DSL Evolution through Composition

Walter Cazzola
Department of Informatics and Communication
Università degli studi di Milano
cazzola@dico.unimi.it

Davide Poletti
Department of Informatics and Communication
Università degli studi di Milano
dav.poletti@gmail.com

ABSTRACT

The use of domain specific languages (DSL), instead of general purpose languages introduces a number of advantages in software development even if could be problematic to maintain the DSL consistent with the evolution of the domain. Traditionally, to develop a compiler/interpreter from scratch but also to modify an existing compiler to support the novel DSL is a long and difficult task. We have developed Neverlang to simplify and speed up the development and maintenance of DSLs. The framework presented in this article not only allows to develop the syntax and the semantic of a new language from scratch but it is particularly focused on the reusability of the language definition. The interpreters/compilers produced with such a framework are modular and it is easy to add remove or modify their sections. This allows to modify the DSL definition in order to follow the evolution of the underneath domain. In this work, we explore the Neverlang framework and try out the adaptability of its language definition.

1. INTRODUCTION

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [17]. In a DSL-based approach the developer concentrates all the knowledge about the domain in the DSL and its supporting component libraries, while all implementation knowledge is condensed in the DSL compiler/interpreter [16]. These languages allow solutions at the problem domain level of abstraction to be expressed in a proper idiom. Consequently, domain experts themselves can understand, validate, modify, and even develop DSL programs. DSLs are exploited by innovative programming techniques. Language oriented programming [3, 6] exploits the idea of building software around a set of domain specific languages to perform a relatively straightforward mapping from the conceptual model into the code. In the *model-driven development* [12] approach, the DSL modeling capability can be used, for example, to free the UML from its dependency of the object-oriented programming paradigm [9].

The use of DSL, introduces a number of benefits also to support application software evolution. Working at a high level of abstraction preserves the user from unsafe or inconsistent program modifications with respect to the original design. Moreover being close to the domain makes the DSL self-documented [13] and the code of the developed applications more clear; simplifying the documentation and reducing the need of updating when the application evolves.

Since the DSL is strictly coupled to the underneath domain when the domain changes the DSL structure itself could become inadequate to deal with it. To overcome this drawback we should develop a new DSL or adapt the existing one to the new features. To develop a new programming language, even if small, implies a great effort because the development of the supporting environment (compiler, interpreter, ...) is a challenging task. Various software tools support the automatic generation of the compiler/interpreter front-end: LeX/Yacc, JavaCC[1] and ANTLR[2]. But these tools have a rudimentary support for further compilation phases, limited to simple semantic actions and tree building during parsing and they can not be used as complete DSL generators. On the other hand, to adapt the existing DSL to the new domain by reusing part of the original DSL implementation requires a particular DSL design. To this regard, in [18] has been showed that DSL *refinement* and *multi-DSL composition* are good strategies to develop a DSL maintaining specificity and accuracy, while simultaneously facilitating reuse to limit the cost creation by modifying an existing language.

The Neverlang framework [2] model strives to support DSL evolution against domain evolution. It does not only support the creation of a DSL providing a complete compiler/interpreter generation, but it also enhances the reusability of the generated compiler/interpreter to ease future modifications of the DSL. These model features allow the user to keep using the DSL also during software evolution by adapting the original language implementation to the domain modifications.

2. THE NEVERLANG FRAMEWORK

2.1 Basic Framework Concepts

A complete compiler/interpreter built up with Neverlang [2] is the result of a compositional process involving several building blocks. In this scenario, to design a programming language consists of implementing a set of basic blocks (each of them coding a single programming feature and the necessary support code, such as type checking and code generation related to such a feature) and composing them together. The whole structure of the compiler/interpreter is the result of the composition of such blocks, in

[1]javacc.dev.java.net
[2]www.antlr.org

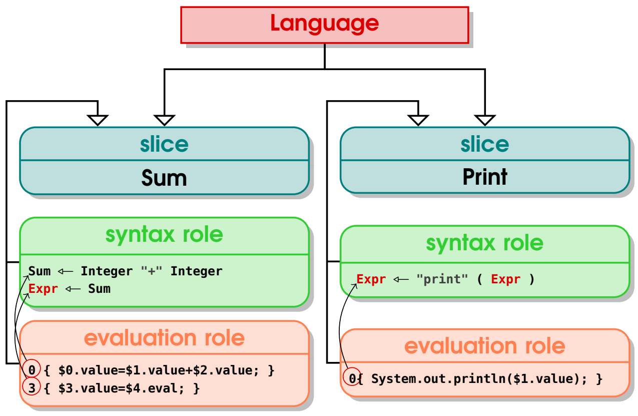
particular of the code necessary to compile/to interpret each single feature. The framework basically provides: a language for writing the building blocks and a mechanism for composing the blocks together and for generating the compiler/interpreter.

The language definition consists of two composition steps: i) the user determines which programming features will be part of the language and composes the basic units necessary to support such a features in *slices*; then ii) such slices are merged together to build the compiler/interpreter for the language. The basic units used in the first step are called *modules*. Each module encapsulates a specific feature or structure of the DSL. A module could contain: the definition of the syntax for a loop structure, the code that implements the type-checking of a comparison or another piece of code implementing the evaluation of an operation.

Traditionally the structure of a compiler is composed of subsequent phases [1]. Each phase transforms the program according to some rules and passes it to the next compilation phase. The compilers generated by using Neverlang follow the same design; each module belongs to a specific compilation/interpretation phase as defined by its *role*. Roles define how modules should be composed by the generator to form the compiler/interpreter. The `syntax` role is compulsory and the whole set of modules with this role defines the syntax of the language. This kind of modules contain one or more grammar productions expressed similarly to the derivation rules of the BNF: a head and a body separated by the $\Leftarrow$ symbol; the left part is a nonterminal, while the right part contains both terminals and nonterminals. Along with this role the user can define one or more semantic roles that specify the behavior of the syntax structure. The modules with a semantic role contain semantic actions that are associated to nonterminals contained in the grammar production of the corresponding syntactical role. The semantic actions are basically pieces of Java code that access to attributes computed in the other semantic modules. What the attributes are and how they are passed from a module to another comes directly from the *syntax-directed translation* mechanism [1]. Attributes are accessed through the nonterminal (by its position prefixed by `$`) they refer to. Syntactic and semantic modules regarding a language feature are grouped together in *slices*. A slice is a collection of modules with different roles pertaining to a specific language feature. A slice could not contain two modules with the same role.

To support the various compilation/interpretation phases, the developer may need some ancillary structures or services that concerns the whole compilation process affecting all the other modules crosswise. Simple examples are the *symbol table* and the code to deal with the *memory management*. A slightly different form of slice, called *endemic*, supports this behavior. The fields and methods defined in an endemic slice are accessible by all the modules in the language independently of the compilation/interpretation phase. To add/to replace an endemic slice permits to easily redefine the whole behavior of the compiler/interpreter.

Figure 1 graphically shows a minimal language definition. The cyan rectangles are the slice definitions and they are composed by two modules each. The green modules are the syntactical ones while the orange modules define the semantic actions used by the evaluation phase. The `Sum` slice implements a simple sum operation between integers: the code in the evaluation modules access to the values of the two operands through the `value` attribute. The `Print` slice contains the implementation of a function that prints the expression evaluation. Then the slices are composed to create the language. The composition in Neverlang is always syntax-driven: semantic modules are glued to the syntactical ones to define the slices thanks to their syntactic structure. As a consequence, all the problems that can rise during the composition are bound to the



```
Sum ⇐ Integer "+" Integer
Expr ⇐ Sum
0 { $0.value=$1.value+$2.value; }
3 { $3.value=$4.eval; }
```

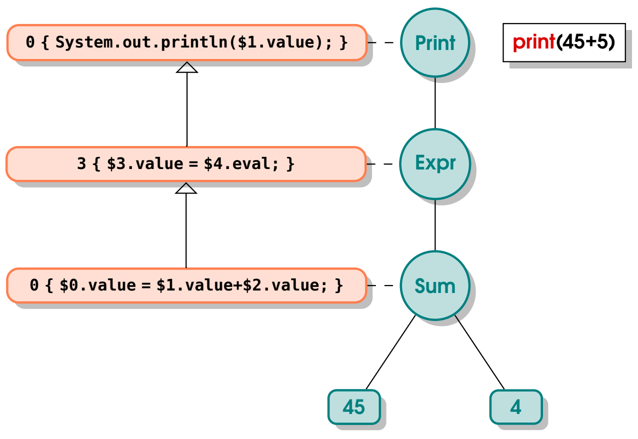
```
Expr ⇐ "print" ( Expr )
0{ System.out.println($1.value); }
```

Figure 1: A Simple Example of Language Definition

syntax (nonterminal clashes or homonymies). However once the syntax structure is correctly composed, the Neverlang framework avoids any other potential conflicts since the semantic actions are associated to the nonterminal position in a given production rather than to their names. The `Sum` and `Print` slices of the language in the example are composed by means of the `Expr` nonterminal highlighted in the figure.

2.2 Framework Implementation Details

The Neverlang framework reads the slices and creates a compiler/interpreter for the corresponding programming language; this is mainly composed of two parts: i) a front-end that parses the source files written in the new language and generates an *abstract syntax tree* (AST) of such source files, and ii) a type-driven back-end that exploits the AST to carry out all compilation/interpretation phases. The grammar productions contained in the syntactic modules define a context-free-grammar of the novel programming language; such a grammar is used by Neverlang to generate the front-end parser. In particular the framework exploits the *Parsing Expression Grammars* (PEGs) [5], this kind of recognition-based formalism lends itself to define such modular grammar being close under composition, intersection and complement. Moreover PEGs



```
0 { System.out.println($1.value); }
3 { $3.value = $4.eval; }
0 { $0.value = $1.value+$2.value; }
```

Figure 2: The decorated AST for `print(45+5)`.

can be parsed in linear time by the *packrat parser*.

The back-end deploys the syntax-directed translation [1] mechanism on the AST produced by the parser. The semantic actions are encapsulated in aspects that AspectJ weaves to the AST node representing the nonterminal which the semantic action refers to. In particular, the compilation phase associated to a role takes place on the traversal of the AST by calling a dummy and empty method on each node; such a method represents the hook used by AspectJ to weave the semantic actions accordingly to the type of the node, the node position in the AST (and consequently the position the corresponding nonterminal has in the applied production) and to the compilation/interpretation phase (that is, the role we are effectively playing). The use of AspectJ permits to overcome the limitations of classical *visitor* pattern [8]-based solutions: roles are implemented without modifying the classes of the AST nodes and the whole role can be easily deployed and played from time to time. Methods and fields defined in endemic slices are wrapped in a static class imported by the generated aspects and classes. Such fields and methods are initialized before the AST visit providing the implemented services for the whole compilation phase.

Figure 2 shows the decorated AST for `print(45+5)` a valid program in the language defined in Fig. 1. The cyan circles are the typed AST nodes, orange boxes contain the semantic actions to be evaluated at the node and the two cyan boxes are the leaves of the AST containing the evaluated integers. The evaluation phase is carried out by a post-order visit of the AST.

3. NEVERLANG FOR SW EVOLUTION

3.1 A First DSL Implementation

To explore the flexibility of the Neverlang model in case of domain evolution we will introduce a simple but realistic example of DSL definition and evolution. Let us consider to be in the need for a small administration utility that allows to automatically manage the logs generated by the system applications. In particular the utility should permit to define some maintenance tasks to: rename, move and backup the log files; basically, this is a minimal version of the Unix `logrotate` utility. Instead of implementing an application that reads a configuration file and executes the corresponding tasks our approach is language driven: the administrator specifies the tasks as small programs written with an *ad hoc* DSL and the tasks validation and execution will be part of the interpretation process lead by the Neverlang generated interpreter.

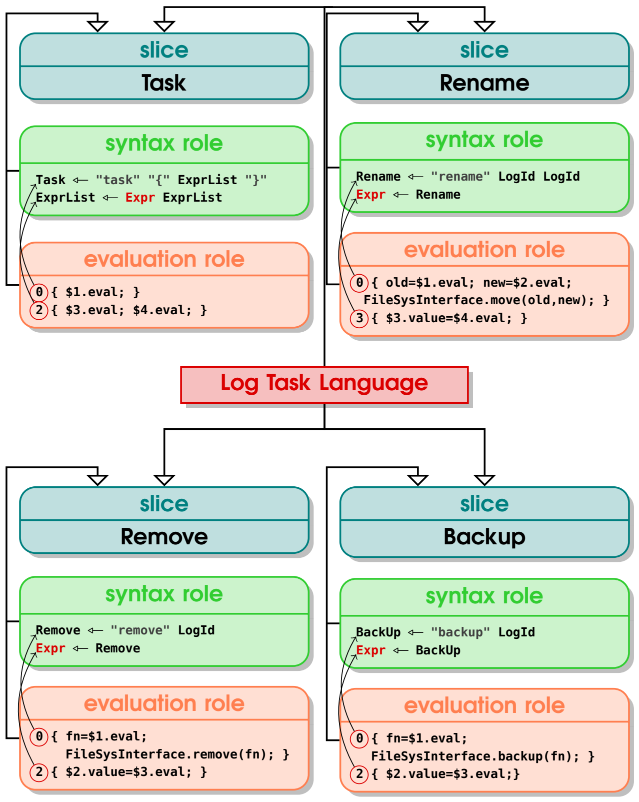
Listing 1 shows two tasks defined in the described DSL: the former task rotates the debug logs of the application named «`application`» by deleting the old log and renaming the current one as old; the latter simply backups access and system error logs.

```
task {
  remove application.debug.old
  rename application.debug application.debug.old
}

task {
  backup access.error
  backup system.error
}
```

Listing 1: An example of management scripts

This approach, compared with a solution based on configuration files, reduces the amount of code necessary to implement an *ad hoc* utility; the DSL implements few features (task, rename, move and backup) that means only four slices. From the point of view of



```
Task ⇽ "task" "{" ExprList "}"
ExprList ⇽ Expr ExprList
```

```
0 { $1.eval; }
2 { $3.eval; $4.eval; }
```

```
Rename ⇽ "rename" LogId LogId
Expr ⇽ Rename
```

```
0 { old=$1.eval; new=$2.eval;
 FileSysInterface.move(old,new); }
3 { $3.value=$4.eval; }
```

```
Remove ⇽ "remove" LogId
Expr ⇽ Remove
```

```
0 { fn=$1.eval;
   FileSysInterface.remove(fn); }
2 { $2.value=$3.eval; }
```

```
BackUp ⇽ "backup" LogId
Expr ⇽ BackUp
```

```
0 { fn=$1.eval;
  FileSysInterface.backup(fn); }
2 { $2.value=$3.eval;}
```

Figure 3: A `logrotate`-based DSL in Neverlang

the user to define the tasks by using a configuration file or by writing a small DSL program does not make any difference as long as the syntax and the semantic of the language provides a good abstraction of the underneath domain. Moreover it is easy to modify the behavior of the application to meet small changes in the problem domain (e.g., to manage log files from a new application, or a change in the policy regarding other log files) so the DSL-based solution results more flexible against domain evolutions.

Figure 3 shows the Neverlang definition for this DSL. There are four slices whose composition relies on the presence of the `Expr` nonterminal in each slice. The `Rename`, `Backup` and `Remove` slices implement the three fundamental operations while the `Task` slice provides the `task` abstraction for a list of expressions. Since each interpreted file can contain several tasks, the complete language definition also contains a slice that defines a program as a list of tasks, however for lacking of space and to keep the picture readable this last (and trivial) slice is omitted. In the semantic actions the operations related to the log files are delegated to the method in the `FileSysInterface` endemic slice (omitted as well).

3.2 DSL Evolution

The Neverlang framework has been designed to support reusability and extensibility; these characteristics are particularly useful to support the evolution of a DSL against the evolution of its domain. To prove this potentiality we are going to introduce some changes to the previous problem domain that cannot be modeled by the current DSL implementation.

First of all, we introduce the possibility to merge two logs in a single file, this implies the definition of a new operation `merge`. To add this new operation to the language we have simply to introduce a new slice (supporting the new feature) in the current language definition. Similar to the other operations, the nonterminal `Expr` will be the point of anchorage for the new operation in the rest of the language. Obviously the endemic slice `FileSystemInterface` containing the low level implementations for the commands in our language must be extended to support the `merge` operation in the file system as well.

The second extension consists in allowing the execution of the log maintenance tasks to several users and not just to the administrator. To avoid security problems, it is necessary to associate each file with different accessing rules based on the user and group id — e.g., the maintenance of `*.critical` log files only relies on the administrator whereas the other log files can only be maintained by their owners. The details about the security management is out of the scope of this discussion, we assume that the role of a task execution is the one of the user who run it. The interesting point is that our utility has now to verify the log access permission before executing each operation and this modification of the original utility concept affects the whole implementation; no new syntax is necessary to support this feature. Obviously, our previous DSL implementation cannot embody this new feature without a modification that potentially involves all the existing slices. Fortunately, the Neverlang model permits to avoid a so massive change by introducing a new semantic phase devoted to the permission check; this novel phase will be run before the evaluation. Note that, basically the operations and their implementation do not change except for the security check and thanks to the introduction of a semantic phase the previous code is completely reused as it is.

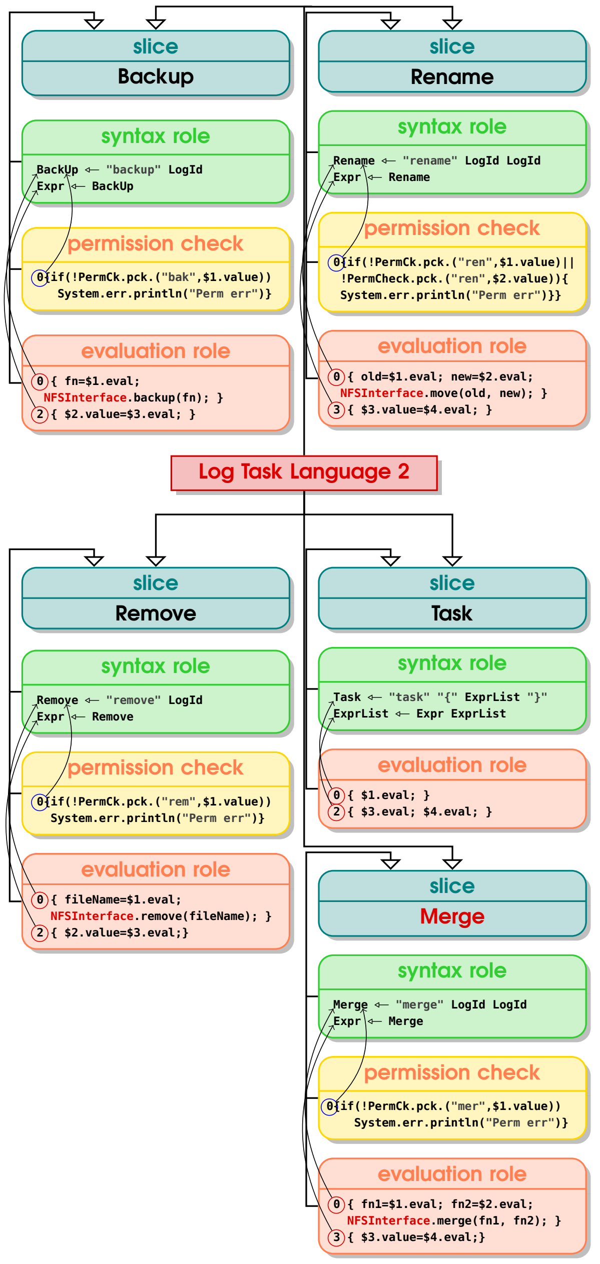
Last extension regards the possibility to store the log files on remote computers to permit their remote administration. As described in the original DSL definition, all the code strictly related to write and to read the log files is delegated to the operations provided by an endemic slice (`FileSystemInterface`). We can simply obtain the new behavior by substituting this slice with a new one (`NFSInterface`) implementing the operations necessary to work on a remote filesystem. Currently, the change of the slice and therefore of its name imposes a change to all the modules that use it; we are working on a composition mechanism that solve this issue.

Figure 4 shows the DSL definition after the proposed extensions. The yellow boxes are the module related to the new semantic phase that performs the permission check. As you can note these semantic actions exploit the `pck` method of the `PermChk` endemic slice that returns true if the operation is allowed on the passed argument; when the operation is forbidden the interpreter simply prints an error message. In the lower right corner of the picture you find the slice introducing the `merge` feature. Finally the calls to `FileSysInterface` endemic slice in the evaluation semantic actions are replaced by calls to the new endemic slice `NFSInterface`. The changes are stressed in red in the figure.

4. DISCUSSION ON NEVERLANG

The example in Sect. 3 shows how a DSL can be adapted to meet the changes in the domain by exploiting the features of the Neverlang model. We showed how easy is to modify the syntax but also the semantic of the DSL by changing only a small subset of its slices to follow the domain evolution. Even if the Neverlang model naturally enhances the reusability and the extendability of the DSL definitions, the user has to adopt an accurate design to exploit its potentiality at the best. In the case study, for example, to easily change the implementation back-end all the operations related to the filesystem has been delegated to an endemic slice.

The compositional model adopted by Neverlang is designed to



```
BackUp ← "backup" LogId
Expr ← BackUp

0{if(!PermCk.pck.("bak",$1.value))
   System.err.println("Perm err")}

0 { fn=$1.eval;
 NFSInterface.backup(fn); }
2 { $2.value=$3.eval; }
```

```
Rename ← "rename" LogId LogId
Expr ← Rename

0{if(!PermCk.pck.("ren",$1.value)||
 !PermCheck.pck.("ren",$2.value)){
 System.err.println("Perm err")}}

0 { old=$1.eval; new=$2.eval;
 NFSInterface.move(old, new); }
3 { $3.value=$4.eval; }
```

```
Remove ← "remove" LogId
Expr ← Remove

0{if(!PermCk.pck.("rem",$1.value))
  System.err.println("Perm err")}

0 { fileName=$1.eval;
  NFSInterface.remove(fileName); }
2 { $2.value=$3.eval;}
```

```
Task ← "task" "{" ExprList "}"
ExprList ← Expr ExprList

0 { $1.eval; }
2 { $3.eval; $4.eval; }
```

```
Merge ← "merge" LogId LogId
Expr ← Merge

0{if(!PermCk.pck.("mer",$1.value))
   System.err.println("Perm err")}

0 { fn1=$1.eval; fn2=$2.eval;
  NFSInterface.merge(fn1, fn2); }
3 { $3.value=$4.eval;}
```

Figure 4: The evolution of the DSL in Fig. 3

improve reusability and extensibility but it does not limit its flexibility: any kind of programming languages can be implemented independently of its complexity or of the adopted paradigm. The toy example presented in this article shows only the implementation of a simple declarative language, however we have developed definitions for functional, object-oriented and concurrent programming languages[3].

Moreover, the compositional model of Neverlang introduces additional benefits in case of changes in the domain. By using a language oriented development strategy the DSL interpreter/compiler becomes the main element of the software, then the slices and modules that compose the language definition are a representation

[3]Available at `homes.dico.unimi.it/~cazzola/neverlang.html`

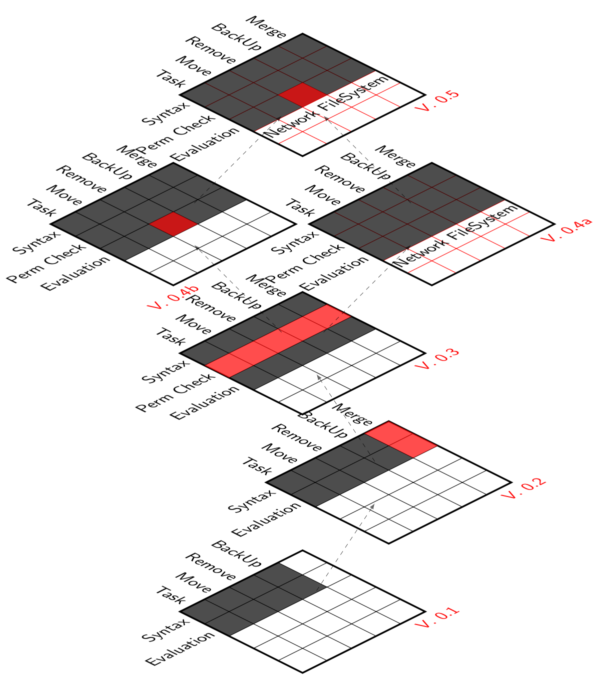


Figure 5: Language evolution over time

of this software. Since all the modifications and the extensions applied to the language are introduced by means of these model units (like in the example in Sect. 3) it is easy to trace the evolution history of the language. Each set of slices and modules that compose the language represent a version of the software, a new version of the software can be defined by adding, by modifying or by removing one or more slices/modules. Besides different version of the software can be created by the merging two other versions.

To clear up this concept, the DSL evolution faced in Sect. 3 has been depicted in Fig. 5. Each layer corresponds to a different DSL version, the history of the DSL evolution can be examined by a bottom-up analysis of the picture. The colored cells in each grid represents the modules, the horizontal position of a cell describes to which slice the module belongs, the vertical position indicates the role of the module; the red parts refer to the innovations with respect to the previous version. The v0.1 is the first implementation described in Fig. 3. Version 0.2 introduces the `merge` operation as a new slice, i.e., a new column. The grid of v0.3 contains a new line corresponding to the new role `permission check`. At the fourth level we fork the evolution: in v0.4 we modify the implementation of the backup algorithm changing the corresponding evaluation module, while in v0.4b we changed the endemic slice to operate on a network file system (to represent the change we highlighted the whole grid structure). Finally, v0.4a and v0.4b are merged in v0.5.

5. NEVERLANG RELATED WORK

In [7] has been analyzed the evolution of a domain specific language embedded in the Java language, even if the aim of that work was different it can be useful to compare a language built up with Neverlang and the final version of the EDSL contained in the final evolution of JMock. In particular the languages built up with the Neverlang framework provide similar benefits for future evolutions: orthogonality, in JMock this means to give the possibility of writing new expectations style and options avoiding code duplication while in Neverlang this means the possibility to add new language feature implemented by the separation of the definition in slices; seamless extensibility, it is possible to modify the language without breaking out the DSL.

Several works that aim to support programming languages creation and extension. In the follow we will analyze some of them.

JastAdd. JastAdd, is a flexible system which allows to conveniently implement the compiler behavior [10]. JastAdd and Neverlang share a very similar object-oriented implementation of the AST [4]. Moreover, they both adopt aspect-oriented programming to extend the language behavior by injecting methods and fields in the AST nodes. On the other side, in Neverlang the AST nodes and their connections come after the grammar productions whereas in JastAdd they can be user-defined granting a major flexibility but the generated code can bloat. Anyway JastAdd do not provide a model that is oriented to the reusability of the language definition. First of all, there is not a unit (like the slice) that encapsulates the language structures. In JastAdd each declared behavior rewrites the AST tree nodes giving the opportunity to add or delay a phase of compilation; behaviors are similar to Neverlang roles, even if Neverlang's modularity (roles) is not limited to compiler phases but straddles the whole compilation/interpretation process via the endemic slices.

Polyglot. Polyglot [14] is an extensible compiler framework that supports the creation of compilers for Java-like languages. Polyglot relies on an *extensible parser generator* that permits to express the language syntactical extensions as changes to the Java grammar. Polyglot extensibility is supported by *delegation*. Each compilation phase is supported by a delegate object present in each AST node type; the delegate object is appropriately replaced in each extension. Neverlang and Polyglot share similar goals, i.e., to support the development of syntactical and semantic extensions to a programming language but Polyglot is limited to Java. Besides, Polyglot extensions are just source-to-source translations from the extended language to pure Java. Modularity and reusability are issues that Polyglot does not face.

xText. xText is an Eclipse plug-in that provides a framework for the development of domain-specific languages. It is tightly integrated with the Eclipse modeling framework [15] to provide a language-specific ODE. Like JastAdd the user is free to define the relation between grammar productions and AST nodes but each parser rule will create a new node in the AST. The language meta-model describes the structure of its AST. xText's generator leverages the *modeling work-flow engine* from Eclipse modeling framework technology and the code is generated from the metamodel created by the parser; the meta-model is similar to the Neverlang semantic back-end. Even if the framework provides some limited possibility to reuse existing grammars and existing metamodels to implement the back-end for different languages, xText does not specifically support the modification or extension of the language definition. Moreover the framework seems oriented to infer a model from a text and to translate it to another model (*model-driven development*) rather than to create real compilers. Other framework like Reuseware [11] can give the opportunity to combine this metamodels with other artifact and actually generate code: however this out from our scope because regards much model driven software developement while Neverlang focus more on compiler/interpreter generation.

6. CONCLUSIONS

In this paper we explain how the Neverlang model speeds up the development of a DSL and allows to adopt a language oriented strategy in the design of a software.

We also showed, by exploiting the modular composition model of Neverlang, how it is easy to extend and to modify a DSL to adapt it to meet the domain evolution.

Neverlang provides several mechanisms to modify an existing language definition while reusing most of its parts. It is possible to change both the syntax and semantic structure of a DSL by adding, by removing or by modifying a slice/role; by means of the endemic slices it is possible also to change the whole behavior of the language compiler/interpreter.

The reusability and extensibility of the language definition are the peculiarities of Neverlang compared to the other language generators.

Acknowledgments

The authors wish to thank Shigeru Chiba that with his view has inspired this work. This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] W. Cazzola and I. Speziale. Sectional Domain Specific Languages. In *Proceedings of the 4th Domain Specific Aspect-Oriented Languages (DSAL'09)*, pages 11–14, Charlottesville, Virginia, USA, on 3rd of Mar. 2009. ACM.

[3] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *OnBoard*, pages 1–13, Nov. 2004.

[4] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, pages 1–18, Montréal, Québec, Canada, Oct. 2007. ACM.

[5] B. Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 111–122, Venice, Italy, Jan. 2004. ACM.

[6] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.

[7] S. Freeman and N. Pryce. Evolving an Embedded Domain-Specific Language in Java. In *Proceedings of the Dynamic Languages Symposium (DLS'06)*, pages 855–865, Portland, Oregon, USA, Oct. 2006. ACM.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[9] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Apllications with Patterns, Models, Frameworks and Tools*. Wiley, 2004.

[10] G. Hedin and E. Magnusson. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, Apr. 2003.

[11] J. Johannes. Controlling Model-Driven Software Development through Composition Systems. In *Proceedings of the 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE'09)*, pages 240–253, Tampere, Finland, Aug. 2009.

[12] S. Kent. Model Driven Engineering. In M. J. Butler, L. Petre, and K. Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, LNCS 2335, pages 286–298, Turku, Finland, May 2002. Springer.

[13] D. A. Ladd and J. C. Ramming. Two Application Languages in Software Production. In *Proceedings of the USENIX 1994 Very High Level Languages Symposium*, pages 1–10, Santa Fe, New Mexico, USA, Oct. 1994.

[14] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*, LNCS 2622, pages 138–152, Warsaw, Poland, Apr. 2003. Springer.

[15] D. Steinberg, D. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Dec. 2008.

[16] A. van Deursen and P. Klint. Little Languages: Little Maintenance. *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, Mar.-Apr. 1998.

[17] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[18] J. White, J. H. Hill, J. Gray, S. Tambe, A. Gokhale, and D. C. Schmidt. Improving Domain-specific Language Reuse with Software Product-line Configuration Techniques. *IEEE Software*, 26(4):47–53, July-Aug. 2009.