Computer Science • Vol. 7 • 2005

Włodzimierz Funika[*], Arkadiusz Janik[**]

INTEROPERABILITY OF MONITORING-RELATED TOOLS

Networking, distributed and grid computing have become the commonly used paradigms of programming. Due to the complicated nature of distributed and grid systems and the increasing complexity of the applications designed for these architectures, the development process needs to be supported by different kinds of tools at every stage of a development process. In order to avoid improper influences of one tool to another these tools must cooperate. The cooperation ability is called interoperability. Tools can interoperate on different levels, from exchanging the data in common format, to a semantical level by executing some action as a result of an event in another tool. In this paper we present some interoperability models, with focus on their advantages and major problems due to their use. We also present an interoperability model designed and used in the JINEXT extension to OMIS specification, intended to provide interoperability for OMIS-compliant tools.

Keywords: *en, interoperability, JINEXT, mediator, monitoring tools, OMIS*

INTEROPERABILNOŚĆ NARZĘDZI MONITORUJĄCYCH

Przetwarzanie rozproszone i gridowe jest obecnie dominującym paradygmatem obliczeniowym. Skomplikowany charakter systemów rozproszonych i gridowych oraz rosnąca złożoność projektowanych aplikacji sprawia, że na każdym etapie tworzenia systemu informatycznego konieczne staje się użycie narzędzi wspierających ten proces. Aby uniknąć zakłócenia pracy jednego narzędzia przez pracę innego, narzędzia te muszą współpracować. Zdolność ta nazywana jest interoperabilnością. Interoperabilność można rozpatrywać na kilku poziomach, począwszy od wspólnego formatu danych, a skończywszy na poziomie semantycznym, na którym jedno z narzędzi reaguje wykonaniem pewnej akcji w odpowiedzi na zdarzenie wygenerowane przez inne z narzędzi. W artykule przedstawiono kilka modeli interoperabilności, opisując zalety i wady każdego z nich. Przedstawiono również model zastosowany w JINEXT, rozszerzeniu specyfikacji OMIS o mechanizm interoperabilności.

Słowa kluczowe: *interoperabilność, JINEXT, mediator, narzędzia, system monitorujący, OMIS*

1. Introduction

In the world of nowaday's computer science creating new applications has become a very difficult, often long-term process. The quick evolution of software engineering and increasing requirements relating to created products in range of commercial systems

[*] Institute of Computer Science, AGH-UST, Kraków, Poland, funika@uci.agh.edu.pl

[**] Institute of Computer Science, AGH-UST, Kraków, Poland, arek.janik@interia.pl

or also scientific applications cause the developing new software becomes more and more difficult. Developers create more and more large, also more complex systems. At the same time parallel, distributed and grid processing has became undoubtedly a predominant computational paradigm. Developing applications to be used on these types of architecture creates many additional problems. A complexity of problems solved with them grows up.

1.1. What is "monitoring tool"?

In order to simplify, accelerate as well as to minimize the number of errors, bugs and mistakes done during a development process, it is necessary to use additional tools supporting this process. The set of applied tools, also known as *toolset*, should be used in every phase of forming a new computer system process from writing a specification to testing the finished software product. Applications supporting the process of creating software systems are called *monitoring tools.*

A simple definition of *monitoring tool* is as follows [7]:

Monitoring tool – is a run time application which is used to observe and/or manipulate the execution of a software system.

Monitoring tools are used to support a development process as well as to trace the workflow of a running program, analyzing some aspects of its work, delivering the user of the system feedback information about it, or even modifying the workflow of the system. Popular examples of such tools are *debugger* and *profiler.* Another well known example is *load balancer* responsible for equal load of each node in a distributed environment.

1.2. The concept of the interoperability of monitoring tools

As one can see the clue of the monitoring issue is that two or more tools are monitoring the target application ***at the same time***. This means that different tools, used to analyze different aspects of a monitored system can run *simultaneously.* These tools are often provided by different vendors, written in different programming languages. To make a simultaneous functioning of tools possible they must *interoperate.*

1.2.1. Definition

While skipping other language-related or system-related aspects of interoperability, we will concentrate on the concept of interoperability as the ability of monitoring tools to cooperate. However, the interoperability term can be defined in a few different ways depending on the aspect chosen as the most important and the most interesting in a particular area. According to ANSI IEEE *interoperability* is defined in the following way:

Interoperability – the ability of two or more systems or components to exchange information and to use the information that has been exchanged.

The above definition stresses the information exchange aspect. As an example a simple situation can be described when two people speaking completely different languages are trying to communicate. As a result they can exchange information but unless they find a way how to interpret the information it will be useless. A similar situation can be considered when talking about an interaction of monitoring tools using completely different data formats. The IEEE's definition of interoperability emphasizes the common semantics of exchanged information which must be shared by all cooperating systems. The semantic level of information is also considered in Fisher's definition (accroding to [3]):

Interoperability – the ability of two or more applications to communicate with each other. Any application in a given toolset can communicate with any other application in a reliable way. Exchange of information is done on syntactical as well as on semantic level.

In literature there can be found some other definitions of interoperability. Wielden and Kaplan underline the fact that cooperating applications are usually written in different programming languages, working on different hardware platforms [12]. They propose the following definition:

Interoperability – the ability of two or more software components written in different programming languages (C++, Java, etc.) to interact and communicate with each other.

The variety of programming languages and different software platforms are stressed in the interoperability definition presented by Howie, Kunz and Law (according to [3]):

Interoperability – the ability of two or more software components to share and process common information regardless of differences between software platform and programming languages they are implemented in.

The interoperability definition proposed by American National Institute of Standards and Technology emphasizes the fact that interoperating applications are distributed, often running on different nodes [13].

Interoperability – the ability of applications started and run on different machines to exchange information and cooperate in processing this information.

Wegner [10] notices another aspect of interoperability as follows:

Interoperability – the property of the autonomous, heterogenous and independently developed software components of interaction between each other.

1.3. Common aspects of the issue of interoperability

Even a short analysis of the monitoring tools' cooperative issue leads to a simple conclusion that the interoperability of the monitoring tools is a very difficult problem, full of potential dangers. There are very few coherent sets of monitoring tools (toolsets) which can be used to support the process of designing, implementing and maintaining software systems. Using systems which come from different vendors may cause a lot of problems due to different techniques of monitoring, different formats of exchanged data produced when monitoring, and also completely different work logics. Using such tools from different vendors may not accelerate the developing process but even disturb it. Let's consider a situation when one monitoring tool, for example a debugger, modifies the code of the application, let's say by inserting a trap instruction in the middle of it. If there is another monitoring tool, e.g. a profiler operating on the same application and that tool is not informed about actions of the debugger, changes to the code may seriously affect the functioning of the second tool, even causing its failure. One way to avoid such situations is to assure that a monitoring system meets some requirements connected with different levels of abstraction. *Structural and logical conflicts*, which are the common problems, will be presented, with special attention to the consistency and transparency aspects of the logical conflicts.

1.3.1. Structural conflicts

The first problem is to run the tools *concurrently.* Concurrency means that monitoring tools are working at the same time being attached to the same software system, to the same application.

In order to solve the concurrency problem, a proper runtime infrastructure should be provided. Monitoring tools are usually based on some module that is responsible for acquiring the necessary data during run-time. This module is called *monitoring system* [7]. A monitoring system may be responsible for providing a mechanism of a concurrent execution of monitoring tools, avoiding a negative influence of concurrently executed monitoring tools on each other, should manage the common resources etc.

All the above problems are called *structural conflicts.* To avoid structural conflicts the functioning of monitoring tools should be *coordinated.* The only way to coordinate monitoring tools is to execute them with a common monitoring system as explained above.

1.3.2. Logical conflicts

Even if all problems resulting from structural conflicts are solved, *logical conflicts* may still be encountered. Logical conflicts are connected with semantical issues.

Consistency aspect. As long as logical conflicts are not solved tools can not *consistently* co-exist. To better understand the consistency problem, let's consider two monitoring tools, a visualizer and a load balancer both monitoring the same target system. In order to balance the CPU load of each node, on which the target system executes, the load balancer may decide to migrate one task of the target system from node A to node B. The visualizer is monitoring the node A of the migrated task and displays information that the task is executed on node A.

In other words, more formally, the consistency problem can be described as *read/write access conflict* [3]. Every operation on the target system can be considered as a read or write access. If the monitoring tool wants to receive information about some aspect of the target system (e.g. the node a task is executed on) it is called *read operation.* If the monitoring tool manipulates and modifies the state of the target system (e.g. migrates a task from one node to another) it is called *write operation.* A *read access* starts at the moment of reading a value of a target system's property.

The end of read access is the moment at which the monitoring tool assumes no longer that the read value is the same as at the beginning. Similarly, *write access* can be defined. Now, a conflict arises if a read access to an object temporally overlaps with a write access.

To resolve logical conflicts, the monitoring system and also monitoring tools have to be notified. On one hand, the monitoring system must provide a mechanism to notify the tools about other tools' access and modification of the target system. On the other hand, monitoring tools must execute some actions on these notifications, process them and update any internally held or assumed information about the target system [5]. These two elements are necessary to provide consistency.

Transparency aspect

By analogy to the consistency conflict, the *transparency* conflict arises if one monitoring tool has *write access* to some value while another tool is still active and possesses the value. In the transparency conflict the tool sees a changed value in place of the original state. This is an unwanted situation because we would like to assume changes to the target system are *transparent* for another tools. Transparency means that in some cases the presence of a monitoring tool should be invisible for other monitoring tools. The observation of changes introduced by a tool is just a side effect. In order to explain we use an example that follows. Let's take a debugger tool which inserts a *trap instruction* into the code of a monitored application. This is necessary to set a breakpoint at a particular address during debugging. If the presence of the debugger and the changes introduced are not transparent, another monitoring tool may read a trap instruction inserted by the debugger as the original part of the target application's code. And this is just a side effect. In case of the transparency problem, the tool which modifies the state of some object needs to intercept accesses to the modified state value and to handle their results in a relevant way [9].

2. An overview of related work

The interoperability term is just one instance of a more general issue of software systems *interconnection* problem in which the nature and granularity of the systems are left unspecified [11]. It is generally recognized that the integration of tools requires three steps:

1. data interoperability – how tools can exchange and share data structures representing application specific information;
2. control interoperability – how tools can communicate and cooperate with each other;
3. user-interface interoperability – how user interfaces can be designed to provide a common, uniform appearance and behavior;

From above ones the most interesting is control interoperability also known as interoperability on *semantic level*. On this level cooperating tools should have possibility to interpret actions and events generated by other ones. They should know what they mean and what to do if a given tool executes some action. The semantic level is needed to solve all structural and logical conflicts described in Section 1.3.

A number of different interoperability models can be found. In the following sections we will present the most popular semantic level interoperability models with an example of the editor and compiler objects. The interoperability of these two objects means that in case of changing a source code in an editor, the compiler will be informed about it and will automatically recompile the code to let it stay up-to-date.

2.1. Encapsulation

The encapsulation method combines software components into a bigger, monolithic system, called *wrapper* which encapsulates individual tools [4]. These tools cannot be accessed directly by clients. Encapsulation forces the client to use the wrapper, ensuring integration. The only access to tools' methods is possible from within the body of wrapper routines which can be called by the client. By calling a wrapper's method by the user, the wrapper not only executes software object's procedures but also does some integrity steps. Without integration the editor could, e.g., save a source file but without the compiler being run. The wrapper provides then its own interface. The scheme of this model is presented in Figure 1. The client is not allowed to call editor's and compiler's methods (grey colored) directly. The client's invocations are shown with arrows.

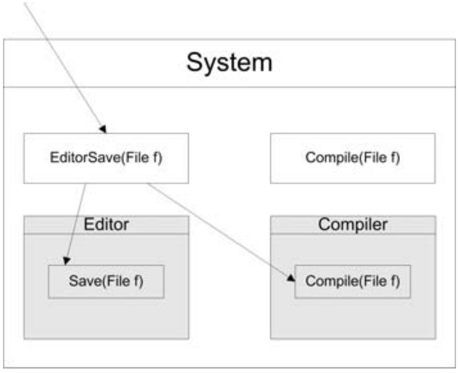

Fig. 1. The encapsulation method schema

The major advantage of this solution is that in case of the integration of different tools there is no need to change their source code. Instead, other code, wrapper's code, has to be added. On the other hand, clients of the system must be changed. The wrapper's *Wrapper.EditorSave()* method has to be called instead of *Editor.Save()*. Another disadvantage is that the wrapper solution results in monolithic types [4]. A single wrapper realizes every aspect of each specification. When more behaviors and relationships between tools are needed, it may lead to large wrappers with complex interfaces. The scalability of such systems is rather poor. So does the evolution.

It means that if the functionality of any tool is changed or a new relation between tools is introduced the source code of the tool and the wrapper have to be modified in many places.

2.2. Hardwiring

On the contrary to the encapsulation method, hardwiring needs the source code of tools be modified. If two monitoring tools need to cooperate, they have to make calls to each other. It means that the programmers have to insert a call to compiler's operation in a source code of the editor's save method, but the clients' code does not have to be changed. Clients depend on the tools' interfaces only. The scheme of this model is presented in Figure 2. Dotted arrows mean that the editor tool calls directly the compiler object to execute the *Compile* method.

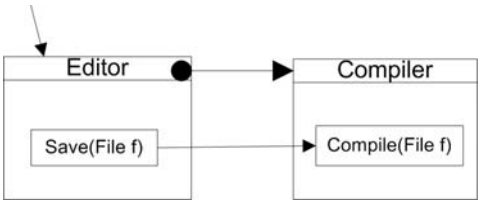

Fig. 2. The hardwiring method schema

Modifications to the functionality of any tool cause some serious problems. Relations between tools are complex and tools must reference each other, so changing tool's functionality needs changes to the source code of both of them.

2.3. Implicit invocation

This solution is very similar to hardwiring. The main difference is providing different relationship structures between tools. A given tool can register another tool to be invoked by its events. Instead of the editor calling the compiler, the compiler can register with the editor to be notified when the editor announces a *file saved* event. Implicit invocation is an old idea. Because of a great similarity to the hardwiring model, problems with the implicit invocation method are the same as in case of the hardwiring model [4]. A scheme of this model is presented in Figure 3.

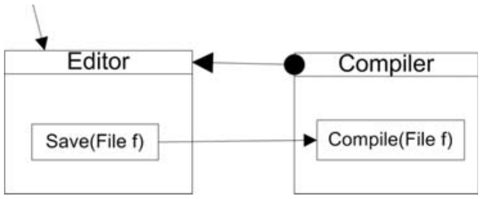

Fig. 3. The implicit invocation scheme

2.4. Broadcast Message Servers

In the broadcast message server (BMS) solution, a specialized server is used to coordinate communication between different tools. Each tool informs the server about the actions it supports. If a given tool wants to invoke an action provided by another tool, the server is used as the intermediator. If a given tool wants to inform other tools about its events also sends a special announcing message to the server. Every tool should subscribe for events it wants to be notified about. In our example the broadcast server calls directly *Compiler.Compile()* method whenever the editor announces the server about saving a modified file.

A scheme of this model is presented in Figure 4. The server is used to remember subscriptions for events.

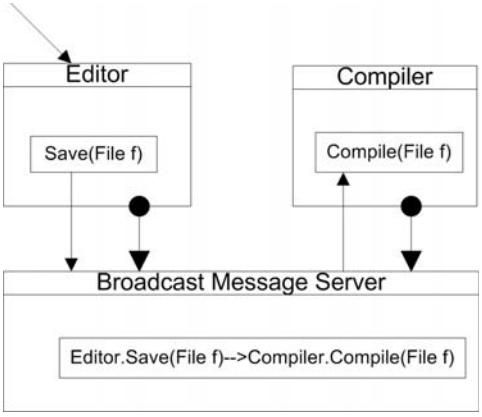

Fig. 4. The broad message server approach scheme

The major advantage of this model is providing an additional level of independence in tools' relations. Cooperating tools rely on interfaces of each other. It means that every change in some tool interface results in the necessity of changing another tools implementation. BMS model could be considered a monolithic solution [4].

2.5. Mediator

The mediator model underlines the separation between a behavior and tools which implements this behavior. On the one side, in the mediator model, there are behaviors provided by tools, on the other side there are behavioral relations. A *behavioral relation* is the connection between two behaviors of two different tools. Each behavior in the model is implemented by an object that represents this behavior. Adding relationships to the behavioral model results in adding an object that represents this relationship. The mediator solution characterizes a high level of dynamics [4].

Each behavior in the mediator model is realized by a proper abstract behavioral type (ABT) [3]. For instance, an editor behavior is realized by *Editor* ABT with an operation to save a file and event *Saved()*. A behavioral relationship is realized by the mediator. Having registered the *UponSave()* operation with this event, it is invoked whenever the event is raised. It should be underlined that the mediator is external to the objects whose behaviors are integrated. A scheme of this model is presented in Figure 5. The editor is viewed as an object which provides the *Save()* action and generates *Saved()* action. A notification about saving a file is done by the broadcast message. The Mediator "captures" this message and calls the compiler's *Compile()* method. The mediator is used to externalize a behavior relationship between Compiler and Editor.

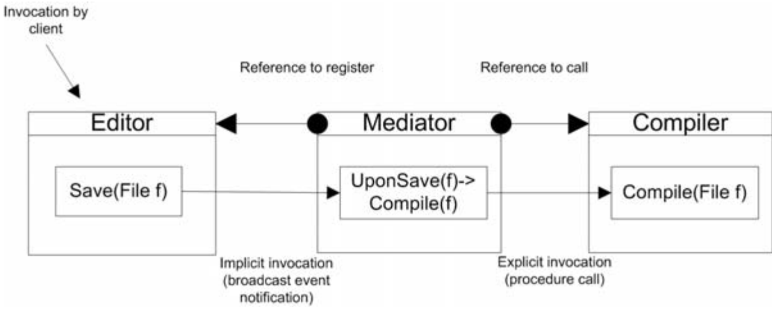

Fig. 5. The mediator approach scheme

The mediator solution guarantees flexibility in changing relations between tools. E.g. if we would like to improve the behavior of a compiler so it recompiles a modified file only if CPU load is low, only the behavioral relationship can be changed. The editor and compiler do not have to be changed. Moreover, if we want to commit the modified file to a repository system (e.g. CVS.) each time it is modified, we can introduce an additional tool. Improving the behavioral model is to add another relation between the new tool and the editor.

The major advantage of this solution is that developers and analysts are enabled to focus clearly on behaviors and their relationships. The behavioral relationship modelling provides a framework, ready to use design scheme that helps think clearly about integrated behaviors. Graph structures can be used to present behaviors and relations between them. Adding a behavior can be mapped to adding a vertex into graph as presented in Figure 6. Introducing a new relationship results in adding a new edge. This causes that the mediator solution simplifies the integration and evolution process. The vertices of the graph represent the behaviors provided by tools and the edges reflects behavioral relations. In case system evolves a new behavioral relation can be added without a system's source code modification. In Figure 6b a new behavioral relation between the *Compiler* and the *Debugger* is added. The new relation (restarting an application in the debugger after recompiling its source code) is represented by an edge added to the graph.

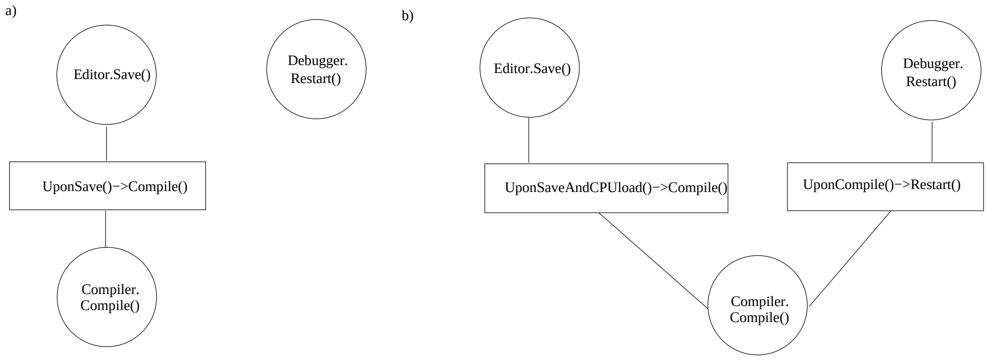

Fig. 6. The mediator graph

2.6. Agent system based models

In order to solve common interoperability problems, models based on agent systems can be used. Of course, an analogy between monitoring tools and agent systems is only fragmentary but if we interpret a monitoring tool as an agent system providing some functionality the analogy is possible. To achieve this, two types of monitoring tools should be distinguished [8]:

1. Provider – a tool which is providing some services. Its features is the ability to satisfy strictly specified requests, e.g. to carry out some measurement, start/suspend some process etc.
2. Requester – a tool which has the ability of sending a request to a provider and consuming information returned by it. The results can be provided to the user in a special form, such as a user understandable chart.

An example of a system realizing such a model is LARKS [8]. There is a specialized layer (realized as an independent agent system) called *matchmaker*, used as an intermediator between the provider and requester. The main task of matchmaker is to find a proper tool registered in the system, which will be able to realize a request submitted by another tool. Each tool which wants to cooperate with other tools should register itself in the matchmaker informing it about the services it can realize. Each request sent by the requester should be forwarded to the matchmaker instead to a particular tool. The matchmaker returns the list of tools which can serve the request. The requester chooses a relevant tool from the list.

In the above solution an exact matching of a request and a service the provided by provider is not necessary. It means that the matchmaker chooses services which are most suitable and similar to the requested one. It is necessary to use a mechanism which will be able to interpret the meaning of a requested service and to choose the best one from all provided ones. The suitability of a service is considered in terms of semantics. It means that in a matching process not only services parameters and a result in taken into consideration but also the semantic meaning of a service. It is more important what a service does than what the service's parameter names are.

An example of the above mechanism is Agents Capability Description Language (ACDL) [8]. This language is used to describe the services provided by a given tool. A description involves both, semantical and structural aspects of a service.

The process of matching services and a called request is realized in the following steps of filtration and selection:

1. Context matching
   From all tools these ones will be filtered which are able to realize services in a given context. By the context term a particular domain is meant. In one context there can be, e.g., on-line monitoring tools, another one may be used to represent monitoring tools operating at the process, network levels etc. Distinguishing between contexts is arbitrary, depending on particular system's requirements.
   Filtering is the process of selecting services which are candidates to support the request. During filtering, useless services are omitted.
2. Syntactical matching
   From all the tools filtered in Step 1, those ones will be filtered which provide services similar to the requested one at the syntactical level, which means the similarity (or identity) of input/output parameters.
3. Semantical matching
   From all tools filtered in Step 2, those tools will be chosen which provide services similar to the requested one at the semantical level. The estimation in this step is carried out based on pre- and post-conditions which are attached to the definition of each service registered in the matchmaker. These conditions must be met as to the input parameters and result of the service.

3. JINEXT model of interoperability

In order to solve the structural/logical problems and to provide interoperability of monitoring tools which comply to the OMIS interface we designed JINEXT – Java INTeroperability EXTension to OMIS. OMIS (On-line Monitoring Interface Specification) [6] is a well known specification of a monitoring system/tools interface architecture and JINEXT is an extension to it aimed at providing interoperability support. For details about JINEXT and its implementation please see references [5]. In this article we will just briefly present the main ideas underlying our model of the interoperability.

The model used in JINEXT is based on the mediator model described in Section 2.5. In JINEXT, each tool must implement a single interface, called *tool type*. This interface simply defines which actions can be executed by such a tool and what events can be raised by it.

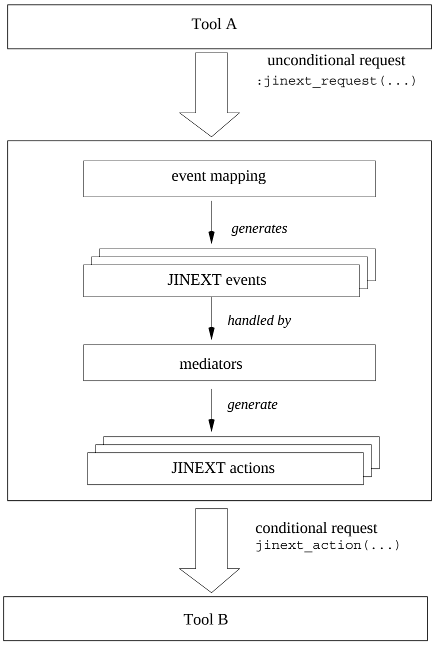

Fig. 7. The JINEXT interoperability model

JINEXT uses the idea of mediators, however it also extends the mediator model in the way presented in Figure 7. If a monitoring tool sends a request to monitoring system (any direct communication with another tool is forbidden), the requested service is translated into one ore more JINEXT events using event mappings. In other words, each requested service (e.g. monitoring system's service) can generate one or more JINEXT events. Special structures called *event mappings* are used during translation. Then mediators come to handle a JINEXT event and generate JINEXT actions executed by destination tools. Each mediator connects a single event generated by a tool of type A with a single action to be executed by tool of type B. What differs the JINEXT mediator of the mediator presented in Section 2.5 is the use of *mediator function*. While the mediator in JINEXT refers to an abstract behavioral relation only, the mediator function implements this relation. A mediator defines that if tool A raises event X then tool B should execute action Y while a mediator function is responsible for translating the parameters of event X into parameters of action Y.

The important advantage of the model used in JINEXT is the simplicity of adding a new relation between the monitoring tools while their implementation stays unmodified. Moreover, the mediator function makes it possible to change the behavior of tool B (used in above example) while the mediator remains the same. It means that monitoring tool's behavior can be flexibly changed by providing new mediator functions, while the very mediators remain untouched, after they have been once well defined.

4. Conclusions and future work

As we mentioned developing a new application in nowadays is becoming an increasingly complicated process. Analysts, designers, and programmers have to solve more and more difficult problems. To do so they need a set of integrated monitoring tools supporting the software development. The tools must interoperate we discussed. Some interoperability models were proposed during the last period. Each of them has advantages and, of course, disadvantages when used. The most interesting model is the mediator one. In JINEXT we extended the idea of mediators so it can be used in an OMIS compliant monitoring system. The mediator model presented in the paper provides a separation of a monitoring tool's behavior and the relations between tools. It means that implementation of monitoring tools can be carried out separately from the design of relationships between them. In this way we can provide a clearly defined graph of relations between the events and actions of different tool types. The mediator function proposed in JINEXT goes further. By changing the mediator function the developer can change the details of the action generated as the result of some event while the graph of relations between different tools remains unmodified. The next step of design the JINEXT model of interoperability is to extend it for grid support.

Acknowledgements

Our thanks go to prof. Roland Wismüller for valuable discussions. This research was partially supported by the KBN grant 4 T11C 032 23.

References

[1] Rackl G.: *Monitoring Globus Components with MIMO.* Institut für Informatik, Technische Universität München, March 2000 (PhD Thesis)

[2] Baliś B., Bubak M., Funika W., Szepieniec T., Wismüller R.: *An Infrastructure for Grid Application Monitoring.* In: Kranzlmüller D., Kacsuk P., Dongarra J., Volkert J. (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, September – October 2002, Linz, Austria, 2474, Lecture Notes in Computer Science, 41–49, Springer-Verlag, 2002

[3] Wismüller R: *Interoperable Laufzeit-Werkzeuge für parallele und verteilte Systeme*, Habilitationsschrift, Institut für Informatik, Technische Universität München, 2001

[4] Sullivan K. J.: *Mediators: Easing the Design and Evolution of Integrated Systems.* Dept. of Computer Sciences and Engineering, Univ. of Washington, USA, 1994. Technical Report 94-08-01. (PhD. Thesis) `ftp://ftp.cs.washington.edu/tr/1994/08/UW-CSE-94-08-01.PS.Z`

[5] Janik A.: *Interoperability of the monitoring tools supporting development of Java Mediators.* Institute of Computer Science, University of Science and Technology, June 2004 (MSc. thesis)

[6] Ludwig T., Wismüller R., Sunderam V., Bode A.: *OMIS – On-line Monitoring Interface Specification (Version 2.0).* Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997)
`http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`

[7] Wismüller R.: *Interoperability Support in the Distributed Monitoring System OCM.* In: R. Wyrzykowski *et al.*, (Eds.), Proc. 3rd International Conference on Parallel Processing and Applied Mathematics – PPAM'99, pages 77-91, Kazimierz Dolny, Poland, September 1999, Technical University of Czestochowa, Poland. Invited Paper

[8] Sycara K., Jianguo L., Klusch M.: *Interoperability among Heterogenous Software Agents on the Internet.* The Robotics Institute Carnegie Mellon University, Pittsburgh, USA, October 1998

[9] Trinitis J., Sunderam V., Ludwig T., Wismüller R.: *Interoperabiity Support in Distributed On-line Monitoring Systems.* In: M. Bubak, H. Afsarmanesh, R. Williams, B. Hertzberger (Eds.), High Performance Computing and Networking, 8th International Conference, HPCN Europe 2000, volume 1823 of Lecture Notes in Computer Science, Springer, 2000

[10] Wegner P.: Tutorial Notes: *Models and Paradigms of Interaction. Technical Report CS-95-21.*, Department of Computer Science, Brown University, Providence Rhode Island 02912, USA, September 1995

[11] Bergstra J. A., Klint P.: *The ToolBus – a component interconnection architecture.* Programming Research Group, University of Amsterdam, Meeting, Band 1697 aus Lecture Notes in Computer Science, page 51–58, Barcelona, Spanien, September 1999.
`ftp://info.mcs.anl.gov/pub/techreports/reports/P754.ps.Z`

[12] Wileden J. C., Kaplan A.: *Software Interoperability: Principles and Practice.* In: Proc. 19th Intl. Conf. on Software Engineering, Boston. ACM Press. May 1997.

[13] Bagwill R., Barkley R. *et al.*: *Security in Open Systems. Special Publication SP 800-7.* National Technical Information Service (NTIS), Springfield, VA, USA, 1994. `http://csrc.nist.gov/nistpubs/800-7/`