J%: Integrating Domain Specific Languages with Java

Vassilios Karakoidas and Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Email: {bkarak,dds}@aueb.gr

***Abstract*—J% (J-mod), is a Java language extension that supports integration with Domain-Specific Languages. The integration is realized through an architecture that permits external modules to support DSLs. The DSL statements can be syntactically checked at compile-time. An additional facility allows the static type checking of Java variables that appear within DSL code. To support this process each DSL module comes as a library that is used both at compile time and during program execution.**

I. Introduction

The multiparadigm programming [1], [2] approach demanded each problem to be dealt with the most suitable programming language. The use of DSLs in the software development process reduces cost and enhances productivity [3], [4]. In modern multiparadigm software development, Domain-specific languages (DSL) are often integrated into General-Purpose Languages (GPL), and according to the categorization of Mernik et al. [3] follow distinct patterns. Mernik also provides guidelines regarding the implementation process of a DSL, and the use of each pattern, according to its notation, design and user community.

DSLs focus on a specific problem domain. For that purpose they sacrifice syntactic flexibility. In the literature, they are often called *micro-languages* or *little languages* [5]. Well known DSLs include regular expressions, SQL, HTML and VHDL. On the other hand, GPLs have a wider scope, providing a set of processing capabilities applicable to various problem domains [5]. Typical examples of GPLs are Java, C, C++, and Python.

Currently the integration of a DSL with a GPL brings forth many practical and research issues. For example, SQL integration in the Java programming language is implemented by the JDBC API application library [6]. This implementation pattern compels the programmer to pass the SQL query as a `String`. The Java compiler is completely unaware of the SQL query and the programmer finds out SQL syntactical errors at runtime, usually by an exception raised by the JDBC driver. Such errors remain undetected, if during the testing phase of the product the SQL query is never invoked.

This paper introduces J%, a DSL-aware extension of the Java programming language. Its purpose is to enrich the current Java syntax in order to effectively support DSLs. The prototype implementation consists of a pre-processor, that translates the J% source code to Java compatible code. Thus, it provides an extensible way to embed DSLs into Java.

J% introduces the following new characteristics:

- **Static Typing** The compilation process of the DSL is type-safe. The compiler is able to perform static typing to the hosted DSLs.
- **DSL Syntax Compile-Time Check** The DSL is syntactically checked at compile time and all errors are reported as compile-time errors. Each hosted DSL retains its syntax. There is only a minimal addition to the grammar of each language to support type mapping with J% (Section II).
- **Meta-Programming Facility** The hosted DSL is not translated into Java, but generates code that facilitates existing Java DSL Application Programming Interface APIs, like JDBC for SQL. This is the main difference between J% and other popular meta-programming systems that transform the DSL into the host language.
- **Common DSL Container** J% provides one common way to embed DSLs into Java. This way, the language grammar is not burdened with custom extensions, each time a new DSL module is included.
- **Extensible** Developers can create their own DSLs module through a well defined application programming interface (API). This way J% can include as many DSLs as possible.

II. The J% Language

J% language adopts the typical Java syntax (v.1.5) with a minimal set of extensions. Each DSL is also extended to support the integration with Java. Our approach follows the *Implementation: Extensible compiler/interpreter* pattern [3].

A. Extending Java

Figure 1 listing contains a simplified version of J% grammar. The following BNF conventions are adopted:

- `[`x`]` denotes zero or one occurrences of x.
- $\{x\}$ denotes zero or more occurrences of x.
- x | y means one of either x or y.

The main rule is `code_unit` which contains declarations of package (`package`), import statements (`import`) and types (`type`). Each type can be either a class, an enumeration or an interface.

The declarations are compatible with those of Java [7]. The difference lies in the `class` rule. Each class can be a class

```
code_unit ::= [ package ]
              { import }
              { type }

type ::= class
      |  interface
      |  enumeration

class ::= typical_class
       |  dsl_class

dsl_class ::= { modifier } 'class' identifier
              'extends' identifier
              '<' identifier '>'
              '{' { any_character } '}'
```

Fig. 1. Simplified J% grammar

```
<external_ref>  ::= '#''['<index>']''<'<field_type>'>'
<field_type>    ::= <base_type> |
                    <object_type> |
                    <array_type>
<base_type>     ::= 'B'|'C'|'D'|'F'|'I'|'J'|'S'|'Z'
<object_type>   ::= 'L'<fullclassname>;
<array_type>    ::= '['<optional_size><field_type>
<optional_size> ::= '0'-'9' { '0'-'9' }
<index>         ::= '1'-'9' { '0'-'9' }
```

Fig. 2. Syntax of a DSL external reference

(`typical_class`) or a DSL class (`dsl_class`). Its block contains the actual DSL code. The `any_character` rule must accept any character as input, because at grammar level we cannot predict the syntax of the embedded DSL language. The actual DSL code is contained in a set brackets, like a typical class or method.

B. DSL Extensions

DSL maintains its own syntax. By doing that, the development phase receives the maximum benefit, since the domain knowledge can be used and expressed through the specific instance of the DSL.

For simple cases, where the DSL does not interact with Java the syntax is completely the same. When the host language needs to interact with the DSL, an *external reference* must be defined; a contract that bridges the type systems between the two languages is required.

The grammar of this DSL extension is presented in Figure 2. The interpretation of the base types (`B`, `C`, etc.) is provided in the Java Virtual Machine Specification [8].

For example, in an SQL query that needs an `int` as a parameter we would write:

```
select * from customer
     where customerId = #[1]<I>
```

The expression `#[1]<I>` defines that the `customerId` expects an `int` base type (`I`). The `[1]` is the parameter index and affects the code generation phase (Section III-A4).

III. The J% Compiler

The compilation process is depicted in Figure 3. The J% compiler accepts the input source files (`.jmod`). The symbol

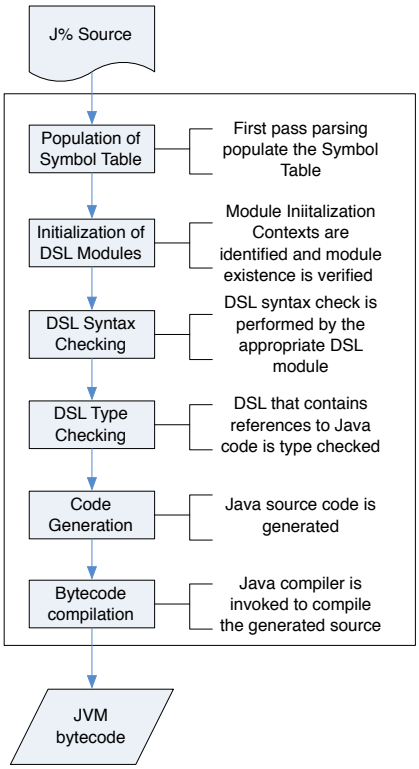

Fig. 3. J% Compilation Process

table is populated and the required DSL modules are identified. In this phase the compiler also verifies each module existence in the classpath and seeks its implementation. The *DSL types* are located, parsed and associated with the appropriate elements in the symbol table.

Afterwards, type checking is initiated for the DSL code. Type mapping information is gathered for each DSL module and the references to external variables are tested about type compatibility and scope.

The code generator is invoked and Java code is generated. Finally, the Java compiler is invoked and it translates the generated Java code into executable JVM bytecode.

A. DSL Modules

The DSL modules have a dual use; they implement the syntax and type checking at *compile-time* and provide an execution environment at *runtime*.

1) Importing: A DSL module is imported using an `import` statement. In the following example, we import the `Regex` runtime class of the regular expressions DSL module.

```
import org.jmod.dsl.regex.Regex;
```

To include third party external modules, the `jar` files must be included in the classpath and the entries must be added in the appropriate configuration file.

2) Initialization: The *compile-time* part of each module is identified in the compiler initialisation phase. The exported DSL types of each module are added in the symbol table

and identified as *DSL types*. The following listing presents a symbol table printout, populated only with the basic types (`java.lang`) and the *DSL types*.

```
$ bin/jmodc -st
[...]
Type:java.lang.IllegalThreadStateException (class)
Type:java.lang.Runnable (interface)
Type:java.lang.ThreadLocal (class)
Type:org.jmod.dsl.regex.Regex (dsl type)
```

3) Configuration: The DSL modules are reconfigured each time they are invoked. This is realised through the `ModuleConfiguration` class (Figure 4). The J% compiler identifies all the classes that are subclasses of the `ModuleConfiguration` class and use them to get at compile-time each DSL's module configuration. Each subclass has a set of public fields that define the configuration. These fields can be only `int`, `String`, `float`, and `boolean` types.

In Figure 4, the `RegexConfiguration` class is a subclass of `ModuleConfiguration` and has two public fields; `optimisation` which is set to "*true*", and `engine` which defines the underlying regular expression engine and is set to "*posix*".

If we declare a class `NumberRegex`,

```
import org.jmod.dsl.regex.Regex;
import org.jmod.dsl.regex.RegexConfiguration;

public class NumberRegex
       extends Regex<RegexConfiguration> {
[0-9]+
}
```

the DSL module is invoked with `optimisation` set to "*true*", and `engine` set to "*posix*". When we want to turn the `optimisation` off, we have to change the `RegexConfiguration` with its subclass `NonOptimisedRegex`, which overrides the public field `optimisation` with the value "*false*". The class field overriding is supported fully by the Java programming language [7].

4) Code Generation: DSL modules are responsible for the code generation. Any third party DSL application library can be used and the only restriction is that the process must produce Java compatible code.

If the DSL block interacts with the main Java program through shared variables (`external references`, then the generated code uses the following convention. The generated class must have a constructor that initialises the shared variable with the correct order, which is defined in their declaration. For example, consider the SQL query:

```
select * from customer where
 cust_id = #[1]<int> and cust_name
        = #[2]<Ljava/lang/String;>
```

The constructor of the class should be `CustomerQuery(int i, String s)` to correctly initialise the two declared types, an `integer` that must be ordered first and a `String` as second.

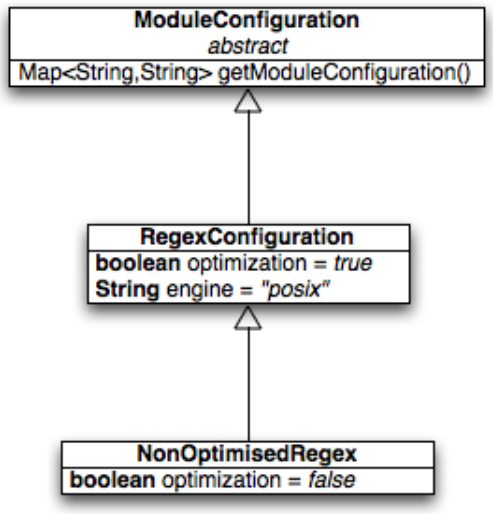

Fig. 4. DSL Module Configuration Hierarchy

IV. Case Study: Regular Expressions

Regular expressions are a standard feature in many programming languages. Java, C#, Perl and Python contain regular expression engines as part of their API. The regular expression API in Java follows the *Implementation:Embedding* pattern and it is realised through an application library.

The Regular Expression Module in J% exports only the type `Regex`. The `RegexConfiguration` class declares the basic configuration parameters. The module performs syntax check in the regular expression, and reports the errors in compile-time. Its main dependency is the standard regular expression library `java.util.regex`, which is distributed as part of the JDK since v.1.4.

Consider that we have the following code, a regular expression that matches an IP Address (IPv4).

```
public class IpAddress
     extends Regex<RegexConfiguration> {
([0-9]{1,3}\.){3}[0-9]{1,3}
}
```

The above code will be transformed into:

```
public class IpAddress
    extends Regex<RegexConfiguration> {
    private String regex;

    public IpAddress() {
       super(new RegexConfiguration());
       this.regex =
        "([0-9]{1,3}\\.){3}[0-9]{1,3}"; }

    public Pattern getPattern() {
       return Pattern.compile(regex); }}
```

The generated code is pretty straightforward and utilises the standard regular expression library. If the regular epxression has a syntactic error, the compiler will report it, during the compilation phase. Regular expressions do not have shared variables with Java, so this module represents the simplest

form of integration between a DSL and J%.

V. Case Study: SQL

The SQL is the standard language for database query and manipulation. This case, is more complex than regular expressions, since it supports also shared variables, in addition with syntax checking. The syntax analysis is based on a custom SQL grammar, and the generated code utilises standard JDBC calls. We follow the type mapping scheme proposed by the JDBC specification [6].

The following listing illustrates an SQL query that perform a select statement from the database table `customers`. The `#[1]<I>` defines that the query needs one integer parameter.

```
public class CustomerSelect
    extends SQL<SQLConfiguration> {
select * from customers where cust_id = #[1]<I>
}
```

The first initialisation parameter for this class will be an `int`. The generated code will look like:

```
public class CustomerSelect
   extends SQL<SQLConfiguration> {
   private int i; private String sql;

   public CustomerSelect(int i) {
      super(new SQLConfiguration());
      this.i = i;
      this.sql =
        "select * from customers where cust_id = ?";
   }

   public PreparedStatement
                getStatement(Connection c) {
      try {
         PreparedStatement r =
                c.prepareStatement(sql);
         r.setInt(1,i)

         return r;
      } catch (Exception e) {
        return null; }
      }
}
```

The `CustomerSelect` class utilises the JDBC API. The constructor accepts one integer parameter. The external reference (`#[1]<I>`) is replaced with a "`?`" in the generated code and a typical *PreparedStatement* is used.

VI. Related Work

We studied approaches that extend functional languages, such as Haskell [9], [10] and other implementation efforts that were based on general-purpose languages like Java and C++ [11], [12], [13]. Table I categorises the case studies that are presented in this section, according to Mernik et al. [3].

There is also a lot of work on the theoretical aspects of the Java programming language, like its type system [14], [15]. Theoretical research also included multi-language systems and their type systems [16], which reveals a research direction, how to efficiently intermix programming languages.

The *Boo* programming language provides a Python inspired syntax and features *String Interpolation* and support for regular expressions implementing the *Implementation:Embedding* pattern.

An noteworthy approach for regular expression embedding is used by Perl [17]. Perl introduces operators that efficiently integrate regular expressions within the language syntax.

Haskell/DB [9] is a host language variant that has been extended to encapsulate SQL queries. This approach completely hides the DSL from the developer, hindering productivity and forbidding domain experts to become involved with the development process. Haskell/DB follows the *Creational:Language Extension*.

Python provides support for regular expressions and SQL. Database tables are encapsulated in classes extending the `SQLObject`. This mechanism permits the developer to execute simple queries without writing SQL, thus partially solving the problem of an erroneous query, simply by generating it automatically through library code.

Powerscript is the core development language for the database development environment Powerbuilder. Powerscript is a specialization of the BASIC programming language that supports integration with SQL. This is a classic example of *Creational:Language Specialisation* pattern, where a general-purpose language (BASIC), is restricted to a specialised development language for database applications. Powerscript provides syntax and type checking in the integrated SQL queries.

C# and Java share many common characteristics in their support of DSL languages. They both support regular expressions and SQL using the *Implementation:Embedding* pattern.

SQL DOM [18] acts as a pre-processor that translates an SQL database schema in C#. The generated collection of objects is used as an API to the main application, thus ensuring type safety and syntax checking. Notably, the SQL statements in this approach are generated with a provided `getSQL()` method. Cω [19] integrated both SQL and XML into its syntax extending the C# programming language.

XJ [11] provides XML static type checking with the extension of additional data types from an XML schema. XACT [12] and JDBC Checker [13] provide a completely different approach; they try to determine possible dynamically generated DSL statements during compile time, and provide error checking. JWIG is an interesting extension of Java for better web service development support [20]. Machete [21] is another Java extension that provides mechanisms towards the unification of pattern matching languages such as regular expressions, structured term patterns, XPATH and bit-level patterns. ILEA (Inter-LanguagE Analysis) [22] is a JVML (Java Virtual Machine Language) extension that permits extensive analysis in C source code, to cover JNI call problems. Jeannie [23] addresses the intermixing of Java and C at the programming language level.

Metaborg [24] utilises similar techniques for code generation, allowing language extensions and utilising existing application libraries. Its main problem is that it does not present a unified method for embedding. On the contrary, it encourages the developers to use their syntactic extensions for

TABLE I
CATEGORISATION OF MULTI-LANGUAGE SYSTEMS

Implementation	Pattern
Boo (regex)	Implementation:Embedding
Haskell/DB (SQL)	Creational:Language Extension
JDBC Checker (SQL)	Implementation:Compiler
Powerscript (SQL)	Creational:Language Specialisation
XJ (XML)	Creational:Language Extension
XACT (XML)	Creational:Language Extension
Cω (SQL, XML)	Creational:Language Extension
Perl (regex)	Creational:Language Extension
Java (regex,SQL)	Implementation:Embedding
SQL DOM (SQL)	Implementation:Preprocessor
J% (any)	Implementation: Extensible compiler/interpreter
JWIG (XML)	Creational:Language Extension
Ruby (regex,SQL)	Implementation:Embedding
Python (regex,SQL)	Implementation: Embedding
Jeannie (C)	Creational: Language Extension
Metaborg (any)	Implementation: Extensible compiler/interpreter

each module.

VII. CONCLUSIONS AND FUTURE WORK

J% is a language extension of Java that integrates DSL in an modular way. It also provides with syntax and type checking between DSL and Java. We also saw a few concrete examples of J% usage through its regular expression and SQL modules.

The modules presented are exhibiting the basic facilities of the J% compiler, and in the future we plan to add more features, exploiting further the compiler's awareness of the DSL code.

Upon maturation of the current prototype, we plan to add support for XML, named parameters for the DSL code blocks, and further study the following open issues:

- **Host Language Support** Current J% design and implementation focus only at the extension of the Java programming language. It would be interesting to explore, how the same set of techniques and methods could be applied on other languages, like C++.
- **Dynamic DSL generation** Our extension deals only for static DSL statements. In the future we plan to utilise existing research [12], [13] and provide mechanisms to check dynamically generated statements.
- **Compile Time Optimisations** In many cases, the domain-specific language module, can act as an optimiser for DSL generated statements, such as code generation for regular expression.
- **Unified Debugging process** Typical debugging solutions are not fit for J%. The debugger must also support plug-ins for all the integrated programming languages. The debugging facilities, such as breakpoints, must work in an inter-unique way.

VIII. AVAILABILITY

A prototype version of the J% compiler is available at http://gaijin.dmst.aueb.gr/~bkarak/programs/jmod/.

ACKNOWLEDGMENT

The research work presented in this publication is funded by AUEB's Funding Programme for Basic Research 2008 (project number 51).

REFERENCES

[1] J. Placer, "Multiparadigm research: a new direction of language design," *SIGPLAN Notices*, vol. 26, no. 3, pp. 9–17, 1991.
[2] P. Zave, "A compositional approach to multiparadigm programming," *IEEE Software*, vol. 6, no. 5, pp. 15–25, 1989.
[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.
[4] A. van Deursen and P. Klint, "Little languages: little maintenance," *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75–92, 1998.
[5] J. Bentley, "Programming pearls: little languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.
[6] M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, 3rd ed. Addison Wesley, 2003.
[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, 3^{rd} edition*. Addison-Wesley, 2005.
[8] T. Lindhorn and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed., ser. The Java Series. Addison-Wesley, 2003.
[9] D. Leijen and E. Meijer, "Domain specific embedded compilers," in *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*. ACM Press, 1999, pp. 109–122.
[10] P. Thiemann, "Programmable type systems for domain specific languages," 2002.
[11] "Xj: Facilitating xml processing in java," *World Wide Web (WWW)*, May 2005, (to appear).
[12] C. Kirkegaard, A. Moller, and M. I. Schwartzbach, "Static analysis of XML transformations in java," *IEEE Transactions on Software Engineering*, vol. 30, no. 3, pp. 181–192, March 2004.
[13] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE, may 2004, pp. 645–654.
[14] S. Drossopoulou, S. Eisenbach, and S. Khurshid, "Is the java type system sound?" *Theory and Practice of Object Systems*, vol. 5, no. 1, pp. 3–24, 1999.
[15] D. Syme, "Proving java type soundness," in *Formal Syntax and Semantics of Java*. London, UK: Springer-Verlag, 1999, pp. 83–118.
[16] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2007.
[17] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*. Sebastopol, CA: O'Reilly, 2000.
[18] R. A. McClure and I. H. Krüger, "SQL DOM: compile time checking of dynamic sql statements," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 88–96.
[19] G. Bierman, E. Meijer, and W. Schulte, "The essence of data access in cw," in *ECOOP 2005: Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005, pp. 287–311.
[20] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Extending java for high-level web service construction," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 6, pp. 814–875, 2003.
[21] M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek, "Matchete: Paths through the pattern matching jungle," in *Practical Aspects of Declarative Languages (PADL08)*, 2008.
[22] G. Tan and G. Morrisett, "Ilea: inter-language analysis across java and c," in *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*. ACM, 2007.
[23] M. Hirzel and R. Grimm, "Jeannie: granting java native interface developers their wishes," in *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*. ACM, 2007.
[24] M. Bravenboer, R. de Groot, and E. Visser, "Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt," in *GTTSE*, 2006, pp. 297–311.