Development of Domain-Specific Languages based on Generic Syntax and Functional Composition

Sergej Chodarev*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
sergej.chodarev@tuke.sk

Abstract

Domain-specific languages allow to raise the level of abstraction by using concepts and operations of the domain. Limited applicability of these languages requires lower development costs compared to general-purpose languages. This means that different development techniques need to be used.

This work is a contribution to the field of development of domain-specific languages. It analyses the current state of the field. Special attention is given to the analysis of language composition methods and identifying widespread composition techniques. Composition is analyzed not only on the level of grammars, but also on the level of language concepts.

The work presents proposed approach for language development based on the standardized generic syntax that serves the role of common substrate for developed languages. The structure of language elements is defined based on principles of functional composition extended with metaprogramming capabilities. The behavior of such language is defined using a general-purpose language that provides a connection with the outer environment. Proposed techniques allow modular language development using libraries of language elements.

A prototype of the system for definition and processing of new languages was developed for experimental verification of proposed techniques. This was used for development of several languages.

*Recommended by thesis supervisor: Prof. Ján Kollár. Defended at Faculty of Electrical Engineering and Informatics, Technical University of Košice on August 31, 2012.

Chodarev, S. Development of DSLs based on Generic Syntax and Functional Composition. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 4, No. 3 (2012) 47-53

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

Keywords

concept composition, domain-specific languages, functional composition, generic syntax, language composition, metaprogramming

1. Introduction

The usage of computers is increasing and expanding to a lot of different domains. This leads to the increased complexity of tasks that are solved using computers.

Software development is one of the most complex areas of human activity. Difficulties come from different sources, such as complexity of solved problems, the need to comply with existing interfaces and specifications, and constant changes in requirements [4]. One of the important sources of the complexity is programming on a low level of abstraction. Abstraction allows to hide implementation details and focus on the essence of solved problem. This is a source of advantages provided by high-level languages. They provide abstractions that hide details of a computer (for example memory management) and also allow programmer to define new abstractions in the form of functions, modules, or classes.

Abstractions integrated in general-purpose languages allow to solve a lot of typical programming problems. But for some problems, even language built-in mechanisms for defining new abstraction are not flexible enough to express concepts and operations from problem domain naturally. In such cases it may be suitable to use a metalinguistic abstraction [1]. This means that a new high-level language is created that includes needed abstractions [17, 5].

Some of such abstractions are general-purpose and it is possible to include them in a general-purpose language (existing or a new one). However, most of them are specific to some domain and so it is suitable to develop domain-specific programming languages for these domains.

2. Domain-Specific Languages

Domain-specific language (DSL) is a programming language or an executable specification language that provides suitable notations and abstractions for achieving

expressive power specialized, and usually restricted, to a domain of problems [14, 12, 7, 13]. Specialization to certain domain allows the language to include constructs specially suited for the domain and therefore to provide higher expressiveness.

Domain-specific languages can be divided into three categories according to their implementation [7]:

External DSLs are using a different programming language comparing to the language used to implement the rest of the application. It can be a completely new language or it can use existing concrete syntax.

Internal DSLs (also called embedded DSLs [8]) use the same general-purpose language that is used to implement the application. However it uses it in some special way to make code readable and closer to the domain.

Language Workbenches are integrated environments for development and the use of domain-specific languages. They use projectional editing, so textual or graphical form of the code is just a projection of the internal graph based representation that becomes a primary form of the code [6].

While external DSLs provide flexibility to define suitable language syntax, they have greater development costs. On the other side, internal DSLs are restricted by the host language and programs written using language workbench do not have convenient textual form suitable for further processing using existing tools.

3. Composition of Languages and Concepts

Domain-specific language is specialized to a certain domain, so it is not possible to develop the whole application using only one DSL. For this reason it is necessary to provide a way to integrate DSL with a general-purpose language or other DSLs. This means that we need to solve a problem of language composition.

Composition is used in different contexts and on different levels in programming. We analyzed these levels of composition:

1. *Concept composition*, that allows to express complex structures and operations using a composition of concepts in a language.
2. *Language composition* — definition of a new programming language by combining elements form existing languages.

Language composition allows to modularize language development. A new language can be developed using existing language components — language libraries that provide reusable language elements [15]. However, language composition is a complex task. It requires combination of grammars and other parts of the language implementation [16, 9, 11].

Concept composition is a simpler task, that is used permanently in programming. We propose the following classification of concept composition:

1. *Structural composition* — composition of concepts based on their position in the code.
2. *Functional composition* expressing a data flow between concepts — functions.
3. *Object composition* expressing relations between data structures and abstract data types.
4. *Aspect composition* expressing interleaving of different aspects with main program logic [10].

One of the ways to avoid the complexity of language composition is to move from the level of grammars to the level of concepts. This requires to decrease the role of grammar in language definition. It can be achieved using these approaches:

1. Using the same concrete syntax for composed languages.
2. Using projectional editing, where concrete syntax is just a projection of abstract code representation (this approach is used by language workbenches [16]).

In both cases elements of languages become concepts of a host language or an internal representation. This makes it possible to move from language composition to concept composition and use its proved methods.

4. DSL Development Based on Generic Syntax

Domain-specific languages did not gain widespread usage even despite of the intensive research in this area. One of the reasons is the cost of language development.

The goal of our work is to contribute to the research of DSL development techniques and language integration. Concrete goals was the following:

1. Increase of the productivity of DSL development (language development should be similar to the development of libraries).
2. Possibility to compose languages and language libraries.
3. Design and experimental verification of tools for DSL development.

A new method for domain-specific languages development is proposed to achieve the goals. It is based on the following principles:

1. *Concept composition* instead of language composition. To allow this, all languages are based on the *common generic language* that defines their concrete syntax.
2. The use of *functional composition* to interconnect elements of created languages (supplemented with structural composition).
3. Definition of language semantics using a general-purpose language that provides *integration of the language* into a larger software system.

Concept composition allows to avoid most of the questions of concrete syntax. At the same time, the use of common syntax allows to preserve the role of concrete syntax as a primary representation of programs. This makes possible to use existing tools that expect textual representation of programs.

The generic language is a basis or a substrate, that is used to build new DSLs. It does not have its own semantics. Languages based on it do not define concrete syntax, they define only new concepts that use syntactic shapes provided by the generic language. These concepts are combined in a program using functional composition and structural composition. This allows to express program structure and flow of data.

Language semantics is defined using the implementation general-purpose language. This language is a metalanguage for developed DSLs. This means that programs written in the implementation language can manipulate values and a code of DSL programs. This allows to connect developed language with the rest of a software system. Furthermore, the use of functional composition principles makes definition of language element semantics similar to definition of a function.

The use of concept composition as a basic language construction mechanism simplifies composition of languages. This makes it possible to modularize the language definition. Language can be divided into several parts that can be used alone or as a part of another language.

4.1 Architecture

It is possible to develop a tool for development and usage of DSLs based on the presented approach. The basis of the tool is a *generic language* that defines common syntax for a whole family of domain-specific languages. A set of tools, that cover different aspects of language definition and usage, can be build around of it (see Fig. 1).

The basic tool is a parser of the generic language that analyses a program text and produces its syntax tree. The tree contains generic language elements that correspond to basic syntactic shapes.

These shapes are used to define elements of the concrete DSL. A DSL is actually a subset of the generic language. Its elements are declared in *a language schema*. The schema contains names and properties of language elements. 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 the language schema. This makes possible to detect some errors before the program evaluation.

The next part of the architecture is *an interpreter*. Its task is to evaluate a program after parsing and validating. The interpreter is actually a framework, that allows language author to define subprograms that would be used for evaluation of DSL elements. The interpreter then traverses a DSL program and executes semantic actions corresponding to language elements. The interpreter can process a program in different ways depending on language needs. It can directly execute the actions, that are described in a program, or it can translate it into other language.

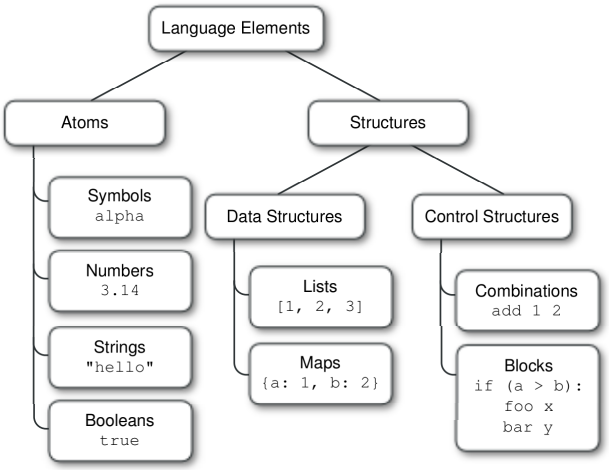

Figure 2: Proposed generic language elements

Semantic actions itself are defined using functions in the implementation language. Every language element has a corresponding *evaluation function* that processes element parameters and calculates of the result of element application. The evaluation function can also communicate with other parts of the evaluation environment.

To simplify language composition, language definition is divided into modules. A module is a language definition unit, that contains several language elements with their declaration and implementation. Every language based on the generic syntax is indeed a set of modules.

5. Generic Syntax

In the proposed method of DSL development, new languages are build based on a generic language. A generic language itself does not have own semantics. It defines only generic syntactic shapes, that can be used to express different elements of other languages. A typical example of a generic language is XML [3].

Parser of a generic language produces a skeleton syntax tree (SST) [2]. In contrast to abstract syntax tree, SST does not reflect concrete language elements, but only generic shapes. Skeleton syntax tree is consequently processed according a language definition.

A new generic language is proposed in the work. The goal of its design is to allow convenient expression of data structures and operations. It should also provide clear and concise concrete syntax. Based on these requirements the proposed language includes several simple elements and shapes for structures as shown on Fig. 2.

Symbols can be used to represent names in programs (both names of language elements and user defined names). Other atoms are used to represent simple data values. The language also provides notation for two types of data structures: lists and maps (associative arrays).

There are also two shapes for expressing control structures and operations:

1. *Combinations* for expressing horizontal structures.
2. *Blocks* for vertical structures.

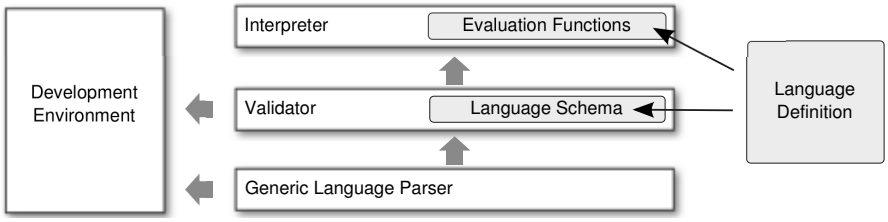

Figure 1: Architecture and supporting tools

```
computer:
    processor {type: x86_64, cores: 2}
    disk {size: 320, speed: 7200, interface: SATA2}
    interfaces [4 * USB2, 2 * IEEE1394]
```

Figure 3: Example of generic syntax

Combination is basically a sequence of language elements. By default it is interpreted as an application of function identified by the first item of the combination.

Concrete syntax for the generic language was chosen to allow clear and concise expression of program code using notations that are familiar to a programmer. Therefore proposed syntax is based on notations used in common programming languages (the most influence comes from Haskell, Python and JavaScript). Indentation is used to express blocks of code and combinations are written simply as a sequence of elements separated by spaces. There is also special syntax for infix operators, which are interpreted as combinations with operator as the first item.

An example of code in proposed generic language is shown on Fig. 3. The example contains a combination. Its first item is the symbol *computer* and second item is a block. The block contains another three combinations that contain examples of data structures and infix operators.

6. Definition of Language Semantics

Proposed evaluation model is based on the evaluation of expressions composed from function applications. Every element of a language has an associated function that ensures its evaluation. If a language element is the first element in a combination, the rest of the combination items are considered as its parameters and evaluation function receives the results of their evaluation.

Processing of a program is divided into several phases:

1. The program is parsed and the skeleton syntax tree is produced.
2. Program is statically checked based on the types of language elements specified by the language schema.
3. Checked code is evaluated using evaluation functions. This can be the final phase. However, in most cases the result of the evaluation is a semantic model or a code in a target language.
4. The last phase is the execution of semantic model or generated code produced in the previous phase. This operation is performed outside of the language development system.

6.1 Values and Evaluation Modes

Arguments can be passed to an evaluation function using one of these modes:

1. by value,
2. symbolically.

The passing mode defines if an argument should be evaluated before the function call or not. If an argument is passed by value, the code that represents it is evaluated and the result is passed as a parameter to the evaluation function.

On the other hand if an argument is passed symbolically, the evaluation function receives a part of a skeleton syntax tree that corresponds to the argument code. Then it can manipulate it freely and even evaluate it or its part when needed.

An evaluation function specifies the passing mode for each parameter. Argument passing mode also affects other parts of the evaluation, for example static program checking.

6.2 Evaluation on Request

Evaluation function can produce different kinds of results, like a computed value, a populated semantic model or generated code in a target language. In case of reusable elements in language libraries it is not known forefront what kind of the result would be needed.

To resolve this problem, evaluation function can produce special object that represents corresponding operation and allows to evaluate it on request. This object also allows to generate code corresponding to the operation in a target language and can itself be stored as a part of a semantic model representing the program. This allows to delay the decision about the way how an operation should be evaluated. This decision is moved to the implementation of a language that uses a language library.

If the evaluation on request is not needed, it is possible to transparently move to simpler evaluation model. Evaluation system checks if a function expects an operation object as a parameter. If it is not expected, the object is evaluated and the function receives the result. And vice versa, values are wrapped in a special object if it is expected. This allows to hide advanced functionality if it is not needed.

6.3 Environment

Names in programs are represented using symbols. Symbols have two roles in a language:

```
@Element
public void define(@Symbolic Symbol symbol,
        Object value) throws EvaluationException {
    if (evaluator.hasSymbol(symbol))
        throw new EvaluationException(
          symbol.getPosition(), "...");
    evaluator.putSymbol(symbol, value);
}
```

Figure 4: Definition of language element *define*

1. They represent language elements.
2. They represent objects defined by a programmer (they can be defined in the same file or externally).

The structure, that can be called symbols table or *environment*, is used to bind names to their meaning. The environment is an associative array with symbols used in a program as keys. Values associated with symbols can be actual values or references to evaluation functions. Environment can be structured and can contain several layers for different scopes and namespaces.

6.4 Implementation of Evaluation Functions

Java was used as an implementation language. This means that evaluation functions are actually methods of some class. Every language module is represented by a class and language elements provided by the module are represented by its methods.

Evaluation functions are marked with the annotation `@Element` to distinguish them from other methods. The name of defined element is equivalent to the name of the method. In a case when the name of the element needs to contain characters that are not allowed in Java identifiers, it is possible to specify it as an argument of the annotation.

Parameters of a language element are mapped to parameters of its evaluation function. Symbolic argument passing is marked in code of using the annotation `@Symbolic`.

An example of evaluation function is presented on Fig. 4. It contains a definition of element *define* that allows to define named constants.

6.5 Static Checking

Static checking is based on the declaration of types and other properties of language elements provided in language schema. Furthermore, it is possible to define check functions in the implementation language, that will provide checking for cases where declarative description is not sufficient.

Element properties are expressed using its type and types of its parameters. The type system is quite simple because it have only a helper function. Types are identified by symbols and are used as tags, defining compatibility of different language elements and possibilities for their composition. Besides of that, it is possible to define type–subtype relations that help to express more complex compatibility rules.

The schema language is used to declare properties of language elements and types. This language is itself based on the generic syntax. Properties of language elements defined in schema include: type of the element, number of its parameters, their types and passing methods. Declaration can also contain a documentation string describing the meaning of the element. This string can be used in an editor to provide interactive help for a programmer.

```
element rgb:
    doc "Create a color from RGB values"
    params [val Number, val Number, val Number]
    type Color

element state:
    doc "Define a state"
    params [sym Symbol, block(val StateProperty)]
    defines_backward 1 State
```

Figure 5: Example of language elements schema

An element can also define new names. This must be declared in the schema, so the new name and its type can be added to the environment and this information can be used during static checking. A name can be defined only forward from the current position in the code, or for the whole block of the code (backward definition). A name can be also defined globally, or locally for the current block.

An example of two declarations of elements is presented in Fig 5. Elements are declared with types of their parameters and other properties. The *state* element also defines a new name corresponding to the symbol from the first parameter and with the *State* type.

7. Experimental Verification

A prototype of DSL development system based on described principles was developed for the verification of proposed solution. The system is implemented in the Java programming language, that is also used for definition of semantics of developed DSLs. The system contains a parser of the generic language, a static checking system and an evaluation framework. Reflection is used for dynamic methods invocation during the evaluation.

Besides of that, the prototype includes language modules for arithmetic operations (addition, subtraction, multiplication and division), relational operations (comparison of numbers), logical operations (and, or, not), and definition of named constants. These modules can be used by several languages and so provide reusable language components.

The prototype was used to develop several experimental languages:

1. State machine definition language.
2. Imperative language for geometric shapes drawing.
3. Data entities definition language.
4. Schema language.

An example program written in the state machine definition language is presented in Fig. 6. It demonstrates the usage of blocks for definition of events, commands and

```
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 6: Program defining a state machine based on the example from [7]

states. Custom operator "`->`" is used to express transitions and operator "`:`" is used to bind a name to an event or command code.

Conducted experiments with the prototype demonstrated that proposed techniques allow to create domain-specific languages that highly decrease costs for development and maintenance of programs in corresponding domain. At the same time, development costs of the language itself are not high and can be returned after the use of the language in a small amount of programs.

8. Conclusions

The work describes several new solutions and approaches for DSL development. Its main contributions include:

- Analysis and systematization of language composition approaches.
- Analysis and classification of concept composition.
- Analysis of the relation between language composition and concept composition.
- Design of the generic host language for DSLs.
- A new method for bonding syntax and semantics of a language.
- Proposal of a static checking system for languages based on the generic syntax.
- Experimental verification of the proposed approach for DSL development.

Proposed techniques can be further enhanced in several directions. For example:

- Enhancement of the possibilities of static and dynamic program checking. Language schema can contain more information about language elements and constraints of their usage. This information can by used for static and dynamic checking and by other tools.
- Development environment for DSLs with interactive help for programmers based on a language definition.
- Design of tools for automation of other aspects of language development.
- Introduction of the possibility to customize language syntax. The generic syntax can be enhanced by domain-specific notations that would be transformed into standard shapes for further processing.

Acknowledgements. This work was supported by VEGA Grant No. 1/0305/11 "Co-evolution of the artifacts written in domain-specific languages driven by language evolution" and by project SK-SI-0003-10 of Slovak–Slovenian Science and Technology Cooperation "Language Patterns in Domain-specific Language Evolution".

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Electrical Engineering and Computer Science. The MIT Press, second edition, 1996.

[2] J. Bachrach and K. Playford. D-expressions: Lisp power, dylan style, 1999. Available at: http://people.csail.mit.edu/jrb/Projects/dexprs.pdf.

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

[4] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, april 1987.

[5] S. Dmitriev. Language oriented programming: The next programming paradigm, November 2004. Available at: http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf.

[6] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. Available at: http://martinfowler.com/articles/languageWorkbench.html.

[7] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[8] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.

[9] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of Onward! 2010*. ACM, 2010.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[11] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. F. Paige, B. Meyer, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer Berlin Heidelberg, 2008.

[12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[13] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[14] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.

[15] M. Völter. From programming to modeling - and back again. *IEEE Software*, 28:20–25, 2011.

[16] M. Völter and E. Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 301–304, New York, NY, USA, 2010. ACM.

[17] M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

Selected Papers by the Author

S. Chodarev. Extensible host language for DSL development. *SCYR 2010: 10th Scientific Conference of Young Researchers of Faculty of Electrical Engineering and Informatics Technical University of Košice*, pages 218–220, Košice, 2010. FEI TU.

J. Kollár, S. Chodarev. Extensible approach to DSL development. *Journal of Information, Control and Management Systems*, 8(3): 207–215, 2010.

S. Chodarev, J. Kollár. XHL: DSL development framework based on fixed syntax. *In CSE 2010: proceedings of International Scientific Conference on Computer Science and Engineering*, pages 141–148, Stará Ľubovňa, 2010. Elfa.

S. Chodarev. Lisp Syntax and its Alternatives. *SCYR 2011: 11th Scientific Conference of Young Researchers of Faculty of Electrical Engineering and Informatics Technical University of Košice*, pages 301–304, Košice, 2011, FEI TU.

S. Chodarev. Generic syntax for domain-specific languages. *Poster 2011: 15th International Student Conference on Electrical Engineering*, pages 1–5, Prague, 2011. ČVUT.

E. Pietriková, Ľ. Wassermann, S. Chodarev. Overview of Parser Generator Tools for Haskell Syntax Processing. *Poster 2011 : 15th International Student Conference on Electrical Engineering*, pages 1–4, Prague, 2011. ČVUT.

S. Chodarev, M. Vagač, J. Kollár. Proposal of generic syntax for domain-specific languages. *Journal of Computer Science and Control Systems*, 4(1): 33–38, 2011.

E. Pietriková, Ľ Wassermann, S. Chodarev, J. Kollár. The effect of abstraction in programming languages. *Journal of Computer Science and Control Systems*, 4(1): 137-142, 2011.

J. Kollár, S. Chodarev, E. Pietriková, Ľ. Wassermann. Identification of patterns through Haskell programs analysis. *FedCSIS: proceedings of the Federated Conference on Computer Science and Information Systems*, pages 891-894, Szczecin, 2011. IEEE, Los Alamitos.

J. Kollár, S. Chodarev, E. Pietriková, Ľ. Wassermann, D. Hrnčič, M. Mernik. Reverse Language Engineering: Program Analysis and Language Inference. *Informatics 2011: proceedings of the Eleventh International Conference on Informatics*, pages 109–114, Rožňava, 2011. TU, Košice.

S. Chodarev. Stratified Framework for DSL Development. *SCYR 2012: 12th Scientific Conference of Young Researchers of Faculty of Electrical Engineering and Informatics Technical University of Košice*, pages 60–63, Košice, 2011. FEI TU.