5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics
January 25-26, 2007 ▒ Poprad, Slovakia

Dynamic Restructuralization of Software Systems Using Aspect-oriented Programming[1]

Marcel Tóth, Miroslav Beličák

Department of Computer and Informatics, Technical University of Košice, Letná 9, 051 20 Košice, Slovak Republic, {Marcel.Toth, Miroslav.Belicak}@tuke.sk

Abstract: Aspect oriented paradigm (AOP) has recently been applied to several software engineering tasks, particularly for solving tyranny of the dominant decomposition problem of object oriented systems. This paper describes using of AOP in cooperation with artificial intelligence agents for slightly different purpose of successful dynamic recomposition of software system without shutdown of the system or disconnection of its users. Suggested architecture that is used as a basis for described system is called open design architecture (ODA). One of core ideas of ODA allows change in design of the system to be reflected back to the implementation and also to the functionality of the system. This paper shows, how can be AOP used in ODA for transformation of modified design to the new implementation.

Keywords: aspect, AOP, aspect oriented programming, weaving, artificial intelligence agent, software agent, design, system, information system architecture, run-time environment

1 Introduction

Major part of the functionality of current software systems is deployed more-or-less in binary form (except of small data part or configuration part in external databases, data files like XML, ...). The functionality (in binary form) is produced through the process of compilation from source code, where most of information about source code structure (and thus also most of programmer's intention) is lost. The process of creation of software system usually follows the path design-implementation-compiled functionality. This course hides, or literally wastes, the structure of system for final user (possibly designer or different role). If the structure is required, it can be re-engineered, but this doesn't work reliably for all

[1] This work was supported by
VEGA 1/4073/07 – Aspect-oriented Evolution of Complex Software Systems
VEGA 1/2176/05 – Technologies for Agent-based and Component-based Distributed Systems Lifecycle Support

systems. The design of IS (by means of different types of diagrams, formal or semiformal notations) is usually separated from implementation parts of IS (for example source codes, system documentation, etc.) [3]. Maintenance of these two factors can be difficult in large and complicated IS. Contemporary object-oriented languages circumvent this problem of lost structure by including metadata with compiled code. This brings solution for including middle-level structure information. We propose next step in this way, information systems architecture with whole design included (linked to compiled binary components). Architecture is named open design architecture (ODA) and it combines ideas of service oriented architecture (SOA), model driven architecture (MDA) and component based architecture (CBA) [2]. The aim is to create architecture, which is fully reconfigurable in run-time. For this purpose, aspect oriented programming was chosen as supporting paradigm. Another proposed architecture for reconfigurable computation can be found in [7, 12].

2 Open Design Architecture (ODA) and it's Basic Characteristics

This section briefly introduces structure of ODA. Exhaustive description is over the scope of this article and will be published in separated works. Figure 1 shows the structure of ODA.

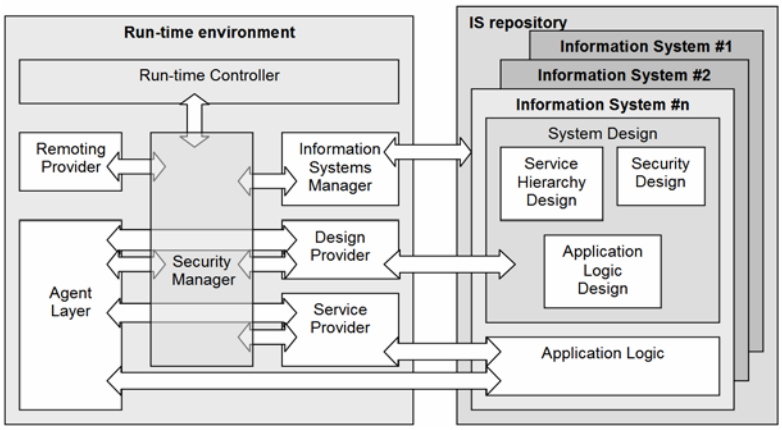

Figure 1
Structure of Open Design Architecture

Main part of ODA is run-time environment ensuring functionality of managed information systems. Co-operation of its individual parts is depicted as arrows in the Figure 1. Run-time environment uses place called IS repository as a storage for all of its information systems. Every information system in ODA consists of

application functionality represented by binary micro-components and **system design** represented by several diagrams. Design of IS consists of three separated sub-designs, which have the form of several diagram types. The first one, Application Logic Design – describes AL. The second one – Security design – designs security aspects – associating user roles with individual services in AL. The third sub-design describes organization structure (hierarchy) of respective services (based on types of service functionality) – Service Hierarchy Diagram. User-designer thus has the possibility of extracting systems design in case of required change. After design extraction, specific integrated development environment (IDE for modifications of design in ODA) can be used to apply required changes – even without breaking of IS's execution and disconnection of users (no consultations with previous designers of system are necessary, because AI agent manages relevant changes). Security considerations of runtime environment and all other security concerns are the responsibility of security manager represented by software agent (SW agent). On the other hand, AI agent's task is to tune-up performance and efficiency for users of IS. It's work includes profiling and setting priorities on behalf of users.

3 Aspect-oriented Programming

3.1 What are Aspects About

Aspect oriented programming [10] and its principles emerged from contemporary most used object-oriented paradigm (OOP), which has matured to the point where its limitations are obvious (e.g.: problem of the tyranny of the dominant decomposition [5]). However, AOP is not only bound to OOP. Instead, it is applicable to almost any paradigm, augmenting it appropriately. Most important advantage, that the AOP is expected to bring, is higher level of decomposition and thus more malleable software systems and easier development (at the cost of steeper learning curve, which is, on the other hand, the problem of almost every new technology or methodology). Elementary principle of AOP is quite simple. Source code is defined in units of modularization of particular language as a model, or directly as a source code (e.g. in classes, when OOP is used), let's call this part of source code *primary code*. Problematic crosscutting functionality [5], which is indecomposable in chosen language is then defined in stand-alone *aspects* that are applied on source code later (exact time depends on principle of AOP engine). Up to this point, the concept of AOP seems to be similar to special kind of configuration files, but substantial difference can be identified, when two basic principles of AOP, obliviousness and quantification [4], are considered.

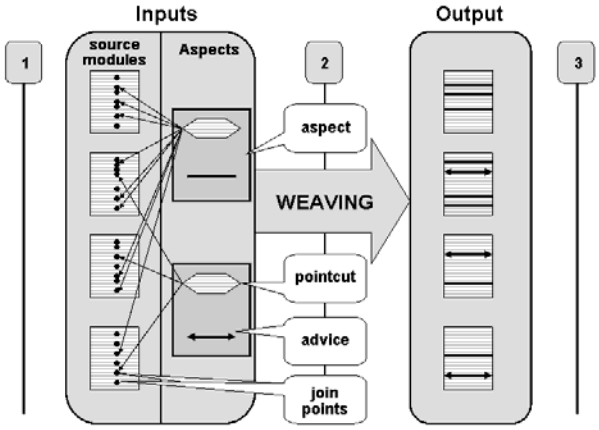

Figure 2
Transformation of service design to source code through aspects

The term **obliviousness** means, that primary code is oblivious to aspects, i.e. designer or programmer of that code doesn't need to know anything about aspects (although aspects will modify the semantics of his code). No special syntactic constructions (no function calls, etc.) related to aspects are included in primary code. Aspects are tied on primary code later, without changes noticeable by primary code. The notion of **quantification** denotes aspect's property of applicability to more than one place in primary code (represented by source code, or some design diagram).

These two properties guarantee independence of primary code from aspect code, which allows applying different aspects to primary code and possibly dynamic switching of aspects. One example for all: logging aspect which records every method call in the program to the file [kircher02xp]. This aspect is useful in time of debugging or profiling (when it will be applied – connected to primary code), but it slows down the regular execution, so it will be disconnected after debugging finishes.

The aspect can be understood as a property of some entities of the system (i.e.: of computation, of software system) which is crosscutting the system in indecomposable (or not readily decomposable) way. From the syntactic view of programming languages, there are more kinds of aspect shapes. We will consider the view of one of the most evolved implementations of aspect oriented paradigm, namely AspectJ [9]. In AspectJ, aspect is an element of decomposition similar to class of object oriented paradigm.

Process of combining aspects with primary code is called **weaving** and its diagrammatical description is in Figure 2. Figure expresses weaving as inputs and outputs of weaving. Inputs are: primary code (marked source modules in the

figure) and aspects, whereas output is modified functionality either in the form of modified source code, or only in the modified run-time functionality (semantics of source code). Aspects are comprised of **pointcuts** and **advices** interconnected in various ways. Pointcuts can be viewed as selectors of relevant **join points** (black dots in the figure) and advice can be viewed as functionality added to join points selected by pointcuts. Join points are not directly visible in primary code, however they are identifiable by pointcuts. Examples of such identifiable join points can be function calls, variable write, variable read, etc.

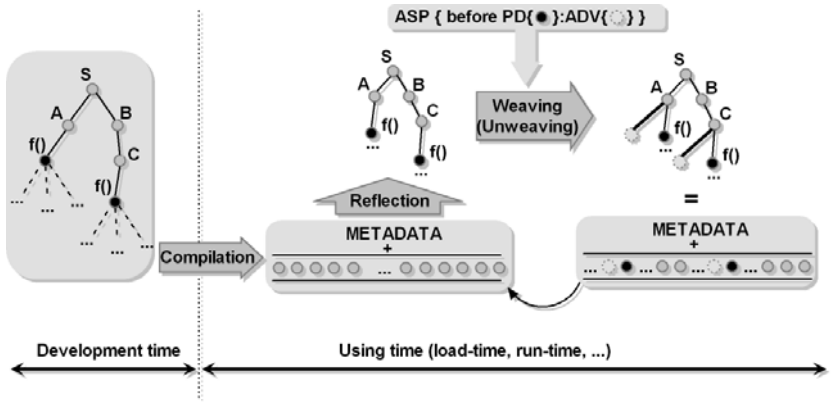

Figure 3
Dynamic weaving principle

Simplified description of aspect's syntax in languages of AspectJ type using EBNF form follows:

```
aspect          :: 'ASPECT_NAME' '{' aspect_parts '}'
aspect_parts    :: aspect_part aspect_parts | ε
aspect_part     :: pointcut_def | advice_def
pointcut_def    :: 'POINTCUT_NAME' pointcut_body
pointcut_body   :: selection_expression
advice_def      :: 'POINTCUT_NAME': advice_body
advice_body     :: where_specifier '{' code_in_programming_language '}'
where_specifier :: 'BEFORE' | 'AROUND' | 'AFTER'
//CAPITALS – terminals, lowercase_letters - nonterminals
//selection_expression and code_in_programming_language explained below
```

Selection expression is so-called primitive pointcut chosen from the set of predefined pointcuts of particular aspect oriented (AO) programming language. Majority of AO languages have primitive pointcuts such as *call(f)*, *set(v)*, meaning call to function *f*, or set a value of *v*, respectively.

3.2 Static and Dynamic AOP

Since the research in the area of AOP identified two basic techniques of weaving [15], both are mentioned, although only dynamic the second one is relevant for our purposes. Main types of weaving are

- static weaving – modification of code before execution,
- dynamic weaving – modification of semantics of executed program.

Static weaving is one form of code refactoring transformation, improving properties of program code (readability, modularity, ...). By contrast, dynamic weaving modifies functionality on the level of individual modules (before loading of required dynamic modules), or directly modifies executing environment, changing semantics of pertinent instructions. Figure 3 shows principle of dynamic weaving. Source code is depicted in the shape of syntactic tree, which is translated to sequence of instructions (black, grey and dashed circles) and included metadata by compilation process. Metadata is then used for re-creating of syntactic tree from sequence of instructions, whenever needed. Inserting of some instructions (dashed grey circles) before instructions for calls to function *f* (black circles) is also depicted.

4 Agents

Agent-based applications and agent systems represent a very robust and theoretically well funded technological paradigm [6]. Despite this, they are not yet widespread at all due to many reasons, one of which is the heavy programming work that still has to be done in order to get efficient and effective agent systems in particular from the point of view of the performance and integration with other applications.

In general, the agent is a concrete entity (a piece of software) that sends and receives messages, while the service is the set of functionality that is provided [8]. We can divide agents into two basic groups:

- Software agents (SW agents), and
- Artificial Intelligence Agents (AI agents).

A software agent is a software entity that possesses an independent thread of execution and has a set of goals or tasks to complete, combined with an internal set of plans that are aimed at achieving of these goals. The independent thread of execution means that the agent determines when to perform tasks based upon observing its ‘environment’ [13].

Another definition of software agent [14]:

A *software agent* is a program, executed at a given place, characterized by: autonomy, communication and learning.

Artificial Intelligence agent (AI agent) has agent several next properties in comparison to SW agent, especially Intelligence, Sociability, Rationality and Character. According to [13] AI agents should also be:

Autonomous, Interactive, Adaptive, Sociable, Mobile, Proxy, Proactive, Intelligent, Rational, Unpredictable, Temporally continuous, Transparent and accountable, Coordinative, Cooperative, Competitive.

For more properties of agents, or more detailed descriptions, see [8, 13, 14].

5 Utilizing Aspects And Agents in Run-time Environment of ODA

In this section, we propose using aspect oriented approach to achieve better properties of code restructuring in ODA architecture and we also outline roles of AI and software agents for security and performance reasons of information systems in ODA.

5.1 Utilizing Aspects

Figure 4 briefly introduces the place for aspect oriented programming (AOP) in development of systems in ODA. Figure needs to be perceived as description of direct steps leading to running system, not as a steps of system's life-cycle or guide to creating system in ODA. For example step 2 (service definition) is regarded as design step and should precede step 1 (compilation - part of system's implementation) in progress of system's design. All steps and their purposes are commented in next paragraphs.

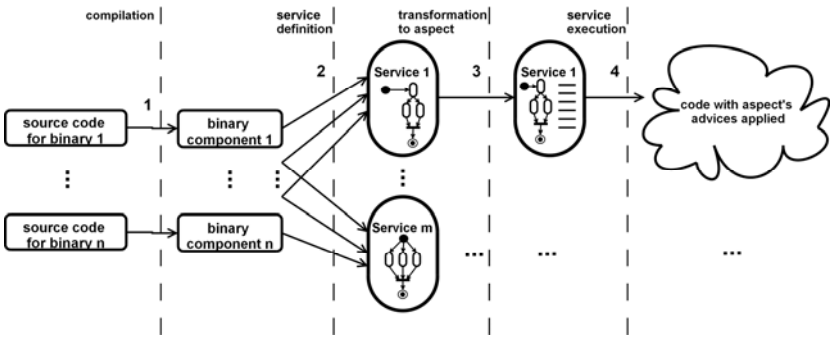

Figure 4
Utilization of aspects in ODA

First step (dashed line marked 1 in Figure 4) is compilation, standard step in programming of systems, transition from source code to executable version of the system. In the figure, this represents transition from source code of binary micro-components (BMCs, as smallest part of system in ODA) to their binary form, compiled binary component. These BMCs have inputs, outputs, description of their functionality and they are stored in common repository supplied by ODA runtime environment (possibly, they are only .dll libraries in file-system).

Second step named 'definition of service' means assembling of services from binary BMCs and/or other services. Practically, it means creation of application functionality design diagrams in supporting design environment (under development). After this step, expected system should be fully designed and implemented. Idea of ODA is to keep system's functionality together with it's design. This is achieved by only keeping design (with aspect code added later), counting on functionality in common repository (in BMCs). Next steps (3-transformation to aspect and 4-service execution) are only ways of bringing working system.

Third step 'transformation to aspect' is most important, because it prescribes how will be design transformed to code. This step is explained in Figure 5 and its description. Part observable in Figure 4 is, that created aspect is stored with design of service and re-created after each change in design of service.

And finally fourth step 'service execution' designates execution of service on demand of user, this includes process of weaving, described later in this article.

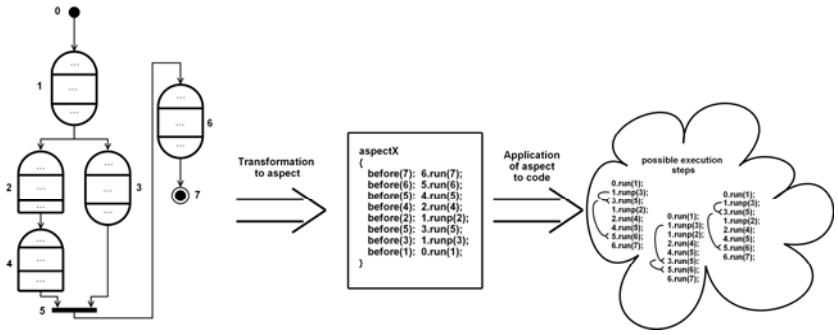

Figure 5
Transformation of service design to source code through aspects

Simplified principle of transformation from application design diagram to AOP aspect and eventually to modified semantics of execution is depicted in Figure 5. Detailed description of these transformations will be offered through description of the picture too. Let us divide the picture to three parts (left, central and right) partitioned by left aiming arrows, for the purpose of explanation.

Symbols used in left part of Figure 5 are application logic diagram symbols of proposed notation in ODA. Symbols marked by 1, 3, 6 are services, i.e.

functionality with some decomposable internal structure. Whole diagram in left part of figure under consideration can be understood as service (on higher level of abstraction, of course). On the other hand, symbols marked by 2, 4 are called binary micro-components (BMCs), which means that they are the smallest part of external functionality usable in application logic diagram. Symbols 0, 5, 7 are start of service, synchronization of parallel execution branches and end of service, respectively. Whole diagram can be regarded as control flow diagram, modeling control flow of application logic and dependencies of functionalities (using simple arrows), data flow description is also included in diagram, but this feature is omitted for the purpose of simplicity in this figure - data flow descriptions are substituted by 'etc.' symbols – '...'. Idea behind whole service described by presented application logic diagram can be grasped as: Service 1 will be executed, and its output will be used as input for two parallel branches of execution. In first branch, BMC 2 followed by BMC 4 runs using output of service 1 for BMC 2 and output of BMC 2 as input for BMC 4. Only service 3 runs in second branch, also using output of service 1 (it is thus clear, that output data of functionality is copied to every branch connected to its output). Execution of the control branches is synchronized in synchronization block 5 (system is waiting for all potential input branches to complete).

The middle part of Figure 5 (produced by 'Transformation to aspect' transformation) shows aspect produced from application control diagram. Details of transformation algorithm are not introduced in full depth, as this is crucial part of ODA still under research, which is subject to change. Mentioned are only important properties of transformation, that should be met. First, only starting point and end point of service are stored in source code of service. All dependencies of application logic (as sequences of other services, parallel execution of micro-components, ...) should be expressed as independent pluggable aspect. Only after this aspect is woven to source code (virtually containing just 2 lines of code for the start and for the end of service), final functionality is achieved. Syntax used in the figure is similar to syntax of most used implementations of AOP principles [1, 9, 11]. Symbolic language of aspect in the figure consists of advices (each line of shape *before(x):y.run(x);* expresses one advice in the figure). Every advice has pointcut part as *before(x),* selecting join point for advice body *(y.run(x))* insertion. Advice body *y.run(x)* runs service *y* and notifies it, that after finishing *y* should call *x*. Keyword *runp* instead of *run* means that functionality is to be executed as individual thread (parallel functionality).

'Application of aspect to code' arrow produces right partition in Figure 5. This arrow means application of aspect to primary code, process called weaving. Seeing that aspect from the middle part of figure is in syntax of AOP extension to some programming language, it is evident that the particular aspect and source code are direct inputs of weaving. With support of latest runtime environment

such as AspectJ's, weaving can be accomplished dynamically[2], during short interval in time of program execution. The cloud in right part of Figure 5 expresses possible steps of execution of individual services and BMCs. All three sequences of execution satisfy the application logic diagram from left part of figure, because execution of service 3 can be done whenever between running of service 1 and running of service 5 (3 is dependent on 1, and 5 is dependent on 3, which is shown in every case of 'possible execution steps' by fork-arrow).

Restructuring of code can also be accomplished by other techniques than aspect oriented principles, for example by individual modules in ODA runtime environment compiling design specifications, etc. By using AOP, well known principle is applied instead of self-constructed technique, meaning that other programmers can more easily gain understanding of our work. Dynamic restructuring of systems is also extensively studied in the field of AOP and thus it is possible, that many unexpected problems will be solved in ODA by using of such elaborated principle as AOP.

5.2 Utilizing Artificial Intelligence Agents

Change in system's functionality is achieved through aspect oriented principles, but some other interrelated issues need to be solved. Examples of such issues can be transfer of connected users to new versions of service, controlling priority of services and their parts, etc. Therefore, AI agent controls the process of functionality modification and ensures correct traverse of all users to new version of service. It also controls execution of services and suggests solutions for distribution of resources of the system. For this purpose, agent stores each user's evidence called **user profile**, consisting of all relevant (agent decides, what is relevant) information about users. The user profile can include statistical data about usage of services by individual users (time of using services, intervals, orders and number of using, ...), precise data about status of used services and other user related data. Based on whole user profile, AI agent can suspend some users for intervals of time, it can also transfer users between versions of service, or even boost system's performance for some task by suppressing other processes. It is evident that utilized agent should have properties like planning, intelligence, and other properties of artificial intelligence agent.

5.3 Utilizing Software Agents

Software agent will be used in ODA for maintaining security of ODA runtime environment (RE). Example of secured processes are remote connections (of users

[2] Research and development in dynamic weaving for AOP is ongoing. This article presents scenario where dynamic weaving is usable, not the definition or means of dynamic weaving which are different in almost every language implementation.

or of other runtime environments) to RE. In the Figure 1 describing structure of ODA RE, block 'Security manager' presents software agent with stated responsibilities. Both AI and software agents need more extensive description which is not allowed by extent of this article, but will be elaborated in consecutive articles describing ODA.

6 Future Work

The work on ODA still requires more precise specifications and operational prototype of runtime environment, that can be used to examine properties of this architecture. Although ODA is designed for better adaptability and maintainability of information systems based on it, expected real-use properties can differ. Subsequent steps in development of ODA will need to try other means of recomposition, not only aspect oriented principles. It is also not clear, whether proposed diagrams (application functionality, service hierarchy and system security) will fully describe required properties of ODA using systems. Therefore more formal description of whole architecture is needed. These aims can bring improvement of quality of information systems.

Conclusions

In this article, we have shown applicability of aspect oriented paradigm, and especially dynamic weaving principle, for recomposition of special kind of software systems. Software systems considered are information systems with newly proposed architecture, called open design architecture (ODA). ODA dictates storing of software system's implementation units together with system's design. Minimal implementation units of functionality are called binary micro-components and their cooperation is managed through design in the form of application functionality diagrams (AFD) of individual services. AFD of every service is transformed to AOP aspect right upon creation or change of service. In this way, aspects and their usage (dynamic weaving) rule control flow and data flow of software system. Advantages and drawbacks of such approach directly correspond to advantages and disadvantages of platforms for dynamic weaving. Drawbacks can include lower performance and enhanced initial complexity than other approaches to recomposition. However, expected benefits (as almost arbitrary changeability and adaptability) clearly outweigh in larger scale systems.

References

[1] Avdičaušević E., Lenic M., Mernik M., Žumer V.: AspectCOOL: An Experiment in Design and Implementation of Aspect-oriented Language. SIGPLAN Not., 36(12):84–94, 2001

[2] Beličák M., Paralič M., Havlice Z.: Distributed Service-Oriented Information Systems with Open Architecture, in Proceedings of Electronic

Computers and Informatics ECI 2006, The 7th Int. Scientific Conf., Košice-Herľany Slovakia, September 20-22, 2006, pp. 8-12

[3] Beličák M.: Architecture Design for Distributed Information Systems implemented in Multiagent Run-time Environment, in Proc. of 6th PhD Student Conf. of FEI TU Košice, May 17, 2006, pp. 19-20

[4] Filman R. E., Friedman D. P.: Aspect-oriented Programming is Quantification and Obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, Workshop on Advanced Separation of Concerns (OOPSLA 2000), October 2000

[5] Filman E. Robert, Elrad Tzilla, Clarke Siobhán, Aksit Mehmet, editors. Aspect-Oriented Software Development. Addison-Wesley, Boston, 2005

[6] Grosso A., Gozzi A., Coccoli M., Boccalatte A.: An Agent Programming Framework Based on the C# Language and the CLI, in Proc. of 1st Int. Workshop on C# and .NET, Computer Graphics, Visualization, Computer Vision and Distrib. Computing February 6-8, 2003, Plzen, Czech Republic

[7] Harakaľ M., Turčaník M.: New Architectures for Reconfigurable Computation, Proc. of Conf. "Nové smery v spracovaní signálov VII", Tatranské Zruby, May 2004, pp. 57-62

[8] Christopher D. Walton: Model Checking Multi-AgentWeb Services, Centre for Intelligent Systems and their Applications (CISA), Informatics, University of Edinburgh, UK

[9] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. An overview of AspectJ. In J. L. Knudsen, editor, Proc. ECOOP 2001, LNCS 2072, pp. 327-353, Berlin, June 2001, Springer-Verlag

[10] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.: Aspect-oriented Programming. Technical Report, February 1997

[11] Kollár J., Tóth M.: An experiment with Aspect Programming Language, Proceedings if the 3rd Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence, Herľany, Slovakia, January 21-22, 2005, pp. 225-235

[12] Mernik M., Zumer V.: Incremental Programming Language Development. Computer Languages Systems and structures, 2005, Issue 31, pp. 1-16

[13] Object Management Group, Agent Platform Special Interest Group: Agent Technology (Green Paper), OMG Document agent/00-09-01, Version 1.0, 1 September 2000

[14] Rykowski J., Cellary W.: Virtual Web Services - Application of Software Agents to Personalization of Web Services, in Proc. of ICEC'04, Sixth Int. Conference of Electronic Commerce, ACM 2004, 1-58113-930-6/04/10

[15] Tóth M.: Static and Dynamic Weaving in Aspect-oriented Programming, in Proc. of 5th PhD Student Conf. of FEI TU Košice, 2005, pp. 121-122