Dynamic Guarantee in Component-Based Distributed Real-Time Systems

Cssia Yuri Tatibana *Romulo Silva de Oliveira* Carlos Montez
Departamento de Automao e Sistemas
Universidade Federal de Santa Catarina
{cytatiba,romulo,montez}@das.ufsc.br

Abstract

This work describes the proposal of a set of mechanisms to be applied in distributed component-based systems to allow the development of real-time applications capable of offering dynamic guarantee for clients. The proposal aims to make the application server able to provide service completion according to all clients imposed timing constraints or the immediate request rejection. To implement these mechanisms, we use an acceptance test at bind time to filter services requests. The acceptance test determines which service request will be executed, based on the system capacity and current load, providing therefore, dynamic guarantee for all the provided services.

1 Introduction

The provisioning of efficient software solutions requires engineering effort to cope with the increasing complexity of this kind of product while still shortening software development cycles and reducing costs.

Reusability is the component technology key factor which allows systems faster evolution by alleviating the maintainability, upgrade and extension process [4, 13]. This technology allows applications to be integrated from components previously developed. Those components that perform successfully in one application can then be reused. The explicit separation of application specific logic and software component common aspects (non-functional part) allows the separation of roles played by different experts while improving software reusability. In software component frameworks the non-functional code is automatically generated according to the component configuration specified through meta-data. One component could be used in different applications providing the same functionality but with different configuration.

*Supported by CNPq.

Component standards specify widely-accepted interfaces that allow independent components from different suppliers (third parties) to be plugged together and even to interoperate across language, compiler, and platform barriers [13]. Well known component standards are OMG CORBA Component Model (CCM) [3], Microsoft COM+ [1] and Sun Microsystems Enterprise Java Beans (EJB) [4].

Component-based models deal successfully with functional attributes, but provide insufficient support for real-time services. Services timeliness and predictability, main issues for real-time systems are not addressed by these models [13].

The separation of concerns provided by component models potentially allows real-time behavior to be configured without affecting the component specific functionality. Therefore, the same component could provide different levels of real-time services according to the application requirements or system resources constraints. As well as the non-functional part of the component, real-time behavior code could as well be generated from the configuration specified by meta-data. Real-time distributed applications become increasingly widespread in all fields, ranging from multimedia to avionic systems.

Real-time component-based development is a relatively new research area. Many of the works [15, 19] in this area focus only the early development steps: planning and designing the component architecture as a mean to ensure better predictability from the resulting system. However, most of these projects do not address interoperability issues or runtime aspects and require specific knowledge and tools for the development of applications based on these components. Others, as described in [2] and [11] present components designed for embedded applications that do not require adaptability and executes in static environments.

Most of the component based approaches designed for embedded real-time systems described in the literature

provides good solutions concerning the application real-time behavior in the target platform. Some of them are capable of offering hard-real time guarantees. However, they are tightly coupled to the underlying platform and demand static and known a priori environment conditions (workload and real-time constraints).

Real-time component-based distributed systems, on the other hand, are usually designed for large scale application in which the environment condition is not known a priori and therefore requires some adaptability aspect from the application. The design of this class of application involves issues like interoperability across different platforms, delays from network communication and the capability to adapt to changes of operational conditions since the workload and resource availability can not be characterized accurately a priori. Given the complexity of such systems the current real-time component-based works addressing distributed environments are usually based on the extension of general purpose existing technology to better suit real-time applications. Therefore, the best effort approach is the one usually found in such applications concerning scheduling issues.

This work proposes real-time dynamic guarantee for open real-time component-based distributed systems. The approach is based in mechanisms capable to enforce real-time behavior while still maintaining software component characteristic properties: opaqueness, composability and isolation [13].

This proposal involves the extension of programming and execution models of component-based application servers to guarantee clients timing constraints or to immediately reject the request. The acceptance test is applied at bind time, when a client requests a time service guarantee for the periodic invocation of a certain component method. The clients request is filtered through an acceptance test in the server side. The test verifies which request will be executed based on the server capacity and the current workload, providing this way, dynamic guarantee for all offered services. This work addresses the issues involved in the development of applications such as supervisory systems, which requires real-time constraints, but do not demand hard real-time guarantees. A previous work regarding this proposal is presented in [10].

In the next section it is presented the characteristics of the domain being treated. In section 3 the proposal is described: the deadline partitioning and scheduling algorithms used in the architecture are presented in subsections. In section 4 related works are briefly described and finally, the conclusions are presented in section 5.

2 Dynamic Guarantee in Open Distributed Systems

In component based real-time systems domain, most of the works described in the literature provide static or best-effort guarantee (figure 1). In this work, when the system endpoints and communication infrastructure are modeled in the application development process, the system is referred as a closed system. Closed systems include applications that have resources and workload known a priori, the number of clients is limited and the system is usually static. Although these properties allow the provisioning of static guarantees, they are too restrictive. Cadena [7], described in section 5, is one example of modeling tool that addresses the development of such systems.

In the context of this paper, open systems are the ones that allow new clients to connect to the server at any moment. In these systems, the workload cannot be predicted, and the system has to manage the resources distribution at runtime. As there is no pre-runtime workload knowledge, most of the open system are only capable to offer a best effort behavior. CIAO, described in [12] is one example of support to develop such systems.

Dynamic guarantee is an alternative to enforce real-time behavior in open systems. Real-time schedulability analysis applied at bind time can keep one application server from dealing with situations in which an excessive number of clients cause the server to fail to accomplish deadlines. As the schedulability test relies on the operating system capabilities, communication infrastructure and the accuracy of component execution time information as well, it can not be detached from these issues, and therefore can be as accurate and effective as these parameters allow it to be.

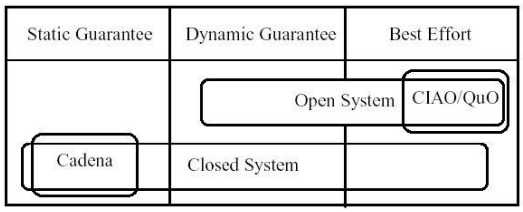

Figure 1. Scheduling approaches.

The objective of this work is to present an extension to the programming model of component-based systems to provide dynamic guarantee for real-time distributed systems according to an open system specification; it considers the timing behavior of the server side only and allows new clients to be included at runtime. This proposal is composed by the combination of a set of mechanisms that allows an acceptance test to be applied at each service invocation to a real-time server. The mechanisms used are

based on clients real-time constraints and the server current capacity.

Although client-server communication is not part of the problem being addressed, it is assumed a server composed by a set of components distributed through different system nodes. Then the predictability problem in a distributed environment is considered. The delay caused by the network communication is accounted in each component method execution time. However, the application timing behavior cannot be isolated from the underlying layers, so it is assumed a bounded delay and synchronized clocks.

3 Proposed Extension

The implementation of dynamic guarantee in any system demands some sort of translation of the system into tasks with timing constraints that can be submitted to scheduling analysis.

In this work, it is proposed mechanisms to deal with the timing behavior of components in three moments: (i) the translation of server-provided services into component method sequences at deployment time, (ii) the specification of timing behavior at bind time and (iii) the accomplishment of real-time constraints at runtime.

In order to guarantee or at least to select service requests with better chances to be completed according to their timing constraint, an acceptance test must take into account static and dynamic properties of servers. At deployment time, information concerning inter-component connections, resources demand and also static and dynamic properties are exchanged.

In the proposed model, the server may reject a client request, the decision regarding the requests acceptance or rejection must happen before the client-server connection is established. The client can then, try to connect to another server or relax its real-time constraints. In this paper context, the bind request is the way a component container receives the clients service request. As bind operations between client and server must happen before any service is effectively executed for the client, the bind time is the adequate time to apply the acceptance test.

The acceptance test applied to each service request is accomplished by scheduling analysis applied to each component involved in providing the requested service.

In the next subsections the three main architecture aspects will be described: the mapping of services into method sequences, the execution flow at bind time and the protocol between containers.

3.1 Services Mapping

In the server, services are sequences of component methods, periodically invoked. As the input of a scheduling analysis is a set of tasks that have known execution time, deadline and period, the mapping of services into method sequences is used to reach better determinism (with respect to timing behavior of server components). The description of component inter-connections and sequences of method executions must be set at design time and specified in a server component deployment descriptor. The set of component containers that participate in one service must exchange information concerning the execution time attributed to each method to be executed as part of this service. At runtime, each method execution is associated with a task.

Each task has its deadline defined by a deadline partitioning algorithm applied to the service request deadline defined by the client. The acceptance test efficiency in the context of this work depends on a good deadline partitioning algorithm, made possible through the service mapping.

The mapping process requires the knowledge of the worst-case execution time of each method. The execution time of each method sequence is acquired at deployment time through configuration methods implemented by the container.

The description of component inter-connections and sequences of method executions must be set at design-time and specified in a component deployment descriptor.

In figure 2, it is represented a set of containers (K_1, K_2, K_3) and components (C_1, C_2, C_3) that implements method sequences described in the rectangles below each container. The arrows connecting the three components methods indicates the method sequence described in the first line of each rectangle and that composes the service initiated by the invocation of method $C_1.m_3$. Each container lists only the direct successor method, which is unique and predefined.

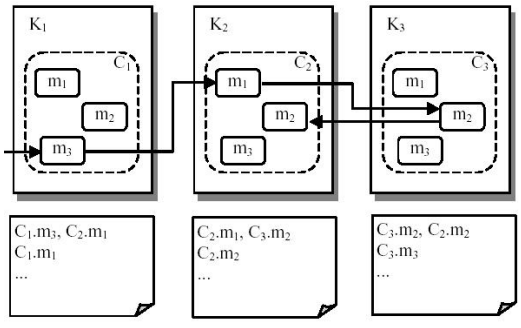

Figure 2. Server service mapping.

As shown in figure 2, the mapping in K_1 indicates that when the method $C_1.m_3$ is executed, the subsequent invocation is to $C_2.m_1$. So, the sequence of method executions started by a service request is determined by the first

server component method invocation. However, a subsequence can be reached by many different methods. In figure 2, the subsequence $C_2.m_1$, $C_3.m_2$ (as indicated by the mapping of component C_2) can be reached from components C_1 or C_3.

Every container table encloses the worst-case execution time values of its component methods. When these methods have a successor method, its execution time is also listed in the container table. This value may indicate one method execution time (when it is the last method to be executed) or it may indicate the sum of a sequence of successor methods. The execution time of the remaining method sequence is attributed to the next successor method only for deadline partitioning purposes. This attribution of execution times in each container table allows a better division of slack time among the methods of the sequence.

The container K_1 table (figure 2) lists the execution time of $C_1.m_3$ and the remainder of the service execution time (sum of $C_2.m_1$, $C_3.m_2$ and $C_2.m_2$ execution times), attributed to $C_2.m_1$. In K_2, the table indicates that the service started by $C_1.m_3$ is composed by the sequence $C_2.m_1$ and $C_3.m_2$.

Although a server may be deployed as an assembly, it should be allowed the posterior addition of new components to the server. When a new component is added to the server, all the other components, directly or indirectly related to it must update its execution time information. Also, the new component container must obtain the information to allow the mapping of the services which it makes part. This information exchange among containers must be made at the new component deployment time, before it is made available for execution.

3.1.1 Deadline Partitioning Algorithm

As each component may be hosted in different nodes of a system, the methods to be executed as part of one service do not share the same resources. The scheduling analysis of each task is bounded to the node resources and dependent on its workload, so the service imposed real-time constraints must be translated to each task. The acceptance test must be independent in each node of the system. The arrival time of a task is given by the deadline of the predecessor task (time-driven model [9]) and the period is the same as attributed for the service, but the task deadline must be partitioned among the tasks involved.

To illustrate the steps for this approach in this work, it was used the model proposed by [8] for deadline partitioning. The scheduling policy adopted for each involved processor is EDF-Earliest Deadline First [9]. Different algorithms for scheduling and deadline partitioning may be applied according to the underlying operating system being used.

In our approach we applied the EQF -Equal Flexibility strategy, which divides the remaining subtasks slack time proportionally to these tasks execution time providing equal flexibility to all the subtasks of the service. The task flexibility is defined by its slack time and execution time ratio.

The acceptance test is based on the scheduling analysis of the task set composed by previously accepted tasks and the task under test. Once the new set of task can be scheduled, the new task is accepted. If this new task set cannot be scheduled, the new task is rejected. This algorithm ensures the real-time constraints of previously accepted tasks and do not allow new tasks to the task set if that harms the guarantee already provided. It will protect the system from accepting more connections than it can actually process.

The application behavior or the acceptance test efficiency depends on the quality of the partitioning deadline algorithm and the underlying software layers timing behavior. The acceptance test result reflects the guarantees that the local operating system and the communication mechanism used are capable to offer. If deterministic operating system and communication mechanisms are used the system allow the proposed architecture to provide dynamic guarantee. On the other hand, if the operating system and communication support have probabilistic timing behavior, the architecture will only be able to offer an acceptance test which will need an overload treatment.

3.2 Acceptance Test at Bind Time

In order to apply the acceptance test before the inclusion of new clients in the system, each bind request must be processed in two steps. Once the bind request arrives at the server, it is put in a service request queue for further analysis and test. The acceptance test works as a filter of bind requests at server-side. The timing constraints used as bind request parameters are related to the periodic component method invocation to be made after the bind request acceptance, and not to the bind request processing itself. No timing constraints are associated with the bind request processing by the server. The set of timing constraints required by the service request must contain at least the period, relative deadline and the maximum time interval the service will be used by the client. Once this interval of time is expired, the resources reserved for the client are released.

At bind time, a method invocation in a container triggers a sequence of steps. The request manager (figure 3) intercepts the request and (i) verifies if the specified deadline is shorter than the sum of the method sequence execution times and needed communication times, (ii) partitions the requested deadline, allocates local processor and (iii)

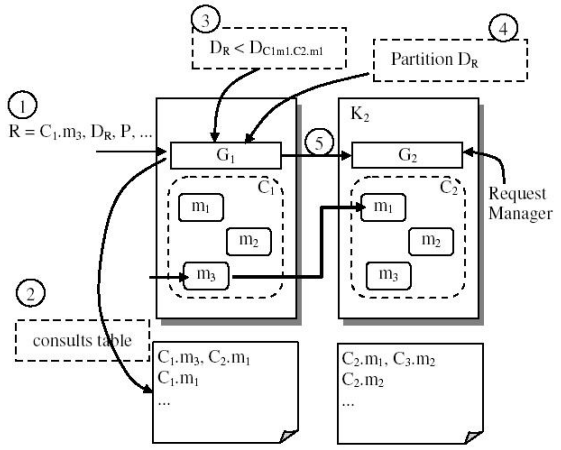

Figure 3. Acceptance Test.

sends the request (period and time interval specified by the client), together with the deadline, to the component container of the successor method to be executed.

This sequence of steps (deadline partitioning, local resource reservation and acceptance test request to other containers) is recursive among server containers. It finishes when the last container that participates in the service is reached.

These steps are enumerated in figure 3: in step 1 the request arrives at the container K_1, it informs the required method, imposed deadline (D_R), period (P) and length during which the service will be needed. In 2, the service mapping table is consulted. In step 3 the container verifies if the required deadline is higher than the method sequence ($D_{C1.m3,C2.m1}$) worst-case execution time. In the affirmative case, the deadline partitioning algorithm is applied to the method sequence started by $C_1.m_3$. In step 5, the container invokes the successor method acceptance test with the parameters calculated by the deadline partitioning algorithm. The recursion used in this sequence of actions allows the deadline partitioning algorithm to occur in each container and, therefore, to take into consideration the time spent by inter-component communication.

As soon as one of the containers (K_1, K_2 or K_3) involved in the service provisioning reject the task ($C_1.m_3$, $C_2.m_1$, $C_3.m_2$ or $C_2.m_2$) in the acceptance test, the service request of this client is rejected. For the method sequence used in this example, if the task that executes $C_2.m_2$ is rejected, the one that executes $C_3.m_2$ is also rejected.

After the service deadline partitioning, the component container K_1 applies the acceptance test to task $C_1.m_2$. If it is accepted, K_1 requests the allocation of processor time while waits for K_2 acceptance test result. If K_1 acceptance test rejects the $C_1.m_3$ execution, this client bind request is rejected no matter what is the result from the test applied in K_2 or K_3. So, the acceptance test does not necessarily need to be completed in all the containers involved in the service.

As the acceptance test demands processor time as well, it must be modeled as a container task that also makes part of the local schedule. It is modeled as a container periodic task that receives the bind requests that arrives at the components interfaces. This periodic task has one processor time slice defined by the application developer. The time reserved for system acceptance tests is bounded by this tasks, the system will not be indefinitely executing acceptance tests.

3.3 Protocol between Containers

The tasks (method execution) that are part of a service provisioning are only taken into account in the system load after they have already been approved by the acceptance test. Before the bind request approval (approval for all tasks of the service) no processor time is allocated for these tasks. From the bind request arrival moment to the confirmation from all the other participant containers, there is a time interval. During this time, the bind request under analysis is put into a waiting queue managed by the container. When all the service participant containers approve their respective tasks, the reservation in each local processor can be confirmed. Then, the service is said to be confirmed.

When an acceptance test invocation R_2 arrives at a container, it is put in a waiting queue until the current request under analysis by this container (assume the test for invocation $R1$) finishes. Only then, the test is applied to R_2. The R_2 acceptance test considers the resource reservation already made for $R1$, even if the service requested by R_1 has not yet been confirmed.

This resource reservation may be confirmed when the service is confirmed or may be undone if one of the participant containers rejects its task (part of the service request R_1).

This procedure does not avoid the situation in which two requests (R_1 and R_2) are rejected even when the server has enough resources to schedule one of them. It is possible for request R_1 to make the resource reservation in K_1 and then be rejected when requests one service in K_2, which has previously reserved resources for R_2. And that R_2 is also rejected when request one service in K_2, because it has already reserved resources for R_1. More sophisticated protocols may be developed in the future to cope with this sort of problems and to make the whole mechanism more efficient.

Considering the scenario in figure 2, at the moment the container K_1 (the container which received the client request) receives the approval of K_2, all the tasks that com-

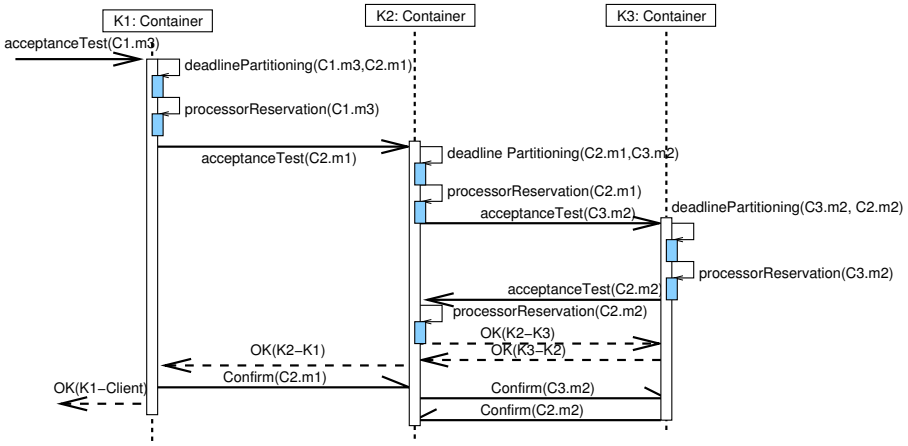

Figure 4. Execution flow: bind request accepted.

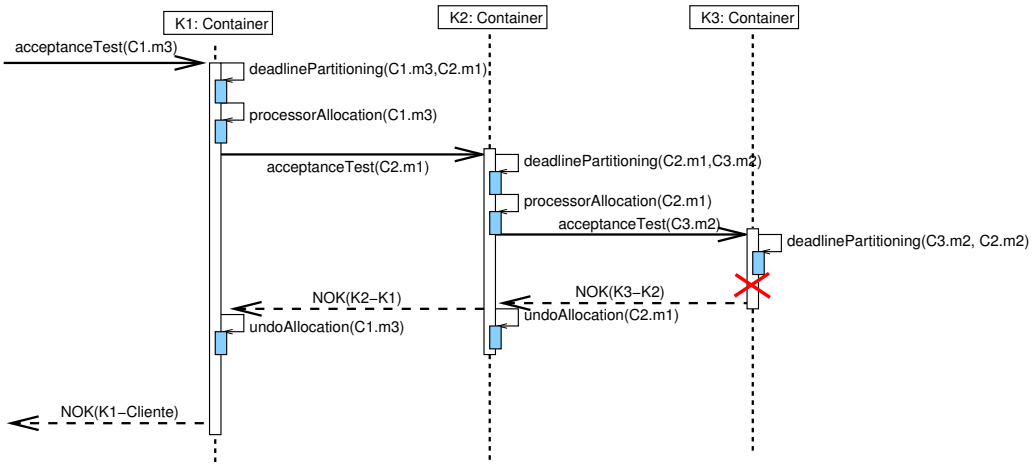

Figure 5. Execution flow: rejected bind request.

pose the service are included in the schedule of each processor and the client is informed about the service acceptance.

In figure 4, the execution flow resulting from the bind request of method $C_1.m_3$ are illustrated. Soon after the request arrival, container K_1 applies the deadline partitioning algorithm to establish the deadlines of the tasks that compose the service. It sends an acceptance test request with all the adjusted parameters to the next container involved in this service. Container K_1 applies the acceptance test to the task that will execute $C_1.m_3$ considering the deadline established by the deadline partitioning algorithm. The local processor time reservation concludes the test procedure. When the bind request is sent to container K_2 through the acceptance test of the task related to $C_2.m_1$, the same steps described for K_1 are repeated.

Figure 5 illustrates the same bind request processing scenario with the rejection of the task that executes $C_3.m_2$. In this case, the rejection that occurred in K_3 causes the test in K_2 to be suspended and the service request is rejected. The containers then release the processor reservations that were already done for this service request.

This work assumes that there are no communication faults. However, fault treatment in distributed systems with synchronized clocks is well known and may be included in this architecture with no great difficulties.

Although the examples presented here assume that each method invokes at most one method only, the architecture admits that a method may call many other methods, since the invocation flow is known at the moment the bind request is evaluated.

4 Related works

Recently, real-time and component-based systems have become the theme of many research projects. The separation of concerns; application logic and common aspects can potentially leverage the complexity inherent to those kinds of systems by detaching application functionality from required real-time behavior. As a result real-time applications need no longer to be tightly coupled to underlying platform; the application real-time behavior is configured instead of hard coded. And the resulting components from this development approach can be reused in different compositions and timing configurations.

Undoubtedly, this combination results in a huge impact in real-time systems development. The direct result is systems growing in scale, robustness and accuracy as their development cycles are shortened through component reuse. However, the same high abstraction level that provides such a powerful approach also demands appropriated tools and platform to support the system development. Otherwise, the system development can be excessively complex and error prone.

The Component Integrated ACE (Adaptive Common Environment) ORB [12], CIAO, is a QoS enabled CORBA 3.X CCM implementation built atop TAO, The ACE ORB. TAO provides support for some of the features of real-time CORBA 1.0 specification plus many real-time scheduling strategies. CIAO applies a range of aspect-oriented development techniques to support separation and composition of real-time behaviors and other configuration concerns that must be consistent throughout a system [12]. Currently, CIAO latest release implements most of OMG CCM. Both, TAO and CIAO are freely available.

CoSMIC [6] is a set of tools for supporting model-integrated computing (MIC) and OMG Model-Driven Architecture (MDA). These tools allow the graphical modeling of component-based applications generating the application metadata used to configure and deploy the modeled application as well as the non-functional code.

Cadena [7] is a development and architecture analysis environment for CCM [3] components of Boeing avionic systems. The application development process is supported by OpenCCM [12]. Cadena works as a layer on top of OpenCCM, adding forms to the applications IDLs under development. These forms allow the specification of timing behavior, inter and intra-components dependencies and distribution of components through system nodes. Cadena translates the designed components, the inserted forms and correction references into an input model for system timing behavior analyzer [5].

However, components built in Cadena are CCM event-triggered components and the physical infrastructure must be known a priori.

Quality Objects (QuO) [14] is a framework for providing quality of service (QoS) in network-centric distributed applications developed by BBN. QuO can be used in conjunction to CIAO, as encapsulated units, to provide dynamic QoS and adaptability. It allows QoS constraints specification, system monitoring and adaptation to requirement changes at runtime.

In [13], an admission control for real-time component-based systems mechanism is presented. This work base the admission control in arrival functions that limits the acceptance of clients based on predefined system CPU utilization. Although the approach assumes components spread over a network, the admission control is centralized. Also, the network delay caused by communication between nodes is not accounted by the approach or modeled in the scheduling analysis.

5 Conclusions

This work presented the mechanisms used to extend the programming and execution model of distributed component-based architecture aiming dynamic guarantee provisioning in real-time systems. The mechanisms provide a flexible architecture that is relatively easy to modify regarding the policies adopted for deadline partitioning and acceptance test.

The acceptance test guarantees the future periodic invocation of methods through bind-time analysis. The test is based on server tasks execution time and considers the system workload.

This work next step is the implementation of such mechanisms in CIAO containers.

Acknowledgment: This work is supported in part by CNPq - The Brazilian National Counsel of Technological and Scientific Development.

References

[1] Microsoft component object model com. http://www.microsoft.com/com/.

[2] Pecos project. http://www.pecos-project.org/.

[3] Corba component model. http://www.omg.org/technology/documents/formal/components.htm, June 2001.

[4] Sun microsystems, java remote method invocation specification. http://java.sun.com/j2se/1.3/docs/guide/rmi, 2005.

[5] C. Demartini, R. Iosif, and R. Sisto. dspin: A dynamic extension of spin. In *In Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, September 1999.

[6] A. Gokhale, B. Natarjan, D. C. Schmidt, A. Nechypurenko, N. Wang, J. Gray, S. Neema, T. Bapty, and J. Parsons. Cosmic: An mda generative tool for distributed real-time and embdedded component middleware and applications. In *in Proceedings of OOPSLA 2002 Workshop on*

Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November 2002.

[7] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *ICSE 2003*, Oregon, 2003.

[8] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8:1268–1274, December 1997. No 12.

[9] C.-L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. volume 20, pages 46–61, New York, NY, USA, January 1973.

[10] C. Y. Tatibana, R. S. de Oliveira, and C. B. Montez. Scheduling approaches for component-based real-time distributed applications. In *Workshop on Quality of Service for Application Servers*, Florianpolis - SC, Outubro 2004.

[11] R. van Ommering, F. van der Linden, and J. Kramer. The koala component model for consumer eletronics software. *IEEE Computer*, March 2000.

[12] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring real-time aspects in component middleware. In *Proceedings of the Conference on Distributed Objects and Applications*, Cyprus, Greece, October 2004.

[13] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 428–437, March 2005.

[14] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1), 1997.