Computing and Informatics, Vol. 29, 2010, 1103–1116

COMBINED APPROACH TO PROGRAM AND LANGUAGE EVOLUTION

Ján Kollár, Michal Forgáč

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
e-mail: Jan.Kollar@tuke.sk, Michal.Forgac@tuke.sk

Manuscript received 2 June 2009; revised 13 May 2010

Communicated by Prabhat Kumar Mahanti

Abstract. Program can be viewed as a sequence of statements that are aimed to produce some result. The execution is done by a platform that interprets the program sequence of statements. The new result of a computation can be achieved by modification of a program, a language interpreter, or both. Software evolution as long-term process can be supported by adaptive language and by environment, which offers reflective possibilities. This paper presents our adaptive approach to both program and language modification in order to support dynamic evolution. Effective software evolution needs to be supported by appropriate execution environment. We have proposed such experimental execution environment, which allows both run-time program and language modification. As we hope, mutating a programming language to a higher abstraction may decrease structural complexity of programs in the future.

Keywords: Software evolution, program modification, language modification, software language engineering, adaptive experimental environment

Mathematics Subject Classification 2000: 68N15, 68N20

1 INTRODUCTION

Modification of complex software systems after their delivery means a 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.

Modification can be understood as software maintenance or software evolution. These terms are often used as synonyms but we incline rather to the claim (according to e.g. [11]) that these terms do not express similar meaning because the purpose of maintenance is mostly in removal of some faults and the purpose of evolution is adaptation of a system according to the new external or internal requirements. According to [12], software evolution is defined as a collection of all programming activities intended to generate a new version from an older and operational version. Two main types of software evolution can be named as static and dynamic evolution [15]. Static evolution consists in evolving the code of an application while it is stopped whereas dynamic evolution consists in evolving an application during its execution, without stopping it. The advantage of the former is that there is no need for state transfer or active thread to solve, whereas main disadvantage is that application is stopped and thus its services are stopped too (there is temporary unavailability). The advantage of the latter is no unavailability but there are some uncertain technical issues. Furthermore, evolution can be anticipated or unanticipated [15]. Anticipated evolution is an evolution that has been foreseen by the programmer while unanticipated evolution consists in evolution that has not been foreseen.

Software evolution also includes language evolution as an actual issue. Some projects may fail not because of bugs in programs, but because of the lack of recognition of language issues. Thus according to [5], software is composed of a program and a language. In this approach, language implementation (e.g. interpreters, compilers, and other language-dependent tools) is regarded as a metalevel of a program.

This paper is composed as follows: Section 2 presents basic information about software language engineering. Section 3 deals with modification of interpreted functionality, namely modification of program, modification of interpreter and modification of both elements in order to produce different result. Section 4 presents our adaptive experimental environment as innovative approach in the field of language engineering. Section 5 contains discussion about our approach and related works. Finally, Section 6 concludes the paper.

2 LANGUAGE ENGINEERING

Every natural language offers means for spoken and written communication. If these languages could be recognized through a computer without any ambiguity, there would be no need for artificial programming languages; but computers need exact specification of statements with unambiguous meaning in order to execute these statements properly.

In order to have a usable computer language, there is a need for exact and complete description. Description of a language can be divided into its structure and its meaning. According to [9], description of a programming language should contain the following elements:

- an abstract syntax description,
- at least one concrete syntax description,
- a mapping from concrete to abstract syntax description,
- a description of the semantics.

Programming language offers notation, which facilitates program construction for given language. This notation is known as concrete syntax. Computer program commonly written by a programmer is in the form of concrete syntax. Original meaning of the term abstract syntax comes from the domain of natural languages [2], where abstract syntax means hidden, basic, unified structure of several sentences (two various sentences represent the same). Abstract syntax is typically used for internal representation of a program.

Description of semantics can be divided into formal or informal description. There are several ways for description of semantics through the formal way, e.g. denotational, operational, action or translational semantics.

Appropriate design of computer language assures expressiveness and understandability for the programmer from one side, and accuracy and simplicity for the computer from other side. In the context of language engineering, there are three types of users [10]. Language user just uses a language with the aim to create software for a need of a customer. Language engineer creates software (modeling or programming) languages in order to create language, which allows effective building software by language user. Meta-language engineer creates tools for language engineers. This language user develops language for description of language specifications and provides support for its utilization.

3 MODIFICATION OF INTERPRETED FUNCTIONALITY

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 interpreter I.

Interpreter may be any virtual machine or in general even CPU. Different result may be obtained by changing at least one of the elements of the couple $\langle P, I \rangle$ [1].

3.1 Modification of Program

This approach is based on modification of the program P_0 from the couple $\langle P_0, I_0 \rangle$. The interpreter I_0 is kept unchanged. A new program P_1 is built from both the application aspects and the program P_0. Just aspect-oriented programming [7], [8] is based on this principle and transformation of the program P_0 into the program P_1 is performed using transformation process named weaving. Originally this approach is named as weaving through program transformation [1].

Modification of semantics through program transformation may be accomplished in various ways. Modification can be realized in various times of program processing in invasive or non-invasive manner.

This modification can be static or dynamic depending on the time when a composition tool is used. If composition is performed before compiling or it is built during the compilation process, this composition is static; if it is performed after compiling, it is dynamic composition. In static composition, compile-time approaches are used. It can help in suggestion of software projects in which it is not important that system must be halted and then changes must be run again. Dynamic composition can be performed during run-time. This is very helpful for applications and services, which require non-stop running because they cannot be shut down due to safety or financial reasons. There is also another case, namely load-time program composition, which can belong to the static or dynamic approaches group depending on whether this composition is realized only once during initial program loading into execution environment (static approach) or this loading is realized also during program execution (in case of program addition).

Modification can be classified also as invasive or non-invasive adaptation depending on whether it performs transformation of base code (in various forms) in order to achieve an additional functionality. Typical case of this invasive modification is modification of a source or binary code. If there is no preservation of information about origin of individual parts (e.g. in the form of metainformation), then after composition, it will not be possible to differentiate the origin of individual parts of code. Typical example of non-invasive composition is run-time weaving in aspect-oriented programming, which works on the principle of interception of certain event in the base functionality and consecutive execution of additional functionality. This type of execution is performed with support of run-time environment. Invasive composition can be performed by addition of program code in well-defined form with new functionality obtained for example by repeated compilation of methods.

3.2 Modification of Interpreter

This approach is based on modification of the interpreter I_0 from the couple $\langle P_0, I_0 \rangle$. The program P_0 is kept unchanged. Originally this approach is named weaving

through interpreter transformation [1] since aspect weaving consists in applying these transformation rules to the initial interpreter I_0 and so a new (or modified) interpreter I_1 is created. The interpreter I_1 is built from both the application aspects and the interpreter I_0.

Our idea of interpreter transformation is extended by transformation of various components of the execution mechanism. Thus, execution is a synonym for transformation in general, such as translation, type checking, code generation, loading, interpretation, modeling, algebraic specification, and even for informal but constructive thinking about algorithmic problems.

Foundations of our adaptive language implementation were started with simple LL(1) language and its adaptive interpreter consisting of lexical analyzer, adaptive translator and evaluator. Depending on the result of interpretation, the LL(1) language was changed, and the next interpretation followed different semantics, i.e. potentially different result of the same source expression.

3.3 Modification of Both Elements

This approach is based on the case in which is attempted to change the second element I from the couple $\langle P, I\rangle$ and consecutively to change the first element P from the presented couple, i.e. from $\langle P_0, I_0\rangle$ to $\langle P_0, I_1\rangle$ and then to $\langle P_1, I_1\rangle$ (Figure 1). Our objective is to achieve this modification during run-time with utilization of the appropriate experimental run-time environment. The designed solution should support partially unanticipated evolution.

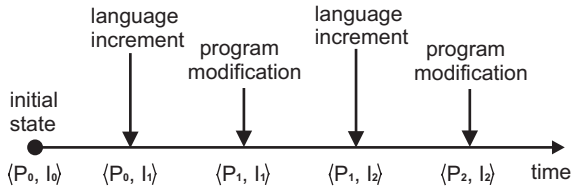

Fig. 1. Modification of both elements from the couple $\langle P, I\rangle$

4 ADAPTIVE EXECUTION ENVIRONMENT

An extensible language allows users to define new language features. These features may include new notation or operations, new or modified control structures, or even elements from different programming paradigms [23].

The value of dynamic evolution of software systems is important in systems, which must provide non-stop availability. Provided that there is a requirement for modification of this system during its execution, but there is also another requirement for modification of its programming language (for example addition of new

language elements for given domain), the proposed adaptive execution environment could represent solution for this task.

4.1 Architecture

Our adaptive execution environment, shown in Figure 2, offers run-time modification of:

- program,
- language for this program.

Program modification means that running program will be modified in the sense of addition, removal or change of selected program statements. Under language modification it is possible to consider 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).

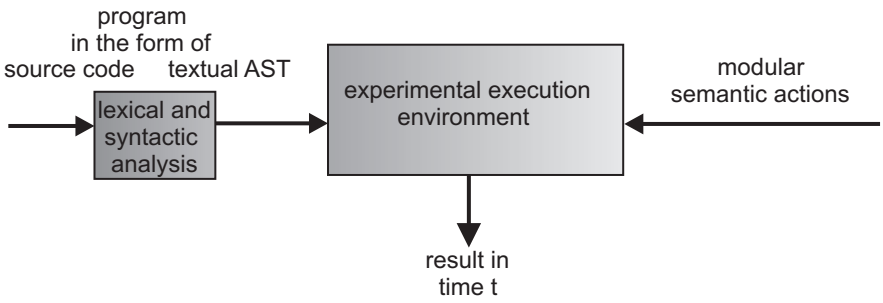

Fig. 2. Adaptive execution environment

Adaptive approach is based on several technologies. The adaptive execution environment is implemented in object-oriented Java programming language with utilization of Javassist [3], which is class library for editing bytecodes in Java. Another utilised mechanism is HotSwap [4], 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 [16], into which we input individual versions of our language grammar and we will have generated program translator from the 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 [13].

Every version of AST (initial or modified during program execution) is inserted into adaptive execution environment and then AST is transformed into internal object form. The basic idea of our run-time adaptation of running program is the fact that only modified subtrees will be changed and references between unchanged

objects remain untouched. Moreover, program statements are evaluated as atomic elements.

Every tree node in AST has assigned semantic action in the form of Java class. Execution environment allows modification of existing semantic actions with utilization of reloading mechanism. Another case is the situation when grammar is modified and after program modification AST has of a new tree node type. Before modified program application it is required to add the relevant semantic action.

4.2 Adaptive Experiment

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

4.2.1 Initial Language and Program

For simplicity, the grammar presented in this section is not in the ANTLR grammar specification form but in more readable Extended Backus-Naur Form (EBNF) with the same meaning. The initial demonstrative language has syntax in EBNF as follows:

```
Program ->  {Statement}
Statement -> Declaration|Block|Assignment
 |WhileStatement|PrintStatement|SleepStatement
Declaration -> int ID
Block -> "{" {Statement} "}"
Assignment -> ID "=" expr
WhileStatement -> "while" "("Condition")"Block
Condition -> "(" atom "<" atom ")"
            |"(" atom "==" atom ")"
            |"(" atom ">" atom ")"
PrintStatement -> "print" expr
SleepStatement -> "sleep" INT
expr -> atom {("+" | "-") atom}
atom -> INT|ID
```

Formulation of INT and ID through regular expressions is as follows:

```
INT :   '0'..'9'+
ID  :   ('a'..'z'|'A'..'Z')+
```

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

The initial computer program is as follows:

```
int a
a=0
while(a<100){
a=a+1
print a
sleep 1
}
```

This program increments the 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
(INT (ID a))
(ASSIGNMENT(ID a)(NUM 0))
 (WHILE
  (CONDITION (SMALLER (ID a) (NUM 100)))
  (BLOCK
   (ASSIGNMENT (ID a)(PLUS (ID a) (NUM 1)))
   (PRINT (ID a))
   (SLEEP (NUM 1))
  )
 )
)
```

This AST form is translated into the object form with references between individual objects.

For example,

```
(ASSIGNMENT (ID a)(PLUS (ID a) (NUM 1)))
```

can be processed into the following object form (of course, names of objects are diferent):

```
Tree t1 = new Tree("ID",a);
Tree t2 = new Tree("NUM",1);
Tree t12 = new Tree("ADDITION",t1,t2);
Tree t3 = new Tree("ID",a);
Tree t4 = new Tree("ASSIGNMENT",t3,t12);
```

The 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 the possibility for observation of program representation in AST form and also of the relevant semantic actions. This functionality shown in Figure 3, where semantic action for ASSIGNMENT statement can be seen. Every operation implements common interface which dictates presence of the execute method.

Fig. 3. AST and semantics actions observation window

4.2.2 Modification of Language and Program

Our environment 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 to the new form:

```
expr -> mulExpr{("+" | "-") mulExpr}
```

The rule `mulExpr` will be added into the EBNF grammar.

```
mulExpr -> atom {"*" 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. It is evident from our approach that individual language elements are represented in modular manner.

We have added semantic action and now we can modify program during run-time into the new form. Special operator '@' for modification of the program will be used.

```
int a
a=0
while(a<100){
a=a+1 @{a=a*2}
print a
sleep 1
}
```

This program multiplies the 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 (ID a)(MULTIPLICATION (ID a)(NUM 2)))
```

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

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

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

5 DISCUSSION AND RELATED WORKS

In general, our experiment offers three possible cases of utilization:

The first case allows program modification without modification of actual language. It is possible to add, remove or replace one or more statements from an existing program during its execution. For example a variable assignment in a while loop can be altered to the new value (e.g. a=a+1 to a=a+2).

The second case is based on a change of the execution environment, which interprets the same program. In this case, the semantics of this program can be changed (e.g. semantics of operator "+" can be altered from addition to subtraction).

The third case offers execution environment change in the sense of new language element addition and consecutive change of used program with utilization of such new language element. For example addition of a print statement into the execution

environment can be accomplished and then this statement can be inserted into a while loop.

Even though our approach offers run-time modification, it can be utilized in the process of language prototyping, development and testing in the sense of static language evolution.

There are several approaches to support program modification in various times, supporting static program modification (typical representative is AspectJ [8]) or dynamic program modification (e.g. PROSE [14] or Jasco [22]). These approaches are mostly connected to the aspect oriented programming paradigm. Our solution also offers dynamic program modification, because it allows run-time modification of every statement in aspect-oriented programming way, i.e. before, after or around particular statement.

Approaches which deal with language modification are connected mainly to the static modification of languages through several possibilities of language engineering tools (e.g. language workbenches [6]). Adaptive language modification is based on the language modification according to several internal conditions and relevant actions were located on the metalevel through predefined rules. This solution was implemented in Haskell functional programming language. Related research was also oriented to another programming platforms, e.g. Java [17, 18, 19]. The approach presented in this paper, however, represents dynamic language modification based on external effects driven by programmers.

From related works, there are for example Lisp macros [13] and its programming mechanisms, which allow data execution as program code (with addition of new functions which can be basically new language elements), thus this solution is related to our approach.

6 CONCLUSIONS

The presented experiment was explained using a language, which is related to the general purpose C programming language. However, our approach can also be utilized in some specific application domains. Domain specific languages are usually simple languages dedicated to some specific domains [20, 21]. Our adaptive configurable environment is open for various languages. If there would be requirement for some domain specific language modification or modification of its program, it would be possible to successfully utilize our environment. For example, we could have control program for an industrial machine and there would be a need for optimization of the manufacturing process during run-time without device stopping. Such optimization would require new language elements with appropriate semantics. Another example may be a computer program, which processes various data in one way and there is a requirement for modification of this data manipulation without program interrupting.

However, there are various challenges for improvement of our experimental solution, whose implementation is now relatively simple. For example, change man-

agement is one of them. Without change management, risk for modification establishment may override the consequences of program stopping and restarting. Thus at least change history could be useful.

Our innovative approach follows our previous language experiment with language adaptation according to the internal conditions. This work is our another contribution in the field of language modification. Although the presented solution is in early phase of research, it would be useful in the future. Practical utilization of our proposal in extended form can be in evolution of complex software system through incremental design of its programming language with the possibility of its modification (mainly run-time modification is of special importance).

Acknowledgement

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 by bilateral project "Software Evolution by Language Adaptation" between Slovakia (Grant No. SK-SI-0001-08) and Slovenia (Grant No. BI-SK/08-09-002).

REFERENCES

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

[2] Črepinšek, M.—Mernik, M.: Inferring Context-Free Grammars for Domain-Specific Languages. Conf. on Language Descriptions, Tools and Applications, LDTA 2005, April 3, Edinburgh, Scotland, UK, pp. 64–81.

[3] Chiba, S.—Nishizawa, M.: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp. 364–376, Springer-Verlag, 2003.

[4] Chiba, S.—Sato, Y.—Tatsubori, M.: Using HotSwap for Implementing Dynamic AOP Systems. ECOOP '03 Workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), July 21st, 2003.

[5] Favre, J. M.: Languages Evolve Too – Changing the Software Time Scale. Eighth International Workshop on Principles of Software Evolution (IWPSE05), 2005, pp. 33–44.

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

[7] Kiczales, G.—Lamping, J.—Mendekar, A.—Maeda, C.—Videira Lopes, C.—Loingtier, J. M.—Irwin, J.: Aspect-Oriented Programming. 11th European Conf. Object-Oriented Programming, volume 1241 of LNCS, Springer Verlag, 1997, pp. 220–242.

[8] Kicszales, G.—Hilsdale, E.—Hugunin, J.—Kersten, M.—Palm, J.—Griswold. W.: An Overview of AspectJ. In Proceedings of ECOOP '01, European Conference on Object/Oriented Programming, Springer-Verlag (LNCS 2072), 2001, pp. 327–355.

[9] Kleppe, A.: A Language Description is More than a Metamodel. In: the 4th International Workshop on (Software) Language Engineering, 2007.

[10] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. 1 edition. Addison-Wesley, 2008, 240 pp.

[11] Lehman, M.: Laws of Software Evolution Revisited. EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp. 108–124.

[12] Lehman, M.—Ramil, J.: Towards a Theory of Software Evolution – and Its Practical Impact. Proceedings of the International Symposium on Principles of Software Evolution, Nov. 2000, Japan, pp. 2–11.

[13] Leitão, A. M.: From Lisp S-Expressions to Java Source Code. COMSiS – Computer Science and Information Systems, Vol. 5, 2008, No. 2, pp. 19–38.

[14] Nicoara, A.—Alonso, G.: Dynamic AOP with PROSE. In: Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005), Porto, Portugal, June 2005.

[15] Oriol, M.: An Approach to the Dynamic Evolution of Software Systems. Ph. D. Thesis, University of Geneva, Geneva, Switzerland, April 2004, 191 pp.

[16] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, May 2007, 376 pp.

[17] Porubän, J.—Václavík, P.: Extensible Language Independent Source Code Refactoring, AEI 2008: International Conference on Applied Electrical Engineering and Informatics, Greece, Athens, September 8–11, Košice, FEI TU, 2008, pp. 58–63.

[18] Porubän, J.—Sabo, M.: Jessine: Integrating Rules in Enterprise Software Applications. Journal of Information, Control and Management Systems, Vol. 7, 2009, No. 1, pp. 81–88.

[19] Sabo, M.—Porubän, J.: Preserving Design Patterns using Source Code Annotations. Journal of Computer Science and Control Systems, Vol. 2, 2009, No. 1, pp. 53–56.

[20] Václavík, P.—Porubän, J.: Template-Based Content Management System, AEI 2008 International Conference on Applied Electrical Engineering and Informatics, Athens, Greece, September 8–11, 2008, pp. 153–157.

[21] Václavík, P.: Application Domain Name-Based Analysis. Journal of Computer Science and Control Systems, Vol. 2, 2009, No. 2, pp. 66–69.

[22] Vanderperren, W.—Suvee, D.—Cibran, M.—Verheecke, B.—Jonckers, V.: Adaptive Programming in JAsCo. In Proceedings of AOSD 2005, ACM Press, Chicago, USA.

[23] Zingaro, D.: Modern Extensible Languages. SQRL Report 47, McMaster University, Hamilton, Ontario, Canada, October 2007.

Ján Kollár is Full Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his M. Sc. summa cum laude in 1978 and his Ph. D. in Computer Science in 1991. In 1978–1981 he was with the Institute of Electrical Machines in Košice. In 1982–1991 he was with Institute of Computer Science at the P. J. Šafárik University in Košice. Since 1992 he is with the Department of Computer and Informatics at the Technical University of Košice. In 1985 he spent 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.

Michal Forgáč is Assistant Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his M. Sc. in 2006 and his Ph. D. in Computer Science, in 2009. Since 2009 he is with the Department of Computers and Informatics at Technical University of Košice. The subject of his research is metaprogramming, programming paradigms, and systems evolution by run-time adaptation.