A Tool for Monitoring Software-Heterogeneous Distributed Object Applications

Krzysztof Zieliński, Aleksander Laurentowski
Jakub Szymaszek, Andrzej Uszok
e-mails: {kz,pinio,jasz,uszok}@ics.agh.edu.pl
Institute of Computer Science
University of Mining & Metallurgy
Al. Mickiewicza 30, Cracow, Poland

Abstract

The next decade will bring radical changes to the way we do information processing, as applications composed of many cooperating distributed subsystems, exploiting different computational paradigms, become more common. This situation increases substantially demands in the area of system management, correctness analysis, understanding, debugging and performance evaluation. Managed Object-based Distributed Monitoring System (MODIMOS) is a project aimed at development of an adaptable platform for visualization of distributed applications, built of interoperating heterogeneous components. MODIMOS is expandable, allowing to add new monitored environments. It employs various management mechanisms to cope with gathered information size and complexity.

1 Introduction

This paper[1] describes an experimental system called MODIMOS (Managed Object-based Distributed Monitoring System) that addresses structure and dynamic behavior visualization of the coming object-based distributed software-heterogeneous applications. The system serves for monitoring and visualization of applications built of many cooperating components, written in different computing environments. We believe that, due to advances in interoperability techniques, such applications will become common in the near future, and software technology based on composition techniques will belong to the dominating ones [6, 7].

[1] This work was sponsored by the Polish State Committee of Scientific Research (KBN) under grant no.: 8 S503 015 06

Majority of existing monitoring and visualization tools are performance-oriented and do not support object-oriented applications [18, 14]. The other are mostly limited to a single programming environment [15] or investigate rather centralized than distributed processing paradigm [16, 13]. Moreover, (1) they support neither interoperability nor cooperation within a software-heterogeneous environment; (2) their architecture is not predestined to easy expansion (by adoption of new environments). The first case eliminates interesting multi-lingual programs from the application area. In the second case, it is impossible to adapt the system to changing requirements, as for instance novel emerging programming environments.

On contrary, MODIMOS investigates mostly structural aspects of the monitored applications. The user may observe changes and shape of the application's structure as well as behavior of its components. The system may serve to detect imbalances of the application's configuration and overloads of its components' interaction, thus allowing appropriate tuning. MODIMOS is adaptable, i.e. it can be expanded by adding new distributed environments, representing different computational models. This was achieved due to a structured multi-layer architecture of the system. The architecture defines interfaces and protocols that offer the user fine-grained control over several fundamental system abstractions, e.g. code instrumentation profile, events collection strategy, interoperability mechanisms, storage of monitored data, and graphic presentation. These features were supported by the chosen implementation technique oriented towards software reuse, with regard to the use of frameworks [4] and design patterns [5].

The key to convenient visualization of this kind of applications (which are inherently large and complex) is a problem-oriented navigation system, that enables collected information to be efficiently managed by the user. We mean that the user should be able to observe only a selected part of the application activity, and focus on it, ignoring unimportant information. In MODIMOS, this is achieved with management, filtering and selection mechanisms, available at each level of the system.

MODIMOS runs in two modes: on-line and off-line visualization. The first one may be used when the events flow is slow enough to be perceived. In other cases, the latter mode (based on recording a trace of events in a file and replaying them during visualization process step-by-step or with a suitable speed) should be utilized.

The rest of this paper is structured as follows. In Section 2 we describe the MODIMOS architecture and introduce the Uniform Model of Object-based Distributed Processing. In Section 3 monitoring framework is shortly specified. Section 4 describes the interoperability framework. Next, visualization of dynamic behavior with the graphic user interface is described. The paper is ended with comments on status and directions of the project.

2 MODIMOS Architecture

As stated in the previous section, MODIMOS allows simultaneous uniform monitoring and visualization of a heterogeneous family of object-based environments. To achieve that, the system has a multi-layer architecture shown on Fig.1. Functionality of these layers may be described as follows.

The Environments Layer consists of an expandable set of popular distributed object-based programming environments. Any new object-based environment can join the Environments Layer, provided it fits at least partially our abstract Uniform Model of information processing, and supports basic mechanisms of communication with the outside world. The Environments Layer consists of Monitored Applications sublayer and Local Monitoring sublayer. The Monitored Application Layer represents original application code, instrumented with notification functions by a set of special preprocessors. Events reported by notification functions are collected in the Local Monitors sublayer. Each local monitor is a managed object written in a language provided by the given environment. It has three interfaces: Monitored Events, Management and Reported Events. Management Interface is used for a monitoring policy setting, that determines which events received via Monitored Events Interface are forwarded through Reported Events Interface. Information sent via Reported Events Interface is structured according to the Uniform Model. Therefore, above the first layer only the abstract semantics is recognized.

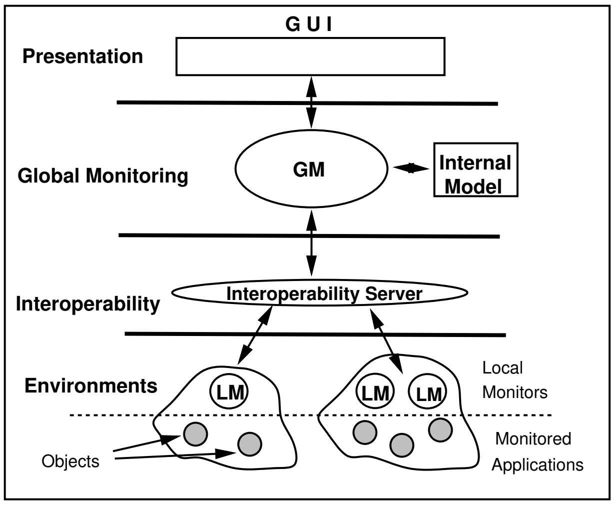

Figure 1: MODIMOS expandable architecture

The second layer deals with interoperability aspects. The aim of the Interoperability Layer is to ensure a universal and general platform for transparent operations dispatching between local monitors and Global Monitoring Layer.

The Global Monitor collects reported events in a database called Internal Model (reflecting the architecture and current state of the monitored application), cooperates with Graphical User Interface in the process of information visualization, implements the selective monitoring policy, and manages domains of local monitors.

The layers mentioned above provide high expandability of MODIMOS system. Each environment and application can possess one or more local monitors, as appropriate. Interoperability Layer can be based on various integration engines, such as, e.g. CORBA [3], ToolTalk, socket gateways. Flexibility of MODIMOS is due to its framework design. Modules from neighboring layers cooperate with each other via fixed interfaces. However, the abstract interfaces located below the Interoperability Layer have to be mapped to the each local language of the Environments Layer.

2.1 The Uniform Model of Information Processing

The monitoring system is assumed to be applicable to a set of diverse object-based distributed environments. To provide a common platform for that purpose, a uniform abstract model of the environments' architecture and activity should be designed, with which they could comply. We have analyzed a set of existing environments and decided that the abstract model should be a superset (union) of their models of computation. This approach guarantees that all the possible levels of abstraction are covered by the abstract model. From a logical point of view, an application in an object-based system consists usually of some number of communicating independent objects. In our Uniform Model of information processing, the following levels of abstraction have been recognized (in order of inclusion): environment, application, container, object, interface and method. The detailed mapping of them into a set of popular programming environments, i.e. ANSA [1], SR [2] and a CORBA-compliant (Orbix) [3, 12] is shown in the Tab. 1.

Unit of abstraction	**Environment**		
	ANSA	SR	Orbix
application	—	makefile	—
container	capsule	virtual machine	process
object	object	resource, global	object
interface	interface	resource specification	interface
method	operation	operation	method

Table 1: Mapping for some concrete environments

To enroll a new environment to MODIMOS, its model of computation should conform at least in part to the Uniform Model.

A short description of the identified Uniform Model components follows below.

Application. In close-architecture environments, like e.g. SR, it is reasonable and straightforward to identify and differentiate an "application" entity, as a single program executed by one user. Problems arise when we consider open systems, like ANSA or CORBA, with publicly available services, shared by many users, written by many independent developers. The only definition that comes without question is functional, i.e. only the user is able to identify the application he is currently using.

Container. Containers are areas of memory, shared by objects, in which they reside. We usually regard entities equivalent with operating system processes as containers.

Object. We decided to regard as objects all the places, where interfaces can be plugged in, to satisfy cases, where one object can possess more than one interface (like e.g. in ANSA or CORBA).

Interface. Interface resides in an object and provides attributes and methods for external use.

Method. Methods are characterized by their signatures. MODIMOS handles each method invocation as a separate entity.

2.2 Monitored Events

A set of relevant events is associated with each abstraction unit. These events are reported to the monitoring system. The following events have been defined in the Uniform Model, as well as a set of corresponding abstract notification functions (see Tab. 2).

Unit	**Events**	**Properties**
Environment	Registration Deregistration	Name, HostList Ticket
Application	Create Destroy	Ident, Name Ident, Reason
Container	Create Destroy	Ident, Name, Host, ProcId Ident, Reason
Object	Create Destroy	Ident, Name Ident, Reason
Interface	Create Destroy	Ident, Name Ident, Reason
Method	Invocation Activation Termination Reply	CallerIdent, Name, SynchType, Parameters Ident, Name, CallerIdent, Parameters Ident, Result CallerIdent, Name, Result

Table 2: Uniform Model Events

This set of functions acts as a pattern for interfaces used for data exchange throughout the system, i.e. in all local monitors and the Global Monitor. For the purposes of particular environments (where the local monitors reside) these interfaces need to be refined according to the local syntactic rules (languages). The interface pattern for the events notification is specified in Interface Definition Language [3], to allow easy porting to various syntactical conventions.

2.3 Entity Identification

A specific problem in the area of software-heterogeneous object environments is to assure a space of globally unique object identifiers in the whole system. Each environment possesses its own, non-compatible format of identifiers. We propose usage of structured identifiers to solve that problem. An identifier in MODIMOS is a concatenation of a *ticket* (a number pertinent for each environment, assigned arbitrarily by the Global Monitoring module), with a native identifier, treated as a sequence of bytes.

3 Information Collection Framework

Monitored information is reported by notification functions inserted into the applications' code by preprocessors during instrumentation phase. The preprocessors are parameterized; the available options cover selection and filtering monitored information (sample switches are: *exclude_level, exclude_events, focus_on_level*). A set of currently used options is reported to the Global Monitor during registration of application.

Invocations of the notification functions, at the time of events' occurrence, are directed to local monitors. A group of local monitors (in some cases a single monitor) maintains the information flow in a given environment. In systems in which the notion of application exists (like SR), the code for local monitoring facilities is automatically linked with an application by preprocessors. In open systems (as ANSA or CORBA-compliant systems), local monitors should be rather independent servers, available publicly for many users and applications. In any case, local monitors collect information in their environments, filter and schedule it. Selected information is forwarded further to the global monitoring subsystem via the Interoperability Layer.

MODIMOS allows dynamic creation and binding of local monitors with the global one. Besides collecting and routing monitored information, local monitors may fulfill many additional functions, e.g. compression, translation of data format, information selecting, filtering, etc., according to the chosen management policy.

The Monitored Events Interface may reflect local specifics, according to the given environment's semantics. It is the role of a local monitor to map them into the Uniform Model terminology, as the only one used by the Reported Events Interface. Let: ME - a set of events generated by the instrumented code; RE - a set of events selected by the monitoring policy. RE is a subset of ME. This relation may be set dynamically by the choice of a monitoring policy performed by the given monitor. ME set depends on instrumentation of application's code and may be changed by the proper setting of flags for instrumenting preprocessor. Hence, in MODIMOS two mechanisms of monitored information filtering are available: (1) monitoring policy performed by a local monitor and set dynamically via the Management Interface, (2) instrumentation options' setting, which is fixed on the application level.

3.1 Consistency of Monitored Data

Monitoring of a distributed application rises a question about consistency of the collected data. This well known problem of achieving causal ordering of events in the absence of global synchronized time may be solved in a satisfactory way via various logical clocks [20] synchronization algorithms, e.g. an optimal variant of vector clocks [21, 22]. Unfortunately, they introduce a substantial overhead, so we have implemented an alternative solution to this problem. The regular, hierarchical structure of the monitored applications and semantics of events introduce a partial order relation between sets of the monitored events, e.g. the *destroy* event concerning object O cannot be delivered if the *create* has not yet been delivered. This allowed us on the first stage to implement a filter based on semantic analysis of incoming events and a set of simple delivery rules of this kind. This is quite enough for monitoring open distributed systems, such as e.g. network management systems, where only occurrence or frequency of certain events (a change of the system's structure), e.g. overload, configuration or alarm are meaningful. However, to satisfy cases where causal relations are crucial for the observation, we plan to implement the vector clock algorithm in the near future.

4 Interoperability Framework

Monitored environments may have various incompatible architectural solutions and protocols making them unable to contact the global monitor (GM) directly in a unified way. Especially, what is crucial for our purpose, dispatching mechanisms of remote invocations may differ significantly. Introduction of an intermediate layer [8, 9, 10], called IL – Interoperability Layer (Fig. 1) into the MODIMOS architecture can be solution to these problems. The aim of IL is to ensure a universal and general platform for dispatching invocations between the local monitors, active in different systems, and GM.

Internal construction of IL can be based on various solutions. Two main technical aspects of its architecture and functionality, which determine possible solutions are:

1. **Bidirectional** communication between local monitors and GM. This relation is not *client – server*, but rather *peer to peer*.
2. **Heterogeneous** set of invocation dispatching

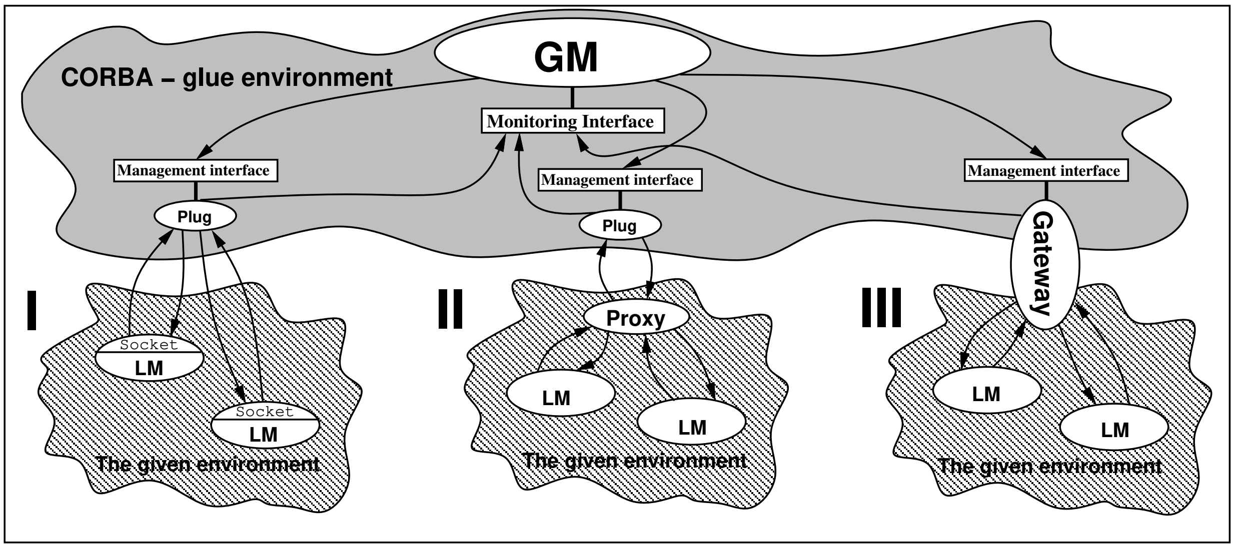

Figure 2: Variants of Interoperability Layer architecture

mechanisms, inherent in monitored environments.

The chosen solution affects configurability, easy extension and efficiency of constructed layer. Possible choices of IL's communication mechanism are:

Proprietary communication mechanism – built especially for IL, based on some primitive communication interface (sockets or similar mechanism), offering only direct communication channel between peers.

Distributed (*glue*) environment – e.g. CORBA (Orbix) [3, 12], ANSA [1] or ToolTalk, offer more abstract notion of communication between partners, provide a lot of services (localization, binding, dispatching, name service, etc.).

Simplicity seems to be the main advantage of the proprietary communication mechanism solution. Information is sent as uninterpreted buffers of bytes. Communication between partners is direct. Its primitiveness ensures its efficiency – transfer of data is fast and it is possible to send a lot of independent messages in a given period of time. Unfortunately, this solution hinders easy configurability and structuring of the Interoperability Layer. All mechanisms and services needed for this purpose have to be implemented from scratch.

These disadvantages disappear when a distributed *glue* environment is used. It enables easy system bootstrapping and changes in its configuration during the system's work. Messages are sent by invocations of functions from remote servers' interfaces, what frees programmer from buffers construction. It also ensures well structuring and encapsulation of IL functionality into environments' objects. The disadvantage could be a lack of efficiency, which can be achieved in proprietary communication mechanism. However, MODIMOS is aimed at selective monitoring and observation, so the amount of exchanged data can be significantly reduced.

4.1 Idea of a *plug*

If a distributed environment is chosen and used as a *glue* for IL, it must be somehow integrated with the monitored environments. It is quite complicated, because of various monitored environments' features. The perfect solution would be construction of an object being a full member of *glue* and a given monitored environment at the same time. It means that this object should be able to call and receive invocations from both these environments, so it may act as a *gateway* between them. However, there is a lot of environments which do not fullfil this assumption.

A solution for this problem is an idea of a *plug* [10]. It is a regular *glue* environment object, which, however, represents a given monitored environment in the *glue* system. It receives remote operations' invocations from the given environment's local monitors, translates them into *glue* system's remote operation calls format, and invokes them on behalf of the local monitors. Invocations from all *plugs* are in an ordinary *glue* system form.

The *glue* environment serves as a communication framework for *plugs*, it localizes the global monitor, binds *plugs* to its interface and dispatches their invocations to it.

4.2 Structure of Interoperability Layer

In MODIMOS, we decided to construct IL with one of *glue* environments – CORBA (Orbix), and to use the idea of a *plug*.

We distinguished three possible variants of *plug* specialization and construction of connection between it and a given environment (Fig. 2):

Socket solution – communication between local monitors and a *plug* is established using *socket* interface. This solution is used when a given environment does not offer threads or other facility to enable implementation of asynchronous communication so it is not possible for local monitors to be a server and a client simultaneously. Then, an event-driven mechanism (based on signals) has to be applied. This solution can be used e.g. for Strand or PCN [11].

Proxy solution – communication between local monitors and a *plug* is forwarded through an intermediate component – *proxy*. It is a regular object of the given environment and local monitors communicate with it using local invocations of its interface. On reverse, it sends messages to them via standard environment's calls (threads have to be available and used in local monitors and *proxy* object). A link between *proxy* and *plug* is set up with a socket interface. We applied this solution to ANSA and SR [10].

Gateway solution – *proxy* and *plug* components are put together in one module – a *gateway*, which is an object available and working in a CORBA-compliant system and a given environment at the same time. It receives invocations from local monitors, translates them and forwards into CORBA-standard calls to GM interface. This solution can be used only when it is possible to construct and initialize object into these two environments simultaneously. *Gateways* are currently being developed for some CORBA-compliant systems [9] (Orbix and DOME).

An instance of an IL's part for a given environment is obtained as a result of framework refinement process [4]. We have developed general construction of Interoperability Layer components, i.e. *plugs*, *proxies*, *gateways* and socket modules linked up with local monitors. These components are chosen and specialized for requirements of a given environment, according to a selected variant of IL architecture.

5 Presentation Layer

Presentation Layer of the MODIMOS monitoring system is implemented by a Graphical User Interface (GUI), that performs two independent functions: management interface and visualization. Applications running in the environments to be visualized consist of items composing a hierarchical structure (a tree), described in terms of a model specific for the particular environment, and in terms of the Uniform Model, common for all environments. The information about the current state of monitored items and structural dependencies between them are stored in the Global Monitor's Internal Model. The structure of IM is also hierarchical, and corresponds with the structure of monitored environments. Each node of IM's tree contains a detailed description of the current state of one monitored item.

GUI's main role is visualization of behavior of monitored systems' selected parts or, in other words, presentation of the whole or parts of the information stored in IM's tree.

5.1 Methods of Items' Presentation

Visualized items are represented on the screen by geometrical figures. Attributes of each figure (shape, color, etc.) depend on the type of the visualized item (its level in the Uniform Model). Performing a specified action (mouse click) on a figure causes displaying the detailed information about an item (e.g. its physical location).

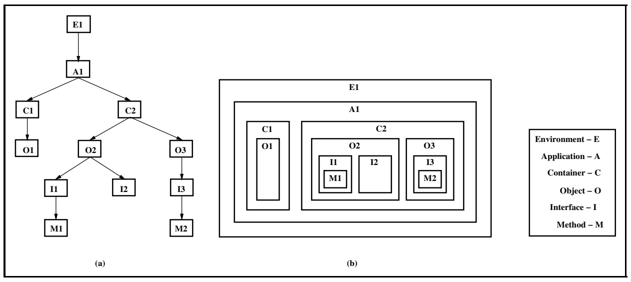

Figure 3: Two methods of tree's presentation: a) tree-like and b) box-like

Visualized items compose a tree structure. One can distinguish two principal methods of graphical representation of tree structures: tree-like (Fig. 5.1a) and box-like (Fig. 5.1b) representation. They differ in the way of presentation of relation between nodes. In tree-like representation, relations between parent node and child nodes are depicted by arcs or lines that connect corresponding figures. In box-like representation, implemented in MODIMOS, figures corresponding with child nodes are placed inside the figure representing the parent node.

Subtrees are displayed in so called containing windows. Roots of all subtrees displayed in a window lie on the same level and belong to the same parent (or more generally the ancestor) item, that is called the root item of the given containing window. Apart from figures composing subtrees, a containing window shows the path describing location of the root item in the items hierarchy. It consists of path figures representing all ancestors of the root. So, the content of the window reflects the current structure and location of the root item. An example containing window is shown on Fig. 4. This window visualizes the application *A1* from the items hierarchy depicted on Fig. 5.1. Hence, this application is the root item of the window. A window includes two subtrees shown in box-like form. Their roots represent items *C1* and *C2* that lie on the same level (they are containers) and belong to the application *A1*. The application is running in *E1* environment. Therefore, the root path consists of two path figures representing *E1 environment*, and the abstract root of the items hierarchy depicted by a ("/") sign.

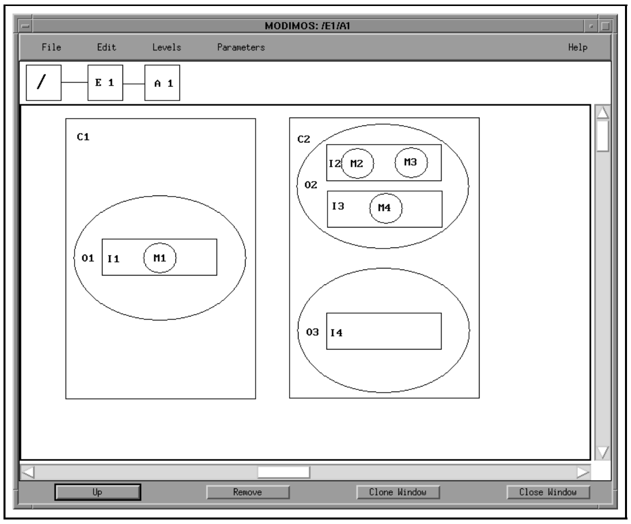

Figure 4: An example containing window

5.2 Navigation in the Items' Hierarchy

The user is able to move over the items' hierarchy and change the root item currently observed in the containing window. In order to go below the root, one of the figures in the window pane should be selected. Selection of one of the path figures causes a move up in the hierarchy. The new root will be the item that corresponds with the selected figure. Its path and content will be displayed in the window. The user investigating the behavior of a distributed system may want to observe many of its parts or layers simultaneously. Therefore, the possibility of creation many containing windows corresponding with different root items is provided. When GUI starts its work, it creates the first containing window corresponding with root of items hierarchy ("/"). It includes figures representing all monitored environments active in the network.

5.3 Selection of Items and Events

In spite of filtering mechanisms provided in code instrumentation and local monitors, the effective visualization of all items composing the root item (and all reported events concerning them) may be unnecessary and often, because of their amount, impossible. For this reason, the ways of visualized items and events' selection are also provided in Presentation Layer. There are two basic methods of items' selection: horizontal and vertical. The horizontal selection (Fig. 5) enables elimination the Uniform Model levels lying under the root item of the containing window. The horizontal selection may be used for: 1) limitation of the "depth" of the view displayed in the window, 2) removal of some non-interesting and unimportant levels.

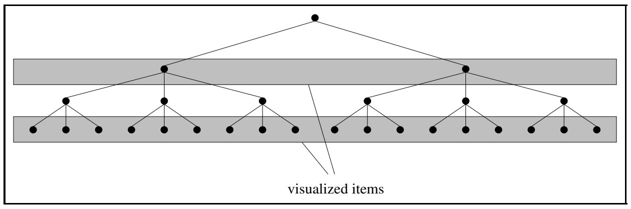

Figure 5: Horizontal selection in a tree

Sometimes, the user may want to focus on selected items of the program. Thus, the monitoring system allows to eliminate of particular visualized items. The vertical selection (Fig. 6) enables choosing and removing undesirable figures from the window. Figures representing descendants of the removed figure are also discarded. A restore operation of all removed figures is also available.

Apart from items' selection, Presentation Layer enables also selection of a set of events non-interesting for the user, that will not be represented in the win-

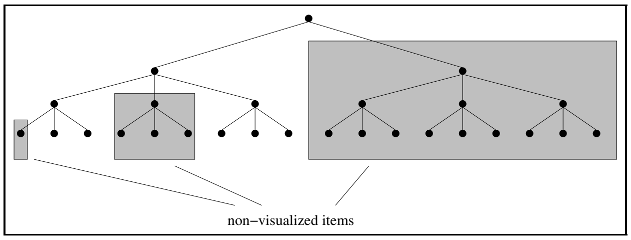

Figure 6: Vertical selection in a tree

dow. The available options are:

- Ignoring all new events – image freezing,
- Ignoring events concerning new items' creation (event create cut-off).

6 Status and Future Directions of the Project

The expandability of MODIMOS architecture has been proved by successful integration of ANSA, SR and Orbix within the Interoperability Layer [10]. This provided the major feedback for the applied technical solutions and proved correctness of this layer's concept. We have implemented a working prototype of MODIMOS with instrumentation for the three environments mentioned above. We harnessed the described 2-D visualization scheme, although we realize that the spectrum of possible solutions spans up to multi-dimensional virtual reality space [17]. In the presence of managed selection mechanisms, such a scheme is sufficient for most of the mentioned application areas.

In the future, we plan to enhance the MODIMOS Presentation Layer with visualization of object interactions, statical program structure (such as e.g. classes and their hierarchies), and the physical location viewpoint.

References

[1] ANSAware 4.0 — Application Programmer's Manual, APM Ltd. Cambridge, 1992.

[2] G.R. Andrews, at el., An overview of the SR language and implementation, *ACM Trans. on Progr. Languages and Systems*, 10(1):51–86, January 1988.

[3] Draft Common Object Request Broker Architecture Revision 1.1, OMG Report 91-12-1, OMG Inc., 1991.

[4] R. Campbell, N. Islam, A Technique for Documenting the Framework of an Object-Oriented System, *Proc. IWOOS'92*, IEEE Computer Society Press, Los Alamitos, CA, September 1992.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1994.

[6] O. Nierstrasz, S. Gibbs, and D. Tsichritzis, Component-Oriented Software Development, *Comm. of the ACM*, 35(9):160–165, September 1992.

[7] J. Udell, Componentware, *Byte* vol. 19 no.5, pp. 46-56, May 1994.

[8] F. Nayeri, B. Hurwitz, F. Manola, Generalizing Dispatching in a Distributed Object System, *Proc. ECOOP'94*, LNCS 821 pp. 450-473, Springer-Verlag, 1994.

[9] Universal Networked Objects, OMG Inc., TC Document 94.9.32, 1994.

[10] A. Uszok, G. Czajkowski, K. Zieliński, Interoperability Gateway Construction for Object-Oriented Distributed Systems, *Proc. of the 6th Nordic Workshop on Programming Environment Research*, Lund, Sweden, TR of the Lund University, June 1994.

[11] G. Czajkowski, A. Uszok, K. Zieliński, Distributed Declarative Systems as Parts of Cooperating Software Environments, *Proc. of the ICLP'94 Post-Conference Workshop on Integration of Declarative Paradigms*, Santa Margerita, Italy, June 1994.

[12] The Orbix Architecture, IONA Technologies Ltd., 1993.

[13] D. F. Jerding and J. T. Stasko, Using Visualization to Foster Object-Oriented Program Understanding, Georgia Institute of Technology, TR GIT-GVU-94-33.

[14] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, Scalable Performance Analysis: The Pablo Performance Analysis Environment, In *Proc. of the Scalable Parallel Libraries Conference*, IEEE Computer Society Press, 1993.

[15] H. Jamrozik, C. Roisin, M. Santana, A Graphical Debugger for Object-Oriented Distributed Programs, *Proc. TOOLS'91*, pp. 117-128, July 1991.

[16] W. De Pauw, R. Helm, D. Kimelman and J. Vlissides, Visualizing the Behavior of Object-Oriented Systems, *Proc. OOPSLA '93*, ACM Press pp. 326-337, SIGPLAN, Washington D.C, October 1993.

[17] J. Y. Vion-Dury and M. Santana, Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems, *Proc. OOPSLA'94*, Portland USA, October 1994.

[18] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim and T. Torzewski, IPS-2: The Second Generation of a Parallel Program Measurement System, *IEEE Transactions on Parallel and Distributed Systems* 1, 2 (April 1990), pp. 206-217.

[19] M. Sloman (ed.), *Network and Distributed Systems Management*, Addison-Wesley, 1994.

[20] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Comm. of the ACM,* vol. 21, pp. 558-565, July 1978.

[21] C. J. Fidge, Logical Time in Distributed Computing Systems, *IEEE Computer*, 24(8) August 1991.

[22] J. Fowler and W. Zwaenepoel, Causal distributed breakpoints, *Proc. 10th ICDCS*, 1990.