An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time System

L. B. Becker [1,2], M. Gergeleit [3], E. Nett [4], C. E. Pereira [1]

[1] Federal University of Rio Grande do Sul, Porto Alegre, Brazil
[2] on leave from University of Cruz Alta, Brazil
[3] German National Research Center for Information Technology (GMD)
[4] Otto-von-Guericke-Universität Magdeburg, Institute for Distributed Systems (IVS)

ABSTRACT

This paper describes a powerful integrated development environment that covers all steps of the development of an object-oriented real-time application from initial design to the final evaluation. The system consists of a set of integrated tools for modeling, simulation, implementation, instrumentation, monitoring, and visualization that all work on the same o-o system model. As all steps of creating a running implementation with visualized monitoring output from the design model are automatic, the environment encourages an iterative approach to the design of complex real-time control applications. This complete environment has been accomplished by integrating the SIMOO-RT modeling, simulation and implementation environment, developed at the Federal University of Rio Grande do Sul, with the Jewel++ object-oriented instrumentation and monitoring tool from GMD, St. Augustin, Germany. The work has been done in the context of the ADOORATA project (A Distributed Object-Oriented Architecture for Real-Time Automation), a Brazilian-German cooperation.

Keywords: modeling, simulation, instrumentation, monitoring, visualization, real-time, object-oriented

1 Introduction

Today industrial automation systems have achieved a high level of complexity that makes them almost intractable for humans. However, next generation industrial automation systems tend to be even more complex, since these systems are expected to be very large, distributed, containing highly dynamic and adaptive behavior, having a long lifetime, and involving complex timing constraints. Therefor, there is a consensus that conventional techniques for developing complex real-time applications are not able to deal with such a complexity, mainly due to their poor abstraction mechanisms. Upcoming techniques like object-orientation are required for dealing with such complex real-time applications. Also, the task of developing modern real-time systems cost-effectively, on time, fulfilling prescribed quality criteria, not only needs the new techniques but also an appropriate system engineering approach. All development phases, from requirements engineering through design, implementation, testing, until maintenance have to be covered and these phases should be handled in a iterative way (spiral life-cycle). Each iteration cycle includes a progression through the same sequence of steps, for each portion of the products and for each of its levels of elaboration, from an overall requirements specification document down to the implementation of each individual program or hardware component. In order to address this challenge and to be able to cover all development phases a consortium of universities, research institutes, and some companies in Brazil and Germany has been set up in joint ADOORATA project. As a first result of this cooperation the SIMOO-RT modeling, simulation and implementation environment, developed at the Federal University of Rio Grande do Sul, has been integrated with the Jewel++ object-oriented instrumentation and monitoring tool from GMD, St. Augustin, Germany. The result is a powerful integrated development environment that covers all steps of the development of an object-oriented real-time application from initial design to the final evaluation. This paper first roughly sketches the functionality of the single components of this environment in section 2. Then, section 3 describes the overall architecture of the integrated environment and the interaction of the components and in section 4, it

gives a summary and an outlook on future work.

2 The Components

2.1 The SIMOO-RT Environment

The SIMOO-RT environment [1, 2] is a result of an ongoing research work that has been developed within the scope of a Master thesis at the Federal University of Rio Grande do Sul. This work extends the original SIMOO [3] environment, that is a framework for development of discrete object-oriented simulation models. The extension provided consists of incorporating specific features that allow the representation of temporal requirements, like deadlines and periodic operations. Also, an automatic code generator for executable code on a real-time operating system has been added. The development of an environment with simulation capacity has been carried on because it is an alternative way of validating the designed model, facilitating the detection of mistakes before the effective implementation of the system.

According to SIMOO-RT methodology, the first step in the development process is the definition of an object-oriented model for the problem under analysis. Temporal restrictions can be imposed to a real-time application at several levels. As our proposed design environment is situated at a higher level of abstraction, it incorporates only modeling restrictions. These restrictions allow the user to specify cyclic operations, e.g. the sensor sample period, and also deadlines for the most critical operations, offering special treatment for those that don't accomplish their deadlines. To avoid a discontinuous transition between the OO simulation model and the generated application, the environment encourages use of state machines to describe the model behavior, as shown in Figure 1 for a tank controlling system. As mentioned before, SIMOO-RT environment also offers support to make automatic code generation for a real-time operating system. The adopted target language is AO/C++. The main features of this language are explained in the next section.

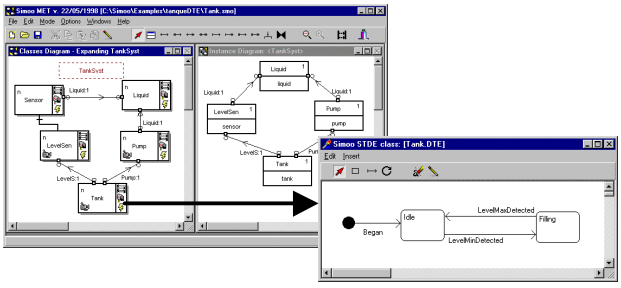

Figure 1: SIMOO-RT Model for a Tank Controlling System.

2.2 The AO/C++ Language

Since the conceptual model adopted in SIMOO-RT is based on the concept of active objects, the use of an object-oriented programming environment supporting the definition of concurrent and distributed objects is desirable for the generation of the executable code for the real entities control program. This would not only ease the task of generating code but also allow a smoothly transition from design to code. AO/C++ [12] has been chosen, since it combines the object-oriented properties of standard C++ with the benefits of the UNIX-RT (e.g. QNX) operating systems. The main idea of AO/C++ is to map the 'logically distributed' model of the object-oriented languages (e.g. C++), with the 'physically distributed' model of the process-oriented UNIX-RT operating systems. AO/C++ adds some primitives to C++ in order to allow the definition of active objects, time-triggered methods, timing constraint specifications (e.g. deadline and its related exception handling code), etc. The number of added primitives to C++ were kept as minimal as possible in order to not heavily modify the usual syntax of C++ programs. The C++ interobject-communication is transparently mapped to the interprocess-communication provided by RT-UNIX operating systems, so that the generated distributed programs are very similar to C++ programs written for sequential, single-machine applications. Both synchronous and asynchronous communication is support by AO/C++. Additionally, all the existing object-oriented features supported by C++ compilers, specially multiple inheritance, polymorphism and overloading are also supported.

```
active class Sensor {
private:
        // reference to other active objects
        Pump REF thePump;
        // or alternatively
    //      Pump * thePump;
        DrvSensor REF theDrvSensor;

 ...// other attributes
        int status;
public:
        void Read(cycle_t){
        begin_cycle
                //Cycle definitions....
        end_cycle
        }
} // end of sensor class
```

Figure 2: AO/C++ code example.

Figure 2 elucidates the language structure through the definition of an 'active' class Sensor, with a cyclic method called 'Read'. Programs written in AO/C++ are parsed by a preprocessor and then converted to a standard C++ code which includes some calls to a specific AO/C++ library.

2.3 The Runtime Environment

As previously mentioned, programs written in AO/C++ are parsed by a preprocessor and then converted to a standard C++ code which includes some calls to an underlying runtime environment. Currently, a code generator for the operating system QNX is available and a version to Linux-RT is under development.

A special constructor is defined for each active class, which generates - at runtime - both a RT-UNIX process as well as a special passive C++ instance. The last has the same interface as the original class but modified methods and should allow a transparent remote method invocation of active class' methods. The real computation of the algorithms specified in the methods of a given AO/C++ active class takes place in the generated RT-UNIX process, which runs concurrently with other processes. The main goal is to keep the AO/C++ code as similar as possible to the normal C++ code. This allows an interesting combination of the characteristics of RT-UNIX processes (concurrent execution, distributed) with those of the C++ runtime system. The latter operates, as usual, on passive instances, allowing overloading, multiple inheritance, and so on.

Regarding in the number of server-threads per object, in the current version each active object (more precisely each active instance of a class) corresponds to a process, i.e. a single-threaded approach is adopted. This can not be considered a limitation, since attributes of AO/C++ active classes are not restricted to be primitive data-types, such as integer, char, but also instances of other active classes are allowed (that means, an active class, like a 'normal' C++ class, can be an aggregate that contains instances of other active classes). Moreover, since the internal active instances encapsulate and protect their internal attributes in case of concurrent access, this kind of intra-object concurrency minimize problems with internal consistency.

2.4 Automatic Code Instrumentation

In many published monitoring systems [8, 10, 11], the required instrumentation has to be inserted manually. In addition, often events have to be named explicitly and this naming has to be done manually as well. This has several drawbacks. Every location in the program, where an interesting event occurs, has to be spotted and a special statement has to be inserted. This requires intimate knowledge of the tested software, not to mention the necessary editing work. Such a procedure of instrumentation is acceptable for a detailed study of stable code, but it becomes impractical as soon as the tool is used in the development cycle of new software. The instrumentation has to be maintained during source changes, it makes the code more difficult to read, and it may become obsolete as soon as the focus of interest changes. This states the demand for a monitoring and instrumentation methodology that can be easily applied to software, that is still under development.

In this case a generic monitoring model helps a lot. Independently of the actual application it specifies a model (typically based on just the structure of the code) that allows to perceive a valuable information about the behavior of the program. A number of such generic models are known from other tools. The well known and still very popular profiling tools *prof* and *gprof*[6] e.g. use a very simple generic model, the call-graph, mainly to present a breakdown of the total execution time. However, as our complete environment is object-oriented, a generic measurement model based on the abstractions of object-orientation, classes, objects, member functions, attributes and their relations seems to be more adequate. Such a model is best supported by an event-based monitoring system. The events needed for the observation of an object-oriented computation have to indicate object creation and deletion, member function invocation and attribute changes. In an object-oriented design, like a system designed with SIMOO-RT, the structure of a program already reflects the overall system model that the developers had in mind when designing the software. Thus, the resulting monitoring model derived from this information will correspond to the software's higher-level structure (Figure 3). This is a desirable state for concurrent performance engineering as this allows to easily identify the critical parts of the design.

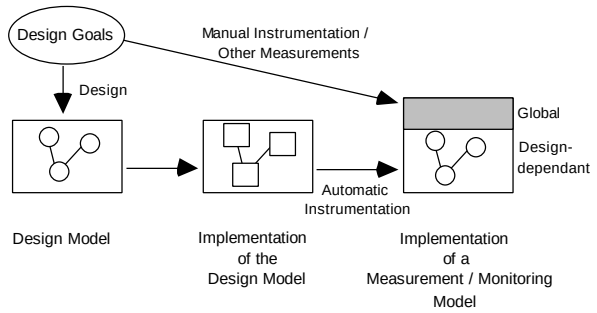

Figure 3: Building a Measurement Model

In our integrated environment instrumentation and event-generation should be highly transparent to the developer. This means, the required instrumentation needed for event generation should be either generic (e.g. as part of the operating system), or applied automatically. In the first case only events are generated when an activity uses the generic object run-time infrastructure, typically when invoking (remote) objects. In the second case instrumentation exploits the information about the structure of the program by inspecting and modifying its sources. In this case software triggers are placed by a compiler-like tool using fixed syntactical rules. Automatic code instrumentation can easily report on the start and the end of method invocations, it can observe value traces of attributes, and, at a very low level, it even allows for a very fine-grained observation of the executions with respect to branches and loops. In order to preserve all information from the o-o environment an event should always contain information about its static context (event-type, class and the method in which it occurred) as well as its dynamic context (object ID, process ID), and a time-stamp.

Fine-grained on-line observation at object (or even intra-object) level requires an event processing system that can handle the high bandwidth of event data. From the experience with the JEWEL modular architecture [7] for collecting and processing events, it became clear that only efficient event selection and filtering mechanisms at an early stage can cope with the potential amount of events. To reduce the amount of events, run- or compile-time filters should be used to discard events if they are (currently) not relevant. This means, whenever it turns out that the fine-grained observation of a certain object (or some of its methods) does not give any additional insight into the behavior of the system, the according events should be filtered out. This might be true either for a whole class (i.e. all object of the same type) or to single instances only. In the first case, compile-time filtering is the most efficient solution, as it allows to place sensors selectively only in those classes and methods that where identified to be relevant for understanding the functional and timing behavior of the system. If the second case applies, run-time filtering is required as it has to be decided on a per object basis, whether the events generated by the same piece of code have to be observed or not.

Compile-time filtering is controlled by the *instrumented* attribute. Classes and methods can be tagged with the instrumented attribute in order to define during compilation time, whether they should contain the additional sensor code or not. Run-time filtering (controlled by a per-object *observed* attribute) allows turning on and off the observation of individual objects at run-time. Currently, it can be only controlled from the class code itself and, of course, it requires that the instrumented attribute of this object it set, i.e. the sensor code is in place. Automatic code instrumentation is performed by a precompiler developed at GMD, called *mc4p* (Martins C Plus Plus Preprocessor) [5, 9]. It translates standard C/C++ (like e.g. the output of the AO/C++ compiler) into the same language again, but adds sensor code. Mc4p's output, the instrumented program, can then be compiled using a standard C/C++ Compiler. This allows for an easy integration into the normal building process of any C++ program with an arbitrary compiler.

2.5 Monitoring with Jewel++

The Jewel++ monitoring tool used in our environment is the object-oriented enhancement of the GMD's measurement and monitoring tool Jewel [7]. Like its predecessor, Jewel++ is event-based and targeted towards a distributed system under test. It is suitable for the observation of real-time applications as it puts emphasis on low-interference, it is capable of using or even creating a system-wide synchronized time-base, and it is able to observe the common system-level events, that are important for any real-time system. In order to achive low interference, the monitor can take advantage of additional monitoring hardware, like an additional processor board (without relying on it as there is a pure software implementation of the monitor available).

However, the original Jewel tool was developed as a completely generic tool with the full freedom (and the need) to configure and adapt it each time to a new system under test. From our experience with Jewel we learned, that the amount of configuration needed to create a new experiment was often prohibitive for doing experiments during the ongoing development of the target system. Especially in the desired iterative development process it is essential that detailed knowledge can be obtained from the current implementation even if it is only in an intermediate state. Therefor, Jewel++, in contrast to the prior system, primarily supports a predefined set of abstractions and event-types. It knows about a certain set of generic system events, that are usually of interest in any kind of real-time system, like thread switches and interrupt services and in addition it understands the application object related events defined by the *mc4p* automatic instrumentation component.

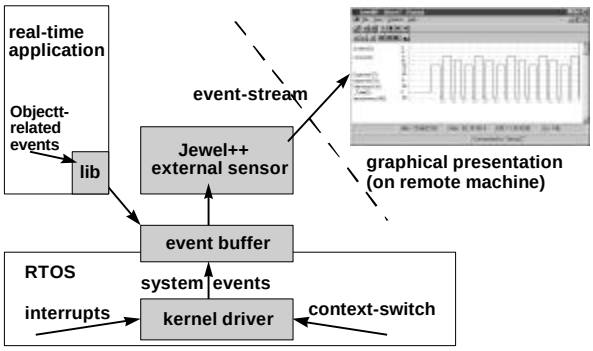

Figure 4: Dataflow in the Jewel++ tool

Jewel++ consists of the three components depicted in Figure 4: an operating system specific driver (first implemented for Windows NT [5] and VxWorks), a remote communication infrastructure (based on RPCs), and a graphical user interface. The driver instruments the OS kernel in order to intercept all thread switches and interrupt handlers. It generates event-descriptions about these system events and stores them with a high-accuracy time-stamp (on an Intel Pentium e.g. provided by the internal tick-counter (resolution < 10 ns)) in one global shared memory buffer per node. Event generation and enqueuing itself introduces an overhead of about 5 µs/event (on a 233 Mhz Pentium machine). By using a library provided with Jewel++ the automatic instrumentation tool *mc4p* also instruments the application code to write the object-related events into the same event-buffer. The remote communication infrastructure (in Figure 4 the „external sensor“ component), a per node low priority process with access to a network, allows to collect the local event-streams from multiple nodes of a distributed system and to forward them to one experiment control node. This can be done off-line after the system terminated or on-line if either network scheduling allows for it or a separate monitoring network is available. On the experiment control node the experimenter can manage the experiments remotely only using one graphical user interface. This interface allows selecting the interesting events, to start and stop monitoring on a selected set of nodes, to initiate data transfer to the central node and to analyze the measured data in detail. The analysis view displays these event stream either as text or as graphically as a Gantt-chart with variable zooming facility (see Figure 5). All thread switches, interrupts and the begin and the exit of object invocations are shown over the time axis. This is a very fine-grained view of the real-time behavior but still very intuitive for the software developer as it still reflects the o-o model used for designing the system.

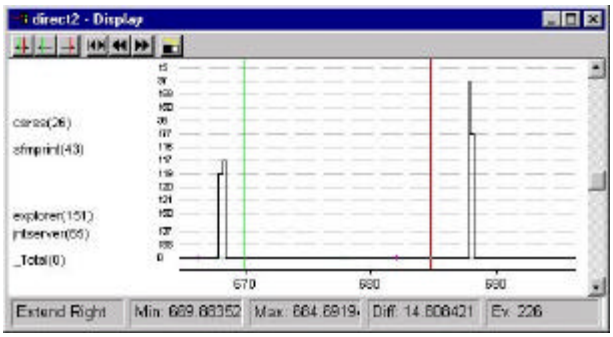

Figure 5: Graphical display of the measured task-switches one node

If not provided by the real-time system itself, the graphical user interface also allows reconstruct a global time base and to merge the local event streams into one global order. This is done assuming that the different local high-resolution clocks have a different offset and a constant drift (i.e. a linear relation) and using the knowledge that certain events occur at nearly the same time on all nodes (e.g. network receive interrupts of a certain packet on a broadcast media, like e.g. the CAN bus or a “short” Ethernet). By selecting two of these events out of any two event streams that have to be merged one can easily compute a linear “correction” function that maps the timestamps of these events from one time-scale to the other. Now this function is applied to all timestamps of one event stream and the two formerly local event streams can be merged in one global order.

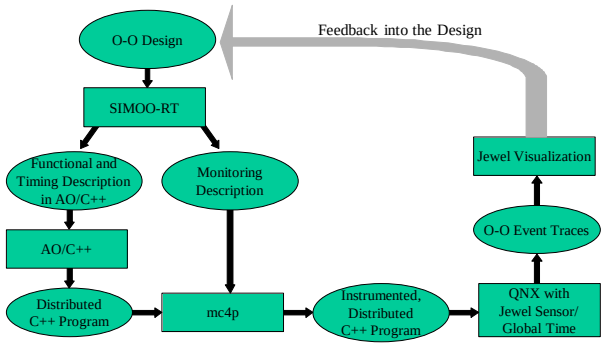

Figure 6: Interaction of the components

3 Tool Integration

So far the functionality of the single components of the system has been described. Each of these tools is valuable for the development of a distributed real-time system, but only the integration of these tools into one environment fully enables an iterative design of an object-oriented real-time application. The interaction of the components within the integrated environment is

depicted in Figure 6. Each cycle of the development starts with an object-oriented design expressed by the means of the SIMOO-RT graphical modeling language. Here, the class and the object structure, the functional code of the methods, as well as the timing of the real-time application are defined. In addition, in our environment a developer also defines the corresponding monitoring model for the test phase by defining the instrumentation status of the application's classes. This is done integrated in the GUI concurrently to the other design activities. Figure 7 shows the according switches in the SIMOO-RT class forms where the designer can select the instrumentation status of complete classes, single methods and even single attributes. The output of the SIMOO-RT tool finally are two files: a functional and timing description written in AO/C++ and a monitoring definition file containing the *instrumented* attributes of the classes. This separation in two files simplifies tool integration as it allows to leave the AO/C++ compiler and its output completely unchanged and also to skip the instrumentation step easily when no monitoring is required (e.g. if only functionality has to be tested).

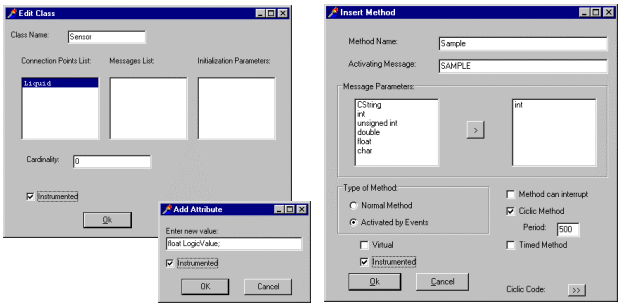

Figure 7: Specification of the instrumented attribute in SIMOO-RT.

All following build steps are now automatic and they can be hidden from the developer by a simple make-script. The AO/C++ code is translated into distributed C++ code (client, server and stubs code) and these output files are then instrumented by running the *mc4p* precompiler on them. Finally, the output of this step is compiled using a standard C++ compiler and linked against a runtime library that contains the primitives of the task management and the remote procedure call invocation of AO/C++ as well as the Jewel++ sensor functionality. The result of this build process is the ready-to-run application that now can be executed in the target environment. A test run will automatically create per node event-traces that are collected and finally visualized be the Jewel++ tool. The graphical monitoring output depicts the timing behavior of the application and it directly corresponds to the abstractions of the object-oriented design. Therefor, it gives the direct feedback for the developer whether his/her design can meet the timing requirements in the observed test cases and it can lead to a new refined design. Since the design as well as the monitoring are controlled solely from central GUI applications and all intermediate steps are completely automatic, the integrated environment supports quick turn-around times and a rapid development of object-oriented real-time applications.

4 Conclusions and Future Work

This work intends to provide computational support for the ADOORATA project. It covers the complete development cycle of real-time object-oriented applications. These has been achieved by the integration of two different tools: SIMOO-RT and mc4p/Jewel++. The first tool is used to help the application development process, including modeling, simulation and code generation. The second one provides means to instrument and monitor the application and analyze its timing behavior. Future work within the ADOORATA project will try to adapt the design and monitoring model to upcoming o-o design standards, to provide a set of non-toy application examples and to extend the base of supported real-time OS platforms.

Through the comparison of the developed environment with other gender tools (see, for instance, [1]), it's noticed that they all incorporate an object-oriented methodology with own characteristics. It happens that, in spite of the advantages and disadvantages of each methodology, none of the other tools have all the desired features for real-time systems development. Within this context, it is also noticed that the ideal environment "mixes" the best concepts of each tool. This done in the proposed environment, as it makes use of different methodologies. I allows for the integrated development of "requirements engineering", "simulation", "executable code generation" and "testing and monitoring". Thus, the SIMOO-RT/Jewel++ combination seeks to supply great part of the lacks found in current commercial tools.

5 Acknowledgments

This work has been partly supported by the CNPq-DLR cooperation program between Brazil and Germany. The Brazilian part of the project has also been partly supported by the research agencies CNPq and FAPERGS.

Thanks are also given to Bernardo Copstein and Flavio Wagner, members of the SIMOO-RT project for

their valuable contribution to the work described in this paper.

6 References

[1] Becker, L. and Pereira, C. SIMOO-RT: An Integrated Object-Oriented Environment for the Development of Distributed Real-Time Systems. Paper submitted to: EUROMICRO'99. York, England, June 1999.

[2] Becker, L, Pardi Jr, W. and Pereira, C. Proposal of an Integrated Object-Oriented Environment for the Design of Supervisory Software for Real-Time Industrial Automation Systems. In: Proc. of WOORDS'99, IEEE Workshop on Object-Oriented Dependable Systems, Santa Barbara, USA, Jan. 1999.

[3] Copstein, B. ,Wagner, F. and Pereira, C. SIMOO - An Environment for the Object-Oriented Discrete Simulation. In: ESS'97 - 9th European Simulation Symposium. Passau, Germany, October 1997. Proceedings, Society for Computer Simulation, 1997.

[4] M. Gergeleit. Automatic Instrumentation of Object-Oriented Programs, Workshop on Visualization and Instrumentation, part of the 8th IEEE Symposium on Parallel and Distributed Processing, New Orleans, Louisiana, 1996.

[5] M. Gergeleit, M. Mock. Real-Time Monitoring of the EIVIS Distributed Video-Server on Windows NT, 18th IEEE Real-Time Systems Symposium, Work in Progress, (RTSS-97), San Francisco, CA, Dec 1997.

[6] S. L. Graham, P. B. Kessler, and M. K. McKusick: gprof: a call graph execution profiler, Proc. SIGPLAN'82 Symp. Compiler Construction, 1982, pp. 120-126

[7] F. Lange, R. Kroeger, M. Gergeleit: JEWEL: Design and Implementation of a Distributed Measurement System, IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 6, November 1992, pp. 657-671

[8] B. P. Miller at al.: IPS-2: The Second Generation of a Parallel Program Measurement System, IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 206-217

[9] E. Nett, M. Gergeleit, and M. Mock: An Adaptive Approach to Object-Oriented Real-Time Computing, Proceedings of ISORC'98, 20-22 April 1998 in Kyoto, Japan

[10] D. M. Ogle, K. Schwan, R. Snodgrass: Application-Deppendent Dynamic Monitoring of Distributed and Parallel Systems, IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 7, July 1993, pp. 762-778]

[11] S. Perl, W. Weihl: Performance Assertion Checking, Proceedings of the 14th ACM Symp. on Operating System Principles in Operating Systems Review, Vol. 27, No. 5, Dec. 1993, pp. 134-145

[12] Pereira, C. Real Time Active Objects in C++/Real-Time UNIX. In: Proc. of ACM SIGPLAN Workshop on Languages, Compiler, and Tool Support for Real-Time Systems. Orlando, EUA. 1994.