A FRAMEWORK FOR DEVELOPING DISTRIBUTED HARD REAL-TIME APPLICATIONS

J. Javier Gutiérrez García, and Michael González Harbour

Departamento de Electrónica y Computadores, Universidad de Cantabria
39005-Santander, SPAIN
e-mail: {gutierjj,mgh}@ctr.unican.es

Abstract: This paper presents a prototyping methodology for the development in Ada 95 of distributed applications with hard real-time requirements. Based upon an architecture-independent description of the system's elementary actions and their exchange of events, and given the system's configuration, real-time analysis of the distributed application can be performed. In addition, using the same system description an executable prototype of the application is created, which enables testing its timing behavior, and, if the actual code for the elementary actions is provided, automatically generating the final code of the application. *Copyright @ 2000 IFAC*

Keywords: Real-Time Systems, Distributed Models, Prototyping, Queues

1. INTRODUCTION[1]

In multiprocessor and distributed systems, the processes or tasks of an application need to cooperate amongst themselves in order to synchronize or exchange information. Therefore, if we try to develop hard real-time applications based on multiprocessor and distributed systems, we need to use networks capable of guaranteeing hard real-time communication. Many conventional interfaces for process or task intercommunication are inadequate for use in real-time applications, because the absence of priorities does not allow the transmission of messages with different degrees of urgency.

In the real-time extensions for the portable operating system interface standard (POSIX, 1996), a message queue interface is defined for communication among real-time processes using a priority-based message-passing mechanism. Although this interface was initially conceived for local communication in non-distributed systems, in a previous work we demonstrated that it could be extended for use in hard real-time distributed systems (Gutiérrez and González, 1996), mainly due to the existence of a priority associated to each one of the messages, which enables the use of a real-time scheduling policy in the communications network.

To make possible the prediction of worst-case response times in hard real-time distributed systems scheduled with fixed-priority methods, schedulability analysis techniques have been developed based on rate monotonic analysis, RMA (Liu and Layland, 1973). These techniques model each network as if it were a processor and each message as if it were a task (Gutiérrez et al., 2000; Klein et al., 1993; Palencia and González, 1998; Tindell and Clark, 1994). In

[1]This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

addition, techniques have been developed to find a feasible assignment of priorities that makes the system schedulable (Gutiérrez and González, 1995; Tindell et al., 1992). In distributed systems, the tasks and messages can undergo a delayed activation effect or *jitter* which has a negative influence on the worst-case response times of all lower priority tasks or messages. It can be eliminated through the use of the sporadic server scheduling algorithm both in the processors and in the communication networks, achieving increases of up to 50% in the resource utilization (Gutiérrez and González, 1996).

In this work we will demonstrate the possibility of using Ada 95 as a platform for the development of distributed hard real-time applications that communicate through the POSIX message-queue interface, and upon which the schedulability analysis techniques based on RMA can be applied. For this purpose, we have designed a distributed real-time application prototyping methodology that uses the Ada 95 priority-based scheduling mechanisms, although we can also use sporadic server schedulers for both processors and communication networks (Sprunt et al., 1989; Gutiérrez and González, 1996). Although we could have used the message queues to implement the partition communication subsystem defined in Ada 95, the application tasks of our prototype use the message queues directly in order to make the implementation simpler and more efficient.

The paper is organized as follows. In Section 2 we provide a quick review of the distributed system model, and of the communication support based on message queues that we use in our distributed application prototype. Section 3 describes the Ada 95 prototype for distributed applications with hard real-time requirements. In Section 4, we show the design of an integrated environment that we are currently developing for programming distributed hard real-time applications using the described prototyping methodology. Finally, Section 5 gives our conclusions and the plans for our future work in this area.

2. MODEL USED FOR DISTRIBUTED HARD REAL-TIME SYSTEMS

We will use the general model of event-driven distributed systems defined in (Gutiérrez et al., 2000), in which there is a set of external events (generated by external devices, timers, etc.) arriving at the system, which generate responses in the form of sequences of actions. These actions can be tasks, distributed in the different processors, or messages, transmitted by the communication networks. The actions activate each other through internal events, that can be generated both by tasks (to activate other tasks in the same processor or to send messages through the

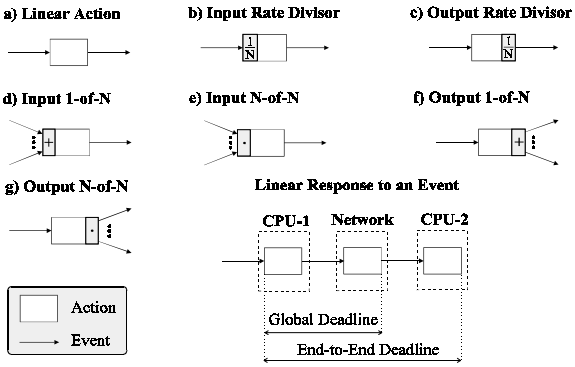

Fig. 1. Model for event-driven distributed systems

communication networks), or messages (to activate the tasks to which they are sent). We assume that both the external event sequences arriving at the system and the arrival rate of these events with hard real-time requirements are known beforehand. In our model we allow complex signal & wait synchronization among tasks by permitting their activation through combinations of multiple events, and we also consider the generation of several events by each task.

Fig. 1 shows the different patterns of action activation and event generation that are supported in our model. The Linear Action is the only one that is triggered by a single event and that generates only one event. In the Input Rate Divisors, the action is activated when it has received *N* instances of a particular event. In the Output Rate Divisor only when the action has *N* times will the corresponding output event or events be generated. In the rest of the cases, combinations of multiple events appear at the input or output, with N-of-N (all events must be present) or 1-of-N (only one of the events is present) patterns. The last action in the response to an event (generally a task) is a special case that corresponds to one of the described input patterns but for which no output event is generated. All the different combinations of event patterns are allowed with the exception of the combination in which an event that is generated by an action with Output 1-of-N gives rise to the activation of an action with Input N-of-N at some point of its response (Gutiérrez et al., 2000). The timing requirements imposed on the actions in their response to the external events may be of different kinds (see Fig. 1). We call *global deadline* the maximum time window that may elapse between the arrival of an external event and the completion of a particular action of its response sequence. We call the global deadlines corresponding to the last action in the response sequence to an external event *end-to-end deadlines*.

The analysis of a distributed system like the one described above can be carried out using RMA techniques, but taking into account the effect of jitter. The solution of this problem for the linear case in which all the response sequences are linear is given in (Palencia and González, 1998; Tindell and Clark,

1994). The analysis technique that can be used to find the response times in the general distributed system model is based on an extension of the analysis for the linear case and is described in (Gutiérrez et al., 2000). This technique transforms the system's model into another equivalent model over which the extended real-time analysis can be applied.

Message queues adapt well to the construction of distributed real-time applications that can be represented using the described general model. In the message queue application program interface that is defined in real-time POSIX, there is a set of operations that allow application tasks to send and receive messages to and from message queues. Since each message carries a priority with it and messages are retrieved in priority order, each message queue is a priority queue into which processes send messages, or from which processes receive messages, independently of whether the message queue is in the same processing node as the task, or in a remote node. Messages with equal priority are treated in FIFO order. Message queues may use a communications subsystem to exchange messages between nodes, through one or more communication networks. We have implemented the exchange of messages between tasks and message queues in different nodes through a mechanism that permits a priority-based communication with two kinds of scheduling policies (fixed priority preemptive scheduling, and sporadic server scheduling) over standard communications subsystems like the VME bus, and point-to-point serial lines (Gutiérrez and González, 1996). This kind of communication is also easy to implement over priority-based communication networks like the CAN bus (Tindell et al., 1994).Therefore, message queues in our distributed system may be of two different kinds: *Local Message Queues* (that connect only tasks located in the same node), and *Global Message Queues* (that connect tasks located in different nodes).

A configuration mechanism is needed to set the information about the allocation of global message queues to processing nodes. In this way, each node knows where each global message queue is allocated, and which communication networks to use when accessing it. Each message queue has attributes that are specified at creation time to fix the maximum number of messages in a queue, and the maximum length of each message. These attributes are very important in order to implement the message queue operations with bounded response times.

3. PROTOTYPING ADA 95 DISTRIBUTED HARD REAL-TIME APPLICATIONS

Distribution of application programs is addressed in Ada 95 in its Annex E (Distributed Systems), which standardizes the communication among partitions using remote procedure calls. However, this Annex does not address the distribution of applications with real-time requirements. Although we could try to implement the Distributed Systems Annex with real-time performance by using our message queue implementation to develop the partition communication subsystem, this implementation is complex and, besides, it would introduce a software layer between the application tasks and the communications that would increase the overhead, and which would have to be modeled so that the system were analyzable. For these reasons, we have used the message queues directly for the distribution of Ada programs, as an alternative to the remote procedure calls defined in Annex E.

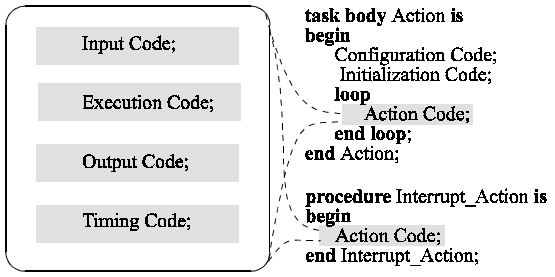

Fig. 2. Action template in a distributed real-time prototype

In this section we describe a prototyping methodology for building distributed applications with real-time requirements. Our prototype uses the message queues as communication mechanism between the tasks distributed in different partitions. The application is described using the model defined in Section 2, and then it is implemented in the prototype using message queues. The same system model that is used to build the prototype is used to apply the real-time analysis to determine whether the imposed timing requirements are met or not.

3.1 Fundamentals of the Prototype

The first aspect we must consider in the design of the prototype is the creation of a set of templates for the implementation of each of the different actions of our distributed real-time system model. These actions are implemented with tasks or with protected interrupt procedures in the processors, and with messages in the communication networks. Each of these actions may be either periodic or aperiodic, and may be scheduled using sporadic servers (both in the processors and in the communication networks) as an aperiodicity control mechanism, or as a mechanism for eliminating the negative effect produced by the presence of jitter in this type of systems.

Fig. 2 shows the code template that is used for periodic or aperiodic tasks and interrupt procedures.

After the initialization and configuration, the action will execute in an iterative way the action Code, which is the code that carries out the work of the action, including the management of the input and output events. The Action Code can be structured in four clearly different parts:

Input Code. It collects the triggering conditions of the action which may be external events, messages received from messages queues, etc. The input code may be null in the case of timed activations.

Execution Code. It carries out the work of the action. This is generally the only code that is supplied by the application developer. Alternatively, an executable stub may be supplied, with a timing behavior similar to the actual execution code.

Output Code. It generates the events that trigger the next actions in the response sequence to the external event; these events are usually messages sent to message queues. The output code is null for the last action of a response.

Timing Code. It establishes the timing control of the actions (periodicity, sporadic server scheduling, etc.)

The internal structure of these four segments of code is conditioned by the way in which the activation and finalization of the external event responses are carried out, by the different patterns defined in the distributed system model, and by the particular timing characteristics of the application actions. We will see the different cases below.

3.2 Distributed Model and Prototype Templates

Each of the input and output patterns of the distributed system model will influence the implementation of the code segments of the action templates in our prototype. Ignoring the special case of the first action in a response that is activated by an external event, we can consider each input event of an action to be a message received from a specific message queue. Normally, the interrupt procedures will be used only as the first action in a response, since they respond to external events produced by hardware devices. The intermediate or final actions in the responses to the external events will be implemented by tasks. When several consecutive actions with a linear structure are allocated to the same processing node they will be implemented as a single task, with several consecutive action code segments in the main loop.

Fig. 3 shows the structure of the input code that corresponds to each of the input patterns of the model, and also the message queues that are necessary to implement them. The input patterns of the model only influence the structure of Input Code segment. We can see that all the templates include the execution of the blocking message receive operation, which suspends the calling task if there are no messages available, over one or more message queues. In the Input Rate Divisor we await the reception of *N* messages before executing the rest of the code segments of the action. The input queue for an Input 1-of-N action contains messages coming from the responses to different event sequences. The order in which the reception operations are called is not important, because if the messages arrive out of order they are stored in the message queues until they are received. When carrying out the analysis it is necessary to consider the worst case in which the first message to be received is the latest to arrive, and so it is necessary to take into account a blocking term equal to the reception time of the rest of the messages in the calculation of the worst-case completion time of the input code.

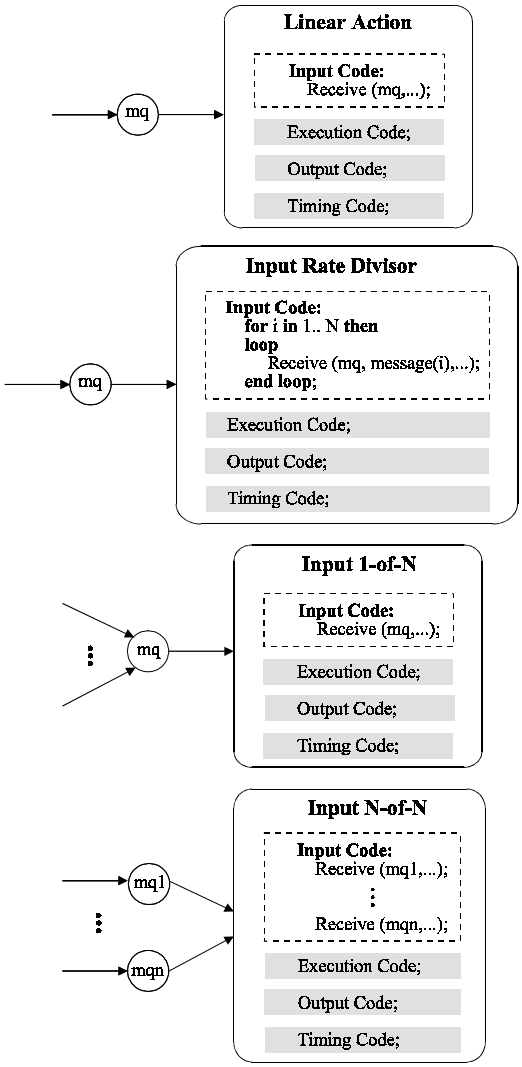

Fig. 3. Action Code templates for the different input patterns of our model

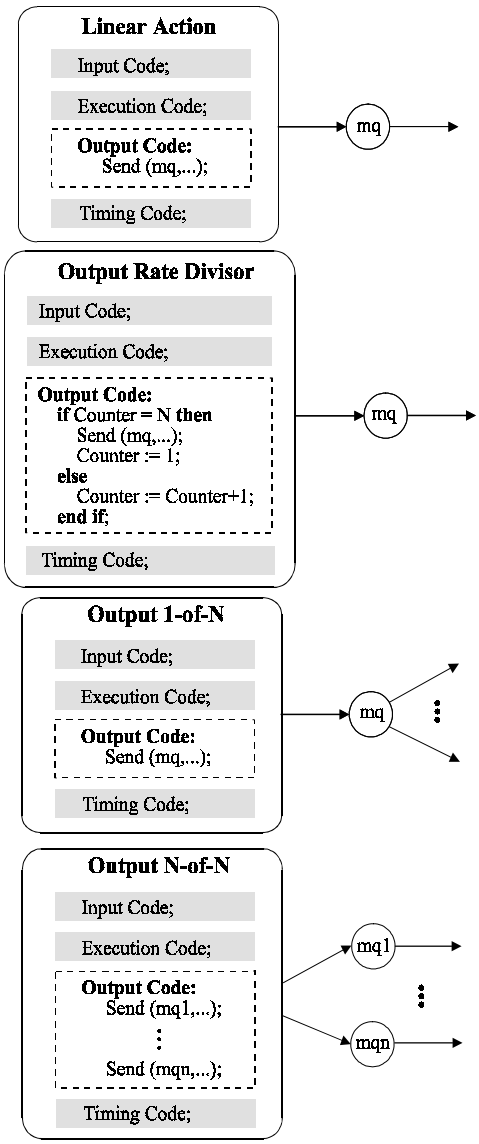

Fig. 4. Action Code templates for the output patterns of the model

In the same way as for the input patterns of the model, Fig. 4 shows the structure of the Output Code segment for the different output patterns, in which the message send operations are executed over one or more message queues, after the execution code of the action has finalized. In the case of the Output Rate Divisor this operation is only executed when the task has executed *N* times. The Output 1-of-N differs from the Linear Action in that there is more than one task available to collect the message from the message queue to which the message is sent. As with the Input 1-of-N case, the order in which we call the send operation for a task with Output N-of-N is not relevant.

An important aspect which still has to be defined in the prototype is the allocation of the message queues that communicate actions from different nodes, or if this allocation is really relevant for implementing the distributed application. The solution to this problem is more justifiable when the model for the real-time analysis is available, and so we will discuss it in a later subsection. However, we can conceive that, for example, in the case of an action with an Input N-of-N pattern it is better to locate the input message queues in the same node as the action. In this way, the time spent receiving the messages is minimized since only local operations need to be invoked.

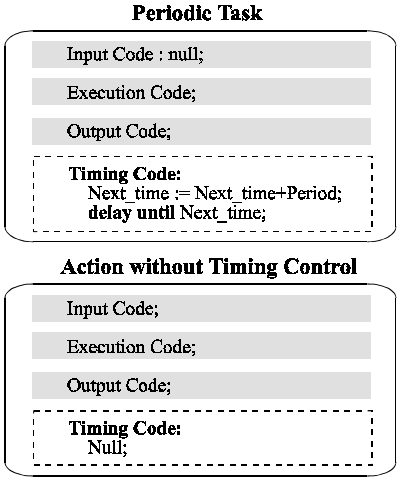

Fig. 5. Timing Code for periodic tasks and actions without timing control

3.3 Timing Control

In this subsection we consider the issues associated with the timing control of the actions, including the activation of periodic activities and the use of sporadic server scheduling for the control of aperiodic activities and the elimination of the negative effects of jitter. These timing control issues affect only the Timing Code segment of the tasks.

Fig. 5 shows the Timing Code of two typical actions in a distributed real-time application: a periodic task and an action without explicit timing control whose activation depends only on the arrival of an event. The timing control for the periodic task is carried out in the usual way, by causing the task to suspend until the next period, after its execution code has finalized, using a delay until sentence. The start of the period, *Next_Time*, is initially made equal to the current time, and then for the next activations the *Period* of the task is added. Usually the periodic task is used as the first action in the implementation of a distributed periodic activity (the external event in this case is generated by the hardware clock used for the delay until sentence). In the actions without timing control the Timing Code is empty. Examples of actions without timing control include the interrupt procedures, and the intermediate and final tasks in the response sequences to the external events (if they do not use sporadic servers).

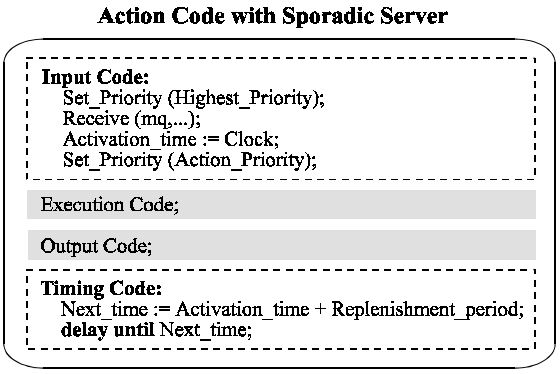

Fig. 6. Timing and Input Code for Sporadic Server Scheduler

The sporadic server (Sprunt et al., 1989) is a technique for scheduling aperiodic events that combines the predictability of the periodic polling with the short response times of the direct processing of events like, for example, interrupt service procedures. It is an extremely flexible solution for the scheduling of a large number of aperiodic activities and, in distributed systems, it can be used to eliminate the negative effects produced by jitter. The sporadic server reserves a certain limited amount of execution time for processing the application task at a given level of priority, in such a way that the effects on lower priority tasks is bounded. A sporadic server controls the execution of one task, and is characterized by two parameters: the execution capacity, and the replenishment time. The execution capacity represents an amount of execution time that can be spent at the priority that was assigned to the task. The initial execution capacity represents the maximum capacity of the server. The replenishment period is the time that must elapse so that the server recovers the execution capacity spent.

The sporadic server can be implemented at the scheduler level, but there are very few commercial Ada run-time systems or operating systems that support this algorithm. Thus, it is normally necessary to implement it at application level. Fig. 6 shows the Timing Code for an action controlled by a simple application-level sporadic server in which the execution capacity is equal to the worst-case execution time of the action, and the replenishment period is equal to the period of the action. The activation time is the instant at which the action is ready to execute, that is, when the Input Code finalizes, and it is necessary to then determine the instant when the execution capacity can be replenished. In order to minimize the difference between the actual event arrival and the recorded activation time, we execute the receive operation and the recording of the activation time at the highest possible priority. In this way, we can avoid preemption of the action during the execution of this operations. Of course this introduces a small blocking term which must be taken into account in the analysis. There are other implementations of the sporadic server at application level with different compromises among performance, overhead, and complexity (González et al., 1997).

3.4 Modeling the Prototype for the Analysis and Allocation of Message Queues

In order to analyze and schedule any kind of real-time system, it is necessary to obtain a suitable model of its timing behavior. The modeling of a distributed system based on the Ada 95 prototype that we have defined consists of the identification for each of the application actions in the system of their worst-case execution times and the blocking times that they may suffer due to synchronization. In this model we consider the processors and the communication networks separately. In the processors we find two types of actions: the application tasks, and the actions corresponding to the implementation of the message queues through which the tasks communicate. In the communication networks we identify the messages that are transmitted, divided into packets, which are the minimum non-preemptable communication units. There are two kinds of messages: application messages and the messages that are necessary for the implementation of the message queue protocol.

The identification of the actions corresponding to the message queues and the calculation of their worst-case execution times are dependent on the implementation. In (González et al., 1995), the modeling for the analysis of the implementation of the message queues is shown in detail. The application actions are the tasks and interrupt procedures considered in the prototype, and the calculation of their worst-case execution times is carried out starting from the templates as the sum of the worst-case times of the associated segments of code (Input, Execution, Output and Timing codes). In this calculation, it is necessary to consider the different input and output patterns, and the segments of code that are executed at higher priorities (due to the use of message queues or sporadic servers) in order to determine the possible blocking terms.

From the point of view of the analysis and correct operation of the system, we can say that the allocation of the message queues that communicate actions in different processor nodes, and which therefore use the communication networks, is irrelevant. However, in order to optimize the system, it will always be more interesting to allocate the message queues in the same node as the action that will receive the message. In this way, the message is always transmitted through the network at the instant when it is sent, and not at the instant when the corresponding action is prepared

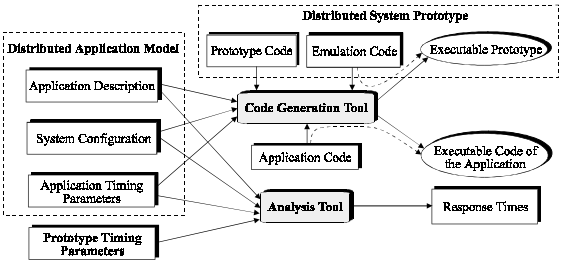

Fig. 7. Environment for developing distributed applications

to receive it (otherwise, this action would have to await the transmission of the complete message). This is closer to the idea of instantaneousness of the events in event-driven systems, in which the messages transmitted through the communication networks carry the events that trigger the subsequent tasks in the response sequences to external events. The only case in which we must locate the queue in the node where the message-sending action is located, is for the Output 1-of-N pattern in the case when the tasks that can receive the messages are located in different nodes. In this case, it is not possible to establish a unique relationship between the message queue and the destination node because there are several destination nodes.

4. INTEGRATED ENVIRONMENT FOR THE DEVELOPMENT OF DISTRIBUTED HARD REAL-TIME APPLICATIONS

In this section we will discuss the general lines of design of an integrated environment for the development of distributed real-time systems and for their programming in Ada 95, which we are now implementing. This environment is based on the distributed system model described in Section 2 and the prototyping methodology that we have extracted from it and that has been presented in this paper. Crespo et al developed an environment for building real-time systems (QUISAP) which provides the users with a specification language, a simulator, a schedulability analyzer, reusable real-time components and an automatic code generator (Crespo et al., 1989; Crespo et al., 1990). Starting from this environment Real et al (Real et al., 1996) carried out a real-time system prototyping using Ada 95. Although these works do not contemplate distribution nor the use of real-time communication networks, they constitute a good reference for approaching the design of an integrated environment for the development of distributed real-time systems.

The environment that we propose has the structure shown in Fig. 7. As it can be observed, the distributed application model along with the prototype templates that we have developed and the timing parameters of the application feed both a code generation tool capable of obtaining the executable code of the application or an executable prototype of it, and also, an analysis tool that generates the worst-case response times of the application. The different elements that constitute this environment are described in more detail below:

Application Description. It consists of a representation of the system at application level in an architecture independent way, that is, without specifying the processors or communication networks used by each task or message. All the actions of the application and their connectivity are represented through events. This model is used both by the code generation tool and by the analysis tool.

System Configuration. It specifies the processors and networks that constitute the distributed system, along with the allocation of each of the application actions (tasks and messages) to these system resources. The scheduling algorithm to be used is also specified (fixed priorities or sporadic server).

Application Timing Parameters. This contains all the timing parameters of the application, including the periodicity of the external events, the deadlines assigned to the different actions, the estimation of the worst-case execution times, and the identification of the critical sections that exist, for the calculation of the synchronization blocking times.

Distributed System Prototype. The prototype that we have developed supplies a complete set of templates to the code generation tool. The Prototype Code includes templates for each of the input and output patterns that have been defined, in order to be able to obtain executable code. Furthermore, within the prototype there is the possibility of supplying an emulation code to the code generation tool, which allows us to obtain an Executable Prototype of the application, even when the Application Code is not available (in part or in whole). The Emulation Code consists of a time consumption code both for the processors (tasks executing a loop for a specific number of times) and for the communication networks (messages of a specific length), using the worst-case execution (or transmission) times defined in the Application Timing Parameters.

Code Generation Tool. This tool links prototype code with either the application of the emulation code, according to the structure defined in the Application Description. As a result, we obtain respectively the executable code of the application or an executable prototype. The application code implements the basic actions of the application, that is, the executable code segment of the tasks and interrupt procedures, and the actual contents of the communication messages. In order to read the application description automatically

we have defined a special-purpose description language. The application itself may be described using the language directly, or using a graphical editor which generates the application description.

Prototype Timing Parameters. These parameters correspond to the real-time model of the prototype and of the message queues that are used for the implementation of the action, including the worst-case execution times of the different code segments along with the blocking times that can be observed.

Analysis Tool. This tool calculates the worst-case response times of the actions of the application so that they can be compared with the imposed deadlines. These times are obtained using the analysis techniques that were mentioned in Section 2.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a prototyping methodology for the development in Ada 95 of distributed applications with hard real-time requirements, which use a communication mechanism based on message queues. In this methodology we have defined an application description model that allows multiple event synchronization, and we have defined a set of templates that are used to implement the various possibilities that are supported in this description model. The application code may be directly built from the system description, the prototype templates and the action code supplied by the application developer. This system is implemented in such a way that allows the realization of a real-time analysis of the distributed application through RMA-based techniques. Furthermore, we have shown the general lines of the design of an integrated environment for the development of distributed real-time systems and for its programming in Ada 95, which is now being implemented.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their constructive comments.

REFERENCES

Crespo A., De la Puente J.A., Espinosa A., and García A. (1989). Ada Tools for Rapid Prototyping of Real-Time Systems. *Ada-Europe Conference*, Madrid, pp. 105-114.

Crespo A., De la Puente J.A., Espinosa A., and García A. (1990). QUISAP: an Environment for Rapid Prototyping of Real-Time Systems. *IEEE Conference on Software Engineering*, Tel Aviv, pp. 502-508.

González M., Gutiérrez J.J., and Palencia J.C. (1997). Implementing Application-Level Sporadic Server Schedulers in Ada 95. *Ada-Europe'97*, London, LNCS 1251, Springer Verlag, pp. 125-136.

Gutiérrez J.J., and González M. (1995). Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems. *Proc. 3rd Workshop on Parallel and Distributed Real-Time Systems*, Santa Barbara, CA, pp. 124-132.

Gutiérrez J.J., and González M. (1996). Minimizing the Effects of Jitter in Distributed Hard Real-Time Systems. *Journal of Systems Architecture 42*, pp. 431-447.

Gutiérrez J.J., Palencia J.C., and González M. (2000). Schedulability Analysis of Distributed Hard Real-Time Systems with Multiple-Event Synchronization. Proc. of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden.

ISO/IEC 9945-1:1998 (1996). Information Technology —Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API) [C Language]. *The Institute of Electrical and Electronics Engineers*.

Klein M., Ralya T., Pollak B., Obenza R., and González M. (1993). A Practitioner's Handbook for Real-Time Systems Analysis. *Kluwer Academic Pub.*.

Liu C.L., and Layland J.W. (1973). Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM, 20 (1)*, pp 46-61.

Palencia J.C., and González M. (1998). Schedulability Analysis for Tasks with Static and Dynamic Offsets. *Proc. of the 19th IEEE Real-Time Systems Symposium*.

Real J., Espinosa A., and Crespo A. (1996). Using Ada 95 for Prototyping Real-Time Systems". *Ada-Europe International Conference on Reliable Software Technologies*, Montreux, Switzerland, pp. 262-274.

Sprunt B., Sha L., and Lehoczky J.P. (1989). Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems, Vol. 1*, pp. 27-60.

Tindell K., Burns A., and Wellings A.J. (1992). Allocating Real-Time Tasks. An NP-Hard Problem Made Easy. *Real-Time Systems Journal*, Vol. 4, No. 2, pp. 145-166.

Tindell K., Burns A., and Wellings A.J. (1994). Calculating Controller area Network (CAN) Message Response Times. *Proc. of the 1994 IFAC Workshop on Distributed Computer Control Systems (DCCS)*, Toledo, Spain.

Tindel K., and Clark (1994). Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing & Microprogramming*, Vol. 50, Nos.2-3, pp. 117-134.