Providing Interoperability for Java-Oriented Monitoring Tools with JINEXT

Włodzimierz Funika and Arkadiusz Janik

Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
funika@uci.agh.edu.pl

Abstract. While Java is getting an increasingly widely used programming language, Java Virtual Machine has become an important platform for networking and distributed computing. Due to the increasing complexity of programs, the demand for monitoring tool support (debuggers, performance analyzers etc.) for efficient computing is growing as well. A special, open interface J-OMIS, which provides an efficient support for monitoring distributed Java programs, is derived from the On-line Monitoring Interface Specification (OMIS) that enables to use multiple monitoring tools simultaneously. OMIS has been developed with interoperability in mind but not all structural and logical conflicts have been solved. Addressing the missing aspects of interoperability support within J-OMIS is intended to increase the simplicity of developing the monitoring tools which will synergetically support each other. In the paper we present the concept of JINEXT, an extension to OMIS, which is aimed to provide interoperability between monitoring tools.

1 Introduction

In the world of nowadays' computer science creating new applications has become a very complicated, difficult, often long-term process. The software systems become larger because the complexity of problems solved with them grows up. In order to simplify, accelerate as well as to minimize the number of errors and bugs done during developing process, it is necessary to use *monitoring tools* supporting this process. Interoperability is the ability of two or more software systems to cooperate with each other. However, using tools from different vendors may not accelerate the developing process but even disrupt it, unless a special coordinating facility, like *monitoring system* provides a kind of interoperability support.

In this paper we present some issues of adding an interoperability-oriented extension to Online Monitoring Interface Specification (OMIS), which solves structural and logical interoperability conflicts not handled by OMIS. In Section 2, we focus on the *mediator* interoperability model. In Section 3, we address the common problems connected with the issue of interoperability . Section 4 presents JINEXT, an extension to the OMIS specification and in Section 5 we present a practical example of using JINEXT. In Section 6 we discuss the ongoing work on JINEXT and further research.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 158–165, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Related Work

There are a number of different concepts of *interoperability* [10]. In the paper we address interoperability in the context of monitoring tools, as a kind of *cooperation* between them on the *semantical* level where interpretation of actions and events provided by each cooperating tool must be guaranteed.

A number of different interoperability models can be found in literature. In the *ToolTalk* system [1] a tools' interaction can be described independently from the underlying communication mechanism. However, debugging tools which use ToolTalk, as well as difficult deadlock analysis are big disadvantages of it. The Eureka Software Factory (ESF) and Purtillo [1] based on the *hardware metaphore* model use a special language to describe mappings between logical modules (tools) of the system. These systems solve interoperability problems on the *control* level, so the semantical issues are out of their interests. There is also a model based on the solution where tools are interpreted as the agent systems. An example is LARKS [6]. There is a specialized layer (called *matchmarker*), used as an intermediator between the tools. The Agents Capability Description Language (ACDL) [6] is a language used to describe services provided by different tools. A description involves both, semantical and structural aspects of a service, but conflicts due to the issue of interoperability are still not solved.

The *mediator* model stresses the separation between a behavior and tools which implement this behavior [3]. A *behavioral relationship* term is used to describe a relation between different tools, which is a connection between two behaviors of two different tools. Each behavior in the mediator model is realized by a proper abstract behavioral type (ABT) [2]. For instance, an editor behavior is realized by *Editor* ABT with an operation to save a file and an event *Saved()*. This event is raised whenever a file is 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. Note that the mediator is external to the objects whose behaviors are integrated. The editor is viewed as an object which provides the *Save()* action and generates *Saved()* event. 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 solution promotes software modularization and is resistant to a behavior evolution. For example, if we would like to improve the behavior of compiler so it recompiles a modified file only when CPU load is low, only the behavioral relationship can be changed. The editor and compiler do not have to be changed. The only modification is an update of the mediator object. Moreover, if we want to send the modified file to a remote backup directory whenever it is modified, we can introduce an additional tool, e.g. a *backup manager*. A way to improve the behavioral model is to add another relation between the improved backup manager and the editor. To implement this change, we have to add a backup manager object and the mediator realizing the relation between the manager and the editor [3].

3 Common Aspects of the Issue of Interoperability

When considering interoperability as the ability of two or more applications to cooperate, we will concentrate on a special case of software systems, called *monitoring tools*, as run-time applications which are used to observe and/or manipulate the execution of a software system. Monitoring tools need a *monitoring system* to be able to work in a proper way based on monitoring data. The monitoring system is part of the tool infrastructure responsible for observing and manipulating the target system, which provides the monitoring functionality for tools, based on some interface. OMIS specifies an interface between a monitoring system and tools [4].

In this context, interoperability is the ability of applying two or more tools to the same software system, at the same time. In order to provide these features some common problems have to be solved.

Structural Conflicts. The first problem is to run the tools *concurrently*, which means to allow them to work at the same time being attached to the same software system. The monitoring system should provide a mechanism for the concurrent operation of monitoring tools, for sharing the common resources (hardware, processes, threads, classes, objects, memory). Without a common monitoring system the tools may access a target system in a completely uncoordinated manner [2].

Logical Conflicts. *Logical conflicts* are connected with semantical issues. As long as logical conflicts are not solved tools cannot co-exist *consistently.* It means that when manipulating on the target system, tools cannot preserve a consistent view of this system, e.g. when a profiler measures the time of a process execution while this process is suspended by a debugger. More formally, the consistency problem can be described as a *read/write access conflict* [2].

4 OMIS Versus the Interoperability of Tools

OMIS and the OCM monitoring system, its reference implementation, are intended to provide a kind of interoperability for tools relating to structural conflicts and conflicts on exclusive objects [5]. All tools that share an object use the same controlling monitor process that coordinates accesses to the monitored objects. Therefore, tools are enabled to coexist. However, the transparency problem was not resolved yet. The support for avoiding logical conflicts in OMIS is incomplete.

The concept of OMIS allowed to extend OMIS and the OCM by monitoring support for Java distributed applications, known as J-OMIS and J-OCM, respectively [7]. To provide a wide interoperability for Java-oriented tools, we designed Java INteroperability EXTension (JINEXT) as an extension to OMIS (see Fig. 1).

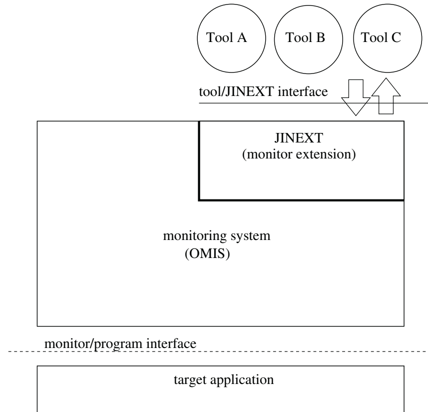

Fig. 1. The overview on JINEXT architecture

In order to be interoperable some conditions have to be met:

- the tools communicate with the JINEXT extension only, instead of calling the monitoring system directly;
- a component implementing the JINEXT specification (a.k.a. *JINEXT compliant system*) is responsible for passing requests to the monitoring system and for synchronization between different tools, as well as for choosing target monitoring tools to be informed about changes due to request;
- each tool must implement a single *interface* (*events* and *actions*) which is used to emphasize that the behavior and functionality provided by each tool is the same for all the tools of a given type (e.g. there can be four different debuggers, developed by different providers, in different languages, all of them sharing a common interface).

The Mediators. *The mediators* in JINEXT follow the *mediator* interoperability model [3]. A mediator is a connection between the event generated by a tool and the action which should be executed by another tool whenever the event occurs. In JINEXT each mediator is used to describe one *behavioral relation* between an event and an action. If the developer wants to introduce a tool implementing a new tool type, she/he has to provide mediators which are responsible for interoperability between the new tool type and all other tool types.

The 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 ease 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.

There are two different modes of tools' cooperative in JINEXT: *non-cooperative mode* and *cooperative mode*.

- *non-cooperative mode* - it is defined w.r.t. a pair of monitoring tools what means that these two tools can work in the non-cooperative mode when considering this particular pair but any of these tools can work in the cooperative mode with some other tool. In the *non-cooperative mode* tools working with the JINEXT compliant system do not have to know that there are other monitoring tools connected to the system, which are monitoring the target application. JINEXT takes care of tools' synchronization.
- *cooperative mode* - two tools work in this mode if they know about each other. Each of them can request another tool for an action execution. It means that the functionality provided by the first tool can be used by the second one, e.g. to extend the functionality of monitoring tools and to build a complex toolset consisting of smaller components, like the *IDE* (Integrated Developer's Environment).

5 Use Case

In order to verify the features of JINEXT we have developed its prototype implementation, called JINTOP. We have also developed four example monitoring tools: an editor, a compiler, a debugger and a profiler. A sample scenario of interoperability between tools is as follows:

1. the user changes the source file of the application *App*
2. the **editor** *saves* the changes;
3. the **compiler** *recompiles* the code and generates the new binary file;
4. the simultaneously used **debugger** *restarts* the debugged application with a new code;
5. during the restart of the application **profiler** *pauses* and *resumes* the previously defined performance measurements.

The sequence diagram for this scenario explaining the role of JINTOP is presented in Fig. 2.

The editor notifies JINTOP about event *source_code_saved* and JINTOP uses mediators to obtain information on the attached tools which are interested in this event. Each event is checked in the context of calling tool so two events e.g. *Debugger.process_stopped* and *Profiler.process_stopped* will be handled by different mediators and may trigger different actions. After having checked events, *callback* method *jinext_action()* is used to execute a tool's action (e.g. *recompile_source_code*). This causes raising the *binary_code_modified* event. If a tool

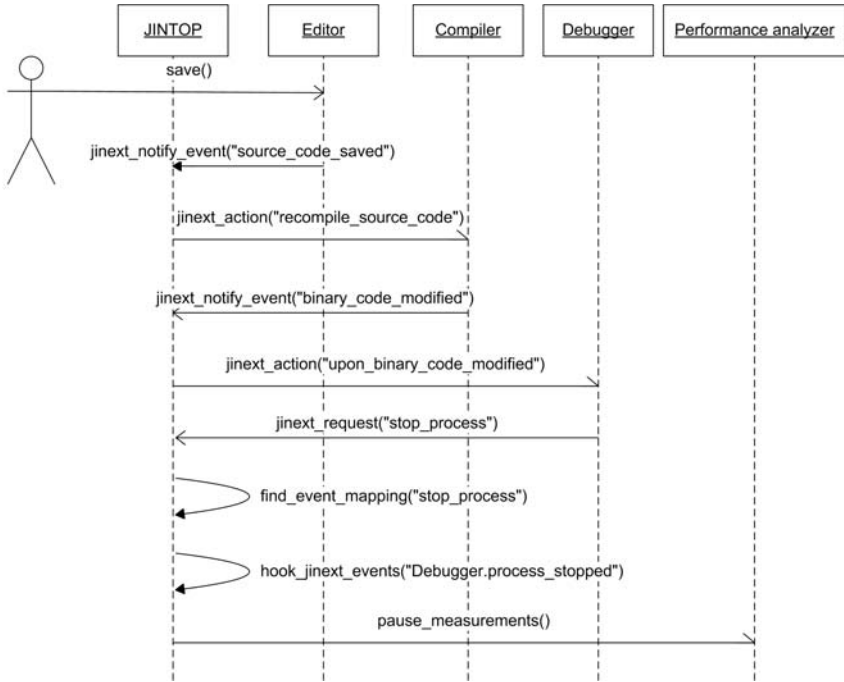

Fig. 2. The sequence diagram for the *save()* operation

requests OMIS/J-OMIS service (e.g. a debugger requesting the *stop_process* service) JINTOP executes this request and hooks zero or more events (*Debugger.process_stopped*) which trigger (via mediators) zero or more actions (*Performancer.pause_measurements*). As one can see, events can be hooked directly by tools (*jinext_notify_event*) or by JINTOP as a result of executing an OMIS/J-OMIS service. JINTOP takes care of consistency after having executed the OMIS/J-OMIS service by the tool. Moreover, JINTOP do not bother any tools which are not attached to the *tokens* (threads, processes, classes, objects etc.) connected with the notified event.

Overhead Due to Using JINTOP. We have carried out experiments to measure the overhead induced due to using JINTOP instead of direct calls to the OCM. The results of our experiment are presented in Fig. 3. The experiment was done on AMD Athlon 2000+ 1,8GHz with 1GB of RAM, under Linux 2.23 OS.

The first observation is that the execution time of processing a request is longer if there are more processes the request refers to. The second conclusion is that the overhead introduced by JINTOP increases if there are more monitoring tools registered by JINTOP. However, it is always less than ca 22% and can be further reduced by changing the internal data structures of JINEXT implementation.

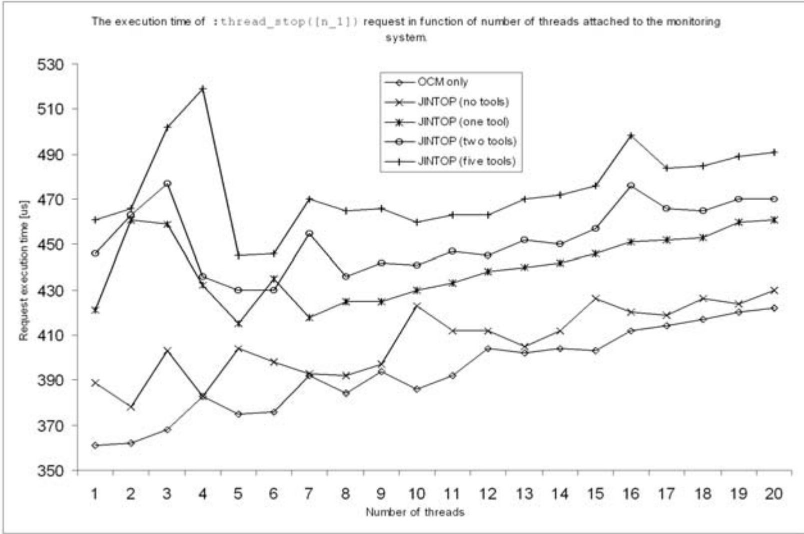

Fig. 3. The execution time of an example request in function of the number of threads attached to the monitoring system

6 Conclusion and Future Work

JINEXT, as an extension to OMIS for tools' interoperability, is designed to enable different monitoring tools to run concurrently, via solving logical and structural conflicts. It enables simultaneous operation of different monitoring tool types, without system failures. The mediator interoperability model used in JINEXT guarantees that an evolution of a behavior of monitoring tools can be done seamlessly from the viewpoint of interoperability.

The first version of JINEXT has been released. However, some useful services should also be added to JINEXT, e.g. these ones which enable dynamic addition of a new tool type allowing the developer to describe the monitoring tool type in an external XML file. The next step in further research is to extend JINEXT to be compatible with OCM-G [8]. Providing the interoperability of monitoring tools working in a grid environment is necessary if we want to allow more than a single user to work on the same target object from within different nodes or sites. Let us consider the situation when one user on node A is doing some performance measurement on an application whereas another user on node B is debugging this application. If these monitoring tools are not interoperable the first user will receive false measurement results. A solution would be to add a special interoperability agent on each node of the grid. The agent analyzes requests and recognizes whether they are coming from a local or remote node of the grid. In the second case, the agent will do all necessary token translations. The agent can also support an additional interoperability-related security policy.

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. 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, Spain, September 1999. `ftp://info.mcs.anl.gov/pub/techreports/reports/P754.ps.Z`
2. Roland Wismüller: Interoperable Laufzeit-Werkzeuge für parallele und verteilte Systeme, Habilitationsschrift, Institut für Informatik, Technische Universität München, 2001
3. K. J. Sullivan. Mediators: Easing the Design and Evolution of Integrated Systems. PhD. thesis, Dept. of Computer Sciences and Engineering, Univ. of Washington, USA, 1994. Technical Report 94-08-01. `ftp://ftp.cs.washington.edu/tr/1994/08/UW-CSE-94-08-01.PS.Z`
4. Ludwig, T., Wismüller, R., Sunderam, V., and 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`
5. Roland Wismüller: 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.
6. Katia Sycara, Jianguo Lu, Matthias Klusch: Interoperability among Heterogenous Software Agents on the Internet, The Robotics Institute Carnegie Mellon University, Pittsburgh, USA, October 1998
7. M. Bubak, W. Funika, M.Smętek, Z. Kiliański, and R. Wismüller: Architecture of Monitoring System for Distributed Java Applications. In: Dongarra, J., Laforenza, D., Orlando, S. (Eds.), Proceedings of 10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, LNCS 2840, Springer, 2003
8. Baliś, B., Bubak, M., Funika, W., Szepieniec, T., and Wismüller, R.: An Infrastructure for Grid Application Monitoring. In: Kranzlmüller, D. and Kacsuk, P. and Dongarra, J. and 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
9. Günther Rackl: Monitoring Globus Components with MIMO, Institut für Informatik, PhD Thesis, Technische Universität München, March 2000
10. Peter Wegner: 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 `http://www.cs.brown.edu/publications/techreports/reports/CS-95-21.html`