Computer Languages, Systems & Structures 41 (2015) 1–20

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A type-safe embedding of SQL into Java using the extensible compiler framework J%

Vassilios Karakoidas *, Dimitris Mitropoulos, Panagiotis Louridas, Diomidis Spinellis

Department of Management Science and Technology, Athens University of Economics and Business, Pattision 76, GR-104 34 Athens, Greece

ARTICLE INFO

Article history:
Received 6 August 2014
Received in revised form
10 December 2014
Accepted 3 January 2015
Available online 10 January 2015

Keywords:
Domain-specific languages
Programming languages

ABSTRACT

J% is an extension of the Java programming language that efficiently supports the integration of domain-specific languages. In particular, J% allows the embedding of domain-specific language code into Java programs in a syntax-checked and type-safe manner. This paper presents J%'s support for the SQL language. J% checks the syntax and semantics of SQL statements at compile-time. It supports query validation against a database schema or through execution to a live database server. The J% compiler generates code that uses standard JDBC API calls, enhancing runtime efficiency and security against SQL injection attacks.

1. Introduction

Domain-specific languages (DSLs) [1–4] are designed specifically to address problems in a particular domain. Furthermore, they are used to improve the efficiency in a software development process [5,6]. Well-known DSLs include regular expressions and SQL.

General-purpose languages (GPLs) have a wider scope, providing a set of processing capabilities applicable to various problem domains [4]. Typical examples of GPLs are Java, C++ and Scala.

Modern software engineering paradigms indicate that DSLs are often used together with GPLs [7–10]. The integration of SQL with various GPLs constitutes a field that drew the attention of researchers and practitioners for many years [11–14]. This integration in the context of Java is realised with a JDBC (Java Database Connectivity) application library [15]. By using it, the programmer has to pass the SQL query to the database as a string. Through this process, the Java compiler is completely unaware of the SQL language contained within the Java code and usually many SQL syntax and type errors are detected at runtime. Such errors remain undetected, even with extensive testing during the development process.

J% (pronounced J-mod and stands for modular Java) is an extension of the Java programming language [16] initially discussed in reference [17]. Its main contribution resides in the development of a generic framework, by which arbitrary DSLs can be embedded in Java programs in a type-safe, syntactically correct fashion. DSL can be included on demand as module plug-ins.

* Corresponding author.
E-mail address: bkarak@aueb.gr (V. Karakoidas).

http://dx.doi.org/10.1016/j.cl.2015.01.001
1477-8424/

This work analyses the case of SQL integration. The prototype implementation focuses on MySQL database backend, but in the future, we plan to expand the module's support to other RDBMS too. The key contribution points are the following:

Query validation: The SQL queries are syntactically checked, and optionally can be validated against a specified SQL database schema. In addition, the queries can also be executed in a live database environment at compile-time to provide real-world testing.

Type safety: The integration between the two languages is type-safe, and all problems are reported at compile-time. The current implementation uses J%'s type mapping facility, which enables the declaration of compatible type relations between languages. This approach follows the standard type mapping conventions proposed in the JDBC APIs (Application Programming Interfaces) [15].

Query compile-time configuration: Each SQL query can be separately configured with different compile-time and runtime features. This is implemented with the utilisation of the *external configuration* concept, which was introduced by the J% compiler architecture [17].

Support for the SQL "in" operator: The current JDBC standard does not support the SQL operator `in` conveniently. Our approach supports it in a better, type-safe way. It simplifies the usage of this operator, which normally requires the programmer to write additional code to handle the translation from the Java composite types to an SQL set.

Compatibility & security: The SQL module uses the existing JDBC specification and does not require from programmers to learn and understand new APIs. The generated code utilises *prepared statements*, therefore securing the application against SQL injection attacks.

2. Motivation

Consider the case of a Java application that stores its data in a relational database. The class `SQLSimpleExample` contains a method called `execSQL` that accepts a string type formal parameter, labelled `id`. A standard implementation for the Java programming language is presented in the listing below:

```
public class SQLSimpleExample {
   public void execSQL(String id) {
      try {
         Statement stmt = connection.createStatement();
         ResultSet rs = stmt.executeQuery(

                  "SELECT * FROM t where id = '" + id + "'");
         [...]
      } catch (SQLException e) { /* error */ }
   }
}
```

The value passed by the parameter is concatenated with the SQL query string. From the previous listing, three issues arise:

1. The SQL query is passed as a `String` and the compiler is not aware that this is an SQL query at compile-time.
2. The parameter (`id`) is concatenated to the SQL query and its actual value is never checked.
3. The above implementation is vulnerable to SQL injection attacks [18].

With heavy duty unit testing [19] and good code coverage some of aforementioned issues can be discovered and corrected in time, but the rest can lead to many undesired situations.

A common technique to address the above issues is the use of prepared statements. This is a JDBC feature that offers an API to parameterise an SQL query. By using the prepared statements, the above code would be transformed into the following listing:

```
public class SQLSimpleExample {
   public void execSQL(String id) {
      try {
         String sql = "SELECT * FROM t where id = ?";
         Statement stmt = connection.prepareStatement(sql);
         stmt.setString(1, id);
         ResultSet rs = stmt.executeQuery();
         [...]
      } catch (SQLException e) { /* error */ }
   }
}
```

In the code above, the effect of the second and the third issue is mitigated. The SQL query parameter is sanitised by the application library, and type safety is guaranteed at compile-time. But the first issue still stands, since the compiler does not syntactically

check the SQL code. To counteract the aforementioned issues, researchers have introduced various approaches and techniques. In the following sections we present their main characteristics, advantages, and flaws.

3. Approaches of integrating SQL with general-purpose languages

Related research in the field can be examined based on the five design integration patterns for GPLs and DSLs, defined by Mernik et al. [20] (see Table 1). The following section is organised by categorising the related research approaches according to them.

Table 2 summarises the approaches and for every approach provides information about two functional characteristics and one non-functional. These are:

Syntax (*functional*): The approach supports DSL syntax checking.
Typing (*functional*): Types are mapped between the DSL and the GPL and errors are detected at compile-time.
Complexity (*non-functional*): How complex is the approach for the average programmer that already knows the GPL. The complexity can be characterised as *low*, *medium*, or *high*. *Low* complexity denotes that a programmer with good

Table 1
Dominant design integration patterns as proposed by Mernik et al. [20].

Design pattern	Description
Design: Language Exploitation (extension)	The DSL extends an existing language with new data types, semantic elements, and syntax
Design: Language Exploitation (specialisation)	The DSL restricts an existing language for purposes of safety, static checking, and optimisations
Implementation: Extensible-Compiler / Interpreter	A GPL compiler / interpreter is extended with domain-specific optimisation rules and domain-specific code generation
Implementation: Embedding	DSL constructs are embedded in an existing GPL (the host language) by defining new abstract data types and operators
Implementation: Preprocessor	The DSL syntax is translated to constructs in an existing, host language

Table 2
DSL integrations score according to syntax checking, static typing and complexity.

Project	Language	Syntax	Typing	Complexity
Java	Java	✗	✗	Low
Python	Python	✗	✗	Low
ODB	C++	✗	✓	Low
Squeryl	Scala	✗	✓	Medium
Slick	Scala	✗	✓	Medium
SOCI	C++	✗	✗	Medium
JPA	Java	✗	✓	High
DTL	C++	✗	✓	High
Perl	Perl	✗	✗	Low
Hibernate	Java	✗	✓	High
Anorm	Scala	✗	✓	Medium
JOOQ	Java, Scala	✗	✓	High
Scala [21]	Scala	✗	✗	Low
JDBC Checker [22]	Java	✓	✓	High
SugarJ [23]	Java	✓	✓	High
SQL Detection Plugin [24]	Java	✗	✗	High
JSquash [25]	Java	✓	✓	High
SQL DOM [26]	C#	✓	✓	Medium
Switch [27]	Ruby	✓	✗	High
Haskell/DB [8]	Haskell	✗	✓	High
Cω [28]	C#	✗	✓	High
SIQ [29]	Scala	✓	✓	High
LINQ [30]	C#	✗	✓	Low
SQLJ [31]	Java	✓	✓	Low
Powerbuilder	Powerscript	✓	✓	Low
J% [17]	Java	✓	✓	Medium

knowledge of the base language could use the DSL integration framework with reasonable effort and *high* means the exact opposite.

3.1. Implementation: embedding

All mainstream programming languages, like Java, Perl, and Python, are using the *Implementation: embedding* pattern to integrate SQL. The concept of this pattern is simple; there is an application library, as JDBC in Java, that implements the SQL connectivity with the server. An API is provided for the host language to access it. The SQL queries are passed as strings to the application library and are executed in the server. All errors are reported upon execution at runtime.

ORM (Object-Relational Mapping) approaches like Hibernate [32], JPA (Java Persistence API) [33], (Scala Language-Integrated Connection Kit) [34], and Squeryl [35] provide a way to access a relational database, by introducing a programming layer, typically supported by a custom querying DSL, or by extending the host language, in order to automatically generate and execute the SQL code. This approach has several advantages; the database entities are closely coupled with the host language and the application is completely decoupled from any database backend. The main disadvantage of this approach is that the programmer needs to learn a new query language. In addition, this language is expressed via strings, thus it suffers from the same problems related to JDBC.

Anorm [36] is another Scala integration approach that enables database access within the Play! web application framework. It adopts the principle that SQL has more than enough features and language richness to cover all database access uses cases that an application could want; thus it offers a Scala library that enhances SQL integration, simplifying common usage patterns via a sophisticated API.

Similar approaches in C++ include ODB [37] and DTL (Database Template Library) [38] SOCI [39] is a C++ library that provides advanced functionality when working with SQL. Technically, it is a C++ extension, but it provides a series of techniques, only by using features of the standard C++ language, to enable SQL embedding; among them, basic query support and basic ORM capabilities.

The main difference of J% against these approaches is that it checks the SQL queries at compile time and reports errors directly to the programmer as detailed error messages. Also, it does not rely on a programmatic layer to efficiently map the SQL abstractions to appropriate GPL ones (e.g. the layers introduced by ORMS).

3.2. Implementation: preprocessor

For this approach, the DSL syntax is translated to constructs in an existing, host language. Typical implementation includes the preprocessors of the C and C++ programming languages, where the macros, which act as the DSL for this case, are translated into C or C++ respectively.

JOOQ [40] is a Java application library and it offers a unique API that maps all database elements into Java code. This is realised with a code generation utility. This utility scans the target database and generates all the models and the code that represent the database as host language types. The generated code uses standard JDBC API calls. The advantage of this approach is the direct mapping of each database table and field to a Java program element. For example, it is impossible for a programmer to mistype a table name or any other SQL identifier, since it is directly mapped into a Java class and the compiler. Hence, the IDE (Integrated Development Environment) will catch the error and present it to the user.

JDBC Checker [22] acts as a preprocessor and searches Java code for JDBC calls and SQL statements to detect possible errors. Notably, the SQL statements are checked against the database schema. Annamaa et al. [24] use Eclipse's compiler infrastructure to efficiently embed SQL queries in Java programs. The plug-in can detect common syntax SQL errors, including misspelled table or column names. To embed DSLs, Erdweg [23] proposes SugarJ, a framework used to extend GPLs with specific syntax. The main contribution of this framework is that can be applied on many languages as host languages (Java, Haskell and Prolog).

JSquash [25] is a tool that analyses Java code and several by-products of a Java application and trace the existence of SQL statements. Then, it automatically links the dynamic parts of SQL queries with the variables of the program. Typically, the tool aims to identify this linkage and replace the contents of these variables accordingly, when the database schema changes.

SQL DOM [26] acts as a preprocessor and translates an SQL database schema into C#. The generated collection of classes is used as an application library, thus ensuring type safety and syntax checking at compile-time.

Switch [27] is a compiler that can be used when developing in the Ruby programming language. Its goal is to provide an alternative for the Ruby on Rails active record library, focusing on query performance.

J% does not translate SQL into Java, or any other intermediate language. It generates Java code, which utilises and exposes to the programmer existing JDBC API calls.

The SQL statements are not altered and the type annotations are translated into prepared statements that use the JDBC mapping for types. Contrary to our work, the aforementioned frameworks generate code that should be included in the program during the compilation, or act as tools that help the programmer work with SQL.

3.3. Design: language exploitation (extension)

This integration pattern dictates that the host language is extended with new data types, semantic elements, and syntax to efficiently support the DSL.

Haskell/DB [8] is a host language variant that has been extended to encapsulate SQL queries. This implementation follows the *Design: Language Exploitation (extension)* pattern. SQL is completely hidden from the developer and queries are expressed through a custom syntax embedded to the host language. This approach hinders productivity and forbids domain experts to become involved with the development process.

Cω[28] integrates both SQL and XML into its syntax, extending the C# programming language and introducing numerous changes to its type system and syntax. SIQ [29] (Scala Integrated Query) follows a similar approach.

LINQ is an integrated component for the .NET languages (C# and Visual Basic) that adds querying capabilities to objects. It can be used to perform declaratively queries in collections of objects.

SQLJ [31] is a language extension of Java to support SQL. It offers type and syntax checking for both languages at compile-time.

These aforementioned frameworks use the GPL compiler to enforce syntax and type checking to the embedded DSL statements. J% adopts this concept. In particular, it allows the inclusion of modules to support an infinite number of DSLs. Thus, it provides a platform on which many DSLs could be used simultaneously and on demand by the programmer.

3.4. Design: language exploitation (specialisation)

This approach focuses on the specialisation of a GPL to efficiently integrate with a DSL. As a result, GPL syntax is enriched with operators and statements to enforce type safety and syntax checking between the integrated languages. A weakness of this approach is that the GPL syntax becomes very complex and loses its generalisation.

Powerscript is the core development language of the rapid application development tool, Powerbuilder [41]. Compile-time query checks against a live database schema are supported through an active database connection.

J% offers similar efficiency in terms of SQL integration, permitting compile-time type checking and execution to an active database, similarly with the aforementioned approach, but without sacrificing the GPL's general features.

3.5. Implementation: extensible compiler/interpreter

This approach proposes the extension of a GPL with domain-specific optimisation rules and code generation. The work presented here is implemented as a module for the J% compiler, which follows this integration pattern [17] to enable this functionality. The basic concepts of the J% language are presented in Section 4.

Several other works [42–46] adopt this approach, but are not focusing on supporting SQL efficiently or any other DSL. Actually, these works focus mainly on enforcing type safety and provide extensible language syntax. They are relevant to J% in terms of methodologies and techniques that are used to implement the language integration. Their similarities stop there, since the J% module exploit the language's features and mechanisms to implement novel functionalities, like compile-time SQL query configuration, etc. All these features are described in the following sections.

4. Design and implementation of the SQL module

The basic features of the J% programming language are presented in reference [17], which covers the basic features of J% like syntax and type mapping. It also presents a premature version of regular expressions and SQL integration, but in the level of a working prototype. The extended version of the SQL embedding module is presented here in more detail. In addition, these works cover all aspects of J%, like type mapping, type annotations and presents an evaluation of J% framework through a series of experiments.

The basic architecture of J% enables the development of compiler modules to support many DSLs. The following section establishes a terminology and describes the J% basic syntax and features. Then type mapping is discussed, focusing on supporting the SQL module. Finally, the module's specific technical aspects are examined in detail, like the overall compilation process and code generation schemes.

First, we need to establish the basic J% terminology:

External module: *External modules* are compiler plug-ins that enable DSL support. Each module exposes two basic elements; one or more *external types* and one *configuration type*. The modules are automatically invoked by the compiler, when it detects DSL usage in the J% program.

External type: They are user-defined Java reference types [16] with extended syntax, like Java enumerations that are marked by the keyword `external` and contain the DSL code. An *external type* always inherits the *external base type* or one of its subtypes. *External types* act like DSL code container. The compiler identifies them at compile-time and calls the specified *external module*. Its concept is similar to the *assimilation type*, introduced by Bravenboer and Visser in reference [42]. Are *External types* are subtypes from the *External Base Type*, which is offered by the J% framework.

Configuration type: Provides compile-time configuration for each *external type*. These typically represent compile-time and runtime options used for the generation and the execution of DSL code. These types are configure through type parameterisation. The concept of external type configuration is analysed in Section 4.2.

External reference: *External references* are type annotations between the Java code and the DSL code. They are used to define type mappings between the DSL statements and the Java code. This mechanism is thoroughly explained in Section 4.3.

4.1. The SQL module basics

The SQL support of J% is implemented in the form of a compiler module. Only one type is declared, namely the `SQLQuery`. The declaration of the `SQLQuery` type is the following:

```
public class ExternalBaseType<T extends ExternalConfiguration>{
    protected ExternalBaseType(T configuration) {...}
    public Map<String, String> getModuleRuntime() {...}
}

public abstract class SQLQuery<T extends SQLConfiguration>
                                    extends ExternalBaseType<T> {
    public String getSQLStatement();
    public abstract PreparedStatement getStatement(Connection c)
                                                throws SQLException;
}
```

The above code fragment contains the declaration of the `SQLQuery`. It extends the `ExternalBaseType` class. All new queries must be `SQLQuery` subclasses. The following notation describes the aforementioned hierarchy:

$$CustomQuery <: SQLQuery <: ExternalBaseType \tag{1}$$

The type must be parameterised by a descendant of `SQLConfiguration` type. The usage of configuration types is presented thoroughly in the next section. Its functionality includes two methods: `getSQLStatement`, which returns the SQL query as a string literal and `getStatement` that returns an instance of a prepare statement object, given an open JDBC database connection.

4.2. Configuring the SQL queries

SQL configuration parameters are used for the compile-time configuration of each SQL query. Practically, they initialise the SQL module with a different context that drives the module's functionality. Configuration types are organised as a hierarchy, such as the one described with the following notation:

$$CustomConfig <: SQLConfiguration <: ExternalConfiguration \tag{2}$$

The SQL module with the default configuration checks only the SQL syntax. Two more checking facilities can be enabled through the configuration system; one that checks the SQL query against the database schema (`SQLMOD_NS_AWARE`), and a second that performs query check against a live database (`SQLMOD_LIVE_TEST`), by executing the queries with default values.

Fig. 1 depicts a configuration type hierarchy based on the options provided by the SQL module. The configuration options typically affect compile-time options or runtime environment strategies. `SQLConfiguration` is the basic configuration type provided by the SQL module. `TestingConf` and `LiveTestingConf` are two subtypes that override and set specific compile-time options. These options are realised as class fields. The following code fragment exhibits the usage of the `TestingConf` configuration type. The `ExampleQuery` type will check the SQL query against the database schema provided by the file `schema.sql`, given as the value of `SQLMOD_NS_URI` configuration option.

```
public external ExampleQuery extends SQLQuery<TestingConf> {
    select * from customer
}
```

Some of the `SQLConfiguration` options are also available at runtime and can be retrieved via the `getRuntimeConfigurartion` method. The following listing contains the declaration of the basic `ExternalConfiguration` class.

```
public abstract class ExternalConfiguration {
    protected abstract Map<String, String> getModuleConfiguration();
    public abstract Map<String, String> getRuntimeConfiguration();
}
```

`SQLConfiguration` and all configuration types must directly inherit this type and provide an implementation for its abstractions. Table 3 summarises all available configuration options offered by the SQL module. If the column *Runtime* is

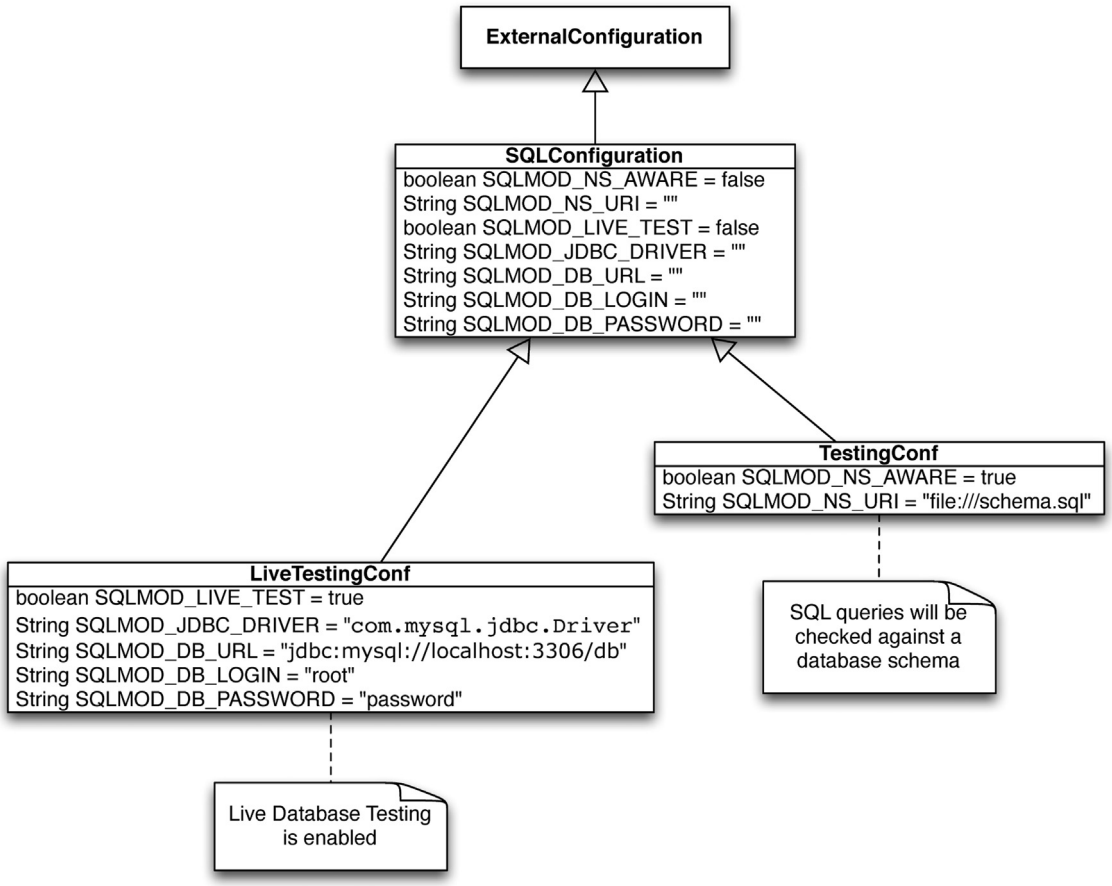

Fig. 1. A typical configuration type hierarchy.

Table 3
Configuration options.

Name	Description	Runtime
SQLMOD_NS_AWARE	SQL check with a database schema flag	✓
SQLMOD_NS_URI	Absolute path of the database schema file	✗
SQLMOD_LIVE_TEST	Live database testing flag	✓
SQLMOD_JDBC_DRIVER	The JDBC driver classpath	✓
SQLMOD_DB_URL	The database URL	✗
SQLMOD_DB_LOGIN	Login of database user	✗
SQLMOD_DB_PASSWORD	Password of the database user	✗

checked, then this configuration option is preserved in the runtime environment, if not, it is stripped during the compilation phase. Also note that the configuration options, which are related to the database connection, are used only in the case of live database testing (SQLMOD_LIVE_TEST).

4.3. Type mapping

J% offers a mechanism to support type mapping. To examine how it works, we introduce the following definitions:

Compatible types: All Java types that can be mapped to a series of DSL types and vice versa are defined as *compatible types*. A compatible type defines a mapping from the assimilated domain type from a DSL to an assimilating type from the GPL and vice versa [42].

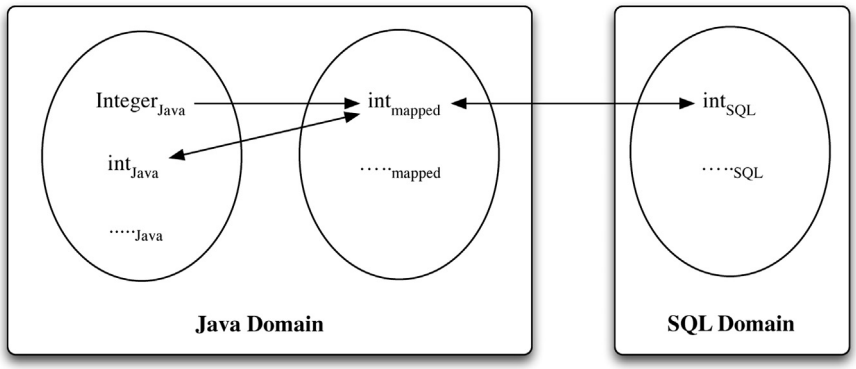

Fig. 2. How type mapping works in J% SQL.

Table 4
Type mapping between Java and SQL, based on MySQL data-type definitions.

Compatible	**Dominant types**		**Compatible**
Java	*Java*	SQL	SQL
java.lang.Byte	byte	tinyint	–
java.lang.Short	short	smallint	–
java.lang.Integer	int	int	integer, mediumint, year
java.lang.Long	long	bigint	–
java.lang.Float	float	float	–
java.lang.Double	double	double	–
–	java.lang.BigDecimal	decimal	numeric
java.lang.Boolean	boolean	bit	bit varying
–	java.lang.String	varchar	character, char, tinytext, text
–	byte[]	BLOB	tinyblob, mediumblob, longblob, longtext
–	java.util.Date	date	–
–	java.util.Timestamp	timestamp	time

Dominant types: Even if the type is compatible with other types, one type must be defined as *dominant* and be preferred when the compiler needs to resolve compile-time ambiguities regarding data conversion between the two languages. These types are referred to as *dominant types*.

Each module defines a series of *compatible* types that are used when the Java language and the embedded DSL need to exchange data. As an example, consider the case where an *integer* for the SQL language (int_{SQL}) that needs to be mapped to a Java type. The following conventions are required:

- the *int* SQL type should be mapped to the *int* Java primitive type as dominant,
- the *int* SQL type should also be mapped to the *java.lang.Integer* as compatible type.

Fig. 2 visualises the aforementioned mapping example. The boxes contain the types declared inside the scope of J% (Java types) and SQL. We declare types in the Java type system, which have a one-to-one relation with the correspondent SQL types. Then, we associate them with the Java types marked as "mapped" to provide the mapping information and conversion functions. The arrowheads illustrate the conventions that are used.

Table 4 lists all the compatible and dominant types for the SQL module. So far, the module supports on MySQL data types.[1] Each database backend should have its own custom type mapping rules, since it is common for RDBMS's to have unique data types. The types listed on the table provide the basic mapping between Java and MySQL. SQL sets and Java collections and composite types are also supported (see Section 4.6).

4.4. External references

In J%, each DSL maintains its own syntax. For simple cases, where the DSL has no type system, such as regular expressions, the DSL is used completely unmodified. When the J% language needs to pass values with the DSL, an *external reference* must be defined, which is a type annotation that describes the convention between the two type systems.

[1] http://dev.mysql.com/doc/refman/5.0/en/data-type-overview.html

```
ExternalRef:
   "#""[" ParameterName "]""<" FieldType ">"

ParameterName:
   Identifier

FieldType:
   Identifier { "." Identifier } { "[" "]" }
```

Fig. 3. External references syntax.

Table 5
External reference examples.

Parameters	Constructor
[char_v] < char >	TypeName(char char_v)
[p1] < int >, [p2] < Integer >	TypeName(int p1, Integer p2)
-No parameters-	TypeName()
[sh] < short[] >, [i] < int >	TypeName(short[] s, int i)

Fig. 3 contains the BNF syntax for the *external references*. The main rule is named `ExternalRef` and it consists of two parts: the *parameter name* and the *field type*.

Parameter name (`ParameterName`) defines the name of the parameter that is going to be used in the code generation phase. The *field type* (`FieldType`) defines the corresponding Java type, according to the naming convention that is commonly used by the Java programming language.

The number of *external references* is bound to the maximum formal parameters that the method can have. This is defined in the Java Language Specification [16].

For example, for an SQL query that accepts an `int` as a parameter as input (`c_id`):

```
public external SimpleSQL extends SQLQuery<SQLConfiguration> {
   select * from customer where customerId = #[c_id]<int>
}
```

The expression [c_id] < int > defines that the `customerId` expects an `int` base type. `[c_id]` is the parameter name and practically generates the following constructor for the `SimpleSQL` *external type*:

```
public SimpleSQL(int c_id) { ... }
```

Table 5 lists a series of *external reference* examples along with their generated constructors.

4.5. The compilation process

The J% compiler adopts the *language processing system* architecture [47], more specifically it follows the *pipe and filter compiler architecture* variation. Fig. 4 illustrates the compilation process. Both the J% compiler and the SQL module are implemented in Java. The compilation process is straightforward; the compiler scans the input source files (*.`jmod` and *.`java`). Then it marks each file as *external* or *Java*. The *external* keyword denotes that an SQL type is declared in this file, and the marker *Java* that it contained pure Java–compatible code. Configuration types are also Java files. All this information populates directly the symbol table and other type information structures. The code generator is invoked and intermediate Java-compatible code is generated. Finally, the Java compiler is invoked, and translates the code into executable JVM bytecode. All information regarding symbols, the *external configuration* and all the specifics of the external code are available to the module at compile-time. After its initialisation, the SQL code is further analysed and checked for discrepancies, which are reported as compile-time errors. Table 6 presents a set of size metrics for the SQL module.

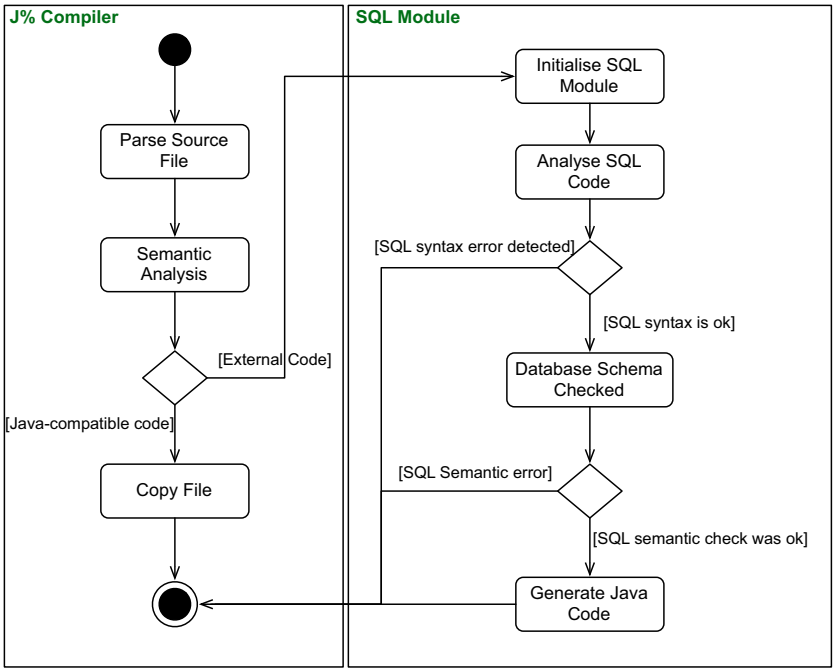

Fig. 4. The compilation process.

Table 6
SQL module size metrics.

Metric	Value
File count	23
Lines of code	2417
Package count	5
Number of classes	22
Number of interfaces	0
Number of enumerations	1

4.6. Code generation

Consider the following program:

```
package jmod.examples.simplesql;

import org.jmod.dsl.sql.SQLQuery;
import examples.simplesql.SimpleConf;

public external SelectExample extends SQLQuery<SimpleConf> {
  select * from sqlexample where sqle_primary = #[prim]<int>
}
```

The `SelectExample` *external type* extends the `SQLQuery` type. The query is written in SQL, enhanced by the usage of *external references*. `[prim]<int>` is the statement of an *external reference*. It declares that the SQL query accepts an integer parameter and thus a formal parameter will implement it, in the form of the generated type's constructor.

The declaration of the *external configuration* type, namely `SimpleConf`, follows:

```
package jmod.examples.simplesql;

import org.jmod.dsl.sql.SQLConfiguration;

public class SimpleConf extends SQLConfiguration {
    public boolean SQLMOD_NS_AWARE = true;
    public String  SQLMOD_NS_URI = "schema.sql";
}
```

The `SimpleConf` type parameter configures the compile-time part of `SQLQuery`. In this example, it instructs the compiler to check the query with the provided database schema. A part of this configuration is passed into the runtime environment, according to Table 3, enabling the developer to retrieve runtime information regarding the specific query. The configuration option `SQLMOD_NS_AWARE` will remain, indicating that the query has been checked against the specified database SQL schema. On the other hand, the `SQLMOD_NS_URI` should be omitted, since the path to the schema is specific to the compilation environment. The configuration type may be different per *external type* without compile-time or runtime overhead. The following listing contains the generated code.

```
package jmod.examples.simplesql;

import org.jmod.dsl.sql.SQLQuery;
import jmod.examples.simplesql.SimpleConf;

import java.sql.*;

public class SelectExample extends SQLQuery<SimpleConf> {
    private int prim;
    private String sqlStatement;

    public SelectExample(int prim) {
        super(new jmod.examples.simplesql.SimpleConf());
        this.sqlStatement = "select max(sqle_primary) " +
                            "from sqlexample where sqle_primary = ?"
        this.prim = prim;
    }

    public PreparedStatement getStatement(Connection c)
                                        throws SQLException {
        PreparedStatement pstmnt = c.prepareStatement(sqlStatement);
        pstmnt.setInt(1, prim);
        return pstmnt;
    }
}
```

The generated constructor accepts one `int` formal parameter, according to the *external reference* declaration. Note that the parameter's name and its internal name is `prim`, which is the name that was used in the *external reference* declaration. The abstract method `getStatement` is implemented. The generated code uses prepared statements from the JDBC API, which carries along benefits, as the extra layer of type safety at compile-time and a transparent performance optimisation at runtime. In addition, the usage of prepared statements enhances application security and protects the application from injection attacks.

J% uses the type mapping mechanism to enhance the code generation process. The SQL module uses prepared statements and according to each type it uses the appropriate method from the JDBC API. This is handled automatically by the SQL module, and the programmer does not need to write anything more than the *external reference* declaration.

The SQL operator `in` is a special case. The standard JDBC API does not provide a standard approach on handling this. Consider the following example:

```
select * from customers
where city in ('Paris', 'London');
```

The standard JDBC API requires to handle the creation of the SQL set manually and a programmer may develop code like the following excerpt:

```
String[] cities = new String[] {"Paris", "London"};
Connection conn; // JDBC connection object

StringBuilder strbld = new StringBuilder();

strbld.append("select * from customers where city in (");

for (int i = 0; i < cities.length;i++) {
    strbld.append("?");
    if (i != (cities.length - 1)) {
        strbld.append(",");
    }
}

strbld.append(")");
PreparedStatement pstmnt =
    conn.prepareStatement(strbld.toString());

for (String city : cities) {
    pstmnt.setString(city);
}
```

J% augments the JDBC API usage. A straightforward implementation would require the creation of an *external type* with the following SQL query:

```
select * from customers where city in #[cities]<String[]>
```

The *external reference* is declared for the composite type `String[]`, which is an array of `java.lang.String` types. Finally, to execute the query:

```
String[] cities = new String[] {"Paris", "London"};

CitiesQuery sq = new CitiesQuery(cities);
ResultSet rs = sq.executeQuery();
```

The underlying implementation of this feature follows in principle the standard approach that is proposed by JDBC.[2] J% provides functions for all Java composite types that derive from the standard supported types (Table 4).

5. Evaluation

We performed the evaluation of our approach with four small experiments; first, we analysed practically what it means for a developer to port a JDBC-based Java code to J% code, we exhibit how the SQL errors are identified at and presented to the programmer, we ported five Java projects that are using SQL, to examine what impact the J% code has on a project in terms of common size metrics, such as LOC, and finally we measured the overall compilation overhead against the standard Java compiler. The results presented in the following sections should be used as an insight to J%'s impact, since to evaluate it thoroughly thousands lines of code from many developers must be written and analysed. Still, we depict some early indications on the effectiveness of our approach.

5.1. A real-world example

Porting Java applications to use J%'s SQL module is straightforward. When a Java source file contains an SQL query, the code is extracted, and a new *external type* is created. After that, we replace the standard JDBC calls with the API offered by the SQL

[2] http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html

module. Consider the file `add_user.java` of the *Examj*[3] project, which contains the following `insert` statement:

```
int add_usr = myStatement.executeUpdate(
"INSERT INTO examj.users(name,surname,email,username,password)
VALUES('" + name + "', '"

        + surname + "', '"
        + email + "', '"
        + usrname
        + "', md5('"+ String.valueOf(usrpassword) + "'))");
```

In J% it should be rewritten as follows:

```
InsertUserQuery iuq = new InsertUserQuery(name, surname, email,
                        usrname, md5(String.valueOf(usrpassword)));
myStatement = iuq.getStatement(connection);
int add_usr = myStatement.executeUpdate();
```

External type `InsertUserQuery` should also be defined:

```
package examj;

import org.jmod.dsl.sql.SQuery;
import org.jmod.dsl.sql.SQLConfiguration;

public external InsertUserQuery extends
                                        SQLQuery<SQLConfiguration> {
INSERT INTO examj.users(name,surname,email,username,password)
VALUES(#[name]<String>,
       #[surname]<String>,
       #[email]<String>,
       #[username]<String>,
       md5(#[password]<String>))
}
```

The above code retains the same functionality and without alterations in its logic, nor its basic structure.

5.2. Compile-time error detection

Consider the `RegisterItem` class in the *RUBiS* project.[4] It contains an SQL statement, which is created by string concatenations.

```
conn.prepareStatement(
  "INSERT INTO items VALUES (NULL, \'"
              + name + "\', \""
              + description + "\", \""
              + initialPrice + "\", \""
              + quantity + "\", \""
              + reservePrice + "\", \""
              + buyNow + "\", 0, 0, \""
              + startDate + "\", \""
              + endDate + "\", \""
              + userId + "\", "
              + categoryId+ ")");
```

[3] https://github.com/bkarak/jmod-ports/blob/master/examj/java/src/add_user.java
[4] https://github.com/bkarak/jmod-ports/blob/master/RUBiS/java/Servlets/edu/rice/rubis/servlets/RegisterItem.java

Table 7
The size metrics that were used in the evaluation process.

Metric	Description
Files	Number of files
Lines of Code (LOC)	Total lines of source code
Ext. LOC	Total lines of the external type declarations
SQL queries	Total number of queries
Classes	Total number of classes
Interfaces	Total number of interfaces
Packages	Total number of packages
External types	Total number of external type declarations

Table 8
Size metrics for JCrontab, Address Book, ExamJ, and RuBiS.

	JCrontab		AddressBook		ExamJ		RUBiS	
Metric Name	*Java*	*J%*	*Java*	*J%*	*Java*	*J%*	*Java*	*J%*
Files	43	48	8	13	15	27	37	65
LOC	3362	3386	1242	1259	3852	3893	8831	8952
Ext. LOC	–	18	–	28	–	40	–	72
SQL queries	5	5	6	5	13	12	44	28
External types	–	5	–	5	–	12	–	28
Classes	40	40	8	8	15	15	37	37
Interfaces	3	3	0	0	0	0	0	0
Packages	6	6	2	2	1	1	3	4

We created the `InsertItemQuery` type, similar to our porting strategy in the previous section:

```
public external InsertItemQuery extends SQLQuery<SQLConfiguration> {
INSERT INTO items
VALUES (NULL,
        #[name]<java.lang.String>,
        #[description]<java.lang.String>,
        #[initial_price]<float>,
        #[quantity]<int>,
        #[reserve_price]<float>,
        #[buy_now]<float>,
        #[start_date]<java.util.Date>,
        #[end_date]<java.util.Date>,
        #[user_id]<int>,
        #[category_id]<int>)
}
```

To compile it, we executed the following command:

```
$ jmodc -compile-with-javac -i RUBiS/jmod/
[...]
[INFO] .../rubis/servlets/RegisterItem.java parsed (Java)
[...]
[INFO] .../rubis/servlets/InsertItemQuery.jmod parsed (External)
[...]
[INFO] SQLModule module called for .../servlets/InsertItemQuery.jmod
```

The above is an excerpt from the compiler output. In the first phase, the compiler identifies the *external type* and then it calls the SQL module to generate the code.

If we accidentally wrote a mistyped input like `insrt` instead of the SQL keyword `insert`, a compile-time the error would be reported:

```
$ jmodc -compile-with-javac -i RUBiS/jmod/
[...]
[INFO] SQLModule module called for .../InsertItemQuery.jmod
[ERROR] Could not parse SQL Statement - INSRT INTO items [...]
[ERROR] .../InsertItemQuery.jmod, error in SQL syntax : [...]
[...]
```

The compiler discovered the error in the SQL query, then it reported it as an error message. Similarly, errors regarding the database schema are reported, like table and column identifier errors or type incompatibilities.

5.3. Measuring J% impact on real-world projects

The evaluation process aims to analyse the impact of J% in real-world projects. J% demands that each SQL code block should be declared as a new *external type*. This can be easily related with significant growth in terms of user-defined types and code. For that reason, the third evaluation experiment will focus on Size & Complexity (Table 7). To isolate efficiently the level of change that our approach imposes, we evaluate each project by comparing its original form with a J% version.

The selected projects include *JCrontab* an open source crontab replacement, *AddressBook*, the sample program that is distributed with the Java DB, *ExamJ* an open source Java editor, which aims to provide a framework to tutor and grade programming language students and the popular benchmarking web platform RUBiS [48]. All these applications use SQL and a relational database as a backend to store and access their data. All ported applications are published in Github.[5]

Table 8 summarises all size metric measurements for all projects. Table 9 contains the list of all the Java files per project that were modified during the porting process. Note that, in general, most files of the projects remained unaffected and only the files that contained SQL statements were altered. The "Java" column contains the original LOC of the file and the "J%" the LOC after the porting.

The results show that by using J%, we should expect slightly increased LOC and number of files. This is the main problem of J% in its current specification. For each SQL query a new *external type* should be declared, which practically means the creation of a new source file. Consequently, the increased LOC can be explained by the declaration of boiler-plate code that each *external type* requires. External LOC (Ext. LOC) counts the lines of code for the *external types*' source files. Since Java does not support *external type*s, the count is always zero for the Java version of the projects. Note that the number of SQL Queries is reduced in *AddressBook*, *ExamJ* and *RUBiS*. J% encourages query reusability and for the same queries, the same *external type* can be reused. For example, the *RUBiS* project has originally 44 SQL queries and 16 of them could be reused.

The SQL code is used almost unmodified, except the *external reference* declarations. These are the source of the second problematic situation. For each reference that is defined in an *external type*, a field parameter in the type's constructor is generated. This is good and bad simultaneously; the strict constructor enforces compile-time validation of each parameter that is passed to the SQL code and in addition is backed-up by a prepared statement when it is passed onto the query. Its disadvantage is that the generation scheme is not very practical when one has many parameters, and it is complex to remember their order, when the *external type* is actually used with the Java code. The example code in Section 5.1 points out to this problem. The resulting generated code of `InsertUserQuery` type has five constructor parameters.

5.4. Compilation overhead

The next step in our evaluation included the measurement of the compiler's overhead, in terms of compilation time. The compilation process includes two phases; the code generation where the SQL checking is performed, and Java code is generated, and the compilation of the generated code to Java bytecode. Consider the following program:

```
package org.jmod;

import java.sql.*;
import java.util.Properties;

public class SimpleSQLProgram {
   public static void main(String[] args) {
      final String jdbcUrl = "jdbc:mysql://localhost/test";
      final String jdbcDriver = "com.mysql.jdbc.Driver";

      try {
         Properties connectionProps = new Properties();
         connectionProps.put("user", "root");
         connectionProps.put("password", "");
         Class.forName(jdbcDriver);
```

[5] https://github.com/bkarak/jmod-ports

Table 9
LoC differentiation per file.

	Java	J%	Diff
JCrontab			
org/jcrontab/data/GenericSQLSource.java	253	236	−17
AddressBook			
com/sun/demo/addressbook/db/AddressDao.java	297	261	−36
ExamJ			
examj/add_user.java	171	159	−12
examj/code_send.java	178	160	−18
examj/load_code_db.java	407	382	−25
examj/remove_user.java	123	122	−1
RUBiS			
edu/rice/rubis/servlets/Auth.java	54	51	−3
edu/rice/rubis/servlets/BrowseCategories.java	152	153	1
edu/rice/rubis/servlets/BrowseRegions.java	85	86	1
edu/rice/rubis/servlets/BuyNow.java	138	138	0
edu/rice/rubis/servlets/PutBid.java	199	194	−5
edu/rice/rubis/servlets/PutComment.java	156	156	0
edu/rice/rubis/servlets/RegisterItem.java	237	225	−12
edu/rice/rubis/servlets/RegisterUser.java	227	215	−12
edu/rice/rubis/servlets/SearchItemsByCategory.java	205	201	−4
edu/rice/rubis/servlets/SearchItemsByRegion.java	203	198	−5
edu/rice/rubis/servlets/ServletPrinter.java	713	710	−3
edu/rice/rubis/servlets/StoreBid.java	269	254	−15
edu/rice/rubis/servlets/StoreBuyNow.java	204	189	−15
edu/rice/rubis/servlets/StoreComment.java	176	162	−14
edu/rice/rubis/servlets/ViewBidHistory.java	168	166	−2
edu/rice/rubis/servlets/ViewItem.java	147	146	−1
edu/rice/rubis/servlets/ViewUserInfo.java	201	199	−2

Table 10
Size metrics for the participants.

Metric Name	java	simple	ns-aware	live-db
Files	1	2	3	3
LOC	31	42	51	54
Ext. LOC	0	9	9	9
SQL queries	1	1	1	1
Classes	1	2	2	2
External types	0	1	1	1

Table 11
Hardware and software configuration.

Operating system	MacOS X 10.9.1
Java runtime env.	1.7.0_40 (64-bit)
System CPU	Intel Core 2 Duo, 3.06 GHz
System memory	12GB

Table 12
Basic descriptive statistic measures (ms).

Name	Min	Max	Range	1st Qrtl	3rd Qrtl
java	676	1320	644	789	696
simple	906	1493	587	935	914
ns-aware	909	1103	194	932	923
live-db	1116	1795	679	1147	1137

Table 13
Basic descriptive statistics measures (cont'd).

Name	Median	Mean	Stddev	Overhead (%)
java	683	705.74	29.31	0.00
simple	921	936.94	29.12	24.68
ns-aware	925	936.45	17.06	24.64
live-db	1123	1153.10	37.69	38.80

Table 14
Basic descriptive statistic measures for compiler overhead (ms).

Name	Min	Max	Range	1st Qrtl	3rd Qrtl
jmodc	1241	2683	1442	1367	1241
javac	1086	1770	684	1185	1240

Table 15
Basic descriptive statistic measures for compiler overhead (cont'd) (ms).

Name	Median	Mean	Stddev	Overhead (%)
jmodc	1333	1463.42	190.33	13.30
javac	1180	1283.50	135.20	0.00

```
            Connection conn = DriverManager.getConnection(
                                        jdbcUrl, connectionProps);
            Statement stmnt = conn.createStatement();
            ResultSet rs = stmnt.executeQuery(
                                        "select * from customer");
            while(rs.next()) {
                System.out.println(rs.getString("customer_name"));
            }
            rs.close();
            stmnt.close();
            conn.close();
        } catch (Exception e) {
            System.err.println("Ooops ... " + e.toString());
        }
    }
}
```

The code opens a connection to a MySQL database, performs a `select` query, then iterates over the results and prints all the customers' names. We ported this program and produced three variants of it; one that performed only a basic check on the SQL statement (*simple*), a second that performed the basic check, then also examined if the query was valid against a specific database schema (*ns-aware*), and a third that executed the query to the actual database to validate it(*live-db*). Table 10 presents a set of size metrics for each variation of the program.

Observe that each version of the program differs in terms of LOC. The original program had 31 lines of code, while the *simple*, *ns-aware*, and *live-db* versions were 26%, 39% and 42% bigger. This phenomenon was analysed in the previous sections, where we identified as the source of the problem, the declaration of the *external types* and their *configurations*.

The actual test included an iteration of 2,000 compilations for each version of the program. Table 11 lists the hardware and software characteristics of the benchmark environment. The results were analysed, and a list of basic statistical measures are presented in Tables 12 and 13. As expected, the Java version compiles faster with an average value of 705.74 ms, while the imposed overhead for the *simple*, *ns-aware*, and *live-db* versions is 24.68% (936.94 ms), 24.64% (936.45 ms), and 38.80% (1153.10 ms) respectively. The results show that is cheap to enable query check against a database schema. Of course, results may vary. In the case of a more complex schema, it is possible that the *ns-aware* could be slower.

The *live-db* variation had the slowest performance, which was also expected, since it requires to connect to a live database server and execute each query.

Tables 14 and 15 contain the result of a much simpler experiment. A simple Java program was compiled by *jmodc* (J%'s compiler) and *javac* to measure compilation time. Since *jmodc* in pure Java code, it copies the file, then it compiles it with *javac*, the results proven that is slower than his contestant. The experiment included 100 iterations and *jmodc* was 13.3% slower than *javac*.

6. Conclusions

The contribution of J% SQL module is that it supports embedding of SQL code in Java programs in a type-safe and syntax checked way. Each SQL statement is checked and all errors are reported at compile-time. In addition, Java types are mapped into SQL types, and all interactions between the two languages are also checked at compile-time.

The integration of the SQL query language to the J% compiler system provides developers with some notable features. The first involves an extensive query configuration. In essence, each query can be configured with different code generation and testing features. This is achieved with the utilisation of the *external configuration* concept introduced by our approach. Query validation is another key feature of J%. If this feature is enabled, queries can be checked against a specified database schema, which is provided as part of the query configuration.

The J% SQL module utilises the existing JDBC specification, without alienating the developer with new APIS or frameworks. In addition, our approach supports the transparent usage of prepared statements. Furthermore, J% employs the performance optimisations offered by the prepare statements compilation in a transparent manner. Finally, by using prepared statements, J% provides shielding against input validation attacks. This feature could be very useful, especially in the case of web applications.

One of the disadvantages of our approach is the fact that for every single query the developer must define a new *external type*, which may lead to significant *external type* pollution in large projects. Features like type mapping and compile-time configuration, greatly support the development process and permit the implementation of advanced features, such as live database testing in the SQL module. To assess J%'s impact to the software development process, more experiments must be performed; thus the measurements presented here were indicative.

7. Future work

Enhancing J% and its SQL support module in both features and design is planned in the future. Such enhancements could involve SQL code reuse by further extensions of the SQL module to support multiple backends. Within the boundaries of our research we intend to examine the following concepts:

Dynamic SQL generation: Our extension deals only with static SQL statements. In the future, we plan to utilise existing research such as [22] and provide mechanisms that support dynamically generated statements.

External configuration via annotations: The introduction of annotations in Java since version 1.5 provided the solid foundation for developers to introduce a series of custom compile-time checks and generation of boilerplate code. J% SQL module could also use annotations to provide compile-time information, a feature that is currently implemented with the concept of configuration types. Consider the following code:

```
@sql {
    SQLMOD_NS_AWARE = true
    SQLMOD_NS_URI = "schema.sql"
}
public external CustomerQuery extends SQLQuery {
    select * from customer where id = #[id]<int>
}
```

The annotation (`@sql`) instructs the compiler to perform compile-time namespace checks to the following SQL query. Traditionally this could be done by the SQL configuration types.

DSL optimisations: The code generation process could be enhanced to include domain-specific optimisations that may boost performance in terms of execution, e.g. caching, like the ones presented in reference [49].

Support more database backends: Our implementation focuses on the MYSQL database dialect. We plan to provide support for other major database systems. The support will include better analysis of custom SQL statements and support for type mapping for unique types that each database backend offers. The work to support each RDBMS will further expand the module's capabilities and provides the starting point to address next-level set of problems, like enhancing database security model at compile-time, etc.

Parameter object class constructor: With the current code generation approach, complex SQL statements with many *external reference* definitions can lead to complex constructors, with many parameters. The code generation mechanism

can be modified to generate parameter objects to group all the constructor parameters in one coherent class definition.

Acknowledgements and code availability

The present research is under the Action 2 of AUEB's[6] Research Funding Program for Excellence and Extroversion of the academic year 2014/2015. It is financed by the Athens University of Economics and Business Research Center, Grant Number EP-2166-01/01-01.

The authors would like to thank George Oikonomou and Christos KK Loverdos for their insightful comments and corrections during the compilation of this paper. J% is available in Github[7] under the GNU Public License v3. A collection of ported programs and the benchmark harness are maintained in https://github.com/bkarak/jmod-ports.

References

[1] Fowler M. Domain-specific languages. Upper Saddle River, NJ: Addison-Wesley; 2011.
[2] Voelter M. DSL engineering: designing, implementing and using domain-specific languages. CreateSpace Independent Publishing Platform; 2013.
[3] van Deursen A, Klint P. Little languages: little maintenance. J Softw Maint 1998;10(2):75–92.
[4] Bentley J. Programming pearls: little languages. Commun ACM 1986;29(8):711–21.
[5] Spinellis D. Reliable software implementation using domain-specific languages. In: Schuëller GI, Kafka P, editors. Proceedings ESREL '99 – the tenth European conference on safety and reliability. ESRA, VDI, TUM. Rotterdam: A.A. Balkema; 1999. p. 627–31.
[6] Spinellis D, Guruprasad V. Lightweight languages as software engineering tools. In: USENIX conference on domain-specific languages. Berkeley: USENIX Association|Washington, DC, United States of America:; p. 67–76.
[7] Tobin-Hochstadt S, Felleisen M. Interlanguage migration: from scripts to programs. In: OOPSLA '06: companion to the 21st ACM SIGPLAN conference on object-oriented programming systems, languages, and applications. New York, NY, USA: ACM Press; 2006. p. 964–74.
[8] Leijen D, Meijer E. Domain specific embedded compilers. In: PLAN '99: proceedings of the 2nd conference on domain-specific languages. Austin, Texas, United States of America: ACM Press; 1999. p. 109–22.
[9] Rhiger M. A foundation for embedded languages. ACM Trans Programm Lang Syst 2003;25(3):291–315.
[10] Heering J, Mernik M. Domain-specific languages for software engineering. In: Proceedings of the 35th Hawaii international conference on system sciences. |Washington, DC, United States of America: IEEE; 2002.
[11] Chaudhuri S, Narasayya V, Syamala M. Bridging the application and DBMS divide using static analysis and dynamic profiling. In: Proceedings of the 35th SIGMOD international conference on management of data. SIGMOD '09. New York, NY, USA: ACM; 2009. p. 1039–42.
[12] Seipel D, Boehm AM, Fröhlich M. JSquash: source code analysis of embedded database applications for determining SQL statements. In: Proceedings of the 18th international conference on applications of declarative programming and knowledge management. INAP'09. Berlin, Heidelberg: Springer-Verlag; 2011. p. 153–69.
[13] Wassermann G, Gould C, Su Z, Devanbu P. Static checking of dynamically generated queries in database applications. ACM Trans Softw Eng Methodol 2007;16(4).
[14] Gil JY, Lenz K. Simple and safe SQL queries with C++ templates. Sci Comput Programm 2010;75:573–95.
[15] Fisher M, Ellis J, Bruce J. JDBC API tutorial and reference. 3rd ed.Addison-Wesley Professional; 2003.
[16] Gosling J, Joy B, Steele G, Bracha G. The Java language specification. 3rd ed.Addison-Wesley; 2005.
[17] Karakoidas V, Spinellis D. J%: integrating domain-specific languages with Java. In: Chrissikopoulos V, Alexandris N, Douligeris C, Sioutas S, editors. PCI 2009: proceedings of 13th Panhelenic conference on informatics. Corfu, Greece: IEEE Computer Society; 2009. p. 109–13.
[18] Halfond WG, Viegas J, Orso A. A classification of SQL-injection attacks and countermeasures. In: Proceedings of the international symposium on secure software engineering; 2006.
[19] Beck K. Test driven development: by example. Addison-Wesley Professional; 2002.
[20] Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Comput Surv 2005;37(4):316–44.
[21] Loverdos CKK, Syropoulos A. Steps in scala: an introduction to object-functional programming. Edinburgh, United Kingdom: Cambridge University Press; 2010.
[22] Gould C, Su Z, Devanbu P. Static checking of dynamically generated queries in database applications. In: Proceedings of the 26th international conference on software engineering (ICSE'04). IEEE; 2004. p. 645–54.
[23] Erdweg S. Extensible languages for flexible and principled domain abstraction [Ph.D. thesis]. Philipps-Universitat Marburg; 2013.
[24] Annamaa A, Breslav A, Kabanov J, Vene V. An interactive tool for analyzing embedded SQL queries. In: Proceedings of the 8th Asian conference on programming languages and systems. APLAS'10. Berlin, Heidelberg: Springer-Verlag; 2010. p. 131–8.
[25] Seipel D, Boehm AM, Fröhlich M. JSquash: source code analysis of embedded database applications for determining SQL statements. In: Proceedings of the 18th international conference on applications of declarative programming and knowledge management. INAP'09. Berlin, Heidelberg: Springer-Verlag; 2011. p. 153–69.
[26] McClure RA, Krüger IH. SQL DOM: compile-time checking of dynamic SQL statements. In: ICSE '05: proceedings of the 27th international conference on software engineering; 2005. p. 88–96. http://dx.doi.org/10.1145/1062455.1062487.
[27] Grust T, Mayr M. A deep embedding of queries into ruby. In: Proceedings of the 2012 IEEE 28th international conference on data engineering. ICDE '12. Washington, DC, USA: IEEE Computer Society; 2012. p. 1257–60. http://dx.doi.org/10.1109/ICDE.2012.121.
[28] Bierman G, Meijer E, Schulte W. The essence of data access in Cw. In: ECOOP 2005: proceedings of the 19th European conference on object-oriented programming; 2005. p. 287–311.
[29] Vogt JC. Type safe integration of query languages into scala [Ph.D. thesis]. RWTH Aachen University; August 2011.
[30] Meijer E, Beckman B, Bierman G. LINQ: reconciling object, relations and XML in the .NET framework. In: SIGMOD '06: proceedings of the 2006 ACM SIGMOD international conference on management of data. New York, NY, USA: ACM Press; 2006. p. 706.
[31] Eisenberg A, Melton J. SQLJ part 1: SQL routines using the Java programming language. SIGMOD Rec 1999;28(4):58–63.
[32] Hibernate. Available online at: ⟨http://www.hibernate.org/⟩; 2014.
[33] Biswas R, Ort E. The Java persistence API – a simpler programming model for entity persistence. Available online at: ⟨http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html⟩; 2006.

[6] Athens University of Economics and Business.
[7] https://github.com/bkarak/jmod

[34] Typesafe. Slick: functional relational mapping for scala. Available online at: ⟨http://slick.typesafe.com⟩; 2014.
[35] Squeryl. Available online at: ⟨http://squeryl.org⟩; 2014.
[36] Typesafe. Anorm is not a object relational mapper. Available online at: ⟨http://www.playframework.com/modules/scala-0.9.1/anorm⟩; 2014.
[37] ODB: C++ Object-Relational Mapping. Available online at: ⟨http://codesynthesis.com/products/odb/⟩; 2014.
[38] Joy C, Gradman M. Database template library. Available online at: ⟨http://dtemplatelib.sourceforge.net/dtl_introduction.htm⟩; 2014.
[39] Maciej S, Hutton S, Zeitlin V, Loskot M. SOCI – the C++ database access library. URL: ⟨http://soci.sourceforge.net/⟩; August 2013.
[40] Jooq. Available online at: ⟨http://www.jooq.org⟩; 2014.
[41] Wikipedia, Powerbuilder. Available online at: ⟨http://en.wikipedia.org/wiki/PowerBuilder⟩; 2014.
[42] Bravenboer M, Visser E. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: OOPSLA '04: proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. New York, NY, USA: ACM; 2004. p. 365–83. http://doi.acm.org/10.1145/1028976.1029007.
[43] Tratt L. Domain-specific language implementation via compile-time meta-programming. ACM Trans Programm Lang Syst 2008;30(6): 1–40, http://dx.doi.org/10.1145/1391956.1391958.
[44] Renggli L, Gírba T, Nierstrasz O. Embedding languages without breaking tools. In: ECOOP 2010: proceedings of the 24th European conference on object-oriented programming; 2010.
[45] Cordy JR. Source transformation, analysis and generation in TXL. In: PEPM '06: proceedings of the 2006 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation. New York, NY, USA: ACM Press; 2006. p. 1–11. http://dx.doi.org/10.1145/1111542.1111544.
[46] Dinkelaker T, Eichberg M, Mezini M. Incremental concrete syntax for embedded languages with support for separate compilation. Sci Comput Programm 2013;78(6):615–32.
[47] Sommerville I. Software engineering. sixth ed. Addison-Wesley; 2001.
[48] Cecchet E, Chanda A, Elnikety S, Marguerite J, Zwaenepoel W. Performance comparison of middleware architectures for generating dynamic web content. In: Proceedings of the ACM/IFIP/USENIX 2003 international conference on middleware. Middleware '03. New York, NY, USA: Springer-Verlag New York, Inc.; 2003. p. 242–61.
[49] Karakoidas V, Spinellis D. FIRE/J: optimizing regular expression searches with generative programming. Softw: Pract Exp 2008;38(6):557–73.