessions. During each session, all subjects had to complete the same task list: the control group did it by using SQL (or LINQ), whereas the experimental group used SyQL. Hence, at the end of both sessions, each subject had taken part both to the control and to the experimental group, trying to complete two distinct lists of tasks. We want to stress this point, because this is an important choice of experimental design we took in order to avoid interference, i.e. the experience gathered by a subject when querying the database. This experience includes different information: field names, relations/classes names, result-sets, tasks

break-downs, etc. For this reason, we also changed database at the end of each session. In addition to that, we want also to stress that the tasks were sorted in order of difficulty, i.e. by increasing the number of SQL (LINQ) features necessary to complete the tasks.
At the beginning of the first session of the first experiment, all 21 subjects filled in a questionnaire to assess their SQL skills. Answers showed that all the subjects had already some experience with SQL, which was not a surprise because SQL is one of the main topics of a first year undergraduate course. In addition, 11 of the 21 students had already used SQL to solve real world problems. On average, the subjects' SQL skills were good, and almost all our technical questions about SQL syntax and semantics were correctly answered (4.8% of wrong answers). On the contrary, in the second experiment, all nine students had no previous experience in LINQ.
During each session, before starting the experiment, subjects belonging to the experimental group received a 20-minute presentation about SyQL, in which the language was introduced by means of examples. In the second experiments, students also received a 20-minute presentation about LINQ.
At the end of both sessions, subjects who were part of the experimental group (i.e., the SyQL users) were requested to fill in a second questionnaire about their "SyQL experience". On average, the experience of using SyQL was considered "Good" by the students who took part to the two experiments (the scale was Excellent, Good, Fair, Poor). Also, no bugs were reported by the subjects that took part in the experiments.

5.2 The Results

In the data coming from the two experiments, we observed the "experience factor" influence and the benefits provided by SyQL.
In the SyQL vs. SQL experiment, after an initial assessment of the data, we decided to analyze only correctness data for one main reason: the users have more experience by using SQL than SyQL, hence comparing the efforts between the two groups may be considered invalid.
In the SyQL vs. LINQ experiment, the situation was opposite, because the users have no experience both in SyQL and in LINQ. Therefore, the comparison of the efforts was more significant than the efforts collected in the first experiment.
The common benefit provided by SyQL was the improvement of the quality of the queries written by the users. The correctness improvement ranges from 9.5% to 66.67% depending from the complexity of the task, and the reduction of the lines of code is up to 52.63%. In addition, the users require less effort to start using SyQL than LINQ, and they obtain better results in terms of correctness.

6 Conclusion and Future Work

This paper presents and describes a query language called System Query Language (SyQL). This language can provide to the final user an easier schema based on methods, object types, and values than the one provided by an underlying relational database. Specifically, we present a schema designed for mapping data from an

AISEMA database. Finally, the benefits resulting from the adoption of SyQL are presented by referring to the results of two controlled experiments.
SyQL improves upon its competitors .QL [12] and LINQ [11] thanks to its flexible data model, which allows to model different domains by hiding the complexity into methods and classes. In the future, the SyQL query engine will be enhanced, so that it will be able to translate into SQL also the aggregate functions whenever it will be possible. In this way, we want to reduce the quantity of data fetched from the DBMS by speeding up the SyQL query execution.

Acknowledgment

I would like to thank my supervisor Prof. Werner Nutt for his advice and guidance when writing this paper.

References

[1] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Chapter 5 - One-pass no backtrack parsing. Prentice-Hall, Inc. (1972)
[2] Bachmann, A., Bernstein, A.: Software process data quality and characteristics: a historical view on open and closed source projects. Proc. of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops (IWPSE-Evol '09), pp. 119--128 (2009)
[3] Bianco, M., Damiani, E.: SyQL: A Controlled Experiment Concerning the Evaluation of its Benefits. In: Proceedings of the Annual Conference on Software Engineering (SE 2010), pp. 61--66 (2010)
[4] Colombo, A., Damiani, E., Frati, F., Oltolina, S., Reed, K., Ruffatti, G.: The Use of a Meta-Model to Support Multi-Project Process Measurement. In: Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC '08), pp. 503--510 (2008)
[5] Coman, I.D.: Adoption and Usage of Automated In-process Software Engineering Measurement and Analysis Systems. Brossura (2008)
[6] Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE TSE 20(6), pp. 476--493, (1994)
[7] Fujigaki, Y.: Stress analysis: A new perspective on peopleware. American Programmer 6(7), pp. 33--38, (1993)
[8] Halstead, M.H.: Elements of Software Science. Operating and programming systems series 7. Elsevier Science Inc. (1977)
[9] Marques-Silva, J., Guerra e Silva, L.: Solving Satisfiability in Combinational Circuits. IEEE Design and Test 20(04), pp. 16--21 (2003)
[10] McCabe, T.J.: A Complexity Measure. IEEE TSE vol.SE-2(4), pp. 308--320 (1976)
[11] Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 706--706 (2006)
[12] Moor, O. d., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J.: Keynote Address: .QL for source code analysis. In: Proceedings of the Seventh IEEE international Working Conference on Source Code Analysis and Manipulation, pp. 3--16 (2007)
[13] Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE?. IEEE Software 23(4), pp. 76--83 (2006)
[14] Robbes, R., Lanza, M.: How Program History Can Improve Code Completion. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 317--326 (2008)
[15] Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. Journal of System Architecture 52(11), pp. 668--675 (2006)
[16] Zadeh, L.A.: The Concept of a Linguistic Variable and its Application to Approximate Reasoning. Information Science 8, pp. 199--249 (1975)