Computing and Informatics, Vol. 35, 2016, 84–110

EXTENSIBLE HOST LANGUAGE FOR DOMAIN-SPECIFIC LANGUAGES

Sergej Chodarev, Ján Kollár

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
e-mail: {sergej.chodarev, jan.kollar}@tuke.sk

Abstract. Programming languages greatly influence the way how programs are created and evolved. This means that the use of appropriate language for solved problem can greatly increase developer productivity. Composition of languages can provide great help in construction of a new language from existing components and for integration of several languages that may be needed to effectively solve a complex problem. In this paper we analyze the composition problem on the two levels: composition of languages and composition of concepts in a language. Possibilities of transition from language composition to concepts composition are also presented. Based on that, we propose a framework of languages construction based on concept composition that aims to support reusability of language elements and tools. It uses common host syntax for developed languages. Their semantics is defined in a general-purpose language. Proposed approach is demonstrated on example languages developed using prototype implementation.

Keywords: Concept composition, domain-specific language, functional composition, generic syntax, language composition, metaprogramming

1 INTRODUCTION

Development of software systems often involves working with concepts from a particular domain of problems. This leads to introduction of new concepts into the programming language using mechanisms that the language provides, for example

classes or functions. However, a general-purpose language may not allow to express operations of the domain naturally. In this case a domain-specific language (DSL) can be introduced that is specially designed for solving problems in the domain [28, 37, 41]. The use of DSLs also improves the ability of programmers to comprehend program code [24].

A downside of this approach is that development of a new language may be complex, involving development of parser, compiler or interpreter, editor, and other tools. Some of the tasks can be automated using existing tools like parser generators Yacc [20], Antlr [32] or YAJCo [33]. But the syntax and semantics of all language elements still need to be defined by a language developer.

At the same time, a lot of elements in different languages may be similar. This includes simple parts as notation for comments and basic types of values, like numbers or lists, and also common operations like arithmetic or logical operators, and many other concepts. This means the process of language development can be simplified if it would be possible to reuse such elements. The language may be composed from other languages or language libraries – collections of language elements intended for reuse [38].

Currently a lot of even basic elements need to be defined repeatedly for every language. To overcome this disadvantage, domain-specific languages are often based on some existing languages. It may be a general-purpose language in case of internal DSLs [19, 16], or some generic language like XML. This simplifies development of language processor and also allows to easily compose languages that share the same base syntax. For example, this allows to embed SVG (Scalable Vector Graphics) images in the XHTML documents.

For these reasons our goal in this paper is to analyze the possibilities to utilize composition in the language development process and to propose a new approach to language development based on that.

In the first part of the paper (Sections 2, 3, and 4), different approaches for language composition problem are discussed. We look at the composition on two levels: the composition of languages and the composition of concepts in a single formal language.

In the second part (Sections 5, 6, and 7) we present a new approach for language development allowing simple composition of language libraries (reusable language fragments). It is based on the common host syntax used by all developed languages.

2 LANGUAGE COMPOSITION

It is hard to define language composition exactly and the term is mostly used without definition expecting some intuitive understanding. We will try to clarify it a bit there.

As we know a language (L) is a set of sequences of symbols from some alphabet T. These sequences are called sentences.

$$L = \{w, w \in T^*\}$$

As the composition of languages combines language elements from two (or more) composed languages, sentences of the resulted language would be combinations of sentences of the composed languages. However, sentences should not be combined at the level of symbols from the alphabet, because this would allow to describe almost every sentence in a language as a combination of some sentences of other languages that use the same alphabet. This means, that it would be too weak definition, so we should restrict the combinations.

For this reason we should define units of higher granularity. Lets call them *meaningful fragments*. Meaningful fragments are all sub-sentences of language sentences that have some specific meaning. For example, if we have sentence "`print 42`" in Python language, sub-sentence "`t 4`" is not meaningful, but sub-sentence "`print`" is meaningful.

Now the composition can be defined. Language L_c (composed) is a composition of languages L_h (host) and L_e (embedded) if every sentence of L_c can be made from the sentence of L_h by inserting meaningful fragments of L_e sentences between meaningful fragments of L_h or instead of them. Let us use the diamond symbol to denote the composition:

$$L_c = L_h \diamond L_e$$

Meaningful fragments would mostly correspond to tokens of languages. For example, let us consider a language with print statements and simple arithmetic expressions (L_p) as defined in Figure 1 and a language for definition of constants (L_d) as defined in Figure 2. We can create a composed language $L_{pd} = L_p \diamond L_d$ (see Figure 3) by allowing fragments of L_d sentences be used inside L_p sentences. More precisely, whole definitions from L_d can be placed between the statements of L_p and identifiers from L_d can replace numbers in L_p.

Statements ::= *Statement* *Statements*
Statement ::= '`print`' *Expression*
Expression ::= *Expression* '`+`' *Expression* | *Expression* '`-`' *Expression* | number

Figure 1. Example of the language of print statements L_p

Definition ::= '`let`' identifier '`=`' number

Figure 2. Example language of constant definitions L_d

Embedded language can be some standalone language that can be used separately from the host language. It can also be *language extension* – a language that cannot be used separately from the host language and mostly it is not even completely defined without the host language. In the same time, a host language can be also incomplete without an embedded language.

$Statements$ ::= $Statement$ $Statements$
$Statement$ ::= '`print`' $Expression$ | $Definition$
$Expression$ ::= $Expression$ '`+`' $Expression$ | $Expression$ '`-`' $Expression$ | number
| identifier
$Definition$ ::= '`let`' identifier '`=`' number

Figure 3. Example composed language $L_{pd} = L_p \diamond L_d$

Examples of standalone embedded languages include JavaScript embedded in HTML documents or C language embedded in Yacc parser generator specifications. In the first case both languages can be used separately or fragments of JavaScript can be inserted inside HTML documents at some specific places.

In the case of Yacc specifications, the host language defines rules of syntax analysis that without the embedded language can be used to generate only a syntax checker. To perform some actions in the parser beside the checking, it is needed to include fragments of the C language. Those fragments, however, are not expressed in the pure C, but in a C augmented with Yacc attribute references (for example `$1`). We may consider this as another case of language composition, where the C language (L_C) is composed with the Yacc references language (L_r) to form an Yacc actions language $L_a = L_C \diamond L_r$. This language is finally embedded into Yacc grammar definition language (L_g) to produce the complete Yacc language (L_y).

$$L_y = L_g \diamond L_a = L_g \diamond (L_C \diamond L_r)$$

An example of language extension is the Blocks extension by Apple Inc. that adds support for anonymous functions to the C language [1]. Another example is provided by additions made in the process of the language development. New versions of languages may include extensions that add some new syntax, for example addition of generic types in Java 5.

In [10] a more implementation-oriented view on the language composition is presented. Language development tools are classified according to types of composition that they support without modification of the original language implementation. It recognizes four types of language composition:

1. *Extension* of a language, where existing base language is modified to include additional elements or some elements are removed from the language.
2. *Language unification*, where two existing languages are combined and interweaved.
3. *Self-extension* of the language, where the language is extended using a program written in the extended language itself.
4. *Extension composition* allowing to compose several extensions of a base language.

Implementation of the language composition is complicated by several factors including [22]:

- *Grammar subclasses.* Most of parser generators allow to use only a subset of context-free grammars – grammar subclass (e.g. LL(k), LALR, etc.). However, these subclasses are mostly not closed under composition. This means that composition of grammars from some grammar subclass may result in a grammar that does not fit into this subclass and therefore cannot be processed using the same parser generator. In such case a manual modification of resulting grammar is required.
- *Separate lexical analysis.* Language processors usually use separate phases of lexical analyzing and parsing. Because of this, it is difficult to compose languages with different lexical grammars, since the same sequence of characters may produce different tokens in composed grammars.

A lot of problems can be solved by using a parser generator, that can process any context-free grammar [22]. On the other hand, even using more restrictive language class, it is possible to compose languages, albeit not in general case.

To avoid the problems of parser generation, it is also possible to combine separate parsers for composed languages using language embedding, where parsers are switched during analysis [25], or preprocessing, where language processors are run sequentially [35].

Semantics composition. Another problem of language composition is composition of semantics. Its realization greatly depends on implementation of language processors. It is obvious, that composition of languages implemented independently and based on different tools would be very difficult and may require substantial changes to language implementation. Moreover, if language semantics is described in a general-purpose language, it may be difficult to extend or compose it. One of the ways how to avoid modification of the base language implementation in the case of language extension is implementation of translator to the base language [4].

On the other hand, if composed languages are implemented using the same tool, it can provide support for composition of semantics. The practicable execution depends on the way how semantics is defined. It may be, for example:

- Modification of translation rules [40]
- Extension or composition of semantics model [40, 39]
- Inheritance of attribute grammar rules [27].

3 CONCEPT COMPOSITION

While the composition of grammars is quite rarely used, the composition of concepts is used all the time. We would use the term *concept composition* for cases where several existing concepts are combined to form a structure. Concepts used in the

composition can be either part of the language itself, or they can be defined by a language user – a programmer. Moreover, the resulting structure can be named and by this way a new concept can be defined.

For example, concepts of multiplication operation ($\times$), circle radius (r) and constants 2 and π can be composed to form a concept of circle perimeter.

$$P = 2 \times \pi \times r$$

The composition of languages or dialects that use the same syntax can also be considered as a concept composition because elements of such languages are actually just different concepts of a host language.

A language specifies the ways how concepts can be composed. It can support several types of composition with different rules and constraints and different types of concepts may be composed differently. We have identified four basic *composition approaches*. Composition rules in languages are mostly based on these types:

1. structural composition (composition of code elements),
2. functional composition (composition of data flow),
3. object composition (composition of data structures),
4. aspect composition (composition by interweaving).

These types are not mutually exclusive. In reality they are just different views on the composition based in different properties of the program. For this reason they are usually not used in a pure form but rather combined. For example procedural languages usually use both the structural and functional composition. Procedures are composed from statements using structural composition, and statements can be composed from several functions using functional composition. At the same time functional composition uses structural composition on the level of program code.

3.1 Structural Composition

The essence of structural composition is combination of concepts based on their position in the program code. Remarkable example is the composition of elements in XML based languages. Language specification in this case defines allowed elements and places in the document structure, where they can be placed. One of the XML schema languages (like DTD, W3C XML Schema or Relax NG) can be used to express these composition rules.

The rules of structural composition define how concepts can be combined and nested in the program code. Structural composition actually corresponds to the language syntax and rules of structural composition can be described using a grammar. This means that the composition of concepts based on the language grammatical structure is actually an instance of the structural composition.

However, rules of the structural composition do not need to be described directly in the language syntax. Part of them can be moved into semantic processing of the

program. This means that they are checked only at the stage of semantic analysis or execution.

This is used by many internal DSLs, or even libraries. For example OpenGL interface defines special functions like *glBegin()* and *glEnd()* that actually provide new control structures and new rules of structural composition that are not checked by the compiler of programming language.

Structural composition can be considered as a low-level composition approach the other approaches build on. It itself is useful for languages oriented towards expressing structured data, where other types of composition may not be needed.

3.2 Functional Composition

The basic principle of functional composition is the application of a function resulting from other functions. This can be expressed using the composition operator:

$$(f \circ g)\ x = f(g\ x)$$

Functional composition expresses the flow of data between functions that process it. It is commonly used in languages as a way to form expressions, where functions and operators (that can be treated a special notation for functions) are composed. In addition to functions, literals and variables are used in such composition to describe data used in computation.

Functional composition can be restricted by a type system, so functions cannot be composed in arbitrary way. In this case, every function has declared types of their arguments and produced value. Composition $f \circ g$ is possible only if argument type of f is compatible with the type of g. This means that the type of functions and values defines composition constraints.

In functional programming languages, functional composition is a basic tool for constructing programs. In these languages the program is formed using functions that are defined as expressions composed from other functions. However, function composition can be used to express manipulation not only with data, but also with computations. This is done using monads [30]. Monad is a data type that allows to abstract execution and data flow between computation steps.

3.3 Object Composition

In this section we would use the term *object* to describe a data structure containing named elements of possibly different types. The elements can be data (simple or other objects) or functions (called methods of the object). Functions associated with an object define its *behavior*. Objects with the same structure are usually grouped into *classes* and their properties are then described in the classes.

If all data members of an object are hidden and the only way to manipulate with object state is provided by its methods, then we can consider the object as an abstract data structure. However, this is not always the case.

Therefore, the basis of the object composition is the composition of data structures. We can distinguish two basic types of the object composition:

Composition and aggregation (term "composition" is used in a narrower meaning there) is a definition of data structure consisting of other data structures that become its elements. More strength relation, where elements cannot exist without a compound structure, is usually called composition. The weaker relation, where elements of a composed structure can also exist separately, can be called aggregation.

Inheritance or extension of data structures is a construction of data structure based on other structure (parent). Child structure inherits elements and behavior from parent structure and can redefine some of them and add its own elements.

Relations between classes of objects can be expressed graphically using UML class diagrams [17]. This allows to visualize relations of structures that are mostly scattered in the program code.

Object composition is suitable to express different types of data structures. So if we consider a program code to be the data structure (for example abstract syntactic tree of graph), then we can see object composition as a generalization of structural composition. This property is used to describe abstract syntax of a programming language using the relations of object composition. For example in [33] programming language syntax is defined using annotated classes and their interconnection.

3.4 Aspect Composition

The aspect composition is an approach where some concepts, called aspects, can alter other concepts behavior in a way that was not envisaged ahead [23]. In other words, "AOP (aspect-oriented programming) can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications on hold over programs written by oblivious programmers" [14].

Aspects can be described as statements in a form "In programs P, whenever condition C arises, perform action A" [14], where condition may be evaluated statically based on the program code or dynamically based on its execution state. The program P itself does not have information on what aspects may be applied to it. The final program is then interweaved from separately defined concepts based on rules specified in aspects.

The aspect composition is not completely independent from other composition types. We can look at the aspects weaving as at the objects modification. Monads from functional programming can also be used for adding aspects into programs [8].

4 TRANSITION FROM LANGUAGE COMPOSITION TO CONCEPT COMPOSITION

While the language composition is still a complex task despite of research in this area [10, 22, 25, 5, 11], the concept composition is well understood and widely used. This is the reason that leads us to use the methods of concept composition for the composition of languages.

This transition requires to lower the role of grammar in language definition. Basically, language elements need to become concepts in some host language. There are at least two ways how to achieve this:

1. to use the same concrete syntax for composed languages,
2. to use projectional editing.

The first way is used in the internal domain-specific languages. They are based on the syntax of the existing general-purpose language, and so their elements are just concepts of the host language.

Another example is provided by languages based on XML or some other existing generalized language. These languages define concepts that use syntactic shapes provided by a generalized language, for example XML elements and attributes.

Projectional editing of the program code, on the other hand, keeps different notations for languages on the surface when using unified representation internally. The main form of program code is an internal graph-based representation. Program is displayed to a programmer using a projection that is directed by a set of rules for transformation of internal structure to a visual form, for example text, diagram, or table [15]. Programmer is then allowed to issue editing commands at the projection which modify the internal structure and the projection is updated accordingly. In case of textual projection, it is important to make editing commands similar to editing text in a conventional editor.

In this case, the syntax becomes only a matter of projection and actual information of a program code is stored in a different form, not visible for a language user. At the same time, there is no text parsing involved, since internal representation is edited directly. Therefore, the elements of languages developed using a language workbench are actually only concepts of an internal representation language (which is usually not textual). This means that in this case the composition of languages corresponds to the composition of concepts inside a single language. Textual composition is only its projection which is not required to be unambiguous.

Projectional editing is used by some language workbenches, for example JetBrains MPS [9] or Intentional Workbench [34].

5 LANGUAGE CONSTRUCTION ON COMMON HOST SYNTAX

The transition described above allows to build tools for language development that would embrace composition in language development process. This would make possible to construct languages based on the existing language components.

Different approaches can be taken to achieve this goal leading to different resulting tools. For example, the choice to use projectional editing may lead to a tool similar to existing language workbenches. We decided to base our approach on the following principles:

1. Concept composition instead of language composition and a *common generalized language* as a tool for this transition.
2. *Functional composition* (and eventually structural composition) for interconnecting language elements in programs.
3. Definition of language semantics using a general-purpose language that would allow *integration of a developed language* into a wider software system.

Common host syntax was chosen instead of projectional editing because it allows to keep textual form of code as a primary form. This makes it possible to use existing tools expecting textual representation of programs, for example version control systems.

Moreover, textual representation of a program is more transparent. This means that all properties of a program are visible directly in the code and it is possible to use any tool to read and manipulate them. On the other hand, projectional editing can hide some program properties from textual projection and it may be required to use different projections to change them.

Another important advantage of common syntax is the reuse of knowledge of language users. In a case where several languages are used in a project, the time of learning a new language can be decreased by the use of a common particular syntax.

However, the use of common host syntax is not suitable for all languages. Host language introduces restrictions on the notation used by a developed language. While it is possible to design a host language that is quite flexible, it obviously cannot match the custom grammar. Increased flexibility of notations can also make the host language more complex and reduce advantages of this approach. So it is possible to effectively use a host language only if notations that it provides are sufficient for description of domain concepts and operations.

Functional composition is used as a basis for composition of language elements in our approach. This allows to nest elements, so inner elements become arguments of outer element. Using this, the flow of data in a program can be expressed naturally. This also allows to express structure of a program by allowing blocks of code to be arguments of other program elements effectively using a structural composition.

Semantics of language elements is expressed using code in a general-purpose language. Fragments of code are associated with each language element and control transformation of elements input into its output together with the manipulation of the execution environment. The implementation general-purpose language plays

the role of meta-language for a developed language, because it allows to manipulate values and code of DSL programs.

This approach makes it possible to easily integrate developed language with the rest of the developed software system. It also allows to implement actions performed by language elements using tools and methods familiar to a programmer. In addition, thanks to the use of functional composition, the definition of language semantics is similar to the definition of functions.

6 EXTENSIBLE HOST LANGUAGE FRAMEWORK

A framework for domain-specific language development was designed based on the described principles. It uses *a generic language* that defines common syntax for a whole family of domain-specific languages. We would call it *Extensible Host Language Framework*[1]. It consists of the host language and a set of tools for processing languages based on it (see Figure 4).

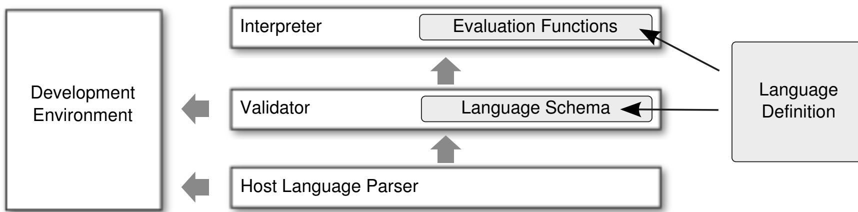

Figure 4. Extensible Host Language architecture

The basic tool is *a parser* of the generic host language that analyses program text and produces a *skeleton syntax tree (SST)* based on it. Unlike the abstract syntax tree that contains concrete elements, the skeleton syntax tree contains generic language elements that correspond to basic syntactic shapes provided by the host language [3].

These shapes are used to define elements of the concrete DSL. A DSL is actually a subset of the generic language. Its elements with their properties are defined in *a language schema*. This information is used by other tools to properly process programs in the language. The most notable of these tools is *a validator*, that performs static checking of programs based on a language schema. This makes possible to detect some errors before program evaluation.

The next part of the architecture is *an interpreter*. Its task is to evaluate a program after it is parsed and validated. The interpreter is actually a framework that allows language author to define subprograms in form of evaluation functions that would take care of actual evaluation of DSL elements. The interpreter then traverses a DSL program and executes evaluation functions corresponding to language

[1] This is obviously an allusion to Extensible Markup Language (XML).

elements. This allows to process a program in different ways depending on language needs. It is possible to immediately execute actions specified in a program, create a model based on it, or translate a program into other language.

Evaluation functions are implemented in a general-purpose language. In our case the implementation language is Java, and evaluation functions have a form of Java methods.

The language definition is also divided into composable *modules*. A module is a language definition unit, that contains several language elements with their declaration and implementation. A language based on the generic syntax can be constructed from one or several modules.

6.1 Syntax

A language that is designed only as a host language for other languages cannot define concrete language structures. These are provided by guest languages. Host language can define only basic simple structures (like symbol or number literal) and generic shapes that allow to create composed structures. Concrete elements of a guest language must be defined as a specializations of these shapes. Result of parsing in case of host language is then a skeleton syntax tree.

Well-known examples of generic host languages include XML [6] and S-expressions [26]. However, both these languages have a significant disadvantage in readability. While XML syntax is too noisy [31], S-expressions are too uniform making it difficult to visually distinguish different language structures.

For this reason the syntax of the proposed host language was designed to make it flexible and yet easily readable. It includes several shapes for common structures (see Figure 5). Two of them are data structures: lists and maps (associative arrays), and other two are control structures that allow to express structure of code: combinations and blocks. *Combination* is simply a sequence of language elements written on the same line. They can be nested using parentheses and by default they are interpreted as a function or operator application. In that way they are quite similar to lists in Lisp. On the other hand, *block* is a sequence of language elements written on separate lines, but with the same level of indentation. We can see combinations as horizontal structures and blocks as vertical ones.

Besides that, the host language defines literals for values of basic data types like numbers, strings, and booleans. There are also symbols, that have a special role – they are used to express names of both language elements and user defined concepts.

The concrete syntax was chosen to avoid unnecessary noise and to make its constructs similar to popular general-purpose languages. Notation for data literals is similar to JSON – data exchange format derived from common data structures in JavaScript [7].

Syntax for combinations is similar to function application in Haskell or a list in Lisp (but with parentheses only for nested combinations). Additionally, infix notation is supported – symbols constructed from special characters are considered as operators and combinations with infix operators are translated into combination

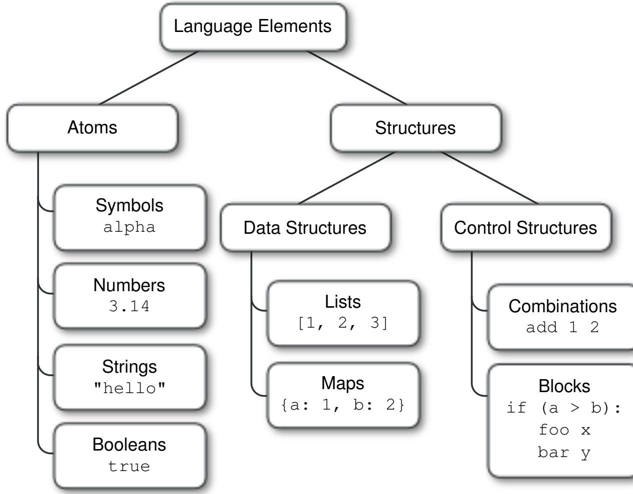

Figure 5. Elements of the proposed generic host language

with operator in the first position. Blocks are defined using indentation, similarly to Haskell or Python. This makes scripts concise and clean.

In Figure 6 you can see the specification of context free grammar of the proposed host language in extended BNF (braces represent repetition of an element 0 to n times). Special lexical symbols *INDENT* and *DEDENT* denote the increase and decrease of line indentation level.

block ::= { *expression NEWLINE* | *expression-with-block* }
expression ::= *combination* { *operator combination* }
expression-with-block ::= *combination* ‘`:`’ *NEWLINE INDENT block DEDENT*
combination ::= *term* { *term* }
term ::= *literal* | ‘`(`’ *expression* ‘`)`’
literal ::= *symbol* | *string* | *number* | *boolean* | *list* | *map* | ‘`none`’
boolean ::= ‘`true`’ | ‘`false`’
list ::= ‘`[]`’ | ‘`[`’ *expression* { ‘`,`’ *expression* } ‘`]`’
map ::= ‘`{}`’ | ‘`{`’ *key-value* { ‘`,`’ *key-value* } ‘`}`’
key-value ::= *symbol* ‘`:`’ *expression*

Figure 6. Grammar of the proposed generic host language

6.2 Evaluation

As we have said, the programming language is defined as a set of sentences that can be constructed using some alphabet. Usually, the alphabet is simply a set of ASCII or Unicode characters. However, if we define lexical analysis as a separate step, we can say that the alphabet is formed by the whole lexical units.

In case of languages based on a generic language, the alphabet consists of the elements or shapes (composed elements) of the generic language, for example S-expressions or XML elements. In Figure 7 you can see the example sentence written as S-expression and its division into basic units that can be considered as units of the alphabet.

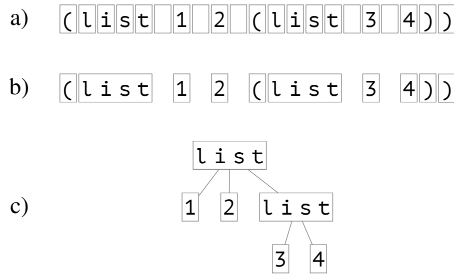

Figure 7. Different views on language alphabet: a) characters, b) lexical units, c) generic language elements

What is important, the elements of the generic language alphabet are not organized in a sequence, but they have a hierarchical structure – they form a skeleton syntax tree. These structured elements are then used as a basis for definition of new languages.

To define a language based on the generic host syntax, it is needed to specify a set of language elements and their properties. Every language element in XHL is represented using *a symbol*. These symbols can occur in a program. In case where element symbol is the first item of a combination, the rest of the combination is considered as arguments of element application, including a block, that may follow a combination.

The processing of a program consists of several steps, where each step receives results of the previous one:

1. syntactical analysis (producing a skeleton syntax tree),
2. static checking (producing an enriched SST),
3. evaluation (potentially producing a model of the program or other representation),
4. execution of the model (if it was produced).

6.3 Evaluation Functions

Every application of language element in a program is processed by its evaluation function. Arguments of the application are passed as arguments to the evaluation function and a return value of the function becomes the result of application. In XHL, evaluation functions are implemented in the Java language as methods of an object corresponding to a language module. During the evaluation, a module instance is created and for each application of language element, corresponding method is executed using Java reflection mechanism.

For example, in Figure 8 an evaluation function for addition operator is presented. Evaluation functions are marked with special annotation `@Element` with optional parameter for element name (for cases where it does not match the method name). If program code would include the combination like "`1 + 2`", the evaluation function would be executed with arguments `1` and `2` and would compute the result of the expression.

```
@Element(name = "+")
public Double plus(Double arg1, Double arg2) {
return arg1 + arg2;
}
```

Figure 8. Evaluation function for addition operator

If some combination is part of another combination (e.g. `f (g 1)`), the result from the evaluation function of the inter element is passed as an argument to the evaluation function of the outer element.

Although the evaluation function is implemented as a method, it have several special properties:

1. It can process code of its parameters instead of evaluating them.
2. It can manipulate the environment of evaluation that contains bindings of program names.
3. It can return a special object instead of the value that will be evaluated on request or will generate code corresponding to the operation in a target language.

The control over the evaluation of arguments is based on the fact that arguments of evaluation function can be passed using one of the two modes:

1. *By value*, where an element representing the argument is evaluated (using its evaluation function) and the result is passed to the evaluation function.
2. *Symbolically*, where the evaluation function receives code of an element representing the argument as a fragment of the skeleton syntax tree.

These properties allow evaluation functions to flexibly evaluate instances of language elements in a program. They can also introduce and manipulate names in

a program and implement custom evaluation strategies using the ability to process code of parameters.

Example in Figure 9 presents evaluation function for the *define* element that allows to define named constants. It has two parameters – the name to be defined and the value that would be assigned to the name. The first parameter is marked as symbolic, it cannot be evaluated since the name should not be defined yet. The function is also an example of a manipulation with the environment – it defines a new binding using the *putSymbol* method. Evaluation function could also create a new environment for local variables.

```
@Element
public void const(@Symbolic Symbol symbol, Double value)
        throws EvaluationException {
    if (evaluator.hasSymbol(symbol))
        throw new EvaluationException(symbol.getPosition(),
            String.format("Symbol %s is already defined", symbol));
    evaluator.putSymbol(symbol, value);
}
```

Figure 9. Evaluation function for *define* element

6.4 Schema

It is useful to perform checking of a program before its evaluation, especially in cases where evaluation includes complex computations or has side effects. Static checking can be performed even interactively while a programmer is writing code. The checking system is based on properties of a language and its elements specified in the language schema.

Language schema provides description of language elements defined by a particular module. It is described using a language that is itself based on XHL. Schema contains a list of all elements and their properties including:

- Type of the value produced by the element.
- Parameters – their type and passing method (by value or symbolically).
- Symbols defined by element application.
- Human-readable description of the language element.

Example of the language schema declaring two elements from previous examples is in Figure 10. Element "+" is an operator with two numeric parameters that are passed by a value. Operator produces a numeric value as a result. Second declaration describes element "const", used to define a numeric constant. Its parameters are a symbol that would be bound, and a value. First parameter must be passed symbolically, because at the point of element application it cannot be evaluated.

The *defines* section says that the element binds a symbol that is given as its first argument to the value of type *Numbers*.

```
element (+):
    doc "Add two numbers."
    params [val Number, val Number]
    type Number

element const:
    doc "Define a numeric constant."
    params [sym Symbol, val Number]
    defines 1 Number
```

Figure 10. Example of language schema definition

Static checking is based on the type system. An element has specified its type and types of expected parameters. Language schema also declares what new names are defined by each language element and what are types of objects bound to these names. Relation of generalization between types can also be specified. The validity of a program is then checked based on the compatibility of types of nested elements.

Language schema is a source of knowledge about the language the use of which is not limited to the static checking. One of the other uses of the knowledge is an interactive help system in a development environment. It can display the documentation for language elements and also provide the automatic completion of code.

6.5 Composition

Presented approach supports composition of languages based on the provided framework by the way of composing language concepts. Language module can import selected concepts from other modules. By this way both language extension and language unification according to [10] are supported, as well as composition of extensions. In case of extension, a new language would add concepts to existing language, or it may also replace existing concepts or remove them. In case of unification, unified languages are imported as modules into a new language and interconnecting concepts are added.

The composition is, however, limited on the level of semantics of language elements. Since semantics is expressed using evaluation functions in the general-purpose language, it cannot be easily altered. This means that the element definition must be replaced with a new one in cases where its modification is required.

Another issue is interconnection of composed elements. XHL provides three way of communication for evaluation functions:

1. values passed as arguments and results of application,

2. evaluation environment – assignment of values to names in program,
3. internal environment – internal state of a module.

The third one poses potential problem for composition because internal environment may not be accessible for other modules and therefore their communication would not be possible. To enable composition, modules should define complete interface for accessing their internal state.

7 EXPERIMENTS

A prototype of the system based on presented concepts, with Java language as an implementation language and a language for definition of evaluation functions, was used to implement several experimental domain-specific languages. There we present two of them to illustrate the use of proposed approach.

7.1 State Machine Language

The state machine language is based on the example from [16]. It allows to define events, commands and states. Each state can execute some commands and allows transitions to different states on specified events.

An example program in the language is presented in Figure 11. You can see that events and commands are defined inside corresponding blocks. A colon operator (`:`) is used to connect event (or command) name and its code. The colon there is used as a language element in a form of infix operator that binds a new object to a name in the evaluation environment[2]. The elements *events* and *commands* each define its own version of the colon operator locally inside a block. State definition also uses a block and the arrow operator (`->`) to express transitions.

You can see a fragment of the language schema in Figure 12. Notice a nested element declaration (the colon operator inside the *events* element). This is used to declare local elements that will be accessible only inside a block defined by the outer element.

The schema also contains information about new names defined by the elements. For example, the colon operator defines a global name corresponding to its first parameter for an object of type *Event*. The *state* element also defines a new name, but in this case it is defined backward. This means that the name can be used in a program before its declaration. Language implementation can retrieve a list of all backward defined symbols and initialize them in the beginning of the evaluation.

A fragment of the implementation is presented in Figure 13. It shows evaluation functions for elements *state* and `->`. The implementation of evaluation functions is very simple. Most of them only alter internal processing state in some way to construct the model representing the state machine.

---

[2] While the colon can be used as custom operator, it is also a part of the host language syntax – it is used to introduce blocks and to separate key from value in maps.

```
events:
    doorClosed:  "D1CL"
    drawOpened:  "D2OP"
    lightOn:     "L1ON"
    doorOpened:  "D1OP"
    panelClosed: "PNCL"
resetEvents [doorOpened]
commands:
    unlockPanel: "PNUL"
    lockPanel:   "PNLK"
    lockDoor:    "D1LK"
    unlockDoor:  "D1UL"
state idle:
    actions [unlockDoor, lockPanel]
    doorClosed -> active
state active:
    drawOpened -> waitingForLight
    lightOn    -> waitingForDraw
state waitingForLight:
    lightOn -> unlockedPanel
state waitingForDraw:
    drawOpened -> unlockedPanel
state unlockedPanel:
    actions [unlockPanel, lockDoor]
    panelClosed -> idle
```

Figure 11. Example program in the state machine language

```
newtype State
newtype Event
newtype Command

element events:
    params [val Block]
    element (:):
        params [sym Symbol, val String]
        defines_global 1 Event

element state:
    params [sym Symbol, val Block]
    defines_backward 1 State

element (->):
    params [val Event, val State]
```

Figure 12. Fragment of the state machine language schema

```
public class StateMachineModule extends GenericModule {
    private Event[] resetEvents = null;
    private State startState = null;
    private State currentState = null;

    @Element
    public void state(@Symbolic Symbol name, @Symbolic Block blk) {
        currentState = (State) evaluator.getSymbol(name);
        evaluator.eval(blk);
        if (startState == null)
            startState = currentState;
        currentState = null;
    }
    @Element(name = "->")
    public void transition(Event trigger, State target) {
        currentState.addTransition(trigger, target);
    }
```

Figure 13. Fragment of the state machine language implementation

7.2 Entities Language

Another developed language provides an example of the composition of language libraries. The language allows to define entities and their properties. It also provides a way to define validation rules as boolean expressions. The language includes modules providing relational and logical operators. In this case, the result of program evaluation is the generated Java code containing classes representing specified entities.

The example program in the entities language is presented in Figure 14. It defines two entities: *Employee* and *Department* and lists their properties with types. Definition also contains validation constraints on length of some properties.

The language implementation defines only a few elements such as *module*, *entity*, or *validate* and property types like *int* and *string*. Operators used in validation rules are imported from external modules. An exception is the *length* function that provides values for operators to work on.

Validation expressions cannot be evaluated in the time of the DSL program evaluation, since they operate on values that would be known only at the stage of execution of the generated code. This means, that the result of validation rules evaluation would be the generated Java code representing these expressions. This requires to use evaluation on request, where language elements do not execute their corresponding operations, but instead return special objects, called *producers*, containing all operation parameters and a method for evaluation of the operation. These objects also provide a method for generation of Java code corresponding to the operation.

```
module company:
    entity Employee:
        id : int
        name : string
        role : string
        worksAt : Department
        validate:
            (length name < 20) & (length name > 0)
    entity Department:
        id : int
        description : string
        validate:
            length description < 20
```

Figure 14. Example of a program defining entities a validation rules

Actually, most of the elements from language libraries (reusable language modules) return producers instead of values. This allows libraries to be used in both cases: where direct evaluation is needed and also for the code generation. At the same time, if other language elements expect result of the evaluation, producers are transparently evaluated by the framework. Other way round, if an element expects a producer, but receives a value, it is automatically wrapped – a new constant producer is created based in the specified value.

An example of this technique is presented in the implementation of the *length* element in Figure 15. It returns a special object implementing the *Producer* interface with *toCode* method for code generation.

8 RELATED WORKS

The goal of the proposed framework is to simplify development of domain-specific languages compared to external DSLs by leaving out the need to specify details of concrete syntax. It also allows the composition of languages and reuse of their parts thanks to the common concrete syntax and evaluation system.

However, several other tools and techniques exist with similar goals. First of all, there are *language workbenches* that provide integrated development environments for development of languages and for the use of the languages [15]. Some of them, like Intentional Workbench [34] and JetBrains MPS [9], use projectional editing. Projectional editing allows composition of languages and language libraries, because it avoids problems of grammar composition [39]. On the other hand, it does not allow to use existing tools expecting textual representation of the code, like revision control systems.

Graphical modeling tools like MetaEdit+ [36] are in the similar position as language workbenches, because they use projectional editing with graphical representation. Multilevel language infrastructure approach [2] even provides support

```
public class EntityModule extends GenericModule {
    ...
    @Element
    public Producer<Double> length(Attribute attr) {
        return new LengthProducer(attr);
    }
    public static class LengthProducer implements Producer<Double> {
        public final Attribute attr;
        public LengthProducer(Attribute attr) { this.attr = attr; }
        @Override
        public Double toValue() {
            throw new UnsupportedOperationException("...");
        }
        @Override
        public String toCode() { return attr.getName() + ".size()"; }
    }
}
```

Figure 15. Evaluation function for the *length* element

for self-extension by allowing to define new concept types in a modeling language itself.

There are also language workbenches that are based on textual representation of programs, for example Spoofax [21] and Xtext [12]. They provide tools to define syntax and semantics of a language and its editor with support for language composition. In contrast to this, the proposed framework does not require to define concrete syntax of a language. This makes language definition simpler, but at the same time less flexible. More detailed discussion of composition in language workbenches is provided in [39].

Composition of languages is also supported by other language processor generators including Lisa that supports composition based on multiple attribute grammar inheritance [27, 29]. A language specification can inherit from several existing languages and inherited properties can be extended or overridden. The use of attribute grammar to express language semantics allows to easily modify semantic rules in language extensions. In contrast to this, in our approach evaluation functions that describe semantics cannot be altered and must be replaced in order to change their behavior. On the other hand, definition of languages based on the generic syntax leads to reuse of lexical and basic syntactic rules across all languages.

Another example is YAJCo parser generator [33] which uses object model as a basis for language definition. A model is implemented using Java classes with annotations that specify concrete syntax and other properties of language element. Composition of languages is therefore based on the object composition, while approach presented in this paper is based on functional composition.

There is another proposal for generic language called Gel [13]. It defines rich generic syntax similar to existing languages like Java or CSS. However, it is focused only on the syntax and does not propose tools for language definition and processing. Even though the Gel syntax is much richer than our proposal which also makes it more complex, it could also be used as a syntactic basis in our approach.

In addition, this paper discusses the problem of language composition that was covered also in [10, 39]. Our work adds the notion of concept composition and its relation with the composition of languages.

9 CONCLUSION

In this paper we presented our proposal of a new language development approach based on the use of concept composition and generic syntax. For this reason a considerable attention was given to the language and concept composition problem.

First of all, we presented a definition of language composition that should clarify the use of the term. We also analyzed and classified concept composition approaches and discussed the relation between composition of languages and concepts.

Presented knowledge was used to propose a new approach for DSL development based on common generic syntax and functional composition. It allows to define a language as a collection of elements. Elements are grouped in modules that can be included in other languages as language libraries.

For the purpose of presented approach the new generic host language was designed. Although it is based on elements used in common programming languages, it generalizes them to allow flexible definition of new languages.

A new approach for evaluation was designed based on evaluation functions in the implementation language. They have the metaprogramming capabilities including the possibility to manipulate directly with the program code and to evaluate operations on request and in different ways depending on actual context. Proposed concept also includes a system for static checking of programs based on properties of the elements defined in the language schema.

The approach was demonstrated on example languages developed using the prototype of the proposed framework. They show that it is possible to modularize language definition based on the proposed approach and to use defined language modules in languages with different evaluation models.

Limitations of the approach are fixed syntax that would not be suitable for every domain and the way of semantic definition that does not allow an extension of already defined concepts, but only their replacement.

Our further research in this area may include development of additional tools for language processing and use including development environment [18]. It may be also possible to automatize the creation of semantic model from the language or even consider different, more declarative, way of the semantic definition.

Other goal is to raise the expressive power of the language schema by supporting more constraints that would check validity of programs and therefore allowing

more precise declarative specification of language properties. This would allow more precise static and also dynamic checking of programs.

Acknowledgement

This work was supported by project VEGA 1/0341/13 "Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication".

REFERENCES

[1] Apple Inc.: Blocks Programming Topics. 2011, Available on: `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Blocks/Blocks.pdf`.

[2] Atkinson, C.—Gutheil, M.—Kennel, B.: A Flexible Infrastructure for Multilevel Language Engineering. IEEE Transactions on Software Engineering, Vol. 35, 2009, No. 6, pp. 742–755.

[3] Bachrach, J.—Playford, K.: D-Expressions: Lisp Power, Dylan Style. 1999, Available on: `http://people.csail.mit.edu/jrb/Projects/dexprs.pdf`.

[4] Bravenboer, M.—Visser, E.: Designing Syntax Embeddings and Assimilations for Language Libraries. In: Giese, H. (Ed.): Models in Software Engineering. Workshops and Symposia at MoDELS 2007. Springer Heidelberg, Lecture Notes in Computer Science, Vol. 5002, 2008, pp. 34–46.

[5] Bravenboer, M.—Visser, E.: Parse Table Composition. Separate Compilation and Binary Extensibility of Grammars. In: Gasevic, D., van Wyk, E. (Eds.): Software Language Engineering (SLE 2008), Springer Heidelberg, Lecture Notes in Computer Science, Vol. 5452, 2009, pp. 74–94.

[6] Bray, T.—Paoli, J.—Sperberg-McQueen, C.—Maler, E.—Yergeau, F.: Extensible Markup Language (XML) 1.0. 1998.

[7] Crockford, D.: The Application/JSON Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. Available on: `http://www.ietf.org/rfc/rfc4627.txt`.

[8] De Meuter, W.: Monads as a Theoretical Foundation for AOP. International Workshop on Aspect-Oriented Programming at ECOOP, Vol. 25, 1997, pp. 31–36.

[9] Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. November 2004. Available on: `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`.

[10] Erdweg, S.—Giarrusso, P. G.—Rendel, T.: Language Composition Untangled. Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA), 2012, Article No. 7.

[11] Erdweg, S.—Rendel, T.—Kästner, C.—Ostermann, K.: SugarJ: Library-Based Syntactic Language Extensibility. Proceedings of Conference on Object-

Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2011, ACM, pp. 391–406.

[12] Eysholdt, M.—Behrens, H.: Xtext: Implement Your Language Faster Than the Quick and Dirty Way. Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '10), The ACM SIGPLAN Conference on Systems Programming Languages and Applications: Software for Humanity (SPLASH '10), ACM, 2010, pp. 307–309.

[13] Falcon, J.—Cook, W. R.: Gel: A Generic Extensible Language. Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages (DSL 2009), Springer-Verlag, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5658, 2009, pp. 58–77.

[14] Filman, R. E.—Friedman, D. P.: Aspect-Oriented Programming Is Quantification and Obliviousness. Technical report, RIACS, 2000. Available on: `http://www.riacs.edu/research/technical_reports/TR_pdf/TR_01.12.pdf`.

[15] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? 2005. Available on: `http://martinfowler.com/articles/languageWorkbench.html`.

[16] Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 2010.

[17] Fowler, M.—Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. 2nd Ed. Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[18] Henriques, P.—Pereira, M. J. V.—Mernik, M.—Lenic, M.—Gray, J.—Wu, H.: Automatic Generation of Language-Based Tools Using the Lisa System. IEEE Proceedings – Software, Vol. 152, 2005, No. 2, pp. 54–69.

[19] Hudak, P.: Modular Domain Specific Languages and Tools. Proceedings of the 5th International Conference on Software Reuse (ICSR '98), IEEE Computer Society, 1998, pp. 134–142.

[20] Johnson, S.: Yacc: Yet Another Compiler-Compiler. Technical report, Bell Laboratories Murray Hill, NJ, 1978.

[21] Kats, L. C. L.—Visser, E.: The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10), ACM, 2010, pp. 444–463.

[22] Kats, L. C. L.—Visser, E.—Wachsmuth, G.: Pure and Declarative Syntax Definition: Paradise Lost and Regained. Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10), ACM, 2010, pp. 918–932.

[23] Kiczales, G.—Lamping, J.—Mendhekar, A.—Maeda, C.—Lopes, C.—Loingtier, J.-M.—Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (Eds.): Object-Oriented Programming (ECOOP '97), Springer Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1241, 1997, pp. 220–242.

[24] Kosar, T.—Oliveira, N.—Mernik, M.—Pereira, M. J. V.—Črepinšek, M.—da Cruz, D.—Henriques, P. R.: Comparing General-Purpose and

Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, 2010, No. 2, 247–264.

[25] Krahn, H.—Rumpe, B.—Völkel, S.: Monticore: Modular Development of Textual Domain Specific Languages. In: Paige, R. F., Meyer, B., Aalst, W., Mylopoulos, J., Rosemann, M., Shaw, M. J., Szyperski, C. (Eds.): Objects, Components, Models and Patterns, Springer Berlin Heidelberg, Lecture Notes in Business Information Processing, Vol. 11, 2008, pp. 297–315.

[26] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM, Vol. 3, 1960, No. 4, pp. 184–195.

[27] Mernik, M.: An Object-Oriented Approach to Language Compositions for Software Language Engineering. Journal of Systems and Software, Vol. 86, 2013, No. 9, pp. 2451–2464.

[28] Mernik, M.—Heering, J.—Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR), Vol. 37, 2005, No. 4, pp. 316–344.

[29] Mernik, M.—Žumer, V.: Incremental Programming Language Development. Computer Languages, Systems & Structures, Vol. 31, 2005, No. 1, pp. 1–16.

[30] Moggi, E.: Notions of Computation and Monads. Information and Computation, Vol. 93, 1991, No. 1, pp. 55–92.

[31] Parr, T.: Humans Should Not Have to Grok XML. 2001, Available on: `http://www-106.ibm.com/developerworks/xml/library/x-sbxml.html`.

[32] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, 2007.

[33] Porubän, J.—Forgáč, M.—Sabo, M.—Běhálek, M.: Annotation Based Parser Generator. Computer Science and Information Systems, Vol. 7, 2010, No. 2, pp. 291–307.

[34] Simonyi, C.—Christerson, M.—Clifford, S.: Intentional Software. Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '06), ACM, 2006, pp. 451–464.

[35] Spinellis, D.: Notable Design Patterns for Domain-Specific Languages. Journal of Systems and Software, Vol. 56, 2001, No. 1, pp. 91–99.

[36] Tolvanen, J.-P.—Pohjonen, R.—Kelly, S.: Advanced Tooling for Domain-Specific Modeling: Metaedit. Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM '07), 2007.

[37] van Deursen, A.—Klint, P.—Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices, Vol. 35, 2000, pp. 26–36.

[38] Voelter, M.: From Programming to Modeling – and Back Again. IEEE Software, Vol. 28, 2011, pp. 20–25.

[39] Voelter, M.: Language and IDE Modularization, Extension and Composition with MPS. Pre-Proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), 2011, pp. 395–431.

[40] Voelter, M.—Visser, E.: Language Extension and Composition with Language Workbenches. Proceedings of the ACM SIGPLAN International Conference on Object

Oriented Programming Systems, Languages and Applications (OOPSLA '10), The ACM SIGPLAN Conference on Systems Programming Languages and Applications: Software for Humanity (SPLASH '10), ACM, 2010, pp. 301–304.

[41] Ward, M. P.: Language-Oriented Programming. Software, Concepts and Tools, Vol. 15, 1994, No. 4, pp. 147–161.

Sergej Chodarev is Assistant Professor of informatics in the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He received his M.Sc. degree in computer science in 2009 and his Ph.D. degree in computer science in 2012. The subject of his research includes domain-specific languages, metaprogramming and programming paradigms.

Ján Kollár is Full Professor of Informatics in the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his M.Sc. summa cum laude in 1978 and his Ph.D. degree in computer science in 1991. In 1978–1981 he worked with the Institute of Electrical Machines in Košice. In 1982–1991 he worked with the Institute of Computer Science at P. J. Šafárik University in Košice. Since 1992 he is with the Department of Computers and Informatics at the Technical University of Košice. In 1985 he worked 3 months in the Joint Institute of Nuclear Research in Dubna, USSR. In 1990 he spent 2 months at the Department of Computer Science at Reading University, UK. He was involved in research projects dealing with real-time systems, the design of microprogramming languages, image processing and remote sensing, dataflow systems, implementation of programming languages, and high performance computing. He is the author of process functional programming paradigm. Currently his research area covers formal languages and automata, programming paradigms, implementation of programming languages, functional programming, and adaptive software and language evolution.