Method of Combined Dynamic Modification of Programs and Languages

Michal Forgáč[*]
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 040 01 Košice, Slovakia
Michal.Forgac@tuke.sk

Abstract

This work presents innovative method of dynamic modification of program and language for purpose of achievement of new functionality during program run-time. The work further presents adaptive execution environment, which allows reaching of modification of language and program during program execution and presents experimental verification of presented environment. Extensible programming language allows user to define new language constructs. This constructs can include new notations and operations, new or modified control structures or even new elements from different programming paradigms. If there is a need for modification of a system during its runtime, and there is also another requirement for extension of used programming language, in which the running system is programmed (for example requirement for addition of new domain-specific constructions), designed execution environment is one from possible solutions, through which there is possibility to realise this task.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.3.2 [**Programming Languages**]: Language Classifications—*design languages, extendible languages*

Keywords

programming language, software evolution, metaprogramming, modification of program, modification of language, experimental execution environment

[*]Recommended by thesis supervisor: Prof. Ján Kollár
Defended at Faculty of Electrical Engineering and Informatics, Technical University of Košice on September 30, 2009.

Forgáč, M. Method of Combined Dynamic Modification of Programs and Languages. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 2, No. 1 (2010) 6-12

1. Introduction

Modification of complex software systems after their delivery means difficult process. The most often reasons for such modification are on the one hand detected faults, which have to be fixed or on the other hand requirements for new functionality, which systems have to include (e.g. replacement of a system from one computational environment into the new one). Such modifications require additional costs. Even implementation of required changes takes in some cases longer than implementation of the first operational software version. Thus there is significant demand for reaching optimal methods in order to achieve effective modification of software systems.

Besides individual programs are important also programming languages, in which these programs are written. The same importance have also tools, which are related to programming and programming languages. Some projects will not fail due to the faults in programs, but due to the problems with programming languages. In this approach an implementation of language (e.g. compilers, interpreters, and other tools related to programming languages) is metalevel of a program.

If there is a requirement for modification of a system during its execution and there is also another requirement for extension of used programming language, in which this system is implemented (e.g. addition of new domain-specific constructs), it is possible to realise this change through experimental execution environment designed in this work.

2. Overview of the State of the Art

Programming language is possible to define as a system of notations for computation description in the form readable by machine and human [23] or as a means, in which programmer defines problems for computer [14]. Programming languages are not static entities, they are changing and are undergoing evolution change. For example, new elements can be added for more appropriate expression of a problem through a given language in designed program. Elements which are no more used can be removed. Language extension ability allows its adaptation on new application domains.

Under software evolution it is possible to understand all programming activities, which are used for creation of newer version of software system from its older operational version [20]. There are two main reasons for realisation of evolution of software systems [19]: the first reason is requirement or groups of requirements which require new functionality, the second reason represents need for removal of problems, which occurred during application use. In some publications, e.g. in [21] or in [24] is the term software evolution used in contrary to the term software maintenance, because maintenance suggests, that software is getting worse, but this is not problem of evolution. Thus also in this work the term evolution has meaning in the sense of requirements realization for new functionality and not removal of problems, which occured during using of software.

Software evolution can be divided into some categories according to the appropriate criteria. According to the time of performing of changes it is possible to discern two types of software evolution [25], namely static and dynamic software evolution. Static evolution consists of modification of code or modification of execution environment, while the execution of application is stopped, whereas dynamic software evolution consists of modification of code or execution environment of application during its execution and thus without application interrupting (or without interrupting, which could have influence on application functionality). Evolution is possible to divide also according to the assumption of possible changes to anticipated and unanticipated evolution [12, 13, 25]. Anticipated evolution is possible to predict in advance, whereas on unanticipated evolution it is not possible to prepare during design of a software system.

Metaprogramming is about writing programs that represent and manipulate other programs or themselves, i.e. metaprograms are programs about programs [4]. The impact of metaprogramming is obvious in traditional development processes, by sorting existing programs as transformational processes with inputs and outputs. An example of metaprogram is e.g. compiler, interpreter or metainterpreter – it is an interpreter, which inputs are at least two programs. An important aspect is time characteristic, because in the case of metaintepreters individual programs can be loaded and processed at any time. Typical example is an interpreter, which supports dynamic modification of programs. New input programs in the form of source code can change behaviour of interpreted functionality.

Reflection [11, 27, 28] is an entity's integral ability to represent, operate on and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter [16]. A metalevel provides information about selected system and makes the software self-aware. A base level includes the application logic.

Program P can be viewed as a sequence of statements that are aimed to produce some result R. This result R is obtained through the execution of the program P. The execution is done by a platform that interprets the program's sequence of statements. The result R of a computation depends on both a program P and an interpreter I. Interpreter may be any virtual machine or in general even CPU. Different result may be obtained by changing at least one element from the couple ⟨P,I⟩ [1]. This work focuses on program modification in the sense of dynamic software evolution. Under interpreter transformation it is possible to understand augmented transformation of components of execution mechanism [16]. According to this approach is execution synonym of transformation in general, such as a translation of code, type checking, code generation, loading, interpretation, modeling and algebraic specification. Also in this work is under interpreter transformation considered augmented transformation of components of execution mechanism, where is under this transformation understood modification of component for program translation (lexical and syntactic analysis) and modification of component for program interpretation (modification of semantic actions). For highlighting of augmented transformation of execution environment components there will be in the next part of this work instead of interpreter I mentioned the execution environment E, thus instead of the couple ⟨P,I⟩ there will be couple ⟨P,E⟩ (Fig. 1).

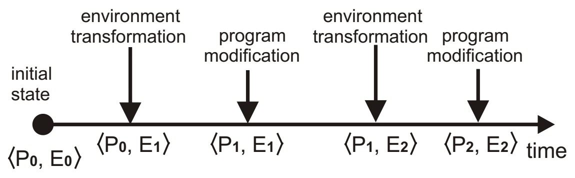


Figure 1: Modification of both elements from the couple ⟨P,E⟩

Our modification approach is based on the transformation of both elements from listed couple. This case allows modification of the second element E from the couple ⟨P,E⟩ and consecutively change the first element P from presented couple, it means from ⟨P_0,E_0⟩ to ⟨P_0,E_1⟩ and then to ⟨P_1,E_1⟩. This modification sequence depends on specific requirements, thus it may be different. Our objective is achieved through designed experimental run-time environment (presented e.g. in [8]). Designed solution supports partially unanticipated evolution.

3. Thesis Objectives

This thesis deals with dynamic mechanisms, which allow modification of programs and programming languages. The main thesis objectives are as follows:

- Proposal of model of combined dynamic modification of program and language.
- Proposal of new method for combined dynamic modification of program and language according to the proposed model.
- Experimental verification of implemented prototype in the form of experimental execution environment through extensible programming language and program.

Implemented prototype is created according to the suggested method, it means, that through verification of prototype will be verified also proposed method.

4. Used Methods and Approaches

This work is connected to the published results in the frame of research of adaptive languages and software systems at the Department of computers and informatics, Technical University in Košice. It is focused on the area of design of new construction and adaptation of languages and software systems, in which it is possible to modify their meaning and behaviour during execution.

This work is based on the results of research, which was performed gradually in the three phases. The first phase was research in the area of aspect-oriented programming with focus on verification of dynamic mechanisms of program composition, important publications of this phase were [5, 7, 29].

From dynamic modification of program moves the research to dynamic modification of languages with verification of mechanisms of metaprogramming and reflection. Important publications in this phase were [15, 17, 18, 19, 16].

The last phase of our research was step from single dynamic modification of language and program toward combined dynamic modification of language and program. Partial results were presented in publications [6, 8, 9, 10].

Before design of model and method for combined dynamic modification of program and language was performed analysis of possibilities of dynamic modification of programs and languages with verification of mechanisms of dynamic aspect-oriented programming, metaprogramming, and reflection. Based on recognized knowledge was designed combined model and on this model was based method for dynamic modification of program and language. Consecutively was designed and implemented adaptive execution environment, which allows modification of a program and a language. This environment was experimentally verified and through its verification was verified also method for combined dynamic modification of programs and languages itself.

5. Model and Method of Combined Dynamic Modification of Programs and Languages

In this work was designed model of combined dynamic modification of program and language (Fig. 2). Proposed model consists of the following inputs:

- initial specification of language and another modified versions of this specification,
- initial program and its modified versions,
- input data inserted interactively or loaded from external source (e.g. from a file).

Outputs of execution environment are:

- result of computation in a given time (displayed e.g. in the execution environment console window)
- another auxiliary output data (for reaching of required computation).

If the program P_{ti} is modified to the new version P_{ti+1} through addition (operator $\cup$) of new code Pn_{ti+1} and removal (operator –) of useless code Po_{ti}, then the new version of program P_{ti+1} will be:

$$P_{ti+1} = (P_{ti} - Po_{ti}) \cup Pn_{ti+1} .$$

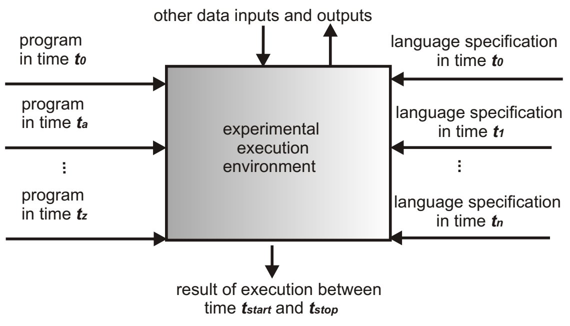

other data inputs and outputs
program in time t_0
program in time t_a
program in time t_z
experimental execution environment
language specification in time t_0
language specification in time t_1
language specification in time t_n
result of execution between time t_{start} and t_{stop}

Figure 2: Model of combined dynamic modification of program and language

If the specification of the language L_{tj} is modified to the new version L_{tj+1} through addition (operator $\cup$) of new elements Ln_{tj+1} and removal (operator –) of not required elements Lo_{tj}, then the new language version will be:

$$L_{tj+1} = (L_{tj} - Lo_{tj}) \cup Ln_{tj+1} .$$

The execution environment E_{tj} will be transformed into the new version E_{tj+1} according to the transformation rules, which will be result from modification of used language specification.

Proposed method is based on the statement, that any execution environment on the metalevel 2, which allows modification of program during its execution is a metasystem of execution environment on the metalevel 1, which allows modification of a language for the program, which is in the environment on the metalevel 1 interpreted (Fig. 3).

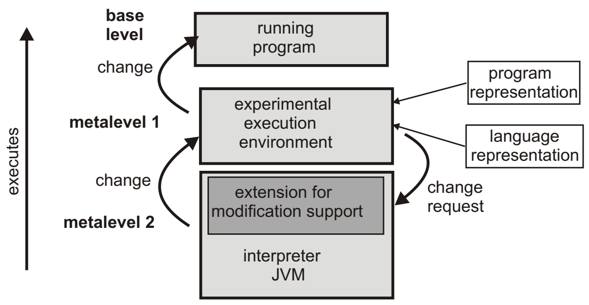


Figure 3: Principle of proposed method

The method for combined modification of programs and languages allows:

- dynamic modification of a program,
- dynamic modification of semantics actions independently from program modification,
- dynamic modification of lexical and syntactic specification of language and semantic actions and consecutive dynamic modification of program,
- dynamic modification of lexical and syntactic specification of language and consecutive modification of program without modification of semantic actions.

Listed possibilities can be combined properly. Objective of this method is reaching of modification of interpreted functionality through properly implemented programmed solution.

6. Experimental Execution Environment

Designed experimental execution environment, based on the proposed method, allows all types of modification presented in this work. This adaptive execution environment is implemented in object-oriented programming language Java with utilization of Javassist [2], which is class library for editing bytecodes in Java. Another utilised mechanism is HotSwap mechanism [3], which allows dynamic reloading of required class file to update the class definition. Lexical and syntactic analysis is performed by program, generated through ANTLR parser generator [26], into which we input individual versions of our language grammar and we will have generated program translator from input program (in the concrete syntax form) to abstract syntax form. This abstract form is represented through textual abstract syntax tree (AST) in the form related to S-expressions [22].

Under language modification it is possible to understand modification of its syntax in the form of modification (or addition) of language grammar and modification of its semantics in the form of modification (or addition) of semantic actions. Under program we mean at first textual representation of program itself in the form of concrete syntax and then textual representation of its abstract syntax tree in the form related to the symbolic expressions known from e.g. LISP programming language.

7. Adaptive Experiment

This adaptive experiment demonstrates possibility and usability of our adaptive approach.

7.1 Initial Language and Program

For simplicity, the grammar presented in this section is not in the ANTLR grammar specification form but it is depicted in more readable Extended Backus-Naur Form (EBNF) with the same meaning for expression of concrete syntax. The abstract syntax form is expressed through expression $^\wedge(parent\{descendant\})$, where sign $^\wedge$ means, that in the parentheses there is tree specification - on the first place there is a lexical element and on the another places there is optionally several subtrees.

The initial demonstrative language has concrete and abstract syntax form as follows:

```
Program ->  {Modify}
        => ^(PROGRAM {Modify})

Modify -> '%' Statement '%' | '@' Statement '@' Statement|
          Statement

        => ^(INSERT Statement)|
           ^(CHANGE Statement Statement)|Statement

Statement -> Declaration|Block|Assignment|WhileStatement|
             IfStatement|PrintStatement|SleepStatement|Nl

          => Declaration|Block|Assignment|WhileStatement|
             IfStatement|PrintStatement|SleepStatement|
             Nothing

Declaration -> "int" Id

            => ^(INTTYPE ^(VARNAME Id))

Assignment -> Id "=" expr

           => ^(ASSIGNMENT ^(VARNAME Id) Expr)

WhileStatement -> "while" "("Condition")"Block
               => ^(WHILE ^(CONDITION Condition) Block)

Condition ->  Atom "<" Atom | Atom "==" Atom|
              Atom ">" Atom

          => ^(SMALLER Atom Atom) | ^(EQUAL Atom Atom)|
             ^(BIGGER Atom Atom)

PrintStatement -> "print" Expr
               => ^(PRINT Expr)

SleepStatement -> "sleep" Int
               => ^(SLEEP ^(INTITEM Int))

Expr -> Atom {("+" | "-") Atom}
     => Atom | ^(ADD {Expr} Atom) |
        ^(SUBTRACT {Expr} Atom)

Atom -> Int|Id
     => ^(INTITEM Int) | ^(VARVAL Id)

Nl -> ("\n")
```

Semantics of proposed language is described through Java programming language, every lexical element has belonging semantic Java class file. Lexical elements are depicted in previous language representation through upper letters.

This language offers several types of language constructs known from e.g. programming language C, such as integer variables, while loops, work with arithmetic expressions and so on. The sleep statement offers possibility for program interruption (time is expressed in seconds) and consecutive continuation. Initial language contains statements for program addition and modification.

The initial computer program may be as follows:

```
int a
a = 0
while(a<1000){
a = a+1
print a
sleep 1
}
```

This program increments value of the variable a, prints its value, sleeps for one second and continues in the next loops while the condition holds.

Internal AST representation of this program in bracket form is as follows:

```
(PROGRAM
 (INTTYPE (VARNAME a))
 (ASSIGNMENT (VARNAME a) (INTITEM 0))
 (WHILE
  (CONDITION
    (SMALLER (VARVAL a) (INTITEM 1000))
  )
  (BLOCK
   (ASSIGNMENT (VARNAME a) (ADD (VARVAL a) (INTITEM 1)))
   (PRINT (VARVAL a))
   (SLEEP (INTITEM 1))
  )
 )
)
```

This AST form is translated into the object form with references between individual objects, for example:

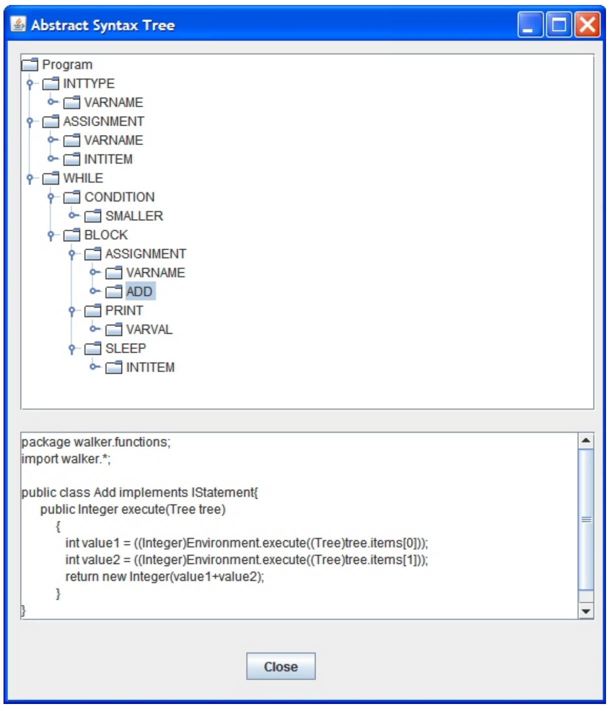
```
(ASSIGNMENT (VARNAME a) (ADD (VARVAL a) (INTITEM 1)))
```

can be processed into the following object form with references on individual objects (the source is depicted in Java programming language and names of objects are in fact different):

```
Leaf l1 = new Leaf("a");
Tree t1 = new Tree("VARVAL",l1);
Leaf l2 = new Leaf("1");
Tree t2 = new Tree("INTITEM",l2);
Tree t12 = new Tree("ADD",t1,t2);
Leaf l3 = new Leaf("a");
Tree t3 = new Tree("VARNAME",l3);
Tree t4 = new Tree("ASSIGNMENT",t3,t12);
```

Execution environment processes this object form according to the token in every object representation. Every token defines operation, which must be performed by execution environment on given objects.

Execution environment offers possibility for observation of program representation in AST form and also belonging semantic actions. This functionality is depicted in the Fig. 4 where it is possible to see semantic action for ADD statement. Every operation implements common interface which dictates presence of execute method.



```
package walker.functions;
import walker.*;

public class Add implements IStatement{
    public Integer execute(Tree tree)
        {
          int value1 = ((Integer)Environment.execute((Tree)tree.items[0]));
          int value2 = ((Integer)Environment.execute((Tree)tree.items[1]));
          return new Integer(value1+value2);
        }
}
```

Figure 4: Abstract syntax tree window

7.2 Modification of Language and Program

Proposed environment (or rather language specification) does not support multiplication by now and we need this functionality, because we want to add multiplication into modified program.

The rule *Expr* will be changed into the new form and the rule *MulExpr* will be added (modification of language specification must be backward compatible):

```
Expr -> MulExpr {("+" | "-") MulExpr}
     => MulExpr | ^(ADD {Expr} MulExpr) |
        ^(SUBTRACT {Expr} MulExpr)

MulExpr -> Atom {("*") Atom}
        => Atom | ^(MULTIPLY {MulExpr} Atom)
```

Execution environment supports run-time modification. The handle for multiplication must be inserted before program modification through new Java class, which is loaded during execution-environment run-time. From our approach is evident, that individual language elements are represented modularly.

We have added semantic action and now we can modify program during run-time into the new form. Special pair operator *@ old_statement @ new_statement* for modification of the program will be used (under new statement we could see also block of statements, because block is statement too). The source code may be as follows:

```
int a
a=0
while(a<1000){
@a=a+1@a=a*2
print a
sleep 1
}
```

This program multiplies value of the variable *a* by the number two in every transition of the while loop. Internal AST representation for modified statement will be changed into the new form:

```
(ASSIGNMENT (VARNAME a)(MULTIPLY (VARVAL a)(INTITEM 2)))
```

Object representation of this statement will be changed into the new one:

```
Tree t5 = new Tree("VARVAL",a);
Tree t6 = new Tree("INTITEM",2);
Tree t56 = new Tree("MULTIPLY",t5,t6);
//t3 without change
//t4 exists yet, but must be modified
t4.changeTree("ASSIGNMENT",t3,t56);
```

If we want to insert new statement into the running program, we can use pair operator *% new_statement %*. For example if we want to insert another sleep statement into the while loop, we insert statement

```
%sleep 1%
```

which will be changed into its internal AST form, then into the new object form, and object form of block code inside the while loop will be modified (object of new statement will be inserted into object, which represents while block).

Run-time modification will be ensured, because only required objects will be modified and references between other objects will stay preserved. Program is modified during its run-time and consists of new language element with relevant semantics.

8. Conclusions

This thesis presented new approach for modification of interpreted functionality through combined language and program modification. This solution can be further extended. It is possible to extend the set of languages that can be added during program execution. This adaptive execution environment is open for various languages (not only for general-purpose programming languages, but also for domain-specific languages), because it is possible to use various language specifications. This approach could be utilised in some specific application domains. For example we could have a control program for an industrial machine and there would be a need for optimisation of its manufacture process during its run-time without device stopping and such optimisation would require new language elements with appropriate semantics.

An interesting idea is to generate an initial programming language according to a specific application domain and then extend such language according to new requirements. Another interesting area of research could be increasing the level of automation in such way that the execution environment could decide according to some rules about loading new language specifications and then modify required program during its execution.

Acknowledgements. This work was supported by:

VEGA Grant No. 1/0015/10 Principles and Methods of Semantic Enrichment and Adaptation of Knowledge-based Languages for Automatic Software Development and

bilateral project Software Evolution by Language Adaptation, Grant No. SK-SI-0001-08 between Slovakia and Slovenia

References

[1] N. Bouraqadi and T. Ledoux. How to weave? In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, June 2001.

[2] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), volume 2830 of Springer Lecture Notes in Computer Science*, pages 364–376. Springer-Verlag, 2003.

[3] S. Chiba, Y. Sato, and M. Tatsubori. Using hotswap for implementing dynamic aop systems. In *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[5] M. Forgáč. Utilization of dynamic weaving in software evolution. In *Proceedings from 7th PhD Student Conference and Scientific and Technical Competition of Students of Faculty of Electrical Engineering and Informatics Technical University of Košice*, pages 105–106, 2007.

[6] M. Forgáč. Adaptive proposal of language modification. In *Proceedings from 9th Scientific Conference of Young Researchers - SCYR 2009*, pages 134–136, 2009.

[7] M. Forgáč and J. Kollár. Static and dynamic approaches to weaving. In *2007 Proceedings, 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, pages 201–210, 2007.

[8] M. Forgáč and J. Kollár. Adaptive approach for language modification. *Journal of Computer Science and Control Systems*, 2(1):9–12, 2009.

[9] M. Forgáč, J. Kollár, and J. Porubän. Reflection as a tool for adaptability of software systems. In *SAMI 2008 Proceedings, 6th International Symposium on Applied Machine Intelligence and Informatics*, pages 179–182, 2008.

[10] M. Forgáč and M. Vagač. Transformation of functionality with utilization of metaprogramming and reflection. In *8th Scientific Conference of Young Researchers of Faculty of Electrical Engineering and Informatics Technical University of Kočice*, pages 95–97, 2008.

[11] R. Hirschfeld and R. Lämmel. Reflective designs. *IEE Journal on Software, Special Issue on Reusable Software Libraries*, 152(1):38–51, 2005.

[12] J. Keeney. *Completely Unanticipated Dynamic Adaptation of Software*. PhD thesis, University of Dublin, 2005.

[13] G. E. Kniesel and R. E. Filman. Unanticipated software evolution: Issue overviews. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):307–308, 2005.

[14] J. Kollár. *Design and Implementation of Programming Languages (in Slovak)*. Elfa, 2002.

[15] J. Kollár, M. Forgáč, and J. Porubän. Adaptiveness of software systems using reflection. *Acta Electrotechnica et Informatica*, 3(7):53–57, 2007.

[16] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Adaptive compiler infrastructure. In *Komunikaèné a informaèné technológie*, pages 4–5, 2007.

[17] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Adaptive language approach to software systems evolution. *Proceedings of the International Multiconference on Computer Science and Information Technology*, 2(2):1081–1091, 2007.

[18] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Functional approach to the adaptation of languages instead of software systems. *COMSiS - Computer Science and Information Systems*, 4(2):115–129, 2007.

[19] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Software evolution from a meta-level compiler perspective. *SCIENCE & MILITARY*, 2(2):29–32, 2007.

[20] M. Lehman and J. Ramil. Towards a theory of software evolution - and its practical impact (working paper). In *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov*, pages 2–11. Press, 2000.

[21] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[22] A. M. Leitao. From lisp s-expressions to java source code. *COMSiS - Computer Science and Information Systems*, 5(2):19–38, 2008.

[23] K. C. Louden. *Programming Languages, Principles and Practice*. Course Technology, 2nd edition, 2002.

[24] T. Mens. Introduction and roadmap: History and challenges of software evolution. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 1–11. Springer, 2008.

[25] M. Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, University of Geneva, Geneva, Switzerland, 2004.

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

[27] F. Rideau. Metaprogramming and free availability of sources, two challenges for computing today, 1999. CNET DTL/ASR.

[28] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, 1996.

[29] M. Tóth, M. Forgáč, and J. Bandáková. Weaving as a mechanism for software evolution. In *Proc. 8th International Conference on Engineering of Modern Electric Systems*, pages 117–122, 2007.

Selected Papers by the Author

J. Kollár, J. Porubän, P. Václavík, J. Bandáková, M. Forgáč. Functional Approach to the Adaptation of Languages instead of Software Systems. *COMSiS - Computer Science and Information Systems*, 4, 2, pages 117–131, 1820-0214, 2007.

J. Kollár, M. Forgáč, J. Porubän. Adaptiveness of Software Systems Using Reflection. *Acta Electrotechnica et Informatica*, 7, 3, pages 53–57, 1335-8243, 2007.

J. Kollár, J. Porubän, P. Václavík, J. Bandáková, M. Forgáč. Software Evolution From A Meta-Level Compiler Perspective. *SCIENCE & MILITARY*, 2, 2, pages 29–32, 1336-8885, 2007.

J. Kollár, J. Porubän, P. Václavík, J. Bandáková, M. Forgáč. How to Adapt Programming Languages instead of Software Systems. *Computer Science and Technology Research survey*, Košice, Elfa, 2, pages 69–79, ISBN 978-80-8086-071-4, 2007.

J. Kollár, J. Porubän, P. Václavík, M. Forgáč, M. Sabo, L'. Wassermann, F. Mrázik, M. Vagač. Software Evolution based on Software Langauge Engineering. *Computer Science and Technology Research survey*, Košice, Elfa, 3, pages 25–30, ISBN 978-80-8086-100-1, 2008.

J. Kollár, J. Porubän, P. Václavík, M. Tóth, J. Bandáková, M. Forgáč. Multi-paradigm Approaches to Systems Evolution. *Computer Science and Technology Research survey*, Košice, Elfa, 1, pages 6–9, ISBN 978-80-8086-046-2, 2007.

M. Tóth, M. Forgáč, J. Bandáková. Weaving as a Mechanism for Software Evolution. *Proc. 8th International Conference on Engineering of Modern Electric Systems*, Felix Spa-Oradea, May 24-26, Oradea, Romania, University of Oradea, pages 117–122, ISSN 1223-2106, 2007.

J. Kollár, J. Porubän, P. Václavík, J. Bandáková, M. Forgáč. Adaptive Language Approach to Software Systems Evolution. *Proceedings from International Multiconference on Computer Science and Information Technology*, 1st Workshop on Advances in Programming Languages (WAPL'07), Wisla, Poland, October 15-17, Polish Information Processing Society, 2, pages 1081–1091, ISSN 1896-7094, 2007.

M. Forgáč, J. Kollár. Adaptive Approach for Language Modification. *Journal of Computer Science and Control Systems*, 2, 1, pages 9–12, ISSN 1844-6043, 2009.

J. Kollár., M. Forgáč, J. Porubän. Adaptiveness of Software Systems Using Reflection. *Acta Electrotechnica et Informatica*, 7, 3, pages 53–57, ISSN 1335-8243, 2007.

J. Kollár., M. Forgáč. Reflective monadic adaptation. *Acta Electrotechnica et Informatica*, 9, 3, pages 43–47, ISSN 1335-8243, 2009.

M. Forgáč, J. Kollár. Static and Dynamic Approaches to Weaving *SAMI 2007 Proceedings*, 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics, Poprad - AquaCity, Slovakia, January 25-26, pages 201–210, ISBN 978-963-7154-56-0, 2007.

M. Forgáč, J. Kollár, J. Porubän. Reflection as a Tool for Adaptability of Software Systems *SAMI 2008 Proceedings*, 6th International Symposium on Applied Machine Intelligence and Informatics, Herl'any, Slovakia, January 25-26, pages 179–182, ISBN 978-1-4244-2106-0, 2008.

M. Forgáč. Utilization of Dynamic Weaving in Software Evolution *Proceedings from 7th PhD Student Conference and Scientific and Technical Competition of Students of Faculty of Electrical Engineering and Informatics Technical University of KoŽice*, Košice, May 23, 2007, Slovakia, Elfa, pages 105–106, ISBN 978-963-7154-56-0, 2007.

M. Forgáč, M. Vagač. Transformation of Functionality with Utilization of Metaprogramming and Reflection *Proceedings from 8th Scientific Conference of Young Researchers of Faculty of Electrical Engineering and Informatics Technical University of Košice*, Košice, May 28, 2008, Slovakia, Elfa, pages 95–97, ISBN 978-80-553-0036-8, 2008.

M. Forgáč. Adaptive Proposal of Language Modification *Proceedings from 9th Scientific Conference of Young Researchers - SCYR 2009*, Košice, May 13, 2009, Slovakia, Elfa, pages 134–136, ISBN 978-80-553-0178-5, 2009.

J. Kollár, M. Forgáč, J. Bandáková. Monadic Adaptation by Reflection *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering*, The Hight Tatras - Stará Lesná, September 24-26, 2008, Slovakia, Elfa, pages 118–125, ISBN 978-80-8086-092-9, 2008.