NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

THE MULTIACTIVITY PARADIGM IN RAPID PROTOTYPING OF EMBEDDED SYSTEMS

by

Raul San Martin Dipl. Ing

a thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60576-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis.
I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Raul San Martin

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Raul San Martin

Abstract

As our society experiences faster and faster rates of progress, technology becomes available to solve the most complicated engineering problems. These new technologies allow to build much more complex systems than what current methodologies allow to design in an orderly and structured manner, which is necessary to permit easy system expansions, upgrading and maintenance. This is particularly true in the case of embedded and real-time systems, which have to exhibit correct functional and temporal behaviors. This thesis focuses on a design methodology for embedded systems that is intended to be used by the application specialists, instead of the computer specialists. This avoids the problems generated when interactions are necessary between the two specialists and when the design is done by those who are unfamiliar with the application, lacking detailed knowledge of the system requirements. The methodology is based on the multiactivity paradigm and uses two system prototypes: the *Specification Prototype*, which is a prototype of the behavioral and functional requirements specifications; and the *Design Prototype*, which is a prototype of the design specifications and can be used to observe its temporal characteristics, to see whether the system will meet the required timing constraints. Finally, the methodology is exemplified and its feasibility demonstrated through various tests that were run using a simulator.

Acknowledgements

This thesis would not have been possible without the help of my supervisor, Dr. Krieger. He was always available for discussions of all aspects of this research and his expert guidance is much appreciated.

I would like to thank all my colleagues and new Canadian friends that not only offered technical help but also made me feel at home in a new society. Particularly, thanks to Amjad Junaid, Robert Joannis and Vojislav Radonjic for numerous discussions which provided many new ideas and insights, but most importantly, for their friendship.

I could not forget my parents who, after all, made many efforts to make this studies possible and provided me with an environment conductive to learning. They were always on my side, in spite of being geographically so far away. I am grateful to my brother, with whom I lived during the first year of my Master's and who helped me whenever I needed.

Finally, I will be forever grateful to a very special person who has always encouraged me. Fernanda, my wife, never denied support in the very difficult times that I passed throughout this thesis, and they were not a few.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter first introduces the scope of this thesis, presenting its motivations and goals. Next, it provides a background on embedded and real-time systems: what they are, how they work and how they are specified and designed. Important characteristics, such as *concurrency, reliability* and *timing requirements* will be presented in some detail; in addition, the required concepts of *multiprocessor systems* and *"real-time operating systems"* will be introduced. Finally, the chapter ends with the thesis outline, in which the contents of each chapter are briefly presented.

1.1 Motivations and Goals

As our society continues to experience faster and faster rates of progress, technology becomes available to satisfy and to provide the tools necessary to solve the most complicated engineering problems. Particularly in the field of electronics, digital hardware and software, new highly advanced components are being released in the market every day. With these new components it is possible to build much more complex systems than what current methodologies allow to design in an orderly, structured manner that permits easy system expansions, upgrading and maintenance [Joa86].

There is today a lack of accepted methodologies for the design of computer-based embedded systems that can be used by the application specialist. So far, traditional

design has been done by computer specialists who may not fully understand the details of each particular application and, therefore, introduce errors in the system's implementation. This type of errors could be avoided if the design were made by the application specialist, especially because it is estimated that most system design errors are introduced when the requirements are being analyzed by the designers [IS90, KLH88].

The objective of this thesis is to present a methodology and to show the feasibility of developing a set of associated design tools (constituting a complete design environment) for embedded or real-time systems that can be used by the application specialist, who is not necessarily a computer specialist. Furthermore, this thesis will present a simulation tool that can be used as a *requirements specification prototype* and as a *design prototype*. These prototypes are used to test and check the specifications of a system (requirements and design) regarding its functional, behavioral and temporal characteristics. The type of tests that should be executed will be presented, as well as how the designer should interpret the results to obtain the design objectives.

1.2 Background

This section has been divided into two subsections. The first presents a brief overview of embedded and real-time systems and the second presents an overview of the design process of these systems.

1.2.1 Embedded Systems and Real-Time Systems

Embedded systems, event-driven systems, real-time systems, behavior intensive systems and other similar terms describe an area of computer engineering that usually deals with application-specific systems. These are built to solve a specific engineering problem and especially those imposing concurrency and timing requirements in their behavior. This category of systems must meet not only the traditional necessities of general systems, that is, correct logical and functional behavior, but correct temporal

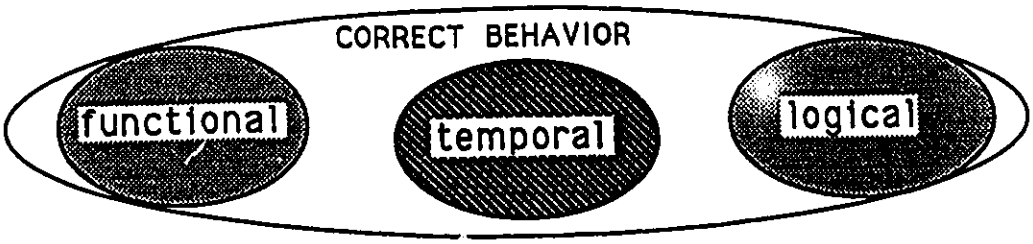

Figure 1.1: Embedded Systems Behavior

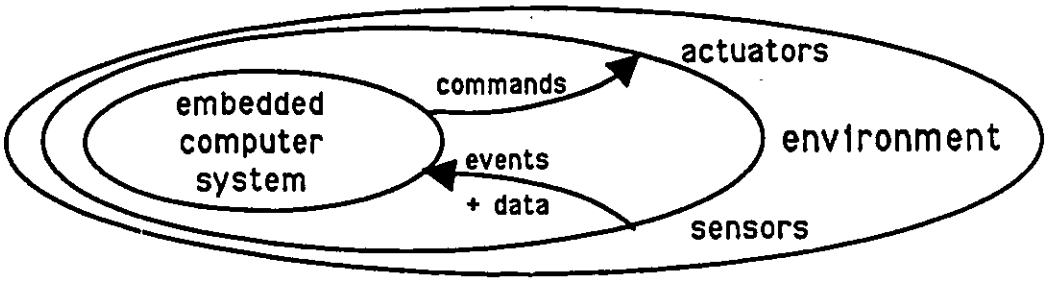

Figure 1.2: Embedded Control Systems

behavior as well (figure 1.1). They must not only do things correctly but they must do them at the correct time [Zav82, Gen88, Sta88a].

Real-time systems may be defined as a particular sub-class of embedded computer systems that have specific timing constraints. They are generally built as dedicated hardware controllers designed with some kind of processing elements, i.e. microprocessors or microcontrollers, which collect data from sensors and issue commands to mechanical, electric or electro-mechanical components, or even to other computers. In addition, these systems usually have to respond to events that occur in the environment (figure 1.2).

A good example of this type of system is a nuclear power reactor control system, which receives information from various sensors distributed over a nuclear plant and executes appropriate actions via some actuators. In this type of control system the time factor can be extremely important as it can transform correct actions into completely wrong actions. It is useless if the controller tries to decrease the flow of a valve

a few microseconds after the plant became unstable or it started leaking radioactive material. This mistake causes an immense cost in material damages and measureless costs in damage to human lives.

This definition of real-time systems being a sub-class of embedded systems is not shared by all researchers (Agrawala, for instance, defines embedded systems as a sub-class of real-time systems [LA90]). What is important is that both systems are highly interrelated event driven systems. In this thesis we will refer indiscriminately to embedded and real-time systems; these terms will be considered as synonymous.

The previous example of the nuclear controller shows a case of hard real-time constraints, where a minor delay cannot be tolerated. Another category of embedded systems has soft real-time constraints, in which minor delays can be tolerated but may cause increasing problems as time increases [Gen88]. In these systems, not meeting the time constraints will not necessarily produce a system failure. Examples of these are most types of industrial applications, such as temperature controllers which can easily afford minor and brief oscillations in temperatures without damages to equipment or products. However, fluctuations of the temperature over a long period of time may cause the equipment to eventually burn or explode.

A different view of soft real-time constraints is that in which the system can afford to miss some events, such as in a radar system, in which inputs (or samples) arrive at a very fast rate. It is very unlikely that a problem will occur when an input's deadline is missed if that input is just one of thousands.

Some of these embedded systems can be designed as sequential event response systems, which are normally represented as finite state machines and can only respond to one event at a time, not being able to take advantage of any concurrency that could otherwise have been possible. Such concurrency may be necessary to achieve the required deadlines, as systems become more complex and the timing requirements more rigorous. This is solved by the utilization of multitasking or multiactivity systems, as will be seen later in this thesis.

Another important requirement of embedded systems is reliability. Embedded

systems must be reliable since they are generally designed to operate costly equipments and/or to deal with human lives. They should have a life time comparable to that of the equipment or environment they are controlling [Gen88]. This is because the computer system is embedded in the environment and sometimes it is placed in spots difficult or dangerous to access. In most cases, a stop for maintenance is infeasible either because of the high costs that may result from an inactive system or from the impossibility of access to it (as in a satellite).

Most real-time systems, even if not expected to operate in non-stop mode are expected to provide fast recovery from failure, graceful degradation or in the worst case, safe shut down. Fault avoidance, masking and tolerance are standard methods that must be employed. The first uses high quality components and proved design techniques, while the second circumvents the errors that eventually occur. Neither method can provide one hundred percent flawless systems: disastrous errors may still occur when these methods are used. Fault tolerance, either by hardware, software, information (data) or temporal redundancy constitute a better, however more expensive alternative [Sea90].

Physical redundancy is achieved using multiprocessor or distributed systems [UPS90]. The former makes use of several processors sharing memory in a tightly coupled architecture, while the latter includes multiple processors arranged in a loosely coupled architecture with nodes connected by communication lines and each node having either uni or multiprocessor architectures.[1]

There is a widespread tendency today to provide reliability through distributed processing. This is in part explained by the success of some important systems, such as the Space Shuttle, which make use of several computers to control specific activities, providing redundancy and reliability. It is a natural way to distribute the control in different nodes, which can compensate for each other's failures.

Reliability is not the only issue. Real-time systems are also distributed for several other reasons. These include processing power (basically multiplying it by the number

[1]In this thesis the term "multiple processor" will be used for either tightly or loosely coupled systems (or simply, non-uniprocessor systems) in which each processor cooperates in some way to the processing of one or more responses.

of processors in the system), application specific processing (some problems require special purpose processors, such as DSP's, which are generally inefficient for general purpose applications), and economical reasons (resources are limited and therefore must be shared by many application tasks running on different processors) [Gen88].

Another important aspect is related to the communication overheads that appear whenever an application is designed as a set of concurrent entities (except in the rare case when these are completely independent and do not need communications and/or synchronization - which is of no interest here). The overheads are present in both uni and multiple processor systems. In the former, control is achieved through virtually concurrent entities, such as tasks or activities, which communicate with each other. In the latter, control is achieved through truly concurrent entities that run in different processors. Distributed architectures, those using communication lines, will also have additional delays due to the transmission of the messages. In addition, they may suffer insertion of errors in the messages and are likely to lose some of them due to the imperfection of the communication channels. To avoid the corruption of messages, it is necessary to include mechanisms capable of checking, detecting and correcting errors. However, they will not recover all errors and will introduce other delays and overheads in the system.

As it can be seen, as one tries to increase the reliability it becomes harder to meet the real-time constraints. A trade-off must be reached, as in most engineering problems.

All the timing requirements and resource constraints created by these complex systems led to the development of new operating systems designed specifically for them. In early multiprogramming systems, performance was improved by using the time previously spent in peripheral related activities, such as busy-waiting and polling, for pseudo-simultaneous execution of independent programs. This allowed many users to share a single expensive CPU. Today, as systems become more and more complex, the necessity of increased efficiency pushed towards the development of multitasking systems, in which many virtually concurrent tasks are "simultaneously" active sharing a single CPU. As technology evolved, multiple CPU's were used instead of a single

one giving rise to truly concurrent multiprocessing systems [BL90, DT90].

To deal specifically with systems that required fast response times and demanded real-time capabilities, concepts such as priorities, interrupt-driven tasks and dynamic scheduling were introduced in the so called *real-time operating systems*. Logically, these and normal operating systems (OS) have many aspects in common. For example, both deal with the limitations of having resources that have to be accessed serially through mutual exclusion and also of having limited resources. The mutual exclusion must be provided by using methods like semaphores, mailboxes or whatever is well suited for meeting the constraints. The difference is that, while in time-sharing systems an operating system is required to manage the system resources so that the user can be dedicated to the application specific issues, a real-time operating system has to be much more aware of the constraints of each particular application. In addition to the requirements of a normal OS, a real-time operating system has to provide time-driven allocation policies, application specific facilities and integrated system-wide scheduling, as pointed out in [Sta88a, Sta88b].

Because of the real-time constraints, the designer has many other aspects to consider. An additional aspect that greatly affects the performance is the way in which responses are divided into tasks. A bad partitioning can cause large communication overheads. In addition, tasks are relatively large pieces of code and they constitute the schedulable entities in multitasking systems. This implies that the scheduler scheme has to be carefully selected.

If the designer faces these types of decisions, then the choice has already been made for the utilization of multitasking systems. However, some of the problems mentioned above can be avoided by using the *multiactivity* paradigm, which is the alternative design method used in this thesis (and which is described in chapter 2).

1.2.2 The Design Process

The traditional design process of embedded systems involves four major phases [Zav84]:

- The Requirement Analysis Phase, in which the requirements of the controlled system are studied and understood;

- The Requirements Specification Phase, which consists of an unambiguous specification of the functional and/or behavioral requirements and also of the additional constraints, such as timing and performance, costs, weight, etc;
- The Design Specification Phase, which translates the requirements specification into a hardware and software specification;
- The System Implementation, Integration and Testing Phase, which is a transformation of the design specification into a real implementation and its testing.

In the life cycle of a product or system there are two additional phases, namely a maintenance phase and an evolution phase. In the maintenance phase, the system is simply kept at its original characteristics and performance. The evolution phase consists of the various alterations that the system will suffer throughout its life, either due to errors in the design or to requirements changes or performance enhancements or even to repairs for bugs [Luq89]. In a sense, the design of a system is never finished because it has to include these permanent phases so that the system operates correctly and according to the expectations.

The correctness of embedded systems depends on several factors besides the reliability of the components used. Major causes of errors are incorrect specifications of the requirements caused by erroneous, incomplete or ambiguous assumptions, and design and implementation errors that result in software and hardware errors in the final product.

The design of embedded systems has become the object of intensive research. So far, little is known about how to deal with them and there are many unanswered questions in how to specify them, how to prove that the specifications correspond in fact to the problem to be solved, how to translate the specifications into implementations in an efficient manner, how to validate and verify these implementations and so on [Sta88b, OW87]. One of the major aspects is how to represent these systems in an easy and clear way that is understandable by the users (application engineers) and by the the designers (the computer specialists). So far, many tools have been designed, each of them with its own advantages and disadvantages and each of them

more or less with specialized application areas.

Written descriptions are not adequate for the precise definition of complex processes which involve several possible concurrent entities. Words usually lack an exact meaning and even if the correct words are found, the description will generally be too complicated to be practical. Thus, there is a need for a precise, unambiguous and simple to use method capable of completely describing a system and its implementation. This description must be universal and has to be understood by all persons involved.

Visual representations are normally easily and rapidly understandable and provide a powerful way of expressing ideas. They are, however, hard to be universally accepted and it is difficult not to direct them too much towards specific applications, to prevent loss of general applicability. Also, *showing* a computer what to do, i.e. entering the representation, is difficult and less successful than giving it instructions via a programming language. However, visual representations convey the ideas more efficiently and have the potential of increasing the programmer's productivity [TIBY89].

The so called "conventional methods of specification" no longer work appropriately with systems built using current technologies since they are unable to cope with their ever increasing complexity [Zav84]. Today, large programs with millions of lines of code are no longer the exclusive domain of business and commercial systems; they do exist in many real-time systems as well. For instance, avionics computers today can easily involve millions or perhaps ten times more lines of code [HP88]. Therefore, it is essential to develop appropriate specification and design methods that can handle such complex systems.

The classic and well known finite state machines (FSM), one of the first tools used for specifying event driven systems, is still one of the most applied techniques. They find application in a wide range of areas, far exceeding the narrow field of real-time systems; they are very appropriate in those problems easily represented by states, events and responses. However, because they are so general, they lack many essential factors and mechanisms such as the representation of time limits and dealing

with data-dependent responses, which can generate extremely large and infeasible machines. Their main advantage is that they are relatively easy to translate into an implementation using simple executives that just execute the actions stored in look-up tables, with little overheads. Many real-time systems can indeed make use of this technique by employing powerful processors or even dedicated processors to handle the event queues or some specific activities. However, a methodology to translate requirements into FSM's is far from being well defined. In addition, the responses are strictly sequential, making real-time constraints difficult to meet; sequential machine response systems do not allow concurrent responses in systems where they would otherwise be possible, hence limiting the overall performance.

Some of these deficiencies are eliminated by extensions of FSM's, such as CSP-Communicating Sequential Processes [Hoa85] and ESTELLE [ISO85], which is mostly used in specification of communication software and protocols research. Time concepts are provided but specifying concurrency with ESTELLE may be a source of difficulties [Bru87]. These problems can be solved with other specification languages such as SDL [CCI88] and LOTOS [ISO86], which are also used mainly in the communication areas and, exactly for this reason, introduce other problems. These languages and techniques have been extended to provide real-time capabilities, such as CELOTOS, Timed LOTOS, Probabilistic CSP, Timed CSP, TCSP, Mini CSP-R, etc., as shown in [FP90].

Another well known tool for modeling concurrent systems are Petri Nets, [Pet77] which provide a very useful visual representation of concurrency and synchronization in general systems. For real-time systems, however, Petri Nets present a major problem since they do not provide means for representation of time constraints. Logical behavior can be easily represented but not temporal behavior. Time has been included in variations of Petri Nets, including Event Graphs or Timed Petri Nets [RH80, CMQV89]. Eventuality and fairness can be analysed with Temporal Petri Nets [SL89], which makes use of temporal logic. These techniques are good for analysing certain specific aspects of concurrent systems but are either quite narrow in scope or too complex for the general user or designer.

More complete design techniques have been developed, such as the COSY system [LTS79], Statemate [HLN+88] and Machine Charts [BKHW89], which integrate the structural, temporal and functional components of a system. Another interesting approach comes from France, the Grafcet or Function Charts [BBFM77] a simple and flexible method for the description of function and behavior of control systems. Grafcet was originally designed for electrotechnical applications but can be used in any field since it is a general way of representing control functions. It corresponds to an industrial application of Petri Nets and provides visual display capabilities for fast and easy comprehension of the system represented. Grafcet designers recognized the problem encountered when designing an embedded system controller: the controlling of a process requires a strong knowledge of the process itself and this knowledge has to be transferred to the design engineers. This is when most problems begin since for each specific application the designers would have to get deeply involved with its particular characteristics.

A major challenge that has to be met by specification techniques is for them to have a notation which is easy to understand, yet can be formally verified and validated. Examples of these are the object-oriented approach described in [Joa90] and PAL, Process Activity Language, which is used in this thesis. Being an executable specification language, a PAL specification can be simulated on a computer system, providing an easy way to analyze systems and to build prototypes. These prototypes can be defined as simpler, scaled down versions but concrete executable models of the system.

This leads to the concept of Rapid Prototyping. Rapid Prototyping can be defined as quickly building initial versions of a system. It is becoming an increasingly popular technique because it provides many advantages over traditional design methods [TY89]. The traditional approaches to system design which are most currently in use are phased refinement methods [YT89]. In these, the design engineers receive a description of the system from the customer and a list of its requirements. Based on this, an almost complete initial version of the system is presented to the user. Hopefully, the user will only require minor modifications and small changes. This method

is also based on the assumption that once the user has established his requirements, these are frozen and will not be altered [Luq89].

In most cases, however, this is not true and major changes may be necessary. Requirements often must be changed after the initial implementations. This would not cause a major problem if after these new modifications either in the requirements or just in the system parameters, the system would run according to the customer's expectations. However, these changes may considerably modify the performance initially foreseen by the designer and the initial goals may not longer be achieved.

The consequences of such changes may be: new interactions with the user, lengthy redesigns, changes in the architecture, etc. The project is likely to last much longer that expected, the budget will be exceeded and even the required constraints may never be achieved! In addition, the maintenance cost of the system can significantly increase since after several modifications the design may not longer be well structured.

The early use of prototypes in the design cycle shows the user what his system will look like, how it will perform and, last but not least, show whether the designer understood the customer's requirements or not. Thus, the prototyping concept seems natural and very logical: by building this simpler, scaled down but concrete executable model of the system, many of the problems can be avoided:

1. The user can provide feedback to the designer, who will know if he correctly understood the requirements and if he is working in the proper direction.

2. The user can check and validate his requirements. The prototype version will help him clarify some ideas and he may realize he had some wrong concepts. He may then want to change some of the requirements.

3. The designer can check if the time constraints are feasible by assigning duration to the various activities involved and simulating the system.

4. The model can show the user and the designer potential errors in the design such as assumptions made on the use of resources that will not be available in the real implementation.

Therefore, prototyping can effectively reduce the time spent in a project and can indeed increase the correctness of the designed system.

The building of a prototype is not in itself a guarantee of a correct design approach. Can the model actually validate the design in all its aspects? How good is this model? Can the model itself be validated to serve as a validation tool for the design? In general, a prototype is a reduced version of the final product in which many aspects will be simplified or may not even be present. Other aspects will have to be simulated, such as the environment. Is the designer's knowledge of the environment good enough to satisfactorily test the model? And how about the reliability expected for the final implementation? Is there any possibility to estimate it based on the prototype? Some of these questions may be impossible to answer; others will be discussed later in this thesis. Some of the crucial aspects that must be present in the model are:

- All external events that may have unpredictable behavior. These include those that may occur at very high or very low rates. They have to be simulated in all possible cases to validate the model's performance;
- The timing constraints of the system and the environment. The prototype has to include actions that are time-dependent on other actions or on events;
- Finite resources: the prototype has to include responses that use resources that will be limited in the final implementation. These include those responses using shared resources and those using resources with limited performance (such as those that will run in special purpose processors).

Certainly, some aspects need not be present in the prototype. Very intricate computations dealing with specific problems related to logical behavior and not temporal behavior will not affect the overall system functionality in terms of meeting the real-time requirements. Therefore, the coding of complicated numeric computations is in general not needed in the early model. Some aspects simply can not be represented correctly or efficiently. Examples may include power failures, aborts of processes, unexpected and unforeseen external events. Some of these may depend on the designer's ability and skill to represent them.

As pointed out by Lauber [Lau89], prototyping must be part of the design cycle in the sense that it should not constitute a separate design phase. As a separate phase, it would demand time and dedication from the designer to learn how to use the tools needed, in addition to the time spent building the prototype itself. This, instead of reducing, may actually increase the design time. Ideally, the final implementation should be just an extension of the prototype. In this way, the prototyping phase becomes an integral and cooperating part of the design cycle and the time spent on it is not an extra time added to the design cycle, but part of it. Indeed, the total time spent on the design will be shorter as the correct final product will be reached more easily and naturally. As it will be seen later in this thesis, PAL[2] meets these requirements because it can be used as system prototypes that later become the design specification itself.

Prototyping is undoubtedly a very powerful tool, but as any other design method, should be used carefully. No tool is safe of failures and all are prone to human mistakes. Problems may arise when doing an excessive number of changes to the prototype. Given an easy to use prototyping tool, the designer may be tempted to exchange design for experimentation. If this occurs, the system may become quite complicated (i.e. a mess) and it may then only be able to be modified and maintained by the designer that made all the changes. Equally bad may be the acceptance of a defective prototype, which could lead to a completely wrong or to an unacceptable system in terms of performance and behavior.

This thesis presents a design methodology based on system prototypes built using PAL and multiactivity systems. These will be discussed in the next chapters.

1.3 Thesis Outline

Chapter 2 presents the PAL-based design environment; multiactivity systems are introduced, followed by a description of the various elements of the Process Activity Language and by an overview of the currently available tools .

[2]For Process Activity Language

Chapter 3 presents one of the tools of this design environment, a simulator that was developed specifically to prove the feasibility of the ideas proposed in this thesis.

Chapter 4 introduces the prototype-based design methodology and explains how two system prototypes are obtained and tested. The first prototype is called a *Specification Prototype* and is a model of the behavioral and functional requirements of the system. It is used to check the correctness of the requirements specifications. The second prototype is the *Design Prototype*, which constitutes a model of the complete system. This model is used to check the temporal behavior of the system being designed, in order to assure that it can meet the performance constraints.

Chapter 5 presents the use of the design prototype to check additional aspects of embedded systems designed as multiactivity systems, such as reliability, the use of redundancy, communication overheads, etc.

Chapter 6 presents the conclusion of this thesis.

This thesis includes two appendix chapters: the simulator's user guide, which presents a more detailed description of the simulator's commands, files, screens, etc.; and an overview of timing in PAL processes, which is necessary to understand how the expected execution time of responses is computed.

Chapter 2

The PAL Environment

The purpose of this chapter is to present a brief overview of PAL, Process Activity Language, and its design environment: the PAL based tools. The chapter first introduces multiactivity systems, in which PAL specifications can be directly implemented, then presents the various language elements of PAL and, finally, introduces the PAL-based tools.

2.1 Multiactivity Systems

In a general embedded application there are two major components: the environment or *controlled entity* and the embedded system or *controlling entity* [KLH88]. The environment may be a very complex dynamic system consisting of many different physical devices, which must be controlled by a computer-based system. This controlling entity has to monitor the environment and coordinate appropriate actions that control the physical components. Most of the elements of an overall system are easily grouped into one of these two categories, such as a computer-based system into the *controlling* entity and a production line into the *controlled* entity; however, there are other elements, such as sensors and actuators, that can be included into either of those categories.

It may be possible for the designers to modify the controlled entity, but it will be assumed here that it is fixed and that it will simply generate events. The embedded

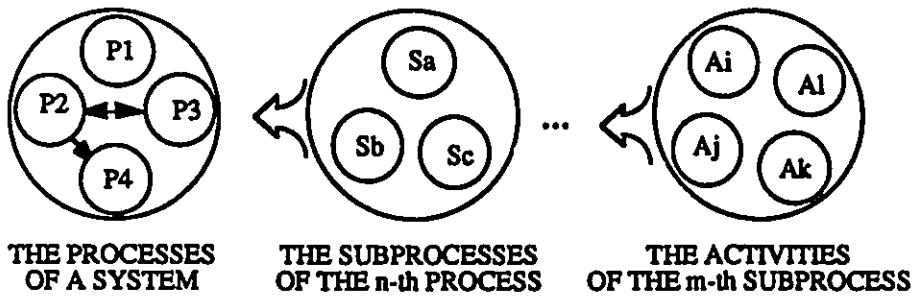

Figure 2.1: Decomposition of Embedded Systems' Responses: Abstraction Levels

system can be specified at various levels of abstractions and can be represented as an event-driven system because it simply has to respond to these events that occur in the environment. As shown in figure 2.1, at the highest level of abstraction, a system can be defined as being the set of responses to events, each response consisting of one or more processes. These processes can be partitioned into subprocesses, which could again be partitioned into smaller subprocesses, and so on until the lowest level of interest to the designers is reached, which is the level of the *activities*. An activity can be defined as a meaningful sequence of operations that can, once started, be executed to completion without having to wait for anything (information from other activities, resources, etc.). In this context, a process corresponds to a set of activities which are executed in a given precedence relation so that a correct response to an event is achieved. The order in which the activities have to be executed depends on the data dependencies between activities of the same process and between activities from different processes as well. Similarly, the granularity of the activities, i.e. how large the activities are, is determined by many factors such as a minimization of the communication overheads between activities, the required resources and the reliability of the overall system, as will be seen in future chapters of this thesis.

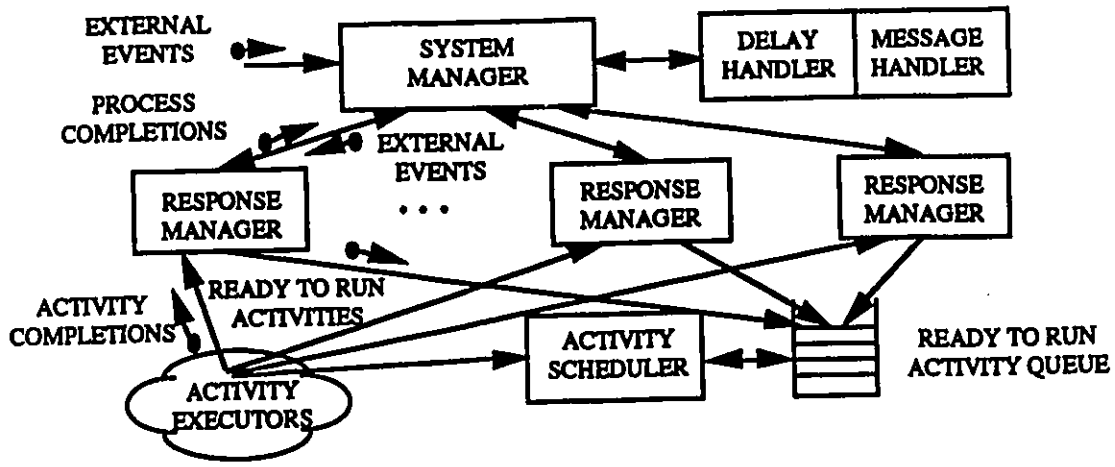

Figure 2.2: A Multiactivity Executive

The processes defined above may have multiple threads of control, as many activities may potentially be executed concurrently. This is in contrast with the definition of a task, which has only one thread of control. In order to meet the real-time requirements of embedded systems, it is necessary to maximize concurrency in the available processors. Traditionally, concurrent operations in multiple processor systems have been achieved through the use of multitasking operating systems, in which tasks are the schedulable entities. The fact that usually there is not a one-to-one mapping between responses to events and tasks introduces a major difficulty in assigning task priorities according to the real-time requirements [SR87]. Furthermore, most general purpose and real-time multitasking operating systems contain additional support facilities that are not often required in embedded systems.

An alternate approach is the use of a *multiactivity* executive in which *activities* that constitute a response are scheduled directly according to the real-time requirements of their parent processes. Such an executive that is responsible for the sequencing and scheduling of the activities, which are considered atomic units of work, can be implemented as the simple three layer structure shown in figure 2.2. It is composed of *a system manager, response managers and an activity scheduler.* These are explained

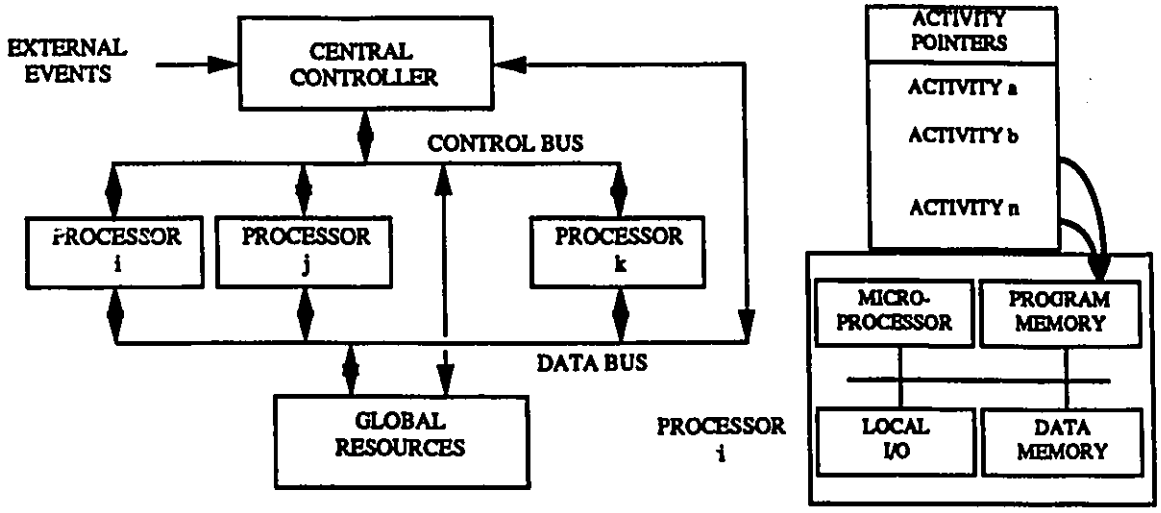

Figure 2.3: A Centrally Coordinated System Architecture

in detail in [KLH88]. The main characteristic of a multiactivity executive is that all relevant information to manage the event responses is kept within the executive. The activities themselves do not include any information relative to the coordination of the responses. The executive coordinates and manages the data required by the activities, which is passed directly by value or by reference through the use of a pointer to the data. Since activities are atomic entities, once started they can run to completion without having to wait for additional data. This characteristic may not only reduce the overhead of context switching but also simplifies the scheduler.

This multiactivity system is well suited for a centrally coordinated implementation because the coordination of the activities is completely separated from their execution. The coordination can be done in a central coordinator, while the activities can be executed in *slave* processors. Such a centrally coordinated architecture is shown in figure 2.3. The central coordinator shown in this figure is responsible for monitoring the environment, accepting the various events that occur, and for running the executive. The slave processors are responsible for executing the activities. The system may also have some global resources such as shared memories or specialized IO devices. These slaves can be all similar or they can be special purpose

processors that are required to execute some specific activities, such as a Fast Fourier Transform (FFT) in Digital Signal Processors (DSP). Each of these has its own local program memory, data memory and IO. The executable code of the various activities that each slave is able to execute is stored permanently in each of their local memories. Therefore, the execution codes need only be loaded at system start-up and no communication overheads will result from transferring these codes at run time. To avoid excessive hardware duplication, to restrict the size of program memories and to provide homogeneous load balancing, not all activities may be executed by all slaves.

This centrally coordinated system has several benefits in terms of system performance, fault tolerance and flexible scheduling. A good activity-to-slave allocation can also reduce communications by allocating activities that share data to the same slave. Regarding fault tolerance, the failure of a slave that was executing an activity will produce little overheads because it will affect only a single activity. In addition, failures can be rapidly detected either via specialized hardware or through time-out mechanisms. Critical activities can be executed redundantly in two or more processors. All these aspects will be analysed in more detail later in this thesis.

This centrally coordinated architecture shares the disadvantages of other centrally controlled systems, namely a possible controller bottleneck, communications bottleneck and the coordinator reliability. Because of the simplicity of the executive, the bottleneck in the coordinator may only appear when there is a very large number of responses, processes, activities and events. A good partitioning of the responses can reduce communications between different processes and an adequate granularity of the activities should not produce too many communication overheads between coordinator and slaves. The problem of the reliability of the coordinator can be solved by using either a very highly reliable processor or a hot stand-by which can assume control in case of a failure.

This centrally coordinated architecture is very flexible in terms of system expansions; additional slaves can be added very easily to increase performance and reduce response times as the requirements change. Such a flexibility is a highly desirable characteristic of embedded systems because these systems are expected to operate

for a very long life time, during which, the requirements usually evolve.

2.2 PAL - Process Activity Language

This section presents a brief description of the various elements of PAL- Process Activity Language, a general purpose high-level language for the specification and design of embedded systems. This language represents a continuation of previous research in this field [Cos83, Joa86, KJ86, Dur87, KLH88].

Being based on an event driven view of a system, PAL fits very well the multiactivity environment because it allows the designer to specify the coordination of event responses independently of the implementation details. For each response, the sequencing relation among the activities and the interactions with other responses are expressed as well formed expressions using constructs similar to those found in other structured high-level languages. In this sense, PAL provides a separation between the coordination of responses to events an the actions that constitute these responses. A system specified in PAL can be easily implemented in a multiactivity system because the coordination can naturally be done in the coordinator and the execution of the activities in the slaves. It should be noted that the activities themselves are not specified using PAL; they are only named and their exact definition can be done using any programming language.

The development of PAL attempted to achieve a balance between the need for understandability, to provide an easy-to-use and "user friendly" language, and the need for formalism, to allow for formal verification. This last requirement is satisfied by having well defined primitives and control constructs similar to those in structured programming languages, while the first requirement is satisfied by limiting the number and complexity of the language elements and basing them on intuitive concepts.

A system in PAL can be defined as a set of responses to events:

$\mathcal{SYS}$ = {Ra, Rb, ..., Rn}

Each of these responses may be defined as a set of one or more processes:

$Ri = \{Pi_1, Pi_2, ..., Pi_k\}$

Primitive	Description	Primitive	Description
ACT	Activity	DLY	Delay
TXM	Transmit msg.	WTM	Wait for msg.
STP	Set Proc. Cond.	STS	Set Sys. Cond.
RDS	Read Sys. Cond	RDE	Read Env. Var.
WTE	Wait For Event		
Construct	**Description**	**Construct**	**Description**
SEQ	Sequence	CON	Concurrence
RPT	Repeat	CAS	Case
SIM	Simultaneous	ALT	Alternation
SEL	Select		

Table 2.1: PAL Elements

Finally, each of the processes has the general form below:

$Pi_j \coloneqq Si.Xi_j.E$

Si represents a starting event or an enabling condition that triggers a response, which may be made of one or more processes. Depending on the requirements, the responses to the events may be handled in a number of ways: a new response can be instantiated for each event that occurs; a finite capacity queue can be used to store the occurrences of events; events can be ignored if the previous response has not yet been completed (*volatile events*).

Xi_j represents the body of a process and may contain any of the primitives whose ordering is defined by the control constructs, both of which will be presented next; E is a delimiter of the expression that simply represents the end of the process. A response is considered completed when all its processes are finished. In general, a response is made of a single process, so the end of a process usually represents the end of a response.

The primitives that a process may contain are represented in table 2.1 and are defined as follows (the mnemonic pseudo-code given in parenthesis):

1. Activity (ACT i):

This is the basic element of the process expressions and constitutes a schedulable unit of work with a bounded execution time. At the implementation level, it is basically translated into a piece of code that is executed by the slave processors. In general, a process in PAL can be seen as a set of activities whose execution ordering is governed by the surrounding control structure.

2. The Conditions Primitives (STP, STS, RDS, RDE):

   These are primitives that deal with the process and/or system conditions, which are simply variables that are associated with the control structure of the system and can be of any of the standard types (*integer, float, long,* etc). In PAL, the variables can be either local to the processes or they can be defined as global system variables, *billboard conditions. STP i exp* - Set Process Condition - and *STS j exp* - Set System Condition - set the local variable i or the global variable j to the value of the expression *exp*; *RDS Si Pj* - Read System Condition - copies the content of the global variable i into the local variable j; *RDE Ei Pj* - Read Environment Condition - copies the content of the environment variable i into the local variable j. Since the billboard conditions can be read and modified by all processes, they can be used for interprocess communications.

3. Delay (DLY t):

   This primitives is used for timing purposes and its execution produces a delay of the specified time t.

4. The Communications Primitives (TXM c, WTM c):

   These primitives are used for direct interactions between responses, either for information exchange or for synchronization. They are similar to the common *send* and *receive* primitives found in other languages and operating systems. Messages are passed between PAL processes via named, one way, finite capacity channels. The interprocess dependencies are explicitly stated in terms of the two primitives *TXM c* - Transmit Message on Channel c - and *WTM c* - Wait for Message on Channel c. If a channel is empty, the process executing the

WTM will wait until the arrival of a message; if a channel is full, a process executing a *TXM* will wait until a message has been received from the channel.

5. Wait For Event (WTE e):

   Through this primitive, a process can wait for an event to occur in the environment via a specified channel. Events are stored in finite capacity buffers, whose sizes define whether the events are volatile (size 0), latched (size 1) or queued (size n); As with messages, events can represent either pure synchronization (messages of length 0), or also the transfer of data items (messages of finite length).

The sequencing and coordination of all these primitives are controlled by the control constructs described below. The basic constructs that control the precedence relations between primitives or sub-expressions are the *sequence* and *concurrence*; additional control constructs are the *repeat, case, simultaneous, alternation and select.* The constructs are also shown in table 2.1.

1. Sequence (SEQ n):

   This construct specifies that n PAL sub-expressions or primitives have to be executed in sequence; each of them can only be started after the previous has been completed. The construct is ended when the last element is executed. For example,

```
SEQ 3
    ACT 1
    ACT 2
    ACT 3
```

   In the above sequence, activity 2 can only be executed after activity 1 has been executed and activity 3 can only be executed after activity 2 has finished execution.

2. Concurrence (CON n):

   This construct specifies that n PAL sub-expressions or primitives are independent and can be executed in parallel if there are enough resources available. The construct is only completed when all its elements have finished execution. For instance,

```
SEQ 2
    CON 3
        ACT 1
        ACT 2
        ACT 3
    DLY (25)
```

   In this sequence, activities 1, 2 and 3 can be executed in parallel and their execution is only limited by the availability of resources. However, the *delay* primitive can only be executed after all three activities have finished execution.

3. Repeat (RPT cond) and Case (CAS n (cond 1) ... (cond n)):

   These constructs provide conditional execution of primitives or sub-expressions. The repeat construct specifies that an expression will be executed repeatedly until the condition becomes true; the case construct executes one of n paths according to the evaluation of the n conditions, which are mutually exclusive; one and only one is true at any given instant. For example,

```
RPT (P2 > 1.9) | (P8/10 = 2)
    CAS 2
        (P2 = 1)          Condition for first element
        (P2! = 1)         Condition for second element
        STP 2 (P2 - 0.1)
        ACT 2
```

In this example, there is a loop which will be repeated until the condition is satisfied (when the process condition 2 is greater than 1.9 or the process condition 6 is equal to 20) [1]. Each time the inner loop is executed, one of the two paths will be chosen, depending on the value of the process condition 2: if it is equal to 1, then it will be decremented by 0.1; if it is not equal to 1, then activity 2 will be executed.

4. Simultaneous (SIM n), Alternation (ALT n) and Select (SEL n):

   The simultaneous and alternation constructs are basically variations of the concurrence construct and are defined only for primitives. In the simultaneous construct, the start of execution of all primitives is delayed until all the necessary resources are available, so that they are allowed to start simultaneously. In the alternation construct, the primitives start execution as resources become available. Contrary to the concurrence construct, it is considered complete when the first primitive finishes execution. The select construct is similar to the one used in ADA; each path is examined for eligibility to be executed, which is determined by the readiness of the first element of the expression. Among all eligible primitives, one is chosen non-deterministically. If none is eligible, then the first path that becomes eligible is selected. This primitive was extended in [Har89a] to include guarded commands, which give the designer more control over the non-deterministic outcome of the statement.

A simple example of PAL expressions represented in functional, pseudo-code and graphical formats is shown in figure 2.4. Note: *The pseudo-code format shown in this figure was defined in [KJ86] and differs slightly from the mnemonic pseudo-code used in this thesis.*

Figure 2.4 also shows an ATC - Activity Timing Chart, which is a graphical notation that indicates the execution times of the components of a process and also helps to visualize the hierarchical nature of PAL. The expressions can be viewed at a higher level of abstraction as a composite expression with an execution time equal

[1] The symbol | represents a logical OR

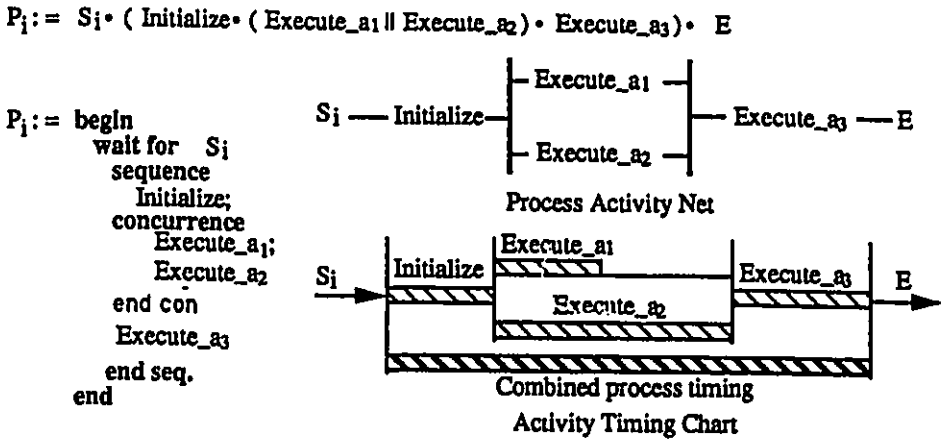

Figure 2.4: PAL Notations

to the combined execution times of the individual components.

2.3 The PAL-Based Tools

The PAL-based design methodology uses a toolset that is briefly described in this section. There are presently three major tools available: The Editor PALED, the Verifier PAL IVS and the Simulator SIMPAL.

2.3.1 The Editor

The PALED Editor [Chu90] allows the user to enter the system specification either as PAL expressions or in the mnemonic pseudo-code format. It provides the following capabilities:

- It allows step-wise refinement of the specifications by allowing abstractions to represent subexpressions, which the user can define at a later stage.

- It performs syntactical and lexical checking of the entered specifications, so that they can be formally verified and validated by another tool.
- It generates rudimentary ATC's of the system processes which provide visual representations of the processes' execution times based on the individual execution times of the activities assuming infinite resources.
- It generates rudimentary PAN's, Process Activity Nets, which are graphical representations of the PAL expressions and show the precedence relations of the various PAL elements.
- It translates the specifications from PAL itself into pseudo-code and vice-versa, generating the required input for the simulator.

2.3.2 The PAL Interactive Verification System

The PAL IVS Verifier [Har89a, Har89b] can be used to check the specifications regarding temporal errors such as deadlocks, starvation and violation of mutual exclusion. It is a tool to verify that a system, specified in PAL as a set of processes that interact with each other, is free of such errors.

This tool is based on the model checking approach to verification defined in [CES83]. In this approach, a system state graph model of the system's temporal behavior, which describes the complete temporal behavior of the system, is first derived from the PAL expressions. This model is then checked against a set of BTL (Branching Temporal Logic) formulae that represent a partial specification of the temporal behavior required of a system in order for it to meet its intended use. If these formulae are satisfied on the state graph model, then the PAL expressions can be considered a valid implementation of the partial specification.

2.3.3 The PAL Simulator

The simulator SIMPAL allows the analysis of PAL specifications by simulating them under various operating conditions. It not only interprets the language elements

but also closes the loop between the execution of the processes and their effects on the environment. The simulator allows the testing of the system regarding many aspects, such as reliability, communication overheads, scheduling schemes, activity-to-processor allocation, granularity of the activities, etc.

This tool was developed specifically for this thesis. The design of the SIMPAL is explained in detail in the next chapter, which presents a description of its internals and its functional modules. This chapter also briefly outlines the various commands available, the simulator environment, processes, activities, etc. A more detailed description of the commands and features can be found in the user manual, which is included as Appendix A.

Chapter 3

The Simulation Tool: SIMPAL

In this section, a general description of the design of the simulator SIMPAL will be presented. The simulator was developed to show the feasibility of the PAL-based design methodology. More specifically, it will be shown how this simulator can be used in the requirements specification phase and in the design specification phase of embedded systems.

The chapter presents a general overview of the simulator, describes how the PAL processes are treated, how the environment is simulated, how the simulator is composed in terms of functional modules, describes the man-machine interface and provides a high-level example of how the simulator is invoked.

A more detailed description of the commands, files required by the simulator and additional features is given in the user's manual (included as Appendix A).

3.1 General Overview and Motivations

The simulator SIMPAL is an essential component of the prototype-based design methodology that will be used and described in detail in the next chapters. It permits the analysis of a system specified in PAL under several expected and unexpected operating conditions. As such, it can be used to determine whether the specification will meet the requirements of the problem in terms of behavior and performance.

The simulator was written in the C programming language and its source code

has approximately 23,000 lines. It is available on PC-compatible machines and on a DEC-3100[1] risc-based family of workstations running in an Unix[2] environment. The program is highly modular so that it can be easily expanded in the future and easily integrated with other PAL tools to provide a complete development environment. Although many functions were developed as needed and are now available, the design followed the general *skeleton tools* idea. This allows easy adaptability of the tool to specific applications. In the *skeleton tools* concept the various tools are kept simple to allow the user to either add other tools or to expand the existing tools to satisfy his particular needs. This provides some advantages over larger, more complex and stiffer tools which may be hard for the user to understand and adapt to the specific requirements of his application.

Specifically, SIMPAL, as the name suggests, performs the simulation of PAL processes written in the mnemonic pseudo-code format of the language. The simulator requires two input files: the processes descriptor file *SIM_PROC.n*, which includes the specification of all the system processes and can either be entered directly in SIMPAL or can be created with the PALED editor; and the environment descriptor file *SIM_ENV.n*, which is created using a small program called Genenv. This file describes the environment in which the processes will run.

The simulator was built over an existent executive [Sha89] for PAL-based multiactivity systems. This executive was developed for the purpose of showing the feasibility of creating a small, simple and fast controller program for the coordinator processor in multiactivity systems. The executive provided a good starting point for the simulator because:

- It proved the feasibility of having an efficient executive in multiactivity systems.
- It provided a realistic perception of the system because the executive was meant to be used in an actual microprocessor implementation.

This executive corresponds more or less to the kernel of the simulator. It is

[1]DEC-3100 is a product of Digital Equipment Corporation
[2]Unix is a trademark of AT&T

basically responsible for the decoding and implementation of many of the language primitives and control constructs.

SIMPAL provides many additional basic features, among them:

- Extensive simulation facilities to emulate the environment, sensors and actuators.
- Powerful user interface, on-line and batch commands and display capabilities that allow the user to see what is happening with the processes and with the environment.
- The implementation of most of the control constructs and primitives with real data flow.
- Features related to reliability and redundancy that allow the user to generate errors and failures and to add redundant activities.
- The use of specific activities, each with its own execution time.
- The use of special purpose processors or processors of different performances (speed).
- The generation of a complete statistics file at the end of each simulation with data on real-time requirements, response times (minimum, average and maximum), processor utilization, interprocess communications, process statistics, etc.
- An on-screen visualization of the processes and a semi-graphical animation of their current status (available on Unix systems only).

3.2 Processes & Activities

The processes that SIMPAL accepts are written in PAL mnemonic pseudo-code and may contain any of the following elements:

- SEQ n - sequence of n elements
- CON n - concurrence of n elements
- RPT cond - repeat until condition is true
- CAS n - case of n elements, followed by the conditions
- ALT n - alternation of n elements
- DLY t - delay of t ticks
- ACT i - activity i
- WTM c - wait for message on channel c
- TXM c - transmit message on channel c
- WTE e - wait for event e
- STP Pi expression - set process condition i to value of expression
- STS Si expression - set system condition i to value of expression
- RDE v Pj - read environment variable v into process variable j
- RDS Si Pj - read system condition i into process variable j

Each process contains its own data variables, similar to variables found in other high level languages. These are local to the processes and cannot be accessed by other processes. In addition to the local variables, there are system variables, or global variables, which may be read and modified by any process in the system. These variables can be of four types: Byte, Integer, Real and Long.

The **activities** executed by the slaves are defined by two parameters:

- Duration of the activity: this is the estimated amount of time that an activity will take to be executed by a slave.

- Slaves in which it can be executed: this indicates to the simulator which of the slaves can execute the activity. In a real implementation certain activities may be executed only by specialized processors (for example an FFT in a DSP) and cannot be executed by any other slave in the system.

The activities of a process are normally considered as simple time delays; they just consume time of the slave processors. In order to close the loop between the controller and the environment, some activities that perform special actions are included. For instance, they can simulate real actuators that will cause modifications in the environment as times passes, they may cause an event to occur after a specified amount of time, may increment or decrement other variables, etc.

SIMPAL also allows the use of redundant activities that must be executed simultaneously either by two slaves (Double Modular Redundant - DMR activities) or by three slaves (Triple Modular Redundant - TMR activities).

As a final note, the simulator assumes that the PAL expressions defined in the input files are correct and contain no lexical or syntactical errors. These are caught and corrected by the PALED editor.

3.3 The Environment

The real world to which an embedded control system is connected has three major elements: actuators, sensors and events. All these can be represented in SIMPAL. These are handled using three types of *environment variables* and four types of events.

The *environment variables* are defined as follows:

- Direct Variables

  In terms of sensors and actuators, SIMPAL simulates them via environment data variables, which represent real world entities such as temperature, pressure, speed, etc. The values of these variables can be modified randomly or by special activities.

- Indirect Variables

These are environment variables that are automatically modified by the simulator according to a flag that is set or reset by special activities. They are used whenever an action has to occur in the environment. For example, the temperature of a heated tank should automatically decrease when a heater is turned off, and should increase when the heater is turned on. This is a simulation facility that provides feedback from the environment in reaction to the activities executed by the slaves.

- Event Variables

  These are environment variables whose values are modified randomly whenever an event occurs. An example is a system in which a mold arrives and has to be filled with a certain quantity of liquid. This quantity is identified by a bar code on the mold and may range from a certain minimum level to a maximum level. This quantity can be generated randomly each time the event "mold arrived" occurs and it constitutes an *event variable*.

In SIMPAL, events are queued as they occur. The size of the queue determines whether the event is volatile (size 0: events cannot be stored), latched (size 1: only 1 occurrence can be stored) or queued (the number of occurrences that can be queued is specified by the size of the queue). In line with this, some of the performance measures that can be obtained are the queue average and maximum sizes; and the average and maximum response completion times (which include the time spent in the queues).

The four types of events that can be used in SIMPAL are:

- Cyclic events: those that occur periodically with the specified period.
- Probabilistic events: those that occur according to a given probability of occurrence.
- Time-interval events: these are similar to probabilistic events, but they will always occur between a minimum and a maximum time. Whenever this kind of

event happens, its next occurrence is determined by generating a random time between the limits provided.

- Delayed Events: these are events that are triggered by any other event after a specified amount of time.

As seen in chapter 2, PAL defines two types of events: *starting events*, which are those that trigger processes, and *condition control events*, which are those that indicate that a certain condition has occurred. Regarding the starting events, they have an associated deadline for execution of the response. For example, a "500" deadline means that all processes that are triggered by this event must finish execution not later than 500 units of time after the event occurred.

In terms of the simulator's clock, basically, a cycle of the simulator consists of a single pass of its main functions, which are described in the next section. This represents a clock tick. Hence, the clock is not tied to any specific time units; the meaning of time is left for the user as he can freely choose the granularity required for the execution time of the activities, which are expressed in terms of the clock ticks.

3.4 The Functional Structure

SIMPAL is composed of eight major modules, each of them including a number of sub-modules. The functional modules and the interactions between them are shown in figure 3.1 and are described below.

1. Initialization Module

   This module is responsible for the system's initialization and includes the following sub-modules: Environment initialization, Processes initialization, Scheduler initialization, Communications initialization, Statistics initialization and Display initialization.

   The Environment initialization sub-module opens the environment descriptor file and loads all information related to events (number of events, types, parameters, processes triggered, due date, etc.) and the environment variables.

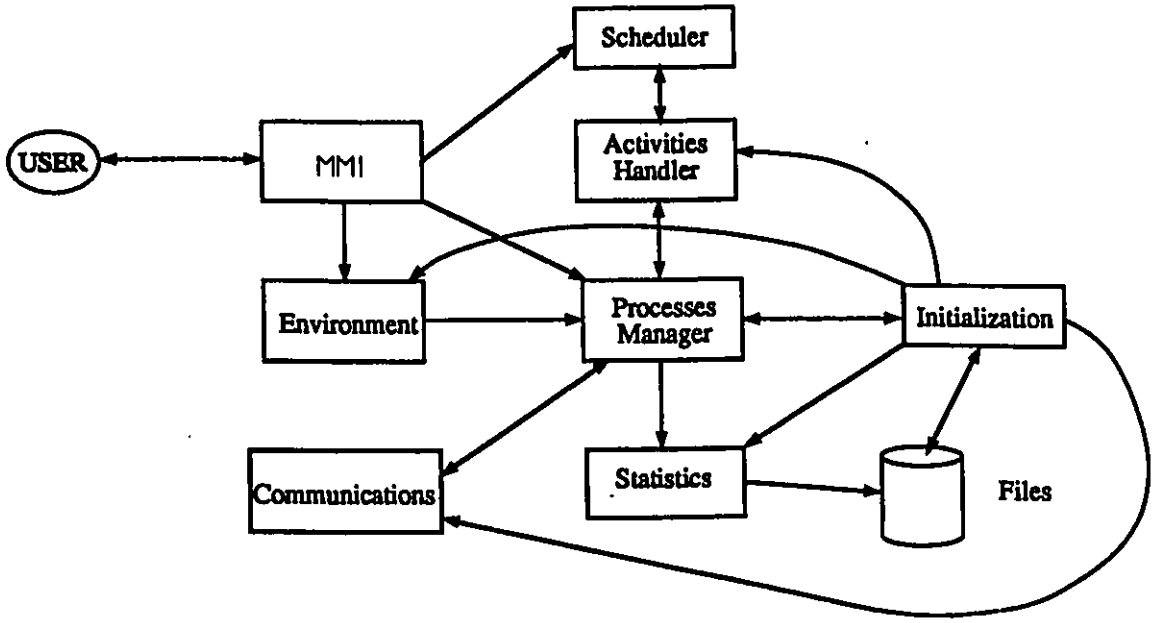

Figure 3.1: Functional Modules

The Processes initialization sub-module opens the processes descriptor file and loads all information related to the global variables in the system and the number of processes. Then for each process it reads its private variables and its language elements written in PAL mnemonic pseudo-code. This sub-module also initializes all processes and sets their initial state to "INACTIVE".

The Scheduler, Communications and Statistics initialization sub-modules mainly load their respective data structures according to the information read by the previous two sub-modules.

The Display initialization sub-module informs the user about the initial parameters of the system and prepares the information fields on the screen.

2. The Man-Machine-Interface Module

This module updates the current status information on the screen, informing the user about the slaves (which activity they are executing and for which process),

the processes (current status), the communication channels, the ready-to-run queue and the events.

It also interacts with the user through a number of on-line commands that allow the user to check and modify most of the system's parameters (these commands are explained later in the section Human Interface & Commands).

3. The Environment Module

   This module is responsible for handling the events that are queued. If an event is queued, it signals all the processes triggered by the event to be awakened. The module contains procedures which are called when an event is detected. These are responsible for accepting the data carried by the event, transferring them to the triggered processes and for updating the system's tables.

   This module also contains the routines that deal with the WTE (Wait for Event), DLY (Delay) and RDE (Read Environment Variable) primitives. In addition, it contains all the routines related to the simulation of events, sensors, activities, etc, and other functions that simulate real world parameters. For instance, temperatures can be automatically increased or decreased at each simulator cycle; random data can be generated each time an event occurs; special activities can set, reset, increment or decrement environment variables or even cause other events to occur after a specified time.

4. The Processes Manager Module

   This module contains the functions responsible for the managing of all system processes. The main function in this module is called at each cycle of the simulator and it basically activates awakened processes or updates currently active processes.

   Other routines are used for tracing the active threads of processes; for unblocking the blocked threads upon completion of the blocking element; for selecting the execution of a language element according to its type; for initializing and finishing control constructs; etc.

5. The Activities Handler

   In general, this module is responsible for the functions related to the execution of activities: queueing activities in the ready-to-run queue; allocating activities to slaves; receiving and transmitting data from and to the slaves. It interacts with the Scheduler module, which will order the activities in ready-to-run queue before the activities are assigned to slaves.

   Other important functions found here provide an interface with the Processes Manager module, accepting all requests for executions of activities and dealing with activities whose execution has been completed.

6. The Scheduler Module

   This modules orders the activities in the ready-to-run queue according to the selected scheduling algorithms (schemes). The possible schemes are described below:

   - FCFS, First Come First Serve: The activities are allocated to the slaves according to the order in which they arrived in the Ready-To-Run queue.
   - LACTF, Longest Activity First: The activity with the longest execution time is allocated first.
   - SACTF, Shortest Activity First: The activity with the shortest execution time is allocated first.
   - EARDF, Earliest Response Due First: The activities of the process with the closest deadline are allocated first.
   - LSSLF, Least Static Slack Time First: The activities of the process with the smallest initial laxity are allocated first.
   - LDSLF, Least Dynamic Slack Time First: The activities of the process with the smallest laxity time are allocated first.
   - HPRIF, Highest Priority First: The activities of the process with the highest priority are allocated first.

7. The Communications Module

   This module contains the functions responsible for message exchanges between PAL processes. The major function, which is called at every simulator cycle, checks all the currently pending transmit and receive requests, servicing the queues as necessary.

   Other routines found in this module insert and obtain messages to and from the buffers (communication channels). They provide an interface with the Processes Manager module, which requests the execution of the transmit and receive primitives (which are handled here).

8. The Statistics Module

   This module contains functions that maintain and check statistical data structures which have to be updated at every simulator cycle.

   One of the major functions compiles and formats these statistics at the end of the simulation and generates a file which can then be examined by the user. These statistics include:

   - Run Duration: the duration of the simulation.
   - Event Handling: the event handling data shows a set of statistics for each of the events in the system. These include the percentage of time that the event happened, the percentage of occurrences of this event compared with the total number of occurrences, the percentage of occurrences that the event was queued or missed, the percentage of the occurred events that were actually used for triggering responses (it will be the same as the queued percentage, except when the event is volatile).

     Next, statistics about the event queues are shown: average size, maximum size, minimum,average and maximum time that an event spent in the queue. The file also shows minimum, average and maximum response completion times.

- Slaves Statistics: the number of slaves and the their utilization are also shown.
- Communication Channels Statistics: in this section, the maximum utilization of the communication channels is shown along with the time it occurred.
- Ready-To-Run Queue Handling: these statistics refer to the size of the ready-to-run queue after the scheduling is done.
- Process Statistics: each process has a set of related statistics that are in general shown as minimum, maximum and average values. They include the time a process spent waiting for an event, waiting for a slave, waiting for a message, the time its activities spent being executed by the slaves, etc.

3.5 Human Interface & Commands

The main screen displayed by the simulator is composed of 7 fields and is shown in figure 3.2.

These fields are:

1. The Process Status Field, which displays information regarding the status of the processes and their due date. The possible statuses are:
   - *READY:* the process is ready to execute.
   - *ACTIVE:* the executive is executing a primitive(s) of this process (which is not an activity).
   - *EXEC:* one or more slaves are executing activities of this process.
   - *BLOCKED:* the process is waiting for the execution of a blocking element, such as the WTM or DLY primitives.
   - *INACTIVE:* the process either has not been triggered yet or has already finished execution.

Figure 3.2: SIMPAL's Main Screen (Unix Version)

2. The Slave Status Table Field, which shows the current status of the slaves, which activity they are executing, for which process and what is the execution time left of the activity. The three possible status of the slaves are BUSY, NOT_BUSY and FAILED.

3. The Communication Channels Field, which shows the available space in each of the first four channels in the system (the others are not shown because of a lack of space on the screen).

4. Elapsed Ticks, Events Occurred and Scheduling Field, which displays the current clock tick, the last event that occurred and its tick of occurrence and the current scheduling algorithm.

5. The Ready-To-Run Queue Field, which displays the contents of the ready-to-run queue in terms of activities and processes.

6. The Commands Field, which simply shows the commands that are currently applicable.

7. The Processes Elements Field (only on Unix Systems), which shows all the processes elements and their current statuses (active, blocked, pending). This provides quite interesting and useful capabilities for tracing the execution of a process and to observe a dynamic animation of the system.

The simulator interacts with the user and accepts the following on-line commands:

1. Abort the simulation: causes the simulation to stop without generating any statistics file.

2. Continuous mode: the simulation enters the "continuous mode", in which cycles are executed continuously without interactions with the user.

3. Events: calls a pop-up menu and a window which shows the status (number of ticks to occurrence) of all cyclic, "time-interval" and "delayed events". In this event window, a menu can be called which allows the user to configure and modify the parameters of the existent events.

4. Failures: creates a new window in which parameters related to system reliability can be modified. Basically, it is possible to modify the probability of a slave failure, to provoke the immediate failure of a slave and to change the parameters of failure frequency and recovery time.

5. Second Screen: an entirely new screen shows the value of all variables in the system, which may be modified.

6. Scheduler Algorithm: this command creates a new window which shows the scheduler algorithms available and asks for a new algorithm.

7. Run Until: creates a small window and asks the user to specify a tick to stop at. The simulation will continue until that tick is reached.

8. Default: pressing space bar or any other key will put the simulation in "single-step" mode and will run the simulator for just one more step.

3.6 Invoking SIMPAL: An Example

In this section, examples are presented to demonstrate the use of SIMPAL.

To run the simulator itself is extremely easy after the two descriptor files (Processes and Environment) have been created.

The syntax to call the simulator is:

SIMPAL [/B:] [/C:k] [/E:k] [/F:f] [/I:] [/N:n] [/P:s:p] [/S:k] [/T:k] [/V:k] [/W:]

The options appear between brackets ([ ]) and are optional(!). They may be used in any order. The options (with their default value) are indicated below:

- B - Batch mode: No
- C - Number of channels: 8
- E - Default activity execution time:1
- F - Failure probability: 0.0

- I - Failure detection at end: No
- N - Number of slaves: 8
- P - Performance factor of slaves: 1.0 (all slaves)
- R - Recovery and failure rates: no failures, no recovery
- S - Scheduling algorithm: FCFS
- T - Simulation time: 2000
- V - Version to load: 0
- W - Wide Screen: No (Unix systems only)

For example, if the user would like to run a simulation of the files of version 7 for 15000 units of time using 4 slaves and in batch mode, the command would be:

SIMPAL /V:7 /T:15000 /N:4 /B:

The batch mode option allows the simulation of any number of different versions automatically, without user intervention, just by writing a batch file. This batch file can look like this:

```
SIMPAL /V:0 /T:150000 /N:1 /B:
SIMPAL /V:0 /T:150000 /N:2 /B:
SIMPAL /V:0 /T:150000 /N:3 /B:
SIMPAL /V:1 /T:150000 /N:1 /B:
SIMPAL /V:1 /T:150000 /N:2 /B:
SIMPAL /V:1 /T:150000 /N:3 /B:
```

In this example, six simulations will run automatically, without the user having to be there. He can just collect the statistics at the end of the batch process.

Chapter 4

PAL-Based Prototyping

This chapter presents a methodology for the generation of two system prototypes, the *Specification Prototype* and the *Design Prototype*. These are used in the prototype-based design cycle shown in figure 4.1. This figure shows a design cycle composed of four major phases:

- The Requirement Analysis, in which the requirements of the controlled system are studied and understood.
- The Requirements Specification, which consists of an unambiguous specification of the behavioral/functional requirements of the system and also includes other requirements, such as timing constraints, costs, etc. In this phase a prototype of the behavioral/functional requirements is developed (a "Specification Prototype"). This phase is as much as possible implementation independent.
- The Design Specification, which translates the requirements specification into a hardware and software specification; it includes the generation of a prototype of the design specifications including the temporal behavior, structure and functionality of the system (a "Design Prototype").
- The System Implementation, Integration and Testing, which is a transformation of the design specification into a real implementation and its testing.

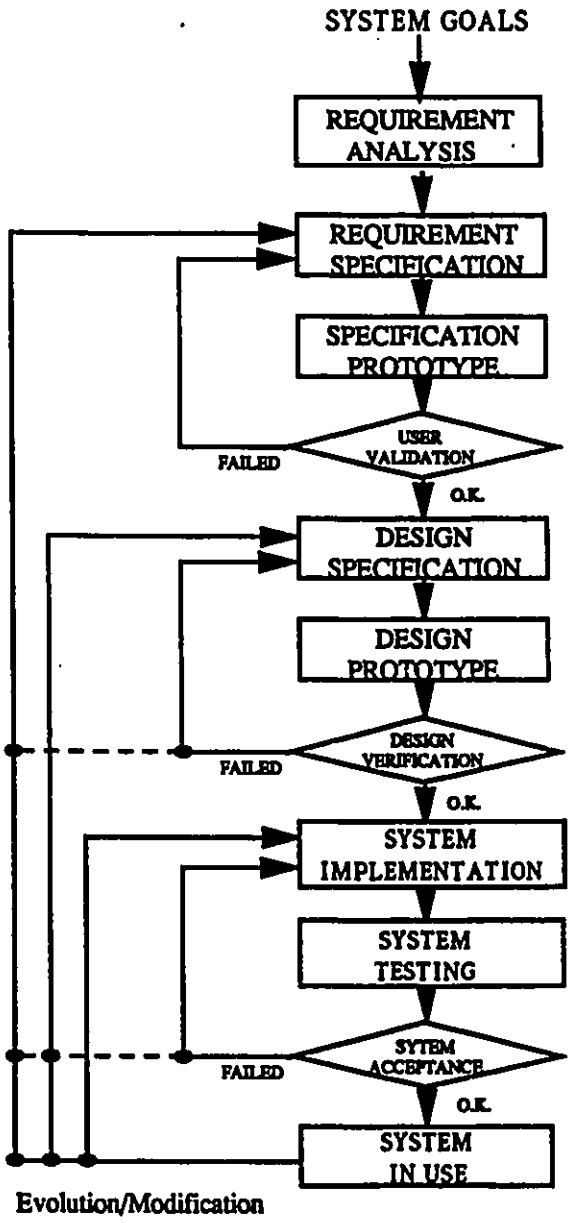

Figure 4.1: A Prototyping-Based Design Cycle

Another phase that is not explicitly shown in figure 4.1 is the maintenance and evolution phase, which consists of the various alterations that the system will suffer throughout its life. These alterations are either to maintain the system at its original performance level or to adapt it to evolving requirements. In a sense, the process of designing a system is never finished because it has to include such a permanent maintenance/evolution phase.

The purpose of this chapter is to present how the PAL simulator can be used (in conjunction with the other PAL tools) to generate the Specification Prototype and especially the Design Prototype shown in figure 4.1.

4.1 PAL-Based Specification Prototype

As mentioned above, the behavioral/functional requirements specifications prototype is called a "Specification Prototype". As such, it can provide early feedback in the design cycle, which will assure that the behavior of the system being designed is in accordance with the behavioral and functional requirements. Furthermore, it will assure that these requirements were correctly specified. This prototype does not take into consideration any of the system's constraints, such as performance desired, timing or costs and does not assume any specific hardware or software platform.

The generation of this prototype is possible if the specifications are done in an executable language such as PAL, because they can be easily simulated using an interpreter of the language. In addition, these specifications become a major part of the design specifications and thus, the prototype saves time and costs in the development phase.

This section provides a description of the different phases for the generation of the Specification Prototype, showing how the various PAL tools are used and, in particular, how the simulator is used.

4.1.1 The Requirements Specification Cycle

The behavioral and functional requirements specification cycle is shown in figure 4.2. It consists of three major phases, each of them using a specific PAL tool:

1. Functional and Behavioral Specification Using the Editor

   This is the first phase and follows directly the requirement analysis of the problem. The specification of the system must reflect the behavioral requirements. It is written either as PAL expressions or in PAL pseudo-code and then entered using the PALED editor [Chu90]. The editor then checks the lexical and syntactical correctness of the specification. If any errors are found, they must be corrected and the process repeated until no more errors are present.

2. Temporal Logic Checking using the Verifier

   In this phase, the specification is checked using the PAL IVS Verifier [Har89a], which will test it with respect to temporal errors, such as deadlocks, critical races and starvation. If errors are encountered, the functional specification will have to be be redefined and the process goes back to step 1 above.

3. Functionality and Behavior Checking using the Simulator

   In the last phase, the specification is simulated using the simulator SIMPAL assuming infinite resources. This phase does not test the system for performance but only with respect to its functional behavior to check whether it does what it is expected to do. If functional errors are found, the specification has to be redefined and the process goes back to step 1.

The editor and the verifier are explained in details in other theses [Chu90, Har89a]. The use of the simulator in this phase is presented in the next section.

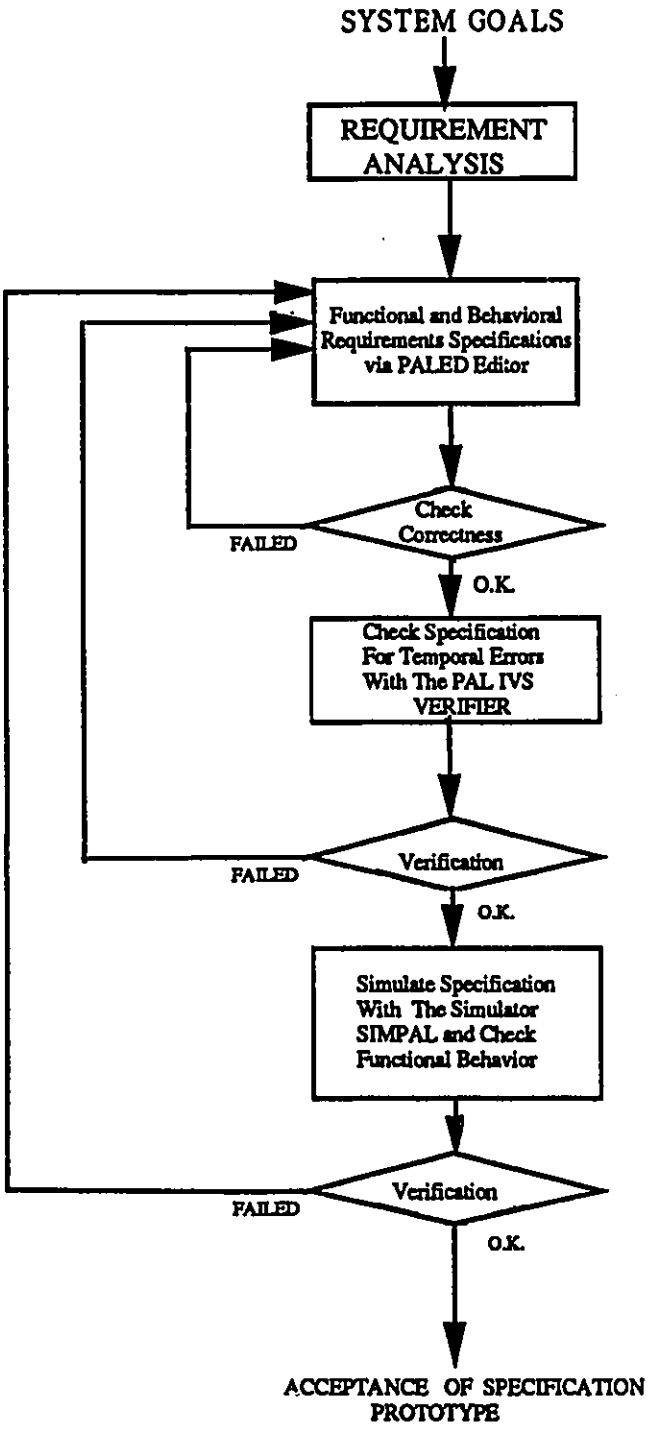

Figure 4.2: The Requirements Specification Cycle

4.1.2 SIMPAL and the Specification Prototype

The last step of the Requirements Specification cycle is the simulation of the specification in order to check its correct functional behavior.

SIMPAL is able to execute any processes written in PAL; it has effective display capabilities that make it very suitable for visualizing what happens in the system as time passes. The control flow of processes and the effects of processes on the environment can be easily observed and checked.

Using SIMPAL the user is able to define all system's events and he is able to activate them individually to analyse specific responses. A simulation of the environment allows the implementation of real world sensors and actuators, which affect and are affected by the processes. This simulation closes the loop between the control entity and the controlled entity, showing the effects of one on the other.

Checking the specification consists of running the system and seeing how it performs, how the processes interact and what are their effects on the environment. At this point, one does not check specific timing or performance requirements and does not assume any specific implementation platform. The Specification Prototype is used assuming an infinite number of resources. The logical and functional behavior can be checked and the user can see if this is really what he was expecting and compare the desired with the observed behavior. If the observed behavior is not acceptable, the requirements were incorrectly specified and changes to the specifications are needed.

It is also possible to detect the presence of some temporal errors using SIMPAL. The simulator can act as a limited temporal error checker which will make apparent to the user when a process hangs-up and becomes permanently blocked. SIMPAL indicates the presence of these errors in the system by making them visible to the user.

When the specification successfully passes through all phases of the Specification Prototype cycle it is considered complete and can then be used and refined in the Design Specification phase, where all the constraints will be checked.

4.2 PAL-Based Design Prototype

In this section, a methodology for building the Design Prototype is presented. Basically, the chapter describes several steps that can be taken to obtain such a prototype and which should be used to test the prototype with the simulator SIMPAL. These tests will determine efficient operating parameters and other aspects of the final implementation.

In order to use the simulator to evaluate timing characteristics of a proposed system, it is necessary to know how to derive the execution time of a process from the individual execution times of its elements (this is explained in Appendix B, where several examples are shown). Since a response can trigger not only one but many processes, its expected completion time will be that of the longest process it triggers. Obviously, these processes must be designed so that the longest execution time among them is shorter than the required response time. This is the first obvious requirement for the feasibility of the solution.

The methodology presented here consists of two steps:

1. An initial estimation of the number of slaves

2. Checking the system's responsiveness

The evaluation and testing a system for all possible operating parameters would require too many different situations to be analysed and too many simulations to be performed. Depending on the case, the process may be practically impossible or at least too extensive to be practically feasible. A simple but consistent methodology is proposed here, based on the principle of "divide and conquer", where the generation of the Design Prototype is divided into the two mentioned steps. Whenever possible, examples of the various suggested tests are presented, usually in a graphical format. These must be analysed carefully since results that are obtained for a specific problem are generally not valid for a different application.

4.2.1 The First Step: Determining The Number Of Slaves

This first step consists of an initial estimation of the number of slaves that will be necessary to implement a specified solution. It is basically a rough analysis of the system's responsiveness to events.

Any "reasonable" guess will provide satisfactory results. For instance, given an event-driven system specified in PAL, the following are two of the approaches can be taken in this first step:

1. The Worst Case Instantaneous Load

   For this approach, a completely deterministic system will be simulated. Using any of the mechanisms provided by SIMPAL, the system must be configured in such a way that each event will occur only once during each simulation (multiple occurrences of an event are not allowed); in addition, all different events in the system must be made to occur simultaneously or within a narrow interval of time. Being a deterministic system, the results would be exactly the same if we performed an analysis based on multiple occurrences of the events (it would just take more time for the simulations).

   Although this approach might look too pessimistic, it is not the worst case with respect to event responsiveness. The reason is that it does not take into consideration the dynamic occurrence of events. In a real system, new instances of events may arrive before the previous ones were serviced. In this case, the response times may increase considerably because events will spend more time in queues while waiting to be serviced.

   The advantage of this approach is its simplicity. It is fast and easy to use and generates reasonable results.

2. Individual Analysis of Events

   In this second approach, the events are divided into periodic and sporadic (probabilistic). For both of these categories, each event response is analysed individually in order to obtain the number of resources (slaves) needed to satisfy its timing requirements.

Such analysis is simple in the case of sporadic events. An event is generated and only the processes triggered by this event start execution. By varying the number of slaves available, one can determine the number needed to satisfy the requirements. A number Ns is obtained (number of slaves needed for that sporadic event).

In the case of periodic events, the analysis will be based on simulations of a relatively large duration. The analysis of the results depend on whether the system can afford to miss events or not. In the former case, only the average response time should be studied. In the latter case, the event queue can never contain more than one element (periodic events that are queued will eventually make the queue overflow). For each periodic event, a number Np is obtained (number of slaves needed for a periodic event).

An initial estimate of the total number of slaves required, Nt, will then be given by:

$$Nt = \sum_{i=1}^{i=last\ periodic\ event} Np_i + \sum_{j=1}^{j=last\ sporadic\ event} Ns_j$$

In both approaches discussed above, the interaction between responses to different events is not being considered. This interactions can actually improve the performance of the real system (by reducing the time processes wait for messages, by releasing resources earlier, etc.).

NOTE: *In some systems it is possible to have processes that are not explicitly triggered by events. They exist as soon as the system is turned on and, in general, they simply monitor an aspect of the environment. This type of processes must be included in the simulated system since they use the resources that could otherwise be available to other responses. These processes can in general be mapped into responses to periodic events and then analysed like the other events; this is not really necessary because they may not have specific timing requirements.*

The steps to follow in this first phase are:

1. Write the processes in PAL.

2. Enter the system in SIMPAL, configuring the environment (where the events are defined) according to one of the approaches described above.

3. Run several simulations varying the number of slaves.

4. Analyse and/or plot the results and choose an initial number of slaves for the subsequent analysis in the second step.

We describe these steps in the next pages.

4.2.1.1 Writing The Processes In PAL

This step is straightforward and presents nothing new so far. To avoid any possible blocking caused by a WTE primitive (wait for event) it should be either eliminated, replaced by delays or the event it is waiting for should be made an independent one. This is necessary because of our assumption that events will occur only once during the simulations (otherwise we may end up blocking the execution of a thread).

4.2.1.2 Entering the system in SIMPAL

Entering the system in SIMPAL is also straightforward. The environment parameters can be entered using the utility Genenv. If the first approach is taken, all events, with the exception of those specifically used for the WTE primitives (as explained earlier), should be made to occur only once during the simulation time, using the mechanisms provided by SIMPAL. Indirect and Event variables may be used in the normal manner. Delayed events will not be necessary in most cases. If the second approach is taken, the events must be configured so that they can be analysed individually.

The processes can be entered using the Editor or can be entered directly in pseudo-code in the SIM_PROC file.

4.2.1.3 Running The Simulations

The user can run a different simulation for each number of slaves tried or use the simulator's batch mode, which will run several simulations at once (transparently to the user). This second approach is probably much more efficient. However, the user may still opt to run each simulation separately in order to observe what is going on in the system.

The number of slaves should be varied from 1 to the maximum possible, according to the expected limits. These limits may be imposed by constraints such as cost and weight, which will restrict the maximum number of processors that can be used.

As stated earlier, the whole process can be automated using a batch file. For example, suppose the processes expressions are written in the file SIM_PROC.n, the events parameters in the corresponding file SIM_ENV.n and we want to run simulations using from 1 to 12 slaves under the FCFS scheduling algorithm. A batch file can be written and the results will be obtained in 12 different statistics files, which are:

STAT.n10, STAT.n20, ..., STAT.n90, STAT.nA0, STAT.nB0, STAT.nC0[1]

These statistics files can then be analysed in order to check the resulting parameters. The user may either read each file and look for some specific results or he may use a simple program, such as Gather, which will open all these files, look for specific data and then gather all data in a single file, which can be read by the user himself or by some other application program[2] Alternately, the user can write other programs to handle and process the statistics in order to meet his specific requirements. A suggested tool is one that reads the data obtained and plots them in a graphical format. Another way would be to develop an optimization tool that will look for maximums and minimums in order to maximize performance versus requirements.

[1] In the files STAT.xyz, x represents the version loaded (the files SIM_PROC.x and SIM_ENV.x), y represents the number of slaves used and z represents the scheduling algorithm (0 for FCFS, 1 for LACTF, etc.)

[2] Gather is a small program which compiles the data obtained about: execution times of processes, delays of processes, execution times of responses, delays of complete responses, usage of slaves, ready-to-run queue handling, etc.

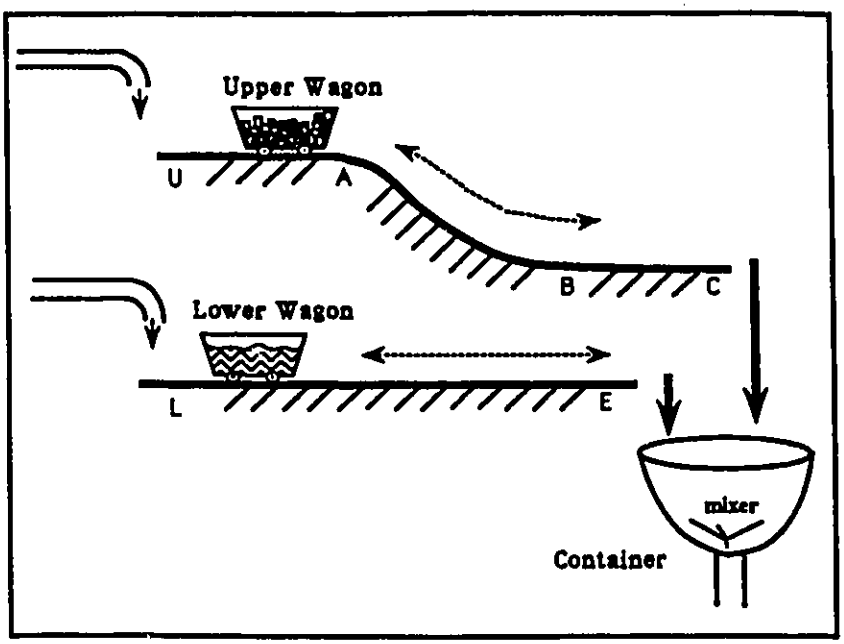

Figure 4.3: The Two Wagons Problem: Physical System

4.2.1.4 A Complete Example

A complete example is presented in this section to show how the steps of the design prototype are executed with SIMPAL. Note that for the purposes of this discussion, the meaning of the problem is irrelevant. Any other set of processes could have been chosen since the goal is simply to show how the tests can be applied. However, in order to make it more interesting to the reader, an example that has some meaningful behavior and that contains a balanced sample of the PAL primitives is used.

The example is the simple system shown in figure 4.3; its specifications, in terms of processes and activities, are given in figure 4.4. The role of the two wagons is to carry a load from the left side to the right side. Two different materials arrive at the left side at the same time and their arrival constitutes an event ("LOAD ARRIVED").

The Upper Wagon carries solid material which has to be carried all the way down to the container, in which it will be dumped. There is a downward slope in the way in which the brakes should be applied and their temperature checked for safety reasons.

Activities and Execution Times

1	Open wagon's container	10
2	Fill wagon	40
3	Warm-up wagon's engine	25
4	Close wagon's container	10
5	Go to point A	45
6	Go to point E	90
7	Turn heater on	6
8	Turn heater off	6
9	Go to point B	45
10	Go to point C	45
11	Dump wagon's contents	40
12	Goback to U	120
13	Open & close valve 1	6
14	Open & close valve 2	6
15	Dump liquid	35
16	Go back to L	70
17	Control brake's temperature	5
18	Prepare load	90
40	Generic Activity	35
41	Generic Activity	100
42	Generic Activity	45
43	Generic Activity	70
44	Generic Activity	35
45	Generic Activity	60

Process 0: UPPER WAGON

SEQ 9	9 serial elements
STP 0 (1)	Local variable 0 =1
ACT 1	Open wagon's container
ACT 2	Fill wagon
CON 2	2 parallel elements
ACT 5	Go to A
ACT 18	Prepare load
CON 2	2 parallel elements
SEQ 2	2 serial elements
ACT 9	Go to B
STP 0 (0)	Local variable 0 =0
RPT (P0=0)	Repeat until B reached
ACT 17	Check brake's temperature
ACT 10	Go to C
CON 2	2 parallel elements
ACT 11	Dump solid
ACT 1	Open wagon's container
TXM 0	Synchr.with Lower Wagon
CON 2	2 parallel elements
ACT 4	Close wagon's container
ACT 12	Go back to point U

Process 1: LOWER WAGON

SEQ 9	9 serial elements
CON 2	2 parallel elements
ACT 2	Fill wagon
ACT 7	Turn Heater on
CON 2	2 parallel elements
ACT 3	Warm-up engine
ACT 4	Close wagon's container
ACT 6	Go to E
WTM 0	Synchr. with Upper Wagon
CON 2	2 parallel lements
ACT 1	Open wagon's container
ACT 8	Turn Heater off
RDE 0 1	Read temperature sensor
CAS 2	A case
(P1>20)	If temp. > 20
(P1<=20)	If temp. <= 20
SEQ 2	2 serial elements
CON 2	2 parallel elements
ACT 13	Open & close valve 1
ACT 17	Check brake's temperature
DLY 3	Wait for 60 units of time
ACT 14	Open & Close valve 2
ACT 15	Dump liquid
ACT 16	Go bach to L

Process 2: Container controller

CON 3	3 parallel elements
SEQ 3	3 serial elements
ACT 40	
ACT 41	
ACT 42	
SEQ 2	2 serial elements
ACT 43	
ACT 44	
ACT 45	

Figure 4.4: The Two Wagons Problem: Processes and Activities

Before dumping its contents into the container, the Upper Wagon has to inform the Lower Wagon that it has arrived in the dumping point on the right side.

The Lower Wagon has a similar role, except that it carries a liquid product that has to be maintained at a specific temperature. For this purpose, there is an electric heater in the wagon. After reaching the container, the wagon has to wait until the Upper Wagon has arrived before it starts pouring the liquid, which will then be mixed in the container.

After dumping their contents, both wagons must return to the initial points on the left side in order to be ready to accept a new load. There is a timing requirement for both wagons to return.

The PAL specification of this problem (shown in figure 4.4) defines two events and three processes. The first event (event 0 - "LOAD ARRIVED") triggers the processes that control both wagons. Since the wagons have time limits to return to their initial position, the response to event 0 has a well defined deadline or due date (900 units of time in this case). The second event (event 1 - "START MIXER") triggers a process that controls the container and its response also has a deadline (1000 units of time). The three processes are: one for the Upper Wagon, one for the Lower Wagon and one for the container controller. The last is just presented here and is not explained (its purpose is merely to increase the load of the system).

Note 1: *There is no intelligence in the sensors and actuators in this problem; the slave processors are assumed busy all the time that they are executing an activity*

Note 2:*As stated earlier, in PAL a single event can trigger more than one concurrent process, which do not necessarily start at the same time. Depending on the system, these processes may be considered as independent responses or they may be considered as a single response. SIMPAL considers a response complete only when all triggered processes are finished. In the example, the response to event 0 is made of two processes and will be considered finished only after both processes finish execution.*

4.2.1.5 Analysing the results

Two important measures of real-time systems are the processes execution times and the lateness of complete responses[3]. Results of the two wagons example are shown in figure 4.5 and figure 4.6 and are explained below:

1. Figure 4.5: Processes Execution Times

   The curves in graph 1 show the behavior of the system with respect to the execution times of the processes, as a function of the number of slaves, assuming the FCFS scheduling algorithm.

   As expected, as the number of slaves increases, the execution time decreases. However, there is a point at which the curves stabilize and no improvement is achieved (in this case, when the number of slaves is 6). This is because there are already enough slaves to handle all possible concurrencies in the system at all times. At this point, more slaves added to the system will just be sitting idle and will never be used (unless there is a failure in one of the slaves). The rate of decrease is much larger in the first region of the curves, where the number of slaves is very low. The reason is that most of the concurrency in this problem is of level two (usually only two concurrent activities are being executed).

2. Figure 4.6: Lateness of Complete Responses

   These curves show how late (in %) the responses finished after their due date (deadline). As it happened with figure 4.5, no improvement is achieved (for event 0, the lateness stabilized at -48% approximately; for event 1, the lateness stabilized at -54% approximately), when 6 or more slaves are used. Between 5 and 6 slaves, the improvement is very small; between 4 and 5 it is a bit larger; finally, the delays are reduced dramatically when the number of slaves is increased from 1 to 2 and then to 3.

The approach used to determine an initial number of slaves will be based on the type of the system involved. If we are dealing with hard real-time constraints,

[3] A "complete response time" includes the time executing the response and the time the event spent in the queue.

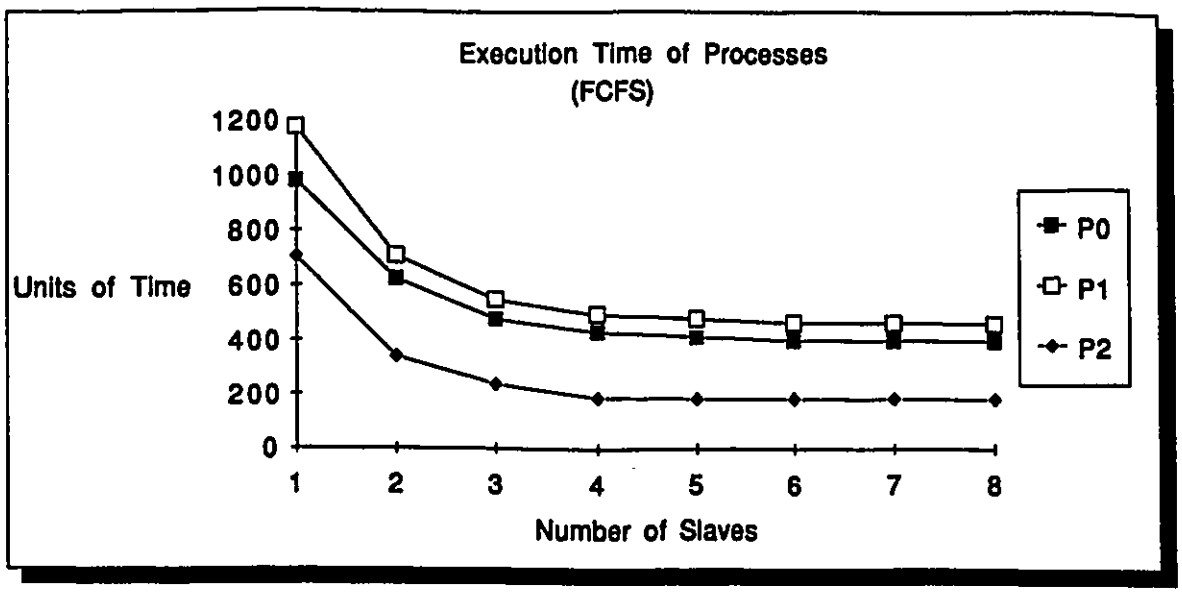

Figure 4.5: Processes Execution Times

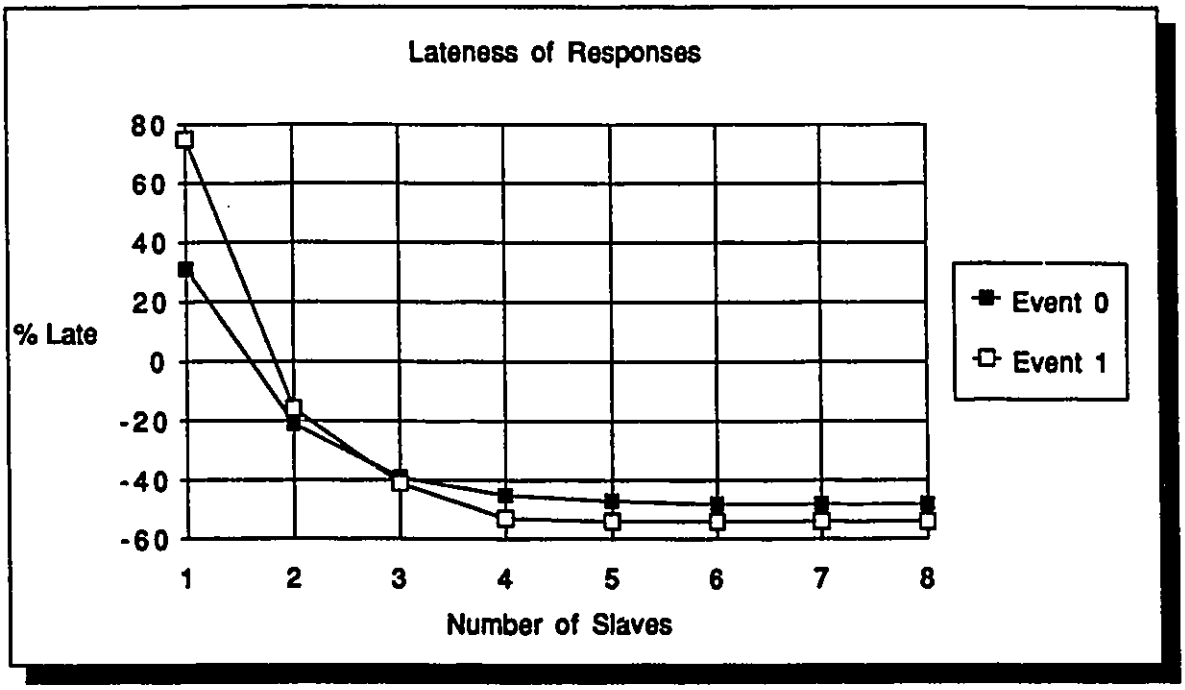

Figure 4.6: Average Lateness of Complete Responses

then graphs that show the responses' lateness such as the one in figure 4.6 should be carefully studied. All cases whose latenesses have positive values should be discarded, which in this case means that the number of slaves should be greater than or equal to 2 (the first negative delay). The real number chosen will depend on the safety margin to be adopted. If a safety margin of 40% is considered a reasonable value, then 4 or 5 slaves should be used. The decision is up to the user and this is when his experience becomes very important. Also, in this case we took the worst case approach. It is expected that the results will be better when events are asynchronous and do not occur simultaneously.

In the case of soft real-time constraints, the average response times shown in figure 4.6 are important. In this type of system, some delays are tolerable; for instance, a delay of 10% may be acceptable once in a while. In order to measure the percentage of responses that are that late, it is necessary to run several simulations and obtain appropriate statistics. This is a more complex analysis and it is considered in the second step (next section). Note that figure 4.6 indicates the performance achieved when no queueing is involved; if events were to be queued, their average response time may increase drastically since events may spend some time waiting in the queue.

The measure of how elastic a "soft real-time" constraint is can be very subjective; in addition, it is certainly dependent on the specific application. It is up to the user to evaluate and choose the correct performance. He may also find it useful to study the responses' delays after their *expected* completion time and/or the individual execution times of the processes participant in the responses (graphs such as those in figure 4.5); it may be acceptable for specific process to suffer a larger delay than another process of the same response.

If queueing must be considered, then safety margins should be adopted when analysing the results. The values adopted need not be too rigid since the results will be checked again in step 2. In the case of the two wagons example, if it is assumed that events cannot be missed and, therefore, a queue must be used to store them, it is reasonable to expect that the average response times will be greater than those shown in the previous figures. Adopting a safety margin of 40%, we obtain from figure 4.5

a required number of slaves equal to 4. So this would be the initial choice for this real-time system.

Other parameters can also be checked, such as slave usage, ready-to-run queue handling, etc..., but it is preferable to analyse them in the next step of the design phase.

4.2.2 The Second Step: Checking System Responsiveness

In the first step a rough estimate on the number of slaves needed was made based either upon an "instantaneous load" assumption or upon the resource requirements of individual responses. This approach did not take into consideration the dynamic occurrence of events (that new instances of events may occur before the responses to the previous occurrences are completed) or other uncertainties in the system, such as slave failures. These aspects are analyzed in this second step.

4.2.2.1 Checking The Uncertainties: Event Responsiveness

The nondeterministic factors of a system must be taken into consideration because they may greatly affect its behavior and performance. In this section we pay particular attention to the event responsiveness as a function of the event's frequency, queueing or type (other uncertainties related to reliability and fault-tolerance are dealt with in the next chapter).

This section shows how to check the ability of a system to respond to dynamic events arrivals; that is, how fast the events can arrive and still be correctly serviced. In the simulator SIMPAL there are three types of events: cyclic events, probabilistic events and time-interval events. The tests may be performed only with the first two types since the third one is more or less a combination of them.

For probabilistic events, the test is to check the performance achieved as a function of the probability of the event, increasing its value between the expected minimum and maximum values. As for cyclic events, the period should also be varied between the minimum and maximum expected values. Note that the user may know the exact event probability or cycle. Obviously, in such cases the number of tests to be

performed will be substantially reduced; the more accurate the user's knowledge, the less the number of tests to perform.

The parameters to be checked depend on the type of system under investigation. A real-time oriented system would emphasize the study of time related characteristics, such as response times and delays with respect to strict deadlines. Depending on the individual application, the study of queue sizes may be important, as may be the percentage of events that were serviced, missed, queued, the maximum time spent in a queue, resource utilization, etc...

It will be briefly demonstrated how such tests can be executed on the two wagons example introduced earlier. Obviously, these tests will not be exactly the same with any other system. Each case is different and must be analysed considering its own particularities, but a general guideline that can be followed by the application engineer and modified as required will be presented. The tests are divided into those with probabilistic events and those with cyclic events.

1. Tests With Probabilistic Events

   In this case, the probability of occurrence of event 0 (which triggers the processes that control the wagons) was varied, while maintaining event 1 as cyclic (this event triggering the process that controls the container). Four slave processors will be used, as determined earlier in the first step.

   In this section we present the results of the example when the probability of event 0 is varied from 0.01% to 0.25%, which corresponds to an event happening on the average between 10,000 and 400 units of time respectively. These probabilities were chosen so that they would be close to the expected execution time of the response, which is around 450 units of time. Event 1 was fixed as periodic with a cycle of 1,000 units of time.

   Figures 4.7 and 4.8 show the latenesses (delays after deadline) obtained for the responses to event 0 and to event 1 respectively. Both the average and the maximum latenesses are shown. They represent the average execution time of all completed responses during the simulation and the maximum values obtained

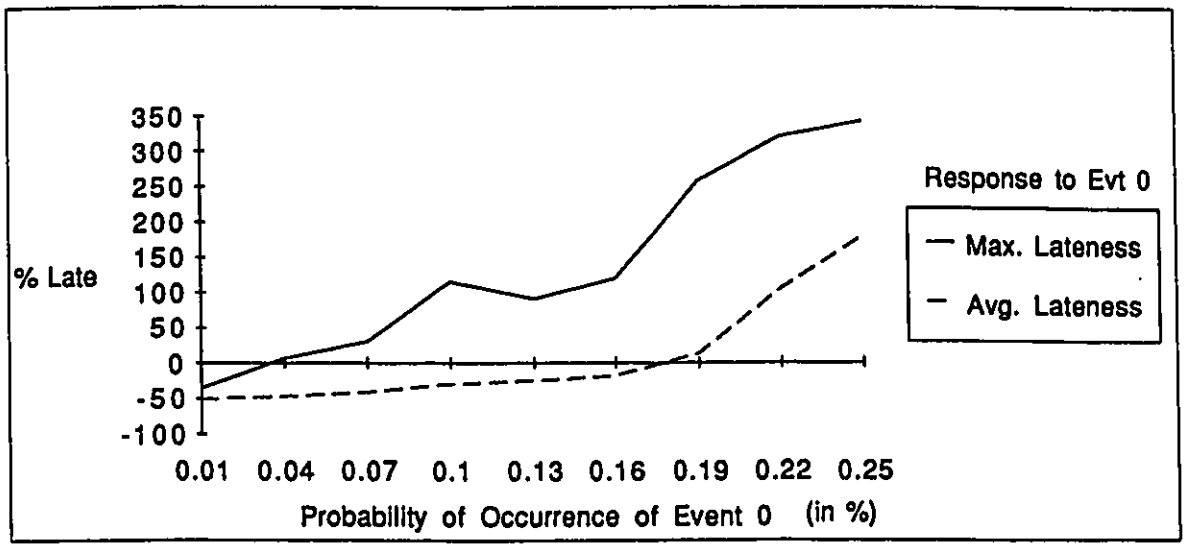

Figure 4.7: Maximum and Average Lateness of Response to Event 0

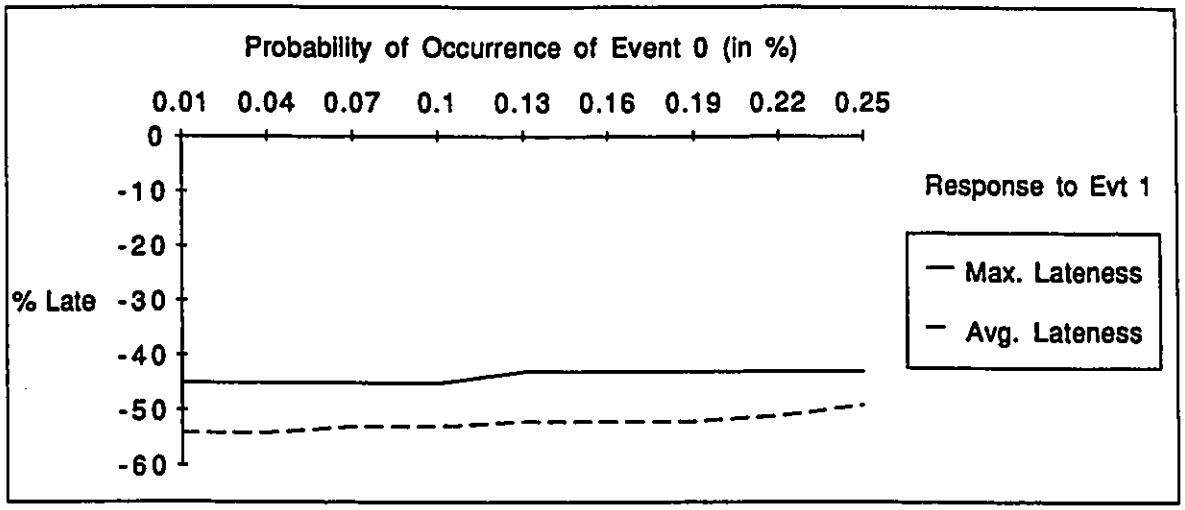

Figure 4.8: Maximum and Average Lateness of Response to Event 1

during the simulation. These numbers were obtained by running 10 simulations lasting 100,000 units of time for each probability shown. The "maximum" value is the greatest delay among these 10 simulations.

Note:*It is preferred to run a few different simulations instead of a single longer one to improve the randomness of the experiment. At each new run, a different seed based on the current time is used to start the pseudo-random number generator.*

For event 0, up to a probability of 0.18% the average lateness is still negative (meaning that on the average the responses are still within their deadline). As a result, a soft-real time system may be able to work well up to this probability. However, it can be seen that the maximum lateness becomes positive (the response is late) after a probability of only 0.04% approximately. A hard real-time system may not tolerate probabilities greater than this, although it should be noted that since the arrival of events is a stochastic variable, it cannot be guaranteed that below this value the maximum lateness will never be positive. Even though we obtained some values for the maximum lateness, it is possible that hundreds of events will arrive consecutively in a very short period of time, in which case the obtained values would be very different. This is not very realistic in our example, which involves a physical system and the events represent the arrival of a load. In such cases, one may be able to guarantee that an event will occur between a minimum and a maximum amount of time, which represent the physical limitations of the system components. This kind of event can also be simulated using SIMPAL just by defining it as a time-interval event.

On the other hand, it can be seen that the response to event 1 is not affected very much by the probability of event 0. One could be tempted to guess the opposite since a higher probability for event 0 should generate a higher usage of the resources and consequently, a lesser availability of them for the response to event 1. This is not the case in this example because of the low concurrency level; there is usually a slave free to execute the response to event 1.

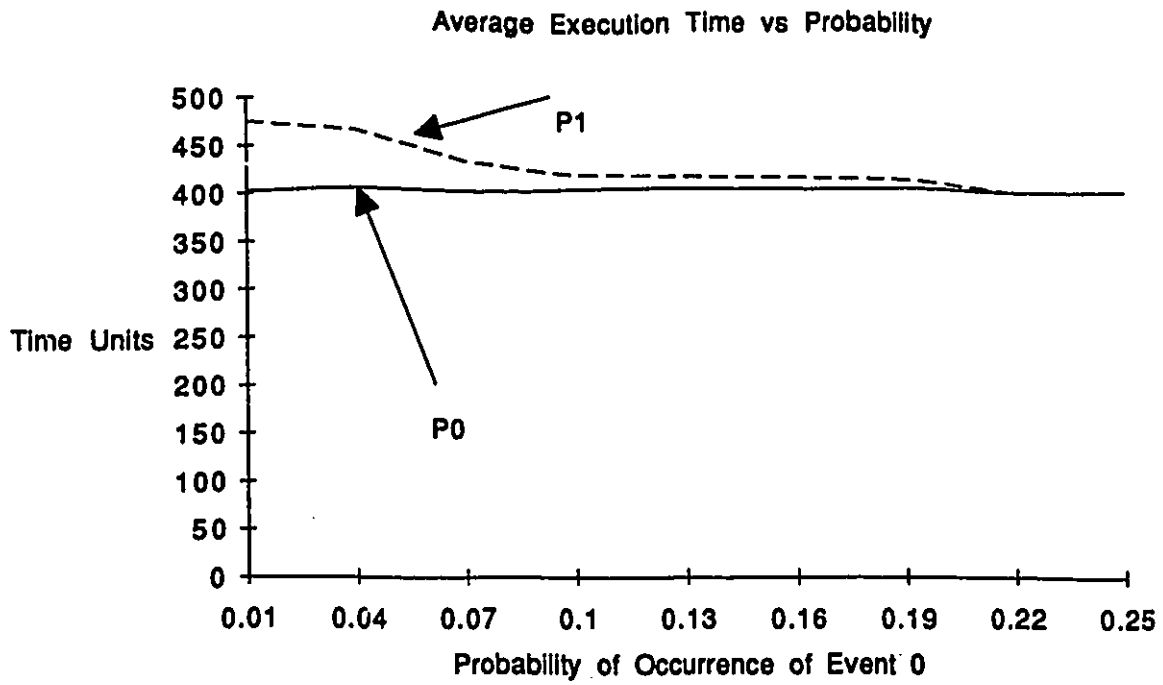

Figure 4.9: Processes' Execution Times

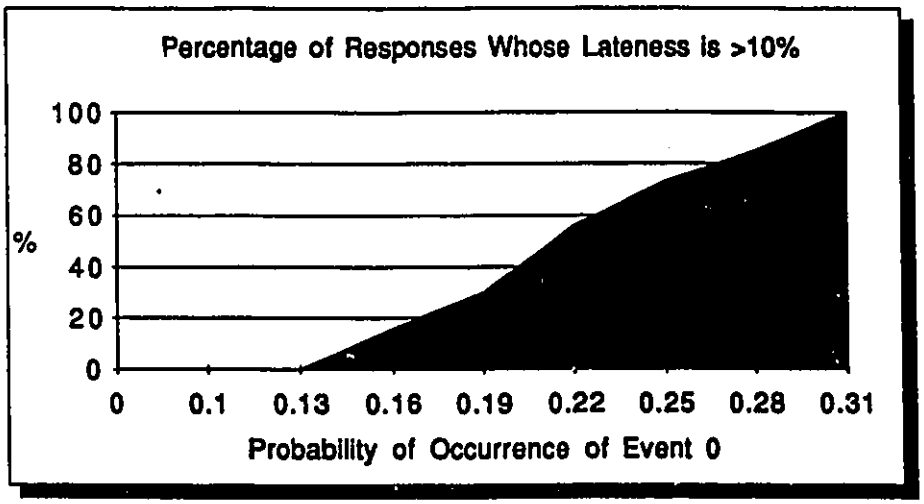

Figure 4.10: Percentage of Responses More Than 10% Late After Deadline

Figure 4.9 shows some interesting results regarding the execution times of process 0 and process 1 as a function of the probability of event 0. The execution time of process 0 is not affected but the performance of process 1 is improved as the probability increases. This is due to the fact that process 1 waits for a message from process 0. The more process 0 occurs, the lesser time process 1 is going to be blocked waiting for a message. Note that this figure does not indicate time spent in the queue by the events (which is considered part of the **response** time, not of the processes execution times).

For soft-real time systems, a better analysis may be obtained by generating statistics on the number of times that a response exceeded a certain delay. For example, it may be acceptable if a response is 10% late (+10% of lateness) for 20% of the responses. Figure 4.10 shows the percentage of times that the responses to event 0 were late more than 10%. It can be concluded that, for this example, the performance would be acceptable only for probabilities less than 0.17%.

Figures 4.11 and 4.12 show the maximum queue sizes and the percentage of events that were missed. Depending on the particular application, it may or may not be acceptable to miss a few events. If events cannot be missed, a solution would be to increase the queue capacity at the expense of a greater average response time. Since the maximum chosen queue size was 10, this represents the saturation value of figure 4.11. The percentage of events missed when the maximum queue size was exceeded is shown in figure 4.12.

If the queue size were modified, the tests should be performed again. A new analysis of the response times must be studied carefully since these are particularly affected by the queue size.

2. The Tests With Periodic Events

The previous tests work very well with probabilistic events (those that occur with a certain rate or probability). Periodic events can be studied in a similar way, varying the cycle (period) of the events instead of their probability.

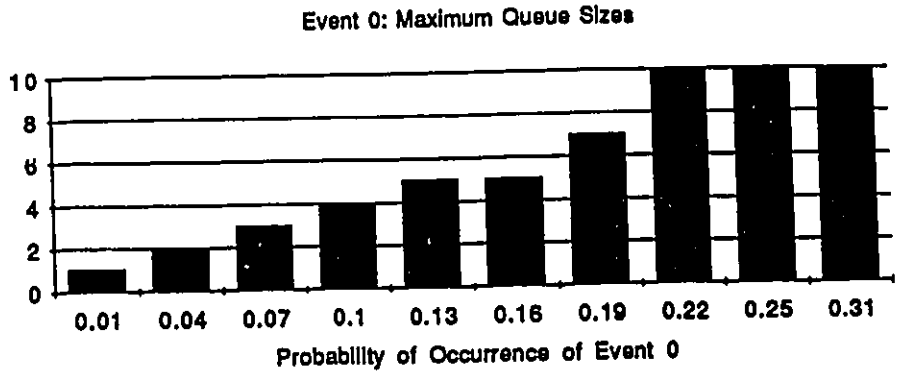

Figure 4.11: Event 0: Queue Sizes

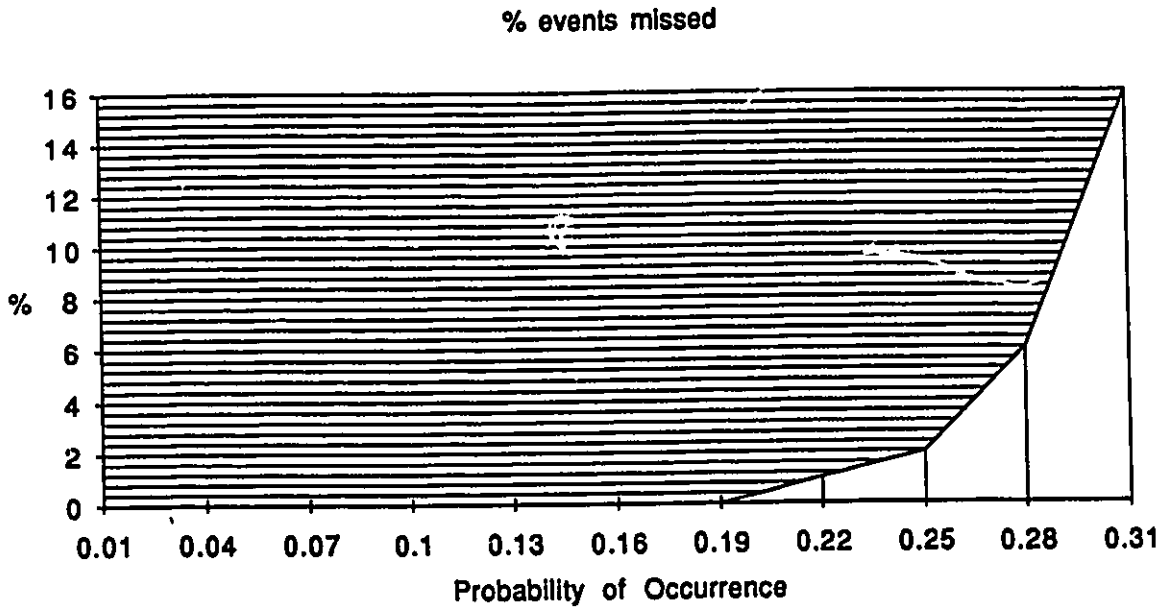

Figure 4.12: Event 0: Percentage of Events Missed

The two wagons example was simulated with event 0 configured as cyclic, with its cycle ranging from 200 to 500 units of time (values close to the expected execution time of the responses). The maximum execution time of a response is in the 400 range, so the results for cycles greater than 500 are the same as those for 500 (all processes triggered by an event will have already finished when a new event occurs).

Figure 4.13a shows the average and maximum latenesses obtained for responses to events 0 and 1, when the cycle of event 0 ranges from 200 to 500 units of time. It is clear that the system responds well for cycles as small as approximately 400 units of time. For events faster than that, the delays in the response to event 0 may become unacceptable depending on the requirements.

Having determined the critical region of operation, the user can now proceed to a more detailed analysis in this range. Figure 4.13b shows the latenesses of the responses to event 0 within the 360–400 range. It can be seen that the point at which the latenesses become late after their deadline is close to 390 units of time.

As a conclusion, we see that the system responds well for cycles up to 390 units of time, which corresponds to a frequency of 0.26%. This means that the responsiveness is better in this case than in the case of probabilistic events, in which the performance was considered acceptable up to a probability of approximately 0.18%. This is explained by the higher determinism existent in periodic events. The probabilistic case has to cope with nondeterministic factors such as many events happening in a very short time, in spite of their average rate of arrival being much larger.

Figure 4.14 shows the queue sizes for cycles ranging from 350 to 400 units of time. With a cycle of 350 units of time, the maximum queue size is 10, which means that the queue has likely overflowed and some events were missed. If this is a problem, an "events missed" curve can also be plotted and analysed.

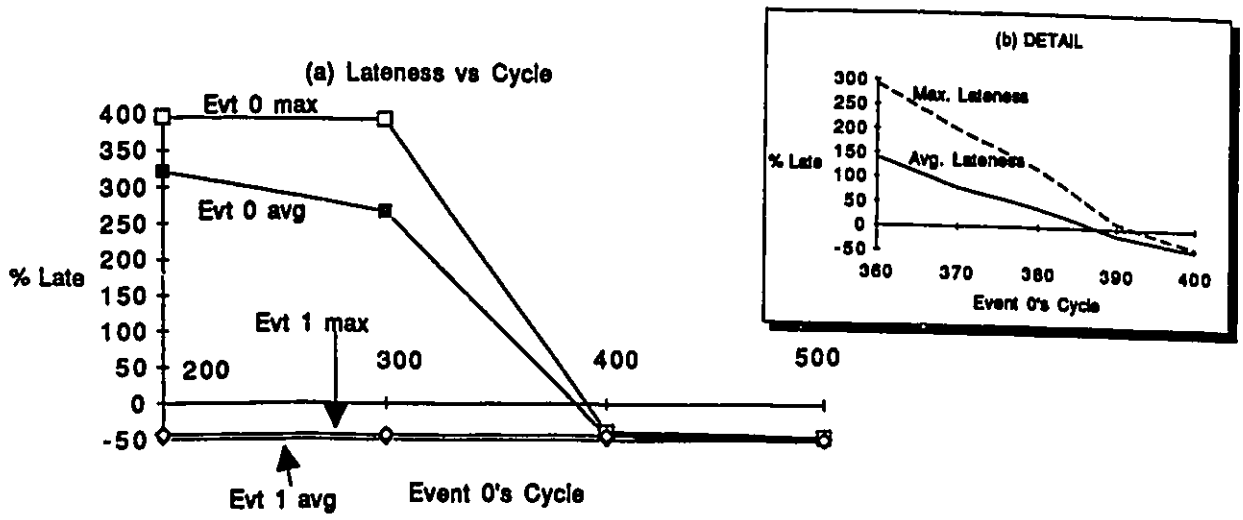

Figure 4.13: Lateness of Responses (a) and Detail (b)

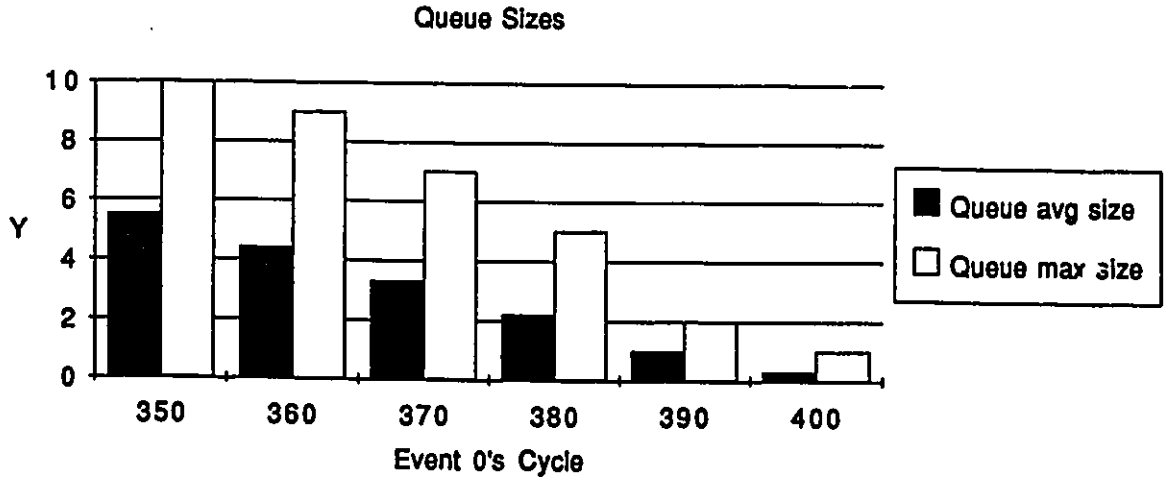

Figure 4.14: Queue Sizes

4.2.2.2 Fine Tuning The System

In the previous section, it was discussed how the uncertainties can be tested using a PAL Design Prototype. In this section, further tests will be shown, specifically those that can be performed to adjust the system with respect to three closely inter-related aspects:

- The scheduling algorithm
- The performance as a function of the capabilities (performance) of the slaves
- Resource usage efficiency

The execution of activities in multiactivity systems can be divided into three phases. First, the activities in the ready-to-run queue are ordered according to the scheduling scheme, which is based upon priorities and/or constraints associated to the various responses. Second, free slaves are assigned to the ordered activities in a sequential manner (the activities with the highest priorities are assigned slaves first). Finally, the activities are executed by the slaves.

Having distinguished scheduling from slave-to-activity assignment, we can now proceed to a detailed discussion of the different scheduling schemes that can be used, the types of processors and the resource usage efficiency (in systems that do not have hard real-time constraints). Each of these will now be examined individually.

1. Maximizing Performance With Scheduling Schemes

   Using SIMPAL, it is possible to simulate the system under several scheduling schemes (or algorithms). It is possible to simulate the system under the different operating conditions expected, and based on this, it is possible to predict which will be the best method to use in the final implementation. These conditions may vary significantly and for each of them a different scheduling algorithm may be best suited. In the actual implementation the executive may use dynamic scheduling, switching schemes based on the previous knowledge acquired in the simulations.

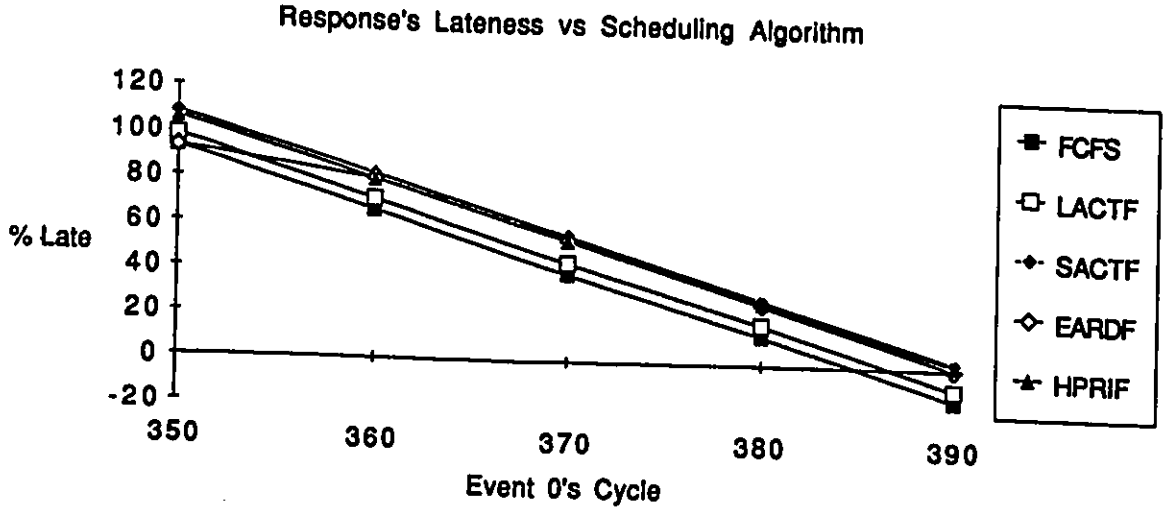

Figure 4.15: Lateness of Responses to Event 0

The following scheduling schemes were implemented in SIMPAL: FCFS-First Come First Serve, SACTF-Shortest Activity First, LACTF-Longest Activity First, the slack-time (or laxity) based algorithms (LDSLF, LSSLF) and HPRIF-Highest Priority First and EARDF-Earliest Response Due First (when there is more than one event).

The latenesses of the responses to event 0 (assumed to be cyclic) are shown in figure 4.15 for several schemes used in the two wagons example. It can be seen that for cycles less than 350 units of time, the FCFS (First Come First Serve) and EARDF (Earliest Response Due First) algorithms are the best. For cycles greater than 350, FCFS becomes the sole best algorithm.

This example does not show large differences between the different schemes but these can become quite significant, as can be seen in [Laced], which presents more detailed discussions on scheduling in multiactivity systems.

2. Varying The Slaves Capabilities

The purpose of this section is to show some interesting aspects related to the different slaves that can be used in a multiactivity system. Particularly, we

are interested in the assignment of slaves to ready-to-run activities after the scheduling is done. There are two aspects to consider when assigning a slave to an activity. First, the system has to check whether the activity requires a special purpose processor or not; this is the case where activities have special requirements which can only be satisfied by specific processors (DSP's, for instance). In this case, a free processor (with the required capabilities) must be assigned right away. If the activity does not require a special purpose processor, then a slave can be assigned in a number of ways; in general, in a system with many free slaves there is a choice of which slave should be assigned to the activity. Particularly, there are two interesting approaches for this choice: Load Balancing and Sequential Allocation.

- Slave-To-Activity Assignment for Load Balancing

  In this case, the assignment of a general purpose slave to a ready-to-run activity tries to achieve a good distribution of work among all slaves. The objective is to make all slaves have more or less the same amount of work. One of the ways to achieve this is to divide the total number of activities into two, three or even more groups. The slaves are also divided into these groups and each group can only execute its respective group of activities.

  The advantage of this approach is that the amount of memory required by the slaves is reduced drastically. Each activity has its own execution code, which always resides in the slaves' memory. By cutting the number of activities that the slave has to execute, the amount of memory it requires is also reduced. As a result, simpler and cheaper processors can be used. The costs are reduced and, in addition, some good secondary effects are obtained, such as an improved reliability.

- Sequential Slave-To-Activity Assignment

  In this case, it is assumed that all slaves can execute all activities. The choice of slave is done by taking the first free slave and assigning it to the activity. This is done in a sequential order, starting from the slave whose

"Id" is 0, followed by 1, 2, ..., until the last slave in the system or until there are no more ready-run activities.

Thus, slave #0 will always have a utilization (percentage of time that is busy executing an activity) greater than or equal to slave #1. The same applies for slave #1 over #2, slave #2 over #3, and so on. A great improvement in the overall system's performance can be obtained just by improving the capacity (performance) of either a single or a few slaves (obviously the ones with the lowest order "Id" numbers).

Figure 4.16 shows the slave usage curve of the two wagons example implemented with four slaves (with event 0 configured as cyclic with a period of 390 units of time).

The utilization of slave 0 is 86%, while slave 1 is 77%, slave 2 is 41% and so on. Based on these ideas it is very obvious that the faster the first slaves are, the better the overall performance will be. To check this out, SIMPAL was used to run the same example changing the speed (or the "performance factor") of slave #0. The responsiveness of the system as a function of slave 0's speed was then plotted and the curve obtained is shown in figure 4.17.

The user can now see that just by multiplying slave 0's performance by 2, the response time average (to event 0) was reduced from 1224 units of time (lateness of 36%) to 529 units of time (lateness of -41%). The result is more than double performance just by doubling the speed of one slave.

The same analysis can be performed by improving the performance of slaves 1, 2, 3, ... Obviously, by doubling the performance of all slaves we should expect more or less double overall performance for "well behaved" processes (those that do not contain too many uncertainties, such as the WTE primitive).

The merits of this approach to increasing system's performance is that it shows that at some point it is not possible to improve responsiveness just by adding an extra slave to the system, as it was seen in the examples

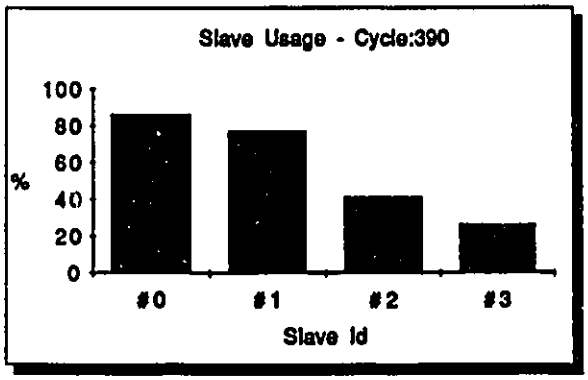

Figure 4.16: Slave Usage

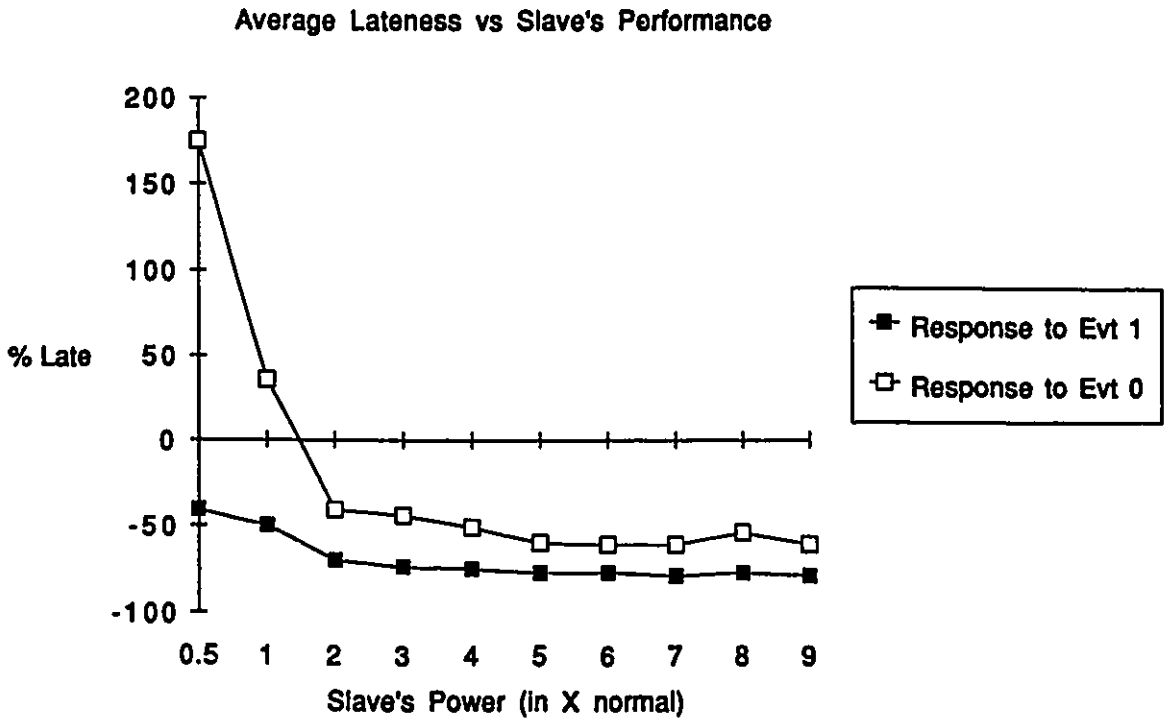

Figure 4.17: Response's Lateness versus Slave 0's Power

presented in the first step (figures 4.5 and 4.6). This is the case for mostly serial processes and when little concurrency is present. For example, in a system with 3 slaves, if there is only a maximum of 2 or 3 concurrent activities at a given time and the performance obtained is poor, no improvement will result from adding a fourth slave to the system because it will remain idle at all times.

Obviously, each case must be studied individually. The performance improvement may only be marginal by adding a faster slave: an activity's execution time may be totally dependent on the external environment or it might have rigid timing constraints (such as specific actions needing to be performed at specific times). In this case, adding a faster slave will not help.

3. Improving the Efficiency of the Slave Usage

The resource utilization of a system is sometimes a very important aspect. Although it is of a lesser importance when dealing with a real-time system, in which the major preoccupations are with the timing constraints (particularly when these are hard real-time constraints). In these systems, to meet the specified deadlines it may be necessary to add resources, even if they are to be used spuriously.

However, in other types of systems, such as soft real-time, in which a minor delay is tolerable, the efficient utilization of the resources is always a desirable characteristic. A poorly used resource translates into wasted investment and increases the overall costs.

In particular, the usage of the slaves is a parameter that is important since microprocessors and microcontrollers are relatively expensive components, not only the CPU's themselves but also all of the associated support hardware. It was seen that statistics on the utilization of the slaves are easy to obtain in multiactivity systems. The simulator SIMPAL offers this statistic after each simulation, allowing immediate checking of whether the slaves are being efficiently used or not. The percentages of utilization that SIMPAL gives are

based on the complete simulation time. The analysis of this parameter, as with any other, must be done carefully since it can be misleading in some cases. An example would be when activities are executed only once or very few times but they require special purpose processors. These processors may present a very low usage rate but they cannot just be deleted from the system; in fact, they are essential to the correct operation of the system.

Again, each case is different, and whether the information of the slave usage will be useful or not depends on the particular application. The only person that knows if slaves can be deleted from the system is the designer himself. Obviously, a good practice would be to rerun some of the tests after the slave has been taken out of the system.

4.2.3 Further Considerations On the Two Steps Approach

In the second step in our design approach, the tests are performed with an already defined number of slaves. This number may be changed during the tests; it just reflects the center of a region of possible candidates (the number of slaves) to the final solution of the problem. As an "average", it has a variance and it is quite probable that the correct number of slaves will be situated within this range. This initial assumption of the number of slaves obtained from step 1 simplifies and reduces geometrically the quantity of tests to be performed. For example, if we wanted to check the system with respect to event probabilities ranging from 1% to 25% with 1% increases at each time, for each of these using 4 scheduling algorithms, the total number of tests would be 100 if the number of slaves were already defined. If it were not, and the number of slaves under consideration ranges from 1 to 19, the total number of tests jumps up to 1,900 as opposed to a total of 176 if the job is divided into two steps (100 tests for the second step and 76 for the first step – 19 slaves with the same 4 scheduling algorithms).

The reduction in the number of tests is not only important because of CPU processing time required to execute them. Rather, it simplifies the analysis of the results.

It is possible to automate the testing process (gathering of data), but it is nearly impossible to automate the analysis and decision process, which still has to be done by the application engineer. This analysis process is a extremely costly procedure that currently no computer can perform completely automatically (a good topic for further research).

With this, it was shown that by using PAL and SIMPAL the user is able to extensively test the system since many parameters can be changed to check how the system reacts. A list of these parameters is given in the simulator's user's manual (Appendix A).

Chapter 5

Reliability, Redundancy and Communications

In the previous chapter the Specification and Design Prototypes were introduced. The major design steps to follow and the corresponding tests using the simulator SIMPAL were also outlined. The purpose of this chapter is to consider in more detail additional aspects of multiactivity systems, such as their reliability, communication overheads and the use of intelligent slaves.

5.1 Hardware Reliability in Multiactivity Systems

Reliability has been a concern since the early times of human history [Eva89] and it is a paramount concern in modern times. It may be the most important quality of an embedded system, yet there is no easy way to achieve it and to quantify it. The ideal system for any designer would be a hundred percent reliable and thus would never fail.

Reality is, however, a much different story. Failures do occur and always will. What the designers have to do is to try to avoid them the best they can and to try to cope with them when they occur.

Multiactivity systems are well suited for applications that require high reliability. In a multiactivity system that has multiple slaves, a failure in one of them may not

cause a major degradation of the system as a whole. The reason for this is the relatively high degree of independence existing among activities and the freedom in allocating them to many different processors (slaves). Such level of independence is harder to obtain in common multiprocessor-multitasking systems, in which each processor includes an operating system that supports interprocessor communications. To provide the required level of concurrency needed by embedded systems, the number of interprocess or interprocessor communications can be quite large. A failure in one of the participating processors may cause widespread propagation of errors and the subsequent blocking of processes. A recovery process in such situations may become quite complicated because in multitasking systems a failure may cause loss of functionality and coordination.

On the other hand, a centrally coordinated multiactivity systems may have many slaves and a failure in one of them will only degrade the system; the activity it was executing will simply be allocated to another slave; the synchronizations and communications among processes are handled by the central coordinator. A failure in a slave processor causes only temporary loss of functionality and **does not** cause any loss of coordination.

Centrally coordinated systems, though, have the problems of using a single central coordinator, which is potentially a major problem for two reasons: reliability and communication bottlenecks between the coordinator and the slaves. The second problem will be dealt with in the last part of this chapter. The first problem is solved by using any method for achieving a extremely reliable processor, which may involve using highly reliable components or by using a hot stand-by. This spare CPU will always have updated data and can assume control of the system at any given time. It can be argued that this solution may have a very high cost but this is not true because there is only one processor involved. The increase in the costs is concentrated in a small part of the system, making the solution highly feasible. From the work with the simulator SIMPAL and the PAL executive, it was also seen that the coordinator can be implemented with a simple modern microprocessor and its software is also simple and small. In addition, by allocating each activity execution code to a number of

slaves, it is possible to shift an activity to another slave in case of failures. However, it is still possible to use hot stand-by processors for the most critical special purpose slaves.

Therefore, centrally coordinated multiactivity systems are well suited for applications that require high reliability. Furthermore, by using the simulator SIMPAL, the reliability of a system can be easily analysed. The simulator has built-in facilities that allow the generation of hardware failures in slave processors during the simulations. The slaves are seen as black boxes in which the activity can or cannot be executed; the exact nature of the failure is unimportant.

5.1.1 Reliability Tests and SIMPAL

There are basically three purposes for the reliability tests:

1. To determine the highest failure rate at which the system is still able to satisfy the performance requirements.

   This analysis is useful because, for many systems, one can only estimate with certainty that the probability of a failure will be less than some specific value; the exact failure rate is usually not known. The standards used by industry, such as the MTBF–Mean Time Before (Between) Failures, are questionable. One of the reasons is the uncertainty generated by statements such as "the MTBF is 100,000 hours". These are hard to trust since one does not know how many components were tested to achieve these results and what kind of projection was made to make an estimate over such a long period of time. It is very unlikely that actual components were tested over that time (equivalent to 12 years!). Other arguments state that an MTBF figure is misleading when dealing with non-sequentially arranged components which have a non-constant failure rate [Mot89]. In any case, this is arguable and very subjective; plenty of literature is available on the subject.

2. To determine the number of slaves needed to meet the required constraints when the failure rate is statistically known.

This analysis is acceptable when the MTBF (or similar measure) of the processors can be accurately determined. Knowing a failure rate makes the tests very straightforward. Based on results (such as lateness of responses) which are obtained for each specified failure rate, the number of slaves should be increased if the performance is not found to be satisfactory.

3. To check how the different slave recovery times (detection, repair and reintegration) affect the behavior of the system, particularly with respect to the real-time requirements.

   The third case checks the behavior of the system when different recovery times are used. When a slave is made to fail in SIMPAL, it will become fully operational again after a certain time specified by the user. Intermittent failures can be easily generated just by making this recovery time immediate. This type of failure does not reduce the availability of slaves in the system. However, it may still significantly affect the performance since the activities in which a failure was detected have to be restarted from the beginning of their execution.

In order to appreciate the results that will be shown later, it is important to know what happens when a failure is generated in the SIMPAL environment.

The simulator has built-in capabilities to generate three types of failures:

1. It is possible to create Immediate Failures. These can be generated in any slave and at any time directly by user interaction, either in single step mode or in continuous mode (by running the simulation until the desired time, generating the failure and then resuming simulation).

2. The user can specify Random Failures. These failures occur in any slave (randomly selected) and are generated randomly at each simulator cycle according to a given probability.

3. It is also possible to introduce Cyclic Failures, which occur periodically according to a given frequency. A slave recovery time can also be specified, which can range from immediate recovery to no recovery at all.

All real-time systems must include a failure detection mechanism; they cannot afford to let a failure be unrecognized for an indefinite (or infinite) period of time. In multiactivity systems, failures can be recognized either during or at the end of the execution of an activity. Earlier failures recognition requires additional system resources (hardware and software), while recognition at the end of the activity causes worse system degradation. In SIMPAL, the user can specify these failures to be recognized either at any time during the execution of an activity or only at the end of its execution. Thus, regardless of the nature of the recovery and repair procedure for failed slaves, it is assumed that this failure detection mechanism exists and that it works in a finite length of time.

In the normal operation mode of the simulator, an activity will be restarted upon recognition of a failure. The actual simulator's procedure puts that activity back in the ready-to-run queue with the highest priority, regardless of the scheduling algorithm. The restart of an activity places an extra load in the system, which may produce some delays. The delays may affect only the process to which the activity belongs or, more likely it may affect other processes as well, since their activities would be picked up from the ready-to-run queue only after the "failed" activity. The amount of extra load and the delays produced depend on the time at which the failure was detected. If it is detected at the very beginning of the execution of an activity, the performance loss may be small and may not even be noticed. However, if it is a very lengthy activity and the failure is only detected at the end of its execution, the performance loss may become very costly. Note that it is assumed that slaves can neither recover automatically from a failure nor tell the system at which point they stopped executing an activity. The possibility of using smarter slaves exists and is an interesting subject for further study.

In addition to the delays produced by the restart of an activity, the major consequence of a slave failure is the unavailability of a slave to the system. If it was designed with very little leeway, the effects may be catastrophic in terms of responsiveness.

5.1.2 Reliability Analysis with SIMPAL

SIMPAL allows the following three main methods of failure analysis (the operational details of each of these ways are outlined in the user's manual in Appendix A):

5.1.2.1 Generation of Worst Case Conditions

The first method allows the user to create an immediate failure in any slave (at any time). This ability to generate a failure on a slave (and its recovery) at run time enables the creation of worst case conditions. For instance, it is possible to systematically generate a failure in a specific slave that is executing a critical activity. By doing so, the system will be highly affected and very long execution and response times will result.

5.1.2.2 Analysing Recovery Times

The second approach allows the user to study the effects of different recovery times on the behavior of the system. This is important because in real systems there must be (and usually there is) recovery procedures which include the repair of failed slaves. It is specially useful for analysing how well a system meets the real-time constraints when different failure rates and recovery times occur. By making the recovery time immediate, this feature allows the study of temporal (intermittent) errors which may occur occasionally anywhere in the hardware. An example of this type of failure are electromagnetic interferences, which sometimes modify the value of a single bit without producing a component failure.

5.1.2.3 A Figure of Degradation

In this approach, slave failures are generated randomly with given probabilities and once a processor has failed, it will remain failed throughout the rest of the simulation. In this case, a measure of the system's degradation can be defined. It is based on the instant of the failure occurrences (in percentage of the complete simulation time) and represents the effective number of slaves over the simulation. This is

equivalent to having a partial number of slaves available. Mathematically, the Figure of Degradation Qn is given by:

$Qn = N - \Sigma(1 - \frac{Tf}{T})$ (over all failed slaves)

where,
N is the initial number of slaves
Tf is the time that the failure occurred
T is the total simulation time

The Figure of Degradation Qn gives the effective number of slaves of the system. For instance, if in a simulation lasting 10,000 units of time and using four slaves, a slave failed after 1,000 units of time, then the Figure of Degradation is given by:

$Qn = 4 - (1 - \frac{1,000}{10,000})$
$Qn = 3.1$

That is, the effective number of slaves in the system is 3.1 instead of 4.

5.1.3 Testing The Two Wagons Example

In this section we first test the two wagons example with respect to intermittent errors. These tests assume that errors occur sporadically in the execution of an activity. As stated earlier, there is a failure detection procedure and a recovery and reinstatement procedure for the slave (so that the slave will **not** remain failed throughout the rest of the simulation). This is opposed to permanent (or semi-permanent) errors, in which a major failure occurs in the hardware making a resource unavailable for a long time before it is brought back to operating conditions.

5.1.3.1 Known Failure Rate

Figure 5.1 indicates the latenesses obtained over a long execution time (100,000 units of time) when the failure rate of slave 1 is known to be 0.5%. In this test, event 0

was configured as periodic with a cycle ranging from 390 to 440 units of time (the expected operating conditions).

It can be seen that for this failure rate the results are unacceptable when four slaves are used and when the event cycles are shorter (faster) than 440 units of time (approximately). Therefore, at this level of failures, a cycle of 440 units of times or more may be considered safe. For faster events one possible solution is the addition of an extra slave, which can also be seen in the same figure. The addition of an extra slave does improve the system's performance in terms of lateness of the responses and these may now be considered acceptable for event cycles of up to 400 units of time (approximately), therefore 40 units of time faster than with 4 slaves.

If it were still not satisfactory, other approaches would have to be tried such as improving the performance of one or more slaves, decreasing the event's frequency or using redundancy.

Another important aspect to consider is the event's maximum queue size, especially in those systems that cannot afford to miss events. In these cases, any long enough simulation that shows a maximum queue size larger that 1 means that eventually the queue is going to blow up. A queue size larger than 1 represents the existence of a composition of events that caused it and it is not possible to guarantee that it will not be repeated. In the previous example, the queue sizes obtained after a simulation of 100,000 units of time are shown in figure 5.2 for systems using four and five slaves.

These results are consistent with the previous choice of a cycle of 440 as a safe one for event 0. An example that analysing only the lateness graph can be misleading is when the system has 5 slaves and the event cycle is 400 units of time. The lateness is still negative, that is, the event is still being responded to before the deadline, but its maximum queue size is 2.

A decision on how long the simulation should run and the type of analysis to be used has to be based on each specific application.

The same type of tests can be applied to probabilistic events by varying the events' probabilities instead of their cycles.

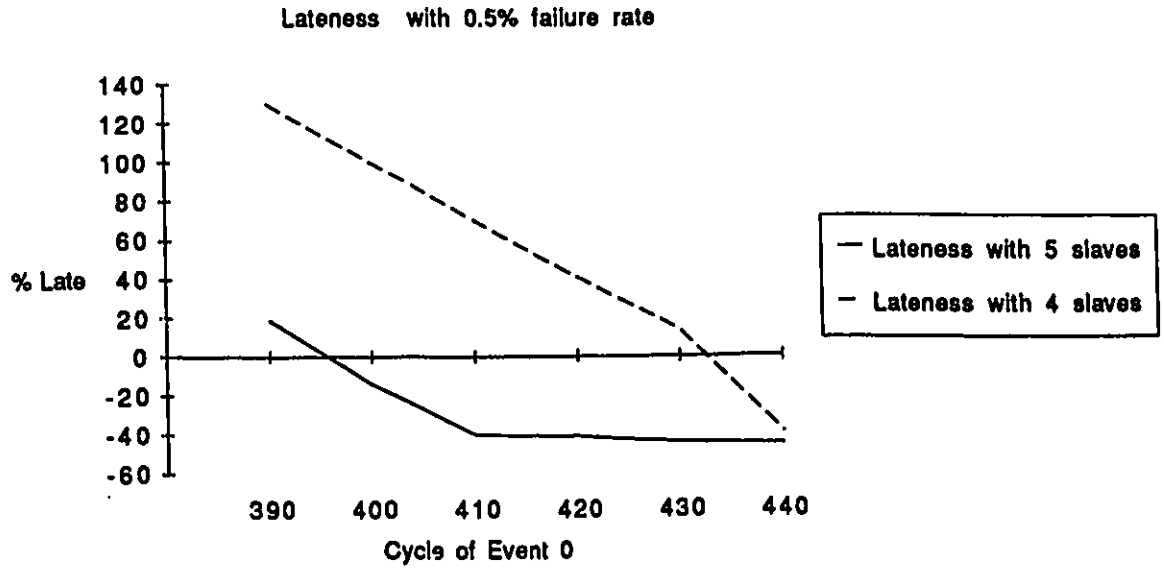

Figure 5.1: Average Lateness of Response to Event 0 With a Failure Rate of 0.5%

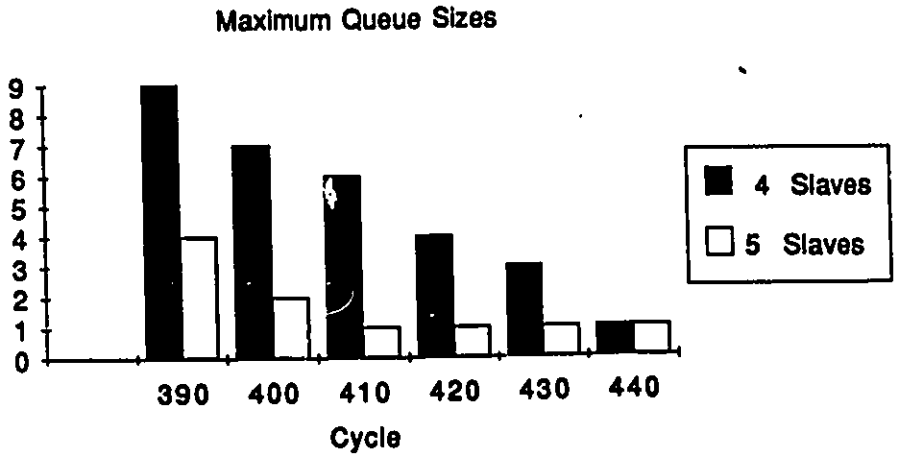

Figure 5.2: Event's Maximum Queue Sizes with 4 and 5 Slaves

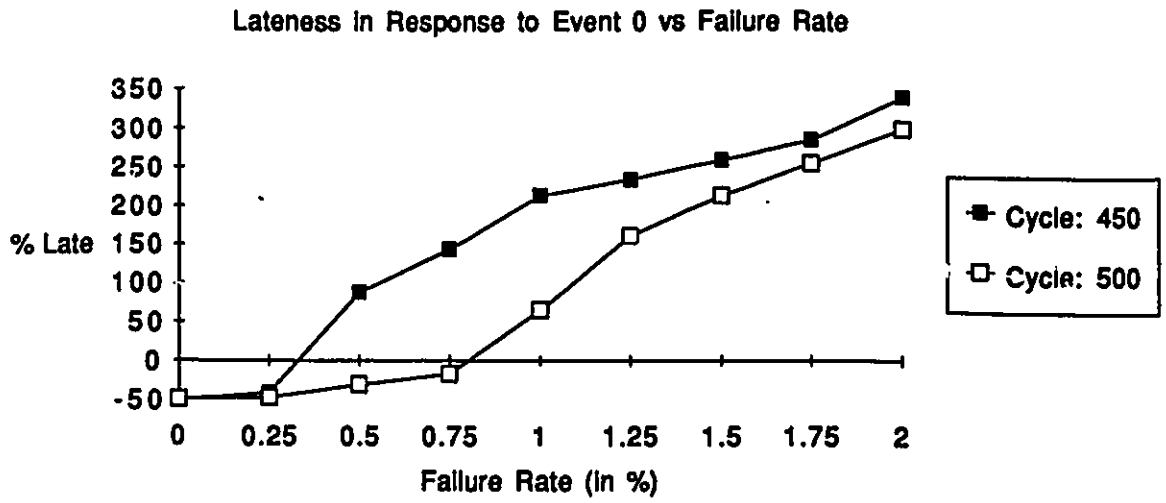

Figure 5.3: Lateness of Response to Event 0 Versus Failure Rate

5.1.3.2 Unknown Failure Rate

Using the simulator, it is possible to check the highest failure rate at which the system is able to maintain an acceptable performance. Figure 5.3 shows the results of tests applied on the two wagons example with relatively light loads (event 0 configured as cyclic with period of 450 and 500 units of time). The graph shows the lateness of the responses to event 0 as a function of the failure rate of slave 1, with the failures being detected only at the end of the execution of activities.

It can be seen that for this example, the results are unacceptable when the failure rate goes above 0.3% for an event's cycle of 450 units of time and when it goes above 0.8% for a cycle of 500 units of time.

5.1.3.3 Varying The Recovery Time

Another interesting test to perform is with respect to the recovery time for slaves that fail. This includes the time to fix the defective unit and the time to reinstate it to the system. Real systems have finite non-instantaneous recovery procedures for hardware failures, in which a resource becomes unavailable until it is fixed (this is

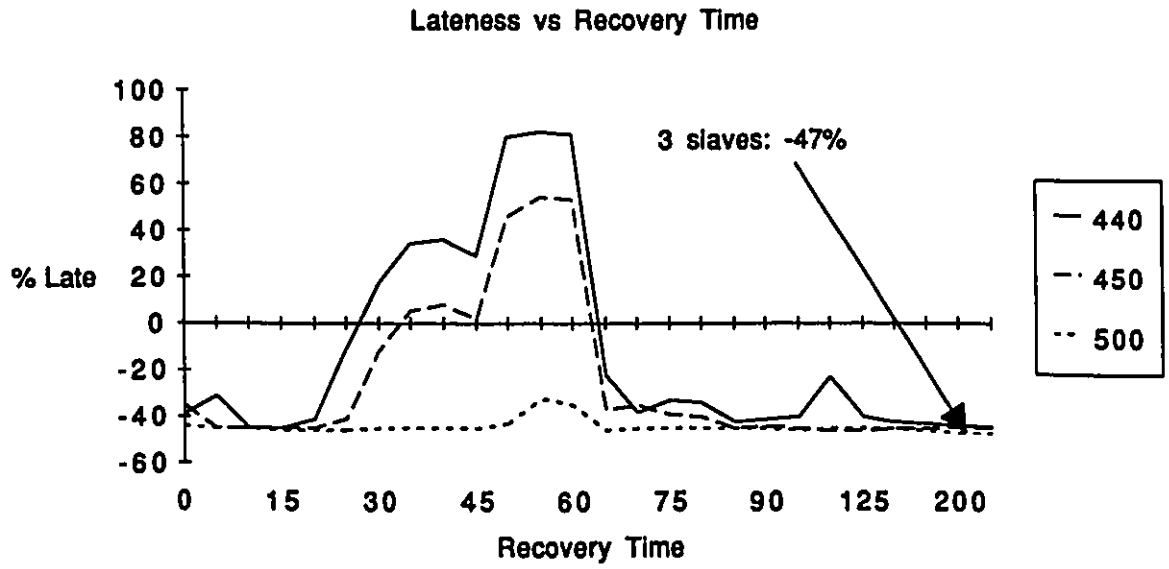

Figure 5.4: Lateness of Response to Event 0 Versus Recovery Time

different from transient or intermittent errors).

Using the simulator, one is able to check the effects of different recovery times on the system's performance. By doing so, it is possible to establish an adequate maintenance and recovery procedure before the final implementation.

In the specific simulations that were performed on the two wagons example interesting results were obtained when event 0 was configured as cyclic. Cyclic failures were generated and any slave could fail but only one per simulation. The results are shown in figure 5.4 for cycles of 440, 450 and 500 units of time.

The graph shows that for a light load (a cycle of 500 units of time) the responsiveness is practically not affected by the recovery times. For the heavier loads, the graph shows relatively large oscillations; it indicates a consistent improvement in the system's performance when large recovery times are used. This is misleading because with very large recovery times the system reverts from a 4 slave system in which slaves can fail to a 3 slave system in which slaves cannot fail; the way the simulation is setup, for large recovery times, a smaller number of failures (i.e. failure rate) can be

generated (a new failure can only be generated after the slave has recovered). Checking the results with three slaves and no failures, the response lateness is -47%, while in the case with 4 slaves and failures, the lateness tends to stabilize at -46%. This might suggest that we should implement the system with only three slaves. However, a system with only three slaves is also subject to failures and a single failure would reduce the number of slaves available by one third, creating a drastic increase in the responses' delays.

5.2 Redundancy in Multiactivity Systems

The addition of an extra slave(s) adds processing power to the system and can prevent events from being missed or responded too late when failures occur. However, in other cases a failure may still cause unacceptable delays in the responses; not only in the ones directly affected by the failure but also in the others because of the extra load added to the system.

Thus, a single failure in the wrong activity can adversely affect the performance of the whole system, particularly if these activities are those that take a long time to execute. They potentially cause the largest disturbances in the case of a failure that was detected very late during the activity execution period (the whole activity has to be restarted from the beginning). In addition, in some systems it is also necessary to guarantee the correctness of intermediate results; further processing can only occur after these intermediate results are checked and proved as valid.

A possible solution to these problems is to execute critical activities redundantly. They are executed simultaneously by two, three or even more slaves and at the end of execution the results are checked either by specialized hardware of by a software routine in the executive.

When executing activities redundantly in multiactivity systems, there are the following possibilities, depending on whether the activities were successfully executed:

1. All results are identical: the activity was successfully executed by two or more slaves, which finished execution and provided all identical results.

2. The results differ: the activity was executed by two or more slaves, all of which finished execution but with one of them giving different results.

3. Execution not finished: not all the slaves that were executing the redundant activity finished execution; a failure occurred in a slave and was not detected (its results never arrived).

4. Failure detection by extra hardware: specialized hardware exists that can detect when a slave is not working properly.

NOTE: *In this discussion we are assuming that only one slave failure can occur; two slave failures can never occur.*

A simple voting algorithm can be used in the case of TMR (triple modular redundancy) or NMR (n>3) when a single slave returns different values from the others. A majority decision is taken and the slave that gave the different result is declared as failed.

In the case of DMR (double modular redundancy), there are two possibilities when a failure occurs:

- When one of the slaves does not provide a result or when one of the slaves is detected as failed by the specialized hardware, the other result is assumed to be valid.

- When both slaves finish execution of the activity but they show different results the activity should be reexecuted in three slaves as a TMR activity.

Therefore, when using DMR and the two slaves show different results, double modular redundant activities will produce a much worse performance than TMR activities, in which there is no need for activities to be reexecuted. But if a slave is detected as failed or if it simply does not finish execution, DMR activities will generate a better performance than TMR activities because they cause a smaller extra-load in the system.

Therefore, redundancy may solve both the problem of guaranteed correct results and the heavy and unpredictable effects of restarting activities. The question is: at what cost?

Redundancy is expensive. It requires more complex coordination and control from the coordinator, thus requiring more of its CPU processing time; it also increases the slave usage. If the system's load is not correctly designed, the use of redundancy may end up inducing delays in other processes. It will not introduce any delays in the process which owns the redundant activity, except for a small processing time in the executive; however, activities from other processes may remain longer in the ready-to-run queue due to the unavailability of slaves. To prevent this, simulation enters the scene.

5.2.1 Testing Redundancy

It has been shown how redundancy can be used to solve many problems relative to the correctness of results and reliability. Questions arise about the costs incurred by running critical activities redundantly. One way to check this problem is to consider the performance degradation produced by the use of redundant activities in a system in which failures do not occur. Another way is to consider the extra number of slaves needed to achieve the same performance of a non-redundant system.

The major purpose of the redundancy tests are:

1. The study of the effects on the performance (responsiveness to events) of the overall system. This is a comparison between the execution time of responses with and without redundancy.

2. An analysis of the required increase in the hardware (and also some extra control software) costs compared with the performance improvements obtained. This checks the increase in the number of resources necessary to meet the timing requirements and considers the trade-off between the better performance and the higher costs.

SIMPAL allows the use of redundant activities very easily by being able to execute three types of activities. The actions taken by the simulator for each of the three types are very simple:

- Normal activities: these do not have redundancy and are executed only once and by only one slave;
- Double redundant activities (DMR): when a process includes any of these activities, a copy of the activity's parameters are given to two slaves and both start executing the same activity simultaneously;
- Triple redundant activities (TMR): when a process includes any of these activities, a copy of the activity's parameters are given to three slaves and all three start executing the same activity simultaneously.

A methodology to check the effects of redundancy in multiactivity systems is outlined below:

1. Determine the activities that require redundancy by:
    - Choosing the activities whose results are critical; the criticalness of an activity can be estimated by checking how many activities require its results; the larger the number, the greater the contamination produced by wrong results.
    - Choosing the activities in which a failure will cause large overheads, such as long activities.
2. Use the simulator SIMPAL, determine the degradation produced when these activities are executed redundantly.
3. Determine the additional number of slaves needed to achieve the same performance of a non-redundant system.

Using the well known two wagons example, several tests were run using SIMPAL adding TMR to the system. The modification to the original processes was simply:

- In process number 0, activity 12 ("Go Back to Point U") became a TMR activity.

NOTE: *When a failure occur in a slave during the simulations, it is assumed that this slave is repaired at the next clock cycle. Therefore, the slave does not become unavailable throughout the rest of the simulation.*

Figure 5.5 shows the performance of systems with 4, 5 and 6 slaves using TMR and also a system with 4 slaves and no redundancy. The purpose of these tests is to show the degradation produced by redundancy when **no failures** occur. It is clear that the performance achieved using TMR activities is very poor when only four slaves are used, mainly because the load in the system increased significantly without an equivalent increase in the number of resources (the average usage of the slaves increased from 56% to 74%). However, when redundancy is used in a system with 5 slaves, the responsiveness becomes very close to that achieved when no redundancy is used. It can be seen as well that very little improvements are obtained when 6 slaves are used instead of 5.

Therefore, in the case when no failures occur, the performance may be reduced considerably but this can be circumvented by the addition of a single slave. This, however, does not tell the whole story. There are significant improvements that are achieved by a redundant system when failures do occur, as shown in figure 5.6.

This graph shows the lateness of the system when a failure occurs systematically at the end of execution of activity 12. Event 0 was configured as cyclic (period of 440 units of time) and 4 and 5 slaves were used. It can be seen that the overall responsiveness became considerably better even when only four slaves were used.

5.3 Communication Overheads and Data Passing in Multiactivity Systems

In this section a brief investigation of the communication aspects of multiactivity systems is presented. The section begins with an introduction to the architectural

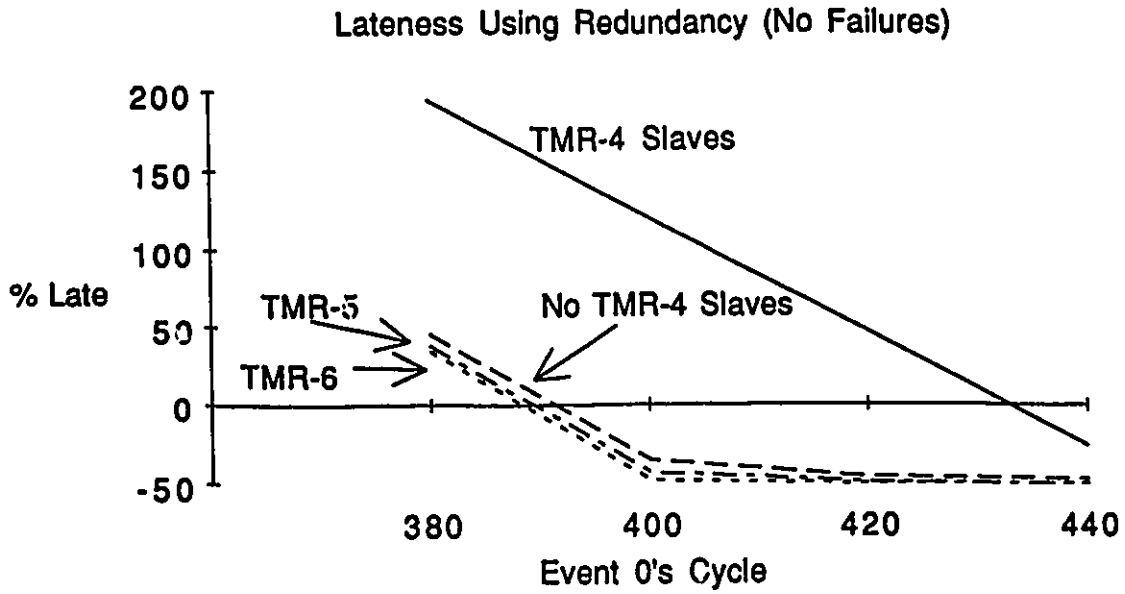

Figure 5.5: Lateness of Response to Event 0 With and Without Redundancy

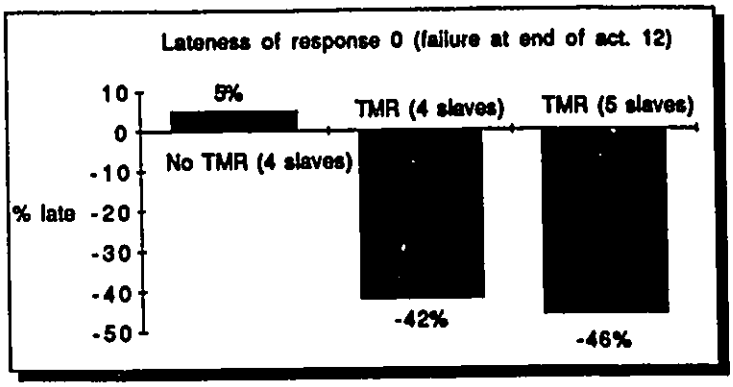

Figure 5.6: Lateness of Response to Event 0 With Failures

alternatives that can be used for data passing in multiactivity systems and then discusses the different types of communications.

5.3.1 Architectures for Communications

There are basically two types of communications in multiactivity systems: Interprocess Communications, which are used for the exchange of information or for synchronization between different processes; and Interactivity Communications, which are used for the sharing of the information generated by the execution of an activity.

In each of these, the amount of data that has to be exchanged may vary from a simple data element consisting of a few bytes to complex data structures containing hundreds or thousands of bytes.

These communications can be implemented in a number of ways, the two extremes being:

1. Shared Memory

   In this scheme each processor may have its own local memory but the communications are done via a shared memory, which they can read *and* modify. Thus, the parameters can be passed very efficiently just by using pointers. There may be significant memory contention when a large number of slaves are accessing the memory.

2. Non-Shared Memory

   This is the case when the coordinator and the slaves have only local memories and the communications are performed through messaging. This represents the worst possible performance condition for the transmission of messages because all parameters have to be passed by value. All the messages may go through the coordinator or alternately, they may be transmitted directly between slaves.

Independent of the architecture used, little communication overheads will result from the transmission of short messages or signals composed of just a few bytes. However, when large amounts of data have to be exchanged, the time lost with

communications may become quite high even if a shared memory is used. This is specially the case if the data of an activity has to be used by many subsequent activities; the data will have to be copied from the global space into each local process' space before it can be modified by the processes, unless some complex consistency control mechanism is used for the global space.

In order to check the performance regarding the various communication overheads involved, the performance can be estimated before the actual implementation, by using the simulator SIMPAL.

5.3.2 Interactivity and Interprocess Communications

Figure 5.7 indicates the various types of communications found in multiactivity systems. They can be data transfers between activities, messaging between processes or synchronization within a process

This indicates many data exchanges in multiactivity systems, which can become very critical. However, embedded systems are usually control oriented, not data oriented. Therefore, one may expect that these communications between coordinator and slaves will consist only of a few bytes of data. In multiactivity systems the slaves already have all the code for the activities in their memory, so this does not need to be passed. What the slaves need is simply an identification of the process that owns the activity. It can then know exactly where to find the data it needs, which sensors to read or which actuators to command, based on a static table loaded at system start-up. With all these information in its own memory, there is little else the slave needs in order to be able to execute an activity. The exceptions may be in the case of massive computations on large amounts of data, which is not likely to occur often in an embedded system.

Another approach that could be taken is the keeping in the slaves of data already processed. The slave would have an area in its memory for each process in the system, where previous data, either received from the processes or obtained during the execution of activities, would be stored. In this case, only the data that has been modified since the last execution request would be retransmitted to the slaves. Only

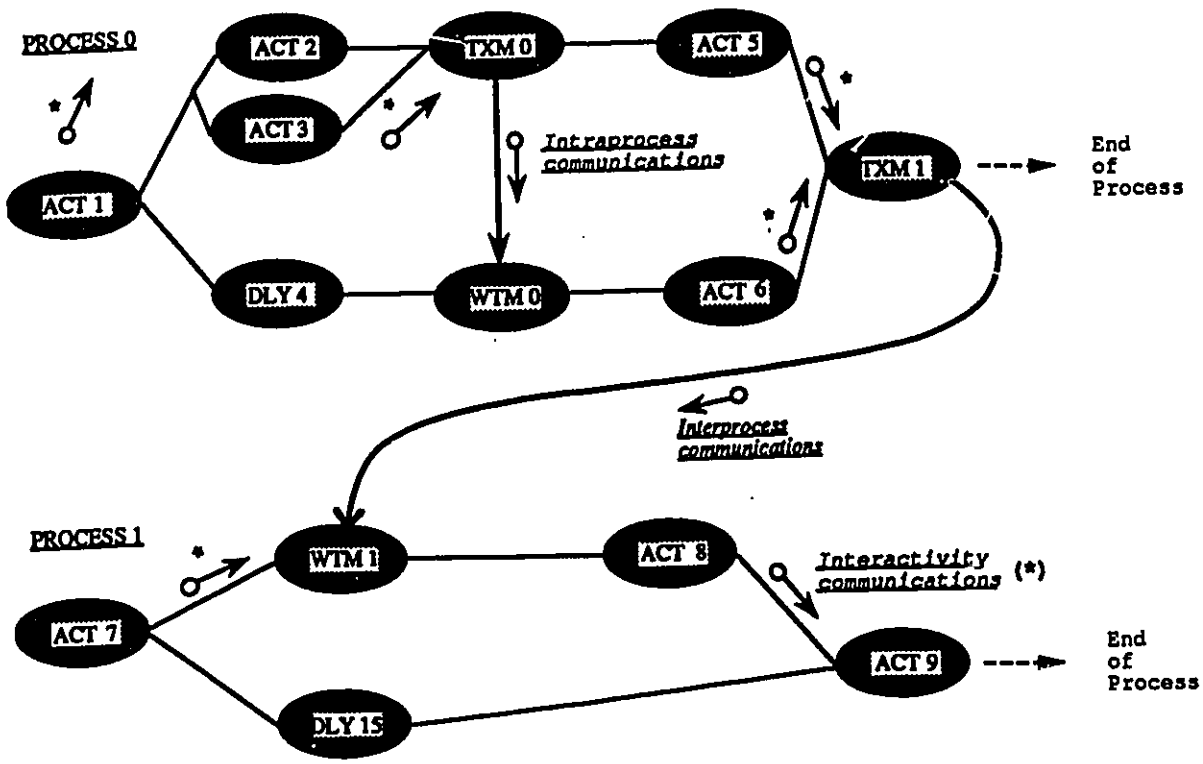

Figure 5.7: Communications in Multiactivity Processes

an initial lengthy transmission would be required at system start-up. This approach requires the use of smarter slaves. Such a scheme is used when, in order to execute an activity, a slave needs data from a previously executed activity. The slave must be able to find and retrieve the correct data. Intelligent slaves are needed in order to avoid the destruction of valid data that will be needed at a future time. A consistency scheme must also exist to assure that the data is still valid after the execution of many other activities.

Interprocess Communications consists of mainly the type of communications between the various processes in the system, although they may also be used to communicate between different threads of a same PAL process (in which case they can be called *intraprocess communications*). In this last case, its use is very restricted and it is basically for synchronization of two independent threads of the same process. Generally, these threads are configured in the same process instead of being two separate processes in order for them to start simultaneously upon occurrence of an event.

Communications between different processes can be used either for information exchange or for synchronization. In the latter case, messages are only signals exchanged by processes through a "blocking receive" type primitive. Since these synchronization messages are usually composed of very few bytes, they do not present a major problem in terms of transmission overheads, regardless of which architecture is used. Overheads can only result from excessive number of synchronizations due to a bad logical partitioning of the system's events and responses or to a bad choice of starting events (those that trigger processes).

In the case of information exchange messages, their sizes can be easily checked. However, this is not the case of the messages frequency, i.e the rate at which these messages are exchanged between the processes. A single transmission of a large message is not as bad as frequent transmissions of smaller messages. In any case, large messages possibly represent a bad partitioning of system responses and are generally due to a bad design. There are some mechanisms available to check the sizes of messages being exchanged between processes but these tools only give a rough idea of the amount of data transferred and they give no idea about the dynamic behavior,

i.e. the frequency with which the messages are exchanged. A tool for checking the communications frequency is still to come (a good topic for further research). In multiactivity systems, the dynamic behavior can be tested with the use of the simulator SIMPAL. The designer can partition the responses into processes in many different ways and each resulting system can then be simulated. Using the statistics generated, the user is then able to choose which logical partitioning gives the best results.

5.3.3 Testing Communication Overheads with SIMPAL

In SIMPAL all communications were implemented using transmissions by value. However, they do not produce any explicit delay due to the size of the messages or the number of parameters passed, in either interactivity or interprocess communications. SIMPAL spends some time for interactivity communications and control but this time is fixed (one clock cycle is lost between the execution of two sequential activities). The additional time spent with large messages is implementation and architecture dependent and therefore it was left for the user to add the delays himself. To do this, two approaches are suggested:

1. When very large interprocess messages are used, the delays can be simulated by the addition of a Delay (DLY) primitive, which allows a process to be delayed for any amount of time.

2. When large number of parameters are used for interactivity communications, the transmission delays can be simulated in two forms:

   - When the slaves themselves have to implement the communications, the communications overheads can be represented by increasing the execution time of the activities. For example, the execution times can be increased in steps of 5 or 10% in order to check the different overheads.
   - When there is some specialized hardware that handles the communications, the overheads can be simulated by the inclusion of a Delay primitive similar

Activity	Original Exec. Time	Decomposed Into	New Execution Time
6	90 units of time	2 activities of	45 units of time
12	120 units of time	3 activities of	40 units of time
16	70 units of time	2 activities of	35 units of time

Table 5.1: Decomposition of Larger Activities in the Two Wagons Example

to case 1 above. This is because the slaves are not directly involved in the communications (but more complex hardware elements are required).

Another important factor is the granularity of the activities, which also affects the execution time of processes. This is because each time an activity finishes execution there are communications between the slave and the coordinator. Despite this fact, there are a few reasons to decompose large activities into smaller ones:

- Improved Reliability: as it was seen in earlier sections, a failure in a very large activity will considerably affect the performance. A failure in smaller activities will produce a much smaller overhead.

- Better Allocation of Resources: this is the problem of allocating all the resources that an activity will need before the activity starts execution. If the resources are only released after the end of execution, they will be much more efficiently used if the activity is decomposed into smaller ones, each with its own resource requirements.

To illustrate some of these ideas, tests were run with SIMPAL which specifically show the trade-off between reliability and communication overheads. The tests were performed on the original two wagons example and on variations with finer granularity. In these variations the three largest activities were decomposed into smaller ones, as shown in table 5.1.

Furthermore, in each of these smaller activities, communication overheads of 0, 10 and 20% were introduced and tested along with the original specification. All cases

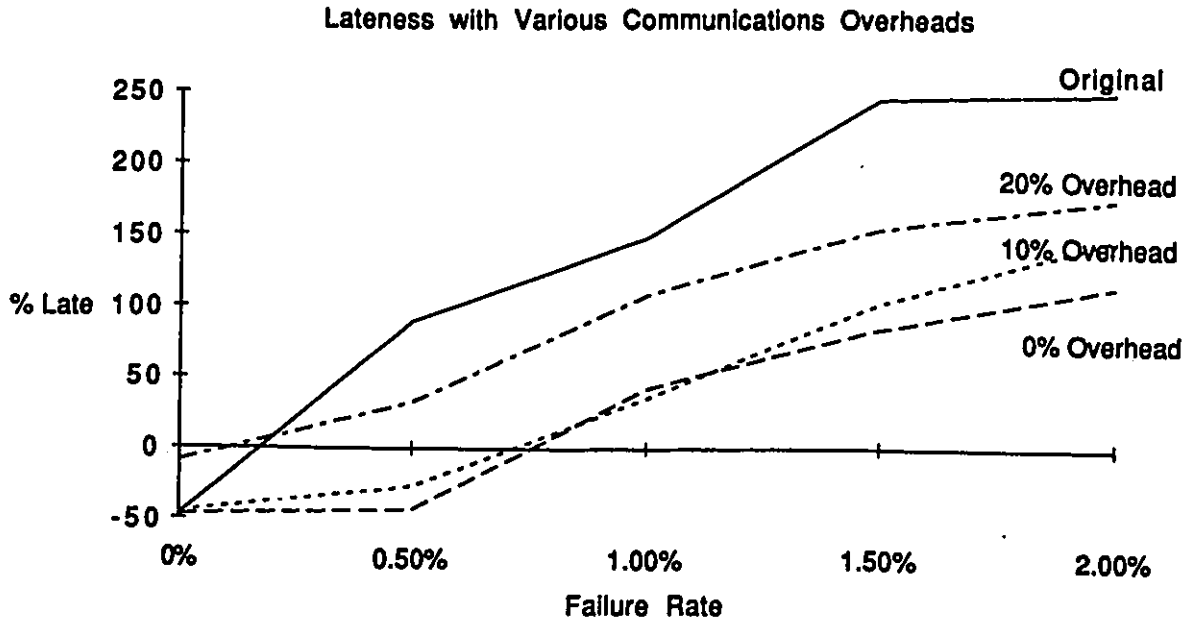

Figure 5.8: Communications Overheads Versus Failure Rate

were tested using 0, 0.5, 1.0, 1.5 and 2% failure rates, with failure recognition at the end of execution of the activities. These results are shown in figure 5.8.

It can be seen that when no failures occur, the specifications with finer granularity perform a little worse than the original system but considerable overheads only start to appear for communications overheads of more than 20%. However, when failures occur, the performance is much better even when communication overheads are up to 20%.

5.4 The Use Of Intelligent Slaves

So far, only the use of passive slaves capable only of executing the activities dictated by the coordinator has been assumed. In this section, we consider the use of smarter slaves and present its major advantages and disadvantages.

Our methodology assumed that the control of the system was done by the executive. It knew when the slaves were busy or free and it decided when to assign the

activities in the ready-to-run queue to the free slaves. There are other alternatives, each of them with its merits and its faults. They can be summarized as follows:

1. A different approach is having the requests for activities being generated from the slaves; whenever a slave finishes the execution of an activity, it will transmit a request to the coordinator, which will then assign to it an activity from the ready-to-run queue. This method relieves the coordinator from the task of keeping track of which slaves are free. However, such tables will have to be maintained anyway if fault tolerance is required; when there is a failure the executive would need to know which activity has failed in order to reexecute it.

2. A second possibility is to provide a mechanism by which each ready-to-run activity is broadcast to all slaves and these decide whether or not they accept the activity. This approach has some advantages (such as reducing some amount of queue handling in the coordinator) but also introduces some problems: a more complex architecture that allows simultaneous transmissions of requests to the slaves (unless an inefficient polling is used) and either the ability to detect or a mechanism to avoid two or more slaves adopting the same activity (such as a daisy chain mechanism). Further interactions between the central coordinator and the slaves may be necessary to decide which slave will really execute the activity; in addition, the slaves may need a more complex kernel.

3. The central coordinator assigns a set of activities to each slave. These accepted activities may be stored in a strictly sequential order (FCFS) or even according to static or dynamic priorities. This idea is potentially very interesting since it reduces some of the load in the coordinator. The cost is an increase in the complexity of the kernel (or operating system) of the slaves. Such an increase will not be too large if fixed priorities (FCFS) are used for the scheduling within the slaves. In this case, all that is needed is the ability to manage a sequential queue. It becomes more complicated when the queue must be ordered according to parameters that must be updated frequently through communications between the coordinator and the slaves.

There are also some possibilities in the choice of from which slave the executive will request the execution of the activities. Such decision can be based on the expected future load of the slaves and on the current timing requirements. There are interesting possibilities in the optimization of this task.

This third approach may have excellent results in those particular applications that require either very complex processes or a very large number of processes, when the coordinator is expected to be very busy. By transferring the job to the slaves, the coordinator is relieved from most of the time consuming operations on the ready-to-run queue. The additional overhead imposed on the slaves will be hardly noticeable. In fact, even the performance of the slaves in terms of jobs executed per unit of time is expected to improve since the slaves can switch much faster from one activity to another, without the need for communications with the coordinator (particularly in the case in which activities share data).

The main disadvantages in this approach are in the fault tolerance aspects. A failure in a slave that has many activities to be executed in its own requests queue will produce a very large overhead with the coordinator having to reallocate all the activities from the failed slave. In order to do this, the coordinator must maintain a table of slaves with their respective allocated activities. It must handle this table, removing already finished activities and keeping it updated.

4. The coordinator sends all the activities in the ready-to-run queue to all the slaves. The slaves store all these requests in a queue and when they start executing an activity they broadcast a message to the other slaves which will then remove that request from their queue.

   The slaves can decide ahead of time how many activities they will execute in a row. Such a decision could be based either on the data that these activities share (in order to minimize the communication overheads) or on the future load of the slaves (so that they are efficiently maintained busy) or even on some priorities defined by the executive (to meet the timing constraints).

   The disadvantages are the need for larger memories for the slaves which may be

required to execute a larger number of different activities; a complex communications scheme; a protection mechanism to avoid the execution of the same activity by more than one slave; and another mechanism to maintain the consistency of the data generated by some activities that is required by other activities.

Other schemes to improve the efficiency of activity-to-slave assignment are possible, which may perhaps include a combination of some of the above. Smarter slaves can also be used for other purposes, as mentioned earlier in this thesis. For example, they could provide a mechanism for detecting a failure during the execution of an activity. This way, instead of making the executive reallocate the entire activity to another slave from the beginning, they could store and preserve the last valid results. The execution of the activity would then be able to resume from an intermediate point. Another example is the use of smarter slaves to implement direct communications between slaves, instead of all communications being handled by the central coordinator.

Chapter 6

Conclusion

This concluding chapter is divided into two sections. The first section presents a brief summary of the thesis, emphasizing the achievements of this research, while the second section discusses directions for future research.

6.1 Thesis Summary

The research done in this thesis has addressed the problem of designing embedded systems. These are also usually referred to as real-time systems because they must satisfy not only behavioral and functional requirements, but also temporal requirements.

A prototyping-based design methodology has been presented. It is based on the multiactivity paradigm, which separates the coordination of activities from their execution, and is intended to be used by the computer literate application specialists; this is in contrast with traditional methodologies that usually restrict the design to computer specialists. This reduces considerably design errors that are introduced when interactions are necessary between both specialists and when the design is done by those who are unfamiliar with the application details. Furthermore, this also avoids the high costs of using the computer specialists for the very small production run usually inherent to embedded systems.

The methodology generates two system prototypes, the *Specification Prototype*

and the *Design Prototype*. The *Specification Prototype* is a model of the behavioral and functional requirements of the system that can be used to check whether the requirements were correctly specified. This is done in three phases. First, the specification is checked with the PALED editor, which detects any lexical and syntactical errors. Second, the specification is tested with the PAL IVS verifier, which tests whether the interactions between processes cause temporal errors, such as deadlocks and starvation. Finally, the specification is simulated with the simulator SIMPAL, which shows whether the specification satisfies the behavioral requirements, without considering performance, timing and other constraints. The generation of the *Specification Prototype* and the use of the editor and the verifier are explained in other theses [Har89a, Chu90], while we concentrated on the use of the simulator in the *Design Prototype*.

The *Design Prototype* is a model of the design specifications of the system, which is used to check if it satisfies the performance and timing requirements and to determine the parameters needed to satisfy them. These include the number and type of slave processors, the activity-to-slave allocation, the granularity of the activities, the reliability required, the scheduling schemes that should be used, etc. The ways in which SIMPAL can be used to check the above parameters were shown in detail.

The design methodology followed in this thesis is based on the multiactivity design paradigm, whose elements were introduced in a number of previous theses [Cos83, Joa86, Dur87]. Additional arguments for its applicability to embedded systems were also presented here.

The simulator SIMPAL was developed to complete, with the other two tools mentioned above (the editor and the verifier), an initial PAL-based toolset. SIMPAL is an easy-to-use program that has many features to allow the testing of a PAL specification, such as: system tracing that shows the execution of processes and their effect on the environment; simulation of system failures to determine the required reliability in order to satisfy the real-time requirements; the use of special purpose processors which can only execute certain activities; the use of pre-defined scheduling schemes, etc. In addition to these, the simulator generates at the end of each simulation a

complete statistics file with data referring to many of the system parameters. As pointed out by Harvey in [Har89a], an important aspect of a methodology is the user acceptance of it: the user's willingness to accept the results depend on his confidence on how these results are obtained. This is why the development environment is very important, specifically the user interface. We believe that the simulator achieved this requirement and constitutes a reliable tool that can boost the user's confidence in the methodology.

6.2 Further Research

Although the implementation of the methodology presented in this thesis is limited, it did show that the methodology is feasible. What is needed now is a final integration of the various tools and an implementation on a hardware platform that is applied to real problems. This is the best way to show how well the methodology can be used and how it can be enhanced. It will indeed be very interesting to see a real application of all these ideas and to check the applications in which multiactivity systems can be used efficiently.

Regarding specifically the design prototyping phase presented in chapters 4 and 5, an additional feature that could be developed in a new tool is the automatic generation of the graphs presented. Such a tool would operate in conjunction with the simulator, obtaining from it the statistical data generated at each simulation run. Furthermore, this new tool could analyse the data and suggest operating parameters for the final system. For instance, the tool could easily be used in the first step of the design prototype, which is the initial estimation of the number of slaves. An algorithm for this would not be hard to develop because it is based on predefined safety margins for the real-time constraints of the responses. The other decisions, such as those regarding scheduling schemes that should be used, the reliability required, the use of redundant activities, etc., may require the development of quite complex computer algorithms or even the use of artificial intelligence methodologies. The number of parameters involved is quite large and there may be too many alternatives to consider. Such an

intelligent tool could certainly relieve the designer from many decision making tasks and could contribute to the automation of the whole process, indeed making it more suitable for application specialists instead of computer specialists.

Regarding scheduling, there are many interesting aspects to consider in order to fulfill the real-time requirements. One possible area of research is to try to find an optimum scheduling scheme for each of the operating conditions. In this way, a system could use dynamic scheduling to adjust to these conditions. The various conditions can be identified by collecting statistical data at run-time and then a scheme can be chosen by comparing the conditions with those previously simulated. Currently, there is another thesis being developed in this area.

Another important aspect that has to be studied is resource allocation in multiactivity systems. In this thesis, we only considered the data requirements of the activities, i.e the data that an activity may require from another activity before it can start execution. It was discussed that there are reasons to increase inter-activity communications, and one of them is the increased performance when failures occur. However, another reason in partitioning large activities into smaller ones is resource allocation. In this thesis, it was assumed that an activity had all its resources allocated before it started execution; these were released only at the end of execution. Other approaches may consider dynamic allocation and release of resources, as the activities are executed. In this case, it may be advisable to use larger activities.

Interesting additional research aspects are related to the communications in multiactivity systems: which types of communications to use (blocking, non-blocking,etc.) and over which underlying architecture. There are alternative approaches to the completely centrally coordinated communications scheme that were assumed in this thesis, such as direct interactions between slaves or the use of a common data areas.

At a higher level of abstraction, a more distributed system can be considered to implement *multi* multiactivity systems. These may be used when very complex activities are used; each of them may become a multiactivity system itself. Such an approach has potentially enormous reliability and performance capabilities and it should be the subject of further research.

Bibliography

[BBFM77] J.C. Bossy, P. Brard, P. Faugere, and C. Merlaud. *Le Grafcet: Sa pratique et ses applications.* educalivre, Paris, France, 1977.

[BKHW89] R.J.A. Buhr, G.M. Karam, C.J. Hayes, and C.M. Woodside. Software CAD: A Revolutionary Approach. *IEEE Transactions on Software Engineering*, SE-15(3):235–249, March 1989.

[BL90] Joseph Boykin and Susan J. LoVerso. Recent Developments in Operating Systems. *IEEE Computer*, 23(5):5-6, May 1990.

[Bru87] J. Bruijning. Evaluation and integration of specification languages. *N/A*, 1987. 0377-5075/87 Elsevier Science Publisher B.V., North Holland.

[CCI88] CCITT. *Recomendation Z.100. Specification and description language SDL*, 1988. AP IX-3 5.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A practical Approach. In *Tenth Annual ACM Symposium on the Principles of Programming Languages*, pages 117–126, Austin, Texas, January 1983.

[Chu90] Arto Chubukjian. The Extended PAL Editor: An Example of Real-Time System Specification Tool. Master's thesis, University of Ottawa, April 1990.

[CMQV89] G. Cohen, P. Moller, J.P. Quadrat, and M. Viot. Algebraic tools for the performance evaluation of discrete event systems. *Proceedings of the IEEE*, 77(1):39–58, January 1989.

[Cos83] Pierre Cosineau. Design of multilayer information systems. Master's thesis, University of Ottawa, 1983.

[DT90] Michel Dubois and Shreekant Thakkar. Cache Architectures in Tightly Coupled Multiprocessors. *IEEE Computer*, 23(6):9–11, June 1990.

[Dur87] Jose M. Duran. Applicability of process activity diagrams as a system design tool. Master's thesis, University of Ottawa, May 1987.

[Eva89] Ralph A. Evans. Reliability: Whence & Whither. *IEEE Transactions on Reliability*, 38(5):513, December 1989.

[FP90] Colin Fidge and Michael Pilling. Specification Languages for Distributed Real-Time Systems. In *Proceedings of the Fifth Australian Software Engineering Conference OASWEC'90*, Australia, 1990.

[Gen88] W.M. Gentleman. Real-Time Applications: Multiprocessors in Harmony. In *Proceedings of BUSCON-88 East*, pages 269–278, New York, NY, October 1988.

[Har89a] R. A. Harvey. Verification of concurrent systems specifications using temporal logic. Master's thesis, University of Ottawa, July 1989.

[Har89b] Randall A. Harvey. Process Activity Language Interactive Verification System: User Guide. User Manual and Software code, 1989.

[HLN+88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: A Working Environment for the Development of Complex Reactive Systems. *IEEE*, April 1988. Reprinted from Proceedings of the 10th international conference on software engineering.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[HP88] D.J. Hatley and I.A. Pirbhai. *Strategies for real-time system specification.* Dorset House Publishing, New York, NY, 1988.

[IS90] Wilson D. Yates III and David A. Shaller. Reliability Engineering as Applied to Software. In *Annual Reliability and Maintainability Symposium,* pages 425–429, Los Angeles, CA, January 1990. IEEE.

[ISO85] ISO International Organization for Standardization. *Estelle: A formal description technique based on an extended transition model,* February 1985. TC97/SC21/N422.

[ISO86] ISO International Organization for Standardization. *LOTOS: A formal description technique based on the temporal ordering of observational behavior,* July 1986. TC97/SC21.

[Joa86] Robert Joannis. Specification and Design of Multiple Microprocessor Systems using Process Activity Diagrams. Master's thesis, University of Ottawa, August 1986.

[Joa90] Robert Joannis. An Object-Oriented Specification Technique for Embedded Systems. Report, 1990.

[KJ86] Moshe Krieger and Robert Joannis. Process Activity Diagrams: A Tool for the Specification and Design of Multiple Microprocessor Systems. In *Proceedings of the ISMM International Symposium on Software and Hardware Applications of Microcomputers,* pages 157–161, Beverly Hills, CA, February 1986. ISMM.

[KJH89] Moshe Krieger, Amjad Junaid, and Randall Harvey. Centrally coordinated systems and their design using process activity language. Unpublished, 1989.

[KLH88] M. Krieger, J.P. Lachance, and R.A. Harvey. Process Activity Language PAL for Planning and Coordination of FMS. In *Proceedings of MAPL 88*, pages 127–135, Winnipeg, Manitoba, June 1988. National Research Council of Canada.

[LA90] Levi and Agrawala. *Real-Time System Design.* MacGraw-Hill, 1990.

[Laced] J.P. Lachance. Scheduling in Multiprocessor Real-Time Systems. Master's thesis, University of Ottawa, To be submitted.

[Lau89] R. J. Lauber. Forecasting Real–Time Behavior During Software Design using a CASE Environment. *Real–Time Systems*, 1(1):61–76, June 1989.

[LTS79] P. Lauer, P.R. Torrigiani, and M.W. Shields. COSY: A system specification language based on path expressions. *ACTA Informatica*, 1979. 12:109:158.

[Luq89] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, May 1989.

[Mot89] Christopher Mott. Comments on: On MTBF (an editorial). *IEEE Transactions on Reliability*, 38(5):516, December 1989.

[OW87] J.S. Ostroff and W.M. Wonham. Modelling, Specifying, and Verifying Real-time Embedded Computer Systems. In *Proceedings of the Real-Time System Symposium*, pages 124–132, San Jose, CA, December 1987. The Computer Society of the IEEE.

[Pet77] James L. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–252, September 1977.

[RH80] I.V. Ramamoorthy and G.S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *IEEE Transactions on Software Engineering*, SE(6):440–449, 1980.

[Sea90] R. W. Sears. Practical Models for Determining Standby Redundancy Levels. In *Annual Reliability and Maintainability Symposium*, pages 120–126, Los Angeles, CA, January 1990. IEEE.

[Sha89] Dan Sharon. A PAL Executive. Preliminary version of Master's Thesis, Carleton University, Ottawa, Canada, 1989.

[SL89] I. Suzuki and H. Lu. Temporal Petri Nets and their application to modeling and analysis of a handshake daisy chain arbiter. *IEEE Transactions on Computer*, 38(5):695–704, May 1989.

[SR87] J.A. Stankovic and K. Ramamrithnam. The design of the Spring Kernel. In *Real-Time Systems Symposium*, pages 146–157, San Jose, California, December 1987.

[Sta88a] John A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, October 1988.

[Sta88b] John A. Stankovic. Real-Time Computing Systems: The Next Generation. University of Massachusets Report, January 1988.

[TIBY89] T.G.Lewis, Fred Handloser III, Sharada Bose, and Sherry Yang. Prototypes From Standard User Interface Management Systems. *IEEE Computer*, 22(5):51–60, May 1989.

[TY89] Schmuel Tyszberowicz and Amiram Yehudai. OBSERV - A Prototyping Language and Environment Combining Object Oriented Approach, State Machines and Logic Programming. University of Maryland: Computer Science Technical Report Series, August 1989.

[UPS90] Shambhu J. Upadhyaya, Hoang Pham, and Kewal K. Saluja. Reliability Enhancement by Submodule Redundancy. In *Annual Reliability and Maintainability Symposium*, pages 127–132, Los Angeles, CA, January 1990. IEEE.

[YT89] Raymond T. Yeh and Murat M. Tanik. Rapid prototyping in software development. *IEEE Computer*, 22(5):9–10, May 1989.

[Zav82] Pamela Zave. An Operational Approach to Requirement Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, March 1982. Reprinted in N. Gehani and A.D. McGettrick, editors, *Software Specifications Techniques*, pages 131-170, Addison-Wesley, Workingham, England, 1986.

[Zav84] Pamela Zave. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):104–118, February 1984.

Appendix A

SIMPAL User's Manual V1.4

A.1 General Description

SIMPAL is a tool for the development of embedded (real-time) systems that allows the simulation of such systems before their actual implementation. Thus, it allows the analysis of the behavior of the desired system under several expected (or unexpected) conditions which will determine whether the proposed solution meets the requirements and the specifications of the problem.

In practice, the simulator reduces the costly and time demanding process of designing a system, implementing it and then having to go back to the design phase after realizing that it does not work well under some of the possible conditions found in the real world.

Specifically, SIMPAL, as the name may suggest, performs the simulation of a system specified as processes written in PAL - Process Activity Language. To use the simulator it is necessary to specify both the processes and the environment in which they will run.

SIMPAL operates on two files of input data, which are read at the beginning of the simulation, then interacts with the user performing the simulation itself and finally generates an output file containing several statistics computed.

These two input files can be generated using two utilities:

1. Genenv: *This is a program that interacts with the user and generates a file containing data that describes the environment. This includes information about events, environment variables, activities, etc... The file is called SIM_ENV.n, where n is the number of the configuration.*

2. The PALED Editor: *This program works as any other high level language editors or text editors. It accepts text from the user, the processes written in PAL, and translates them into pseudo-code, generating a file called SIM_PROC.n. It has a few other features such as generating ATC's (Activity Timing Charts) and performing syntax checking on the entered expressions.*

At the beginning of execution, SIMPAL will load both files, SIM_ENV.n and SIM_PROC.n with the same extension number. Thus, the extension number should be used to characterize different versions, configurations or even completely different problems to solve.

A.1.1 The Processes In Simpal

The processes that SIMPAL understands are written in PAL mnemonic pseudo-code and should contain the following elements:

- *SEQ n - sequence of n elements*
- *CON n - concurrent of n elements*
- *RPT cond - repeat until condition is true*
- *CAS n - case of n elements, followed by the conditions*
- *ALT n - alternate of n elements*
- *DLY t - delay of t ticks*
- *ACT i - activity i*
- *WTM c - wait for message on channel c*

- *TXM c - transmit message on channel c*
- *WTE e - wait for event e*
- *STP Pi expression - set process condition i to value of expression*
- *STS Si expression - set system condition i to value of expression*
- *RDE v j - read environment variable v into process variable j*
- *RDS Si j- read system condition i into process variable j*

NOTE: When using the PALED Editor, you are able to write the different primitives and control constructs as PAL expressions, instead of the mnemonic pseudo-code presented above.

Each process contains its own data variables, similar to variables found in any other high level languages. These are local to the process and cannot be accessed by other processes. These local variables are referenced as P0, P1, ..., Pn.

In addition to the local variables, there are system variables, or global variables which may be read and modified by any process in the system. These are known also as billboard conditions. They are referenced as SJ, S1,...,Sn.

A.1.1.1 The Process and System Variables

The local variables Pi and the global variables (billboard conditions) Si can be of four types and are declared together with their initial value. These types are:

1. *BYTE: An integer value contained in 8 bits.*
2. *INTG: Any integer value contained in 16 bits.*
3. *LONG: Any integer value contained in 32 bits.*
4. *REAL: A floating point value.*

A.1.1.2 Logical Conditions and Expressions

In SIMPAL, conditions can have up to 4 levels of parentheses (nesting) and they may involve any process or system variable, as well as any constant. The common mathematical logical operators are accepted. Examples of valid conditions are:

- (P0>1)
- (P2<S9)
- ((P1=2)&(S3!=0))
- (((S0>0)&(S0<2))|((S2!=S3)&(P12=5.7)))

An invalid condition: (P0>S1+1)

Similarly, expressions can have up to four levels of parentheses and may involve any variable. The difference is that common mathematical arithmetic operators are also accepted. Examples of valid expressions:

- P0
- (P1+3)
- (S2/3)+((P1+P2+P3)*(S3-0.5)) *(no blanks in between)*

Conditions are used by RPT and CAS. Expressions are used by STP and STS.

A.1.1.3 The Communication Channels

Interprocess communication in PAL is done either via billboard conditions (global variables) or by channels. A process may transmit a message through a channel or may receive a message from a specific channel.

Channels are simply buffers of limited size. The messages can consist of any number of bytes. In fact, a message is specified as being an IO LIST, which can contain any number of variables.

A.1.1.4 Event triggered, priorities, due time and expected execution time

The complete specification of a process requires the definition of these parameters:

1. Event Triggered Process

   This is a flag that tells the the simulator that a process will only start execution after the occurrence of an event.

2. Priority:

   All processes in SIMPAL have a priority of execution, which is evaluated when the scheduling algorithm is HPRIF (Highest Priority First). The lowest value (0) has the highest priority.

3. Due Date:

   All processes, being in one way or another a response to an event, must have a maximum execution time (a deadline for completion of the response). Note that this value is used by processes that are not triggered by events (which are actually triggered by the event START execution). When processes that are triggered by events are awakened, their due date will become the response's due date based on the instant that the event occurred (and not the process' due date).

   These values are used by the scheduler (in the EARDF algorithm).

4. Expected Execution Time:

   These values, one for each process, must also be given to the simulator. They must be computed or estimated by the user by adding the execution time of individual activities in the case of serial threads (a SEQ- sequence) or by taking the maximum execution time when in parallel threads (a CON-concurrent). Other primitives must be estimated. These include: WTM (waiting for a message), WTE (waiting for an event), etc....

A.1.1.5 Example Of A Process

A simple PAL process is shown below, written in pseudo-code.

```
BYTE 28          ; Declaration of a byte variable (initial value 28)
INTG 1000        ; Declaration of an int. var. (initial value 1000)
INTG -290        ; Declaration of an int. var. (initial value -290)
REAL 0.45        ; Declaration of a float var. (initial value 0.45)
LONG 60000       ; Declaration of a long var. (initial value 60000)


SEQ 3                          ; sequence of 3 elements
   WTM 0                       ; wait for message on channel 0
   TXM 1                       ; transmit message on channel 1
   RPT ((P1=0)&(S4<P0))        ; repeat until cond. true
      SEQ 2                    ; sequence of 2 elements
         RDE 3                 ; read environment. var. #3
         CON 2                 ; concurrent of 2 elements
            ACT 42             ; execute activity 42
            CAS 2              ; case of 2 elements
            (P1>2)             ; condition for first element
            (P0=S4)            ; condition for second element
               ACT 12          ; first elem.: execute activity 12
               ACT 5           ; second elem.: execute activity 5
```

A.1.2 The Simulator Clock

SIMPAL works with cycles and ticks. A clock tick is defined as the execution of a complete cycle by the simulator. But what is a cycle?

The simulator reads the processes written in the PAL language pseudo-code and interprets the various primitives and control constructs in a continuous way without

interruption until it cannot execute any non-blocking element any more. The simulator starts performing these operations on process #0 and continues until the last process in the system, always switching from one process to the next one when all parallel threads of the process are blocking elements. At the end of the last process a single simulator cycle is completed.

NOTE: Blocking elements are: ACT, WTM, TXM, WTE, DLY. Non- blocking elements are: STS, STP, RDE, SEQ, CON, ALT, CAS, RPT.

From this we can see that it is a very dangerous mistake to have a PAL loop of non-blocking elements since it can (and almost always will) generate a deadlock.

An example of this kind of mistake is:

```
RPT (P0 > 1)
    SEQ 2
        STP 1 (P1+1)
        STS 3 (P0/2)
```

Since the elements inside the repeat loop are non-blocking, if the initial value of P0 is less or equal than 1, the simulator will only start executing the elements of the next process whenever the condition of this repeat is satisfied, i.e., when P0 > 1. However, since the elements inside the loop do not affect the value of P0, the simulator will never exit the loop.

This case is similar to a deadlock in a multitasking system, in which other tasks would not modify the value of P0. However, the situation is not exactly the same. Another task could indeed modify P0 and the repeat would be ended when this task is executed after the next context switch. Such a concept does not exist in PAL since the coordinator, which executes the processes' elements, is supposed to be a sequential uniprocessor machine. The user could write another PAL process that would modify the value of P0, but this process would never get access to the CPU after the execution of the repeat started.

NOTE on NOP: To obtain a NOP instruction using the PAL language, without

generating a system deadlock, a DLY (0) should be used. This is a delay that will take one clock tick to execute.

A.2 The Environment

The real world to where an industrial control system is connected is made of three major elements: Actuators, sensors and events. All these can be represented using SIMPAL. These are handled through the use of environment variables, automatic variables, arrival variables and events.

A.2.1 Direct Variables

In terms of sensors and actuators, SIMPAL simulates them via environment data variables. These are variables declared as in other high level languages, having a type (byte, integer, real or long) and an initial value. The purpose of these variables is to represent real world entities such as temperature, pressure, speed, etc. The values of these variables can be modified randomly or by special activities, as we will see later.

A.2.2 Indirect Variables

These are environment variables that are automatically modified by the simulator according to a specified flag. They are used whenever an action has to occur in the environment, independent of the PAL activities themselves. For example, the temperature of a heated tank is should automatically decrease when the heater is turned off, and should increase when the heater is turned on.

The previous case would be represented as:

5 3 +0.10 3 -0.15

The action taken by the simulator is: If variable #5 is true, the variable #3 is incremented by 0.1. If variable #5 is false, the variable #3 is decremented by 0.15. Variable #5 represents the heater (value 1 when ON, 0 when OFF) and variable #3 represents the tank temperature.

A.2.3 Event Variables

These are variables whose values are modified randomly whenever an event occurs. For example:

1 2 4 50

This means that event #1 will modify the value of variable #2 whenever it occurs. The value will be randomly generated between the limits 4 and 50.

A.2.4 Events

In terms of events, SIMPAL needs to know how many events are configured in the system and for each of them, its type and associated parameters:

1. *Type 0: The event is cyclic.*

   Parameters: period, time of first occurrence

2. *Type 1: The event is probabilistic.*

   Parameters: probability of occurrence

3. *Type 2: The event is of type time-interval.*

   Parameters: None

Cyclic events are those that occur periodically (!) with the specified period. Probabilistic events are those whose probability of occurrence is the one given. Time-interval events are similar to probabilistic events, but they will always occur between a minimum and a maximum time. Whenever this kind of event happens, its next occurrence is determined by generating a random time between the limits provided.

Each event also has an associated deadline for execution of its response. For example, a '500' deadline means that all processes that are triggered by this event must finish execution until 500 units of time after the event occurred.

Following the description of the type of the event, it is necessary to specify if the event triggers a delayed event or not.

A.2.4.1 Delayed Events:

These are events that are triggered by any other event (of type 0 ,1 or 2) after a specified amount of time.

The next items to specify are the event IO list, the size of its queue and number of processes that are triggered by the event (the processes that only start execution when the event occurs). It is necessary to specify for each process which IO list it is using.

A.2.4.2 IO Lists

They are lists of variables belonging either to a process or to an event. For example, when an event triggers a process, the environment variables specified in the event's IO list are transferred to the process. The process' variables specified in the process' IO list will receive the values of the environment variables.

A.2.4.3 Queues

The size of the queue of the event will determine whether the event is volatile (size 0: no event can be stored), latched (size 1: only 1 occurrence can be stored) or queued (the number of occurrences queued specified by the size of the queue).

A.2.4.4 Time-Interval Events Parameters

Finally, after the description of the events, it is necessary to specify minimum and maximum time of the time interval events.

A.3 Definition Of The Activities

The activities executed by the slaves for the processes are defined by two parameters:

- *Duration: This is the specified amount of time that an activity will take to be executed by an slave. The default value is 1 tick.*
- *Slaves: This represents the number of slaves in which an activity may be executed. In the real world certain activities may be only executed by very specialized*

processors (for example an FFT in a DSP) and can not be executed by any other slave in the system. The specification is done by telling the simulator in how many slaves the activity can be executed and then which they are. A 0 (zero) number of slaves means that the activity can be executed by all the slaves in the system.

```
3              ; 3 activities to be defined
0 20 1 4       ; Act. #0 lasts 20 ticks - only 1 slave (#4)
1 10 0         ; Act. #1 lasts 10 ticks - all slaves
2 5 2 0 7      ; Act. #2 lasts 5 ticks - two slaves (#0,#7)
```

Activities not listed or not defined by the user will have the default execution time and may be executed by all slaves.

A.3.1 Special Activities

Some activities have special meanings. They are necessary to simulate actuators and to have some feedback from the environment after activities are executed.

NOTE: These activities are built in the system. They do not (and cannot) have their parameters modified.

They are:

- *DEL_EVT (#95): The execution of this activity will schedule the delayed event whose id is specified in the process variable #0 (P0, which must be of type byte). The event will occur after the time randomly calculated by the simulator (which is between the minimum and maximum specified).*
- *INC_VAR (#96): When this activity is executed, the environment variable specified in the process variable #0 (P0) is incremented by 1.*
- *DEC_VAR (#97): When this activity is executed, the environment variable specified in the process variable #0 (P0) is decremented by 1.*

- *SET_VAR (#98): This activity simply sets the environment variable specified in the process variable #0 (P0). The environment variable will be used as a flag by other special activities.*
- *RESET_VAR (#99): Resets the environment variable whose number is specified in the process variable #0 (P0).*

A.3.2 Redundant Activities

SIMPAL allows the use of some special activities that must be executed simultaneously by more that 2 slaves in a redundant manner. The objective is to prove how well PAL performs when reliability is needed and estimate which are the overheads imposed by such activities. These are:

1. *DMR - Double Redundant Activities: These activities are executed simultaneously by two slaves. DMR activities are those whose id number ranges from 20 to 29.*
2. *TMR - Triple Redundant Activities: These activities are executed simultaneously by three slaves. TMR activities are those whose id number ranges from 30 to 39.*

Note that if one of these activities is first (after scheduling) in the ready-to-run queue, it will only be allocated to the slaves if the two or three slaves are available. If they are not, in order to assure that the activity will eventually be allocated to the slaves, no activity at all will be allocated, even if slaves are available for non redundant activities.

A.4 The Main Window and On-Line Commands

In this section we explain the main window that you see on the screen and then discuss the commands available.

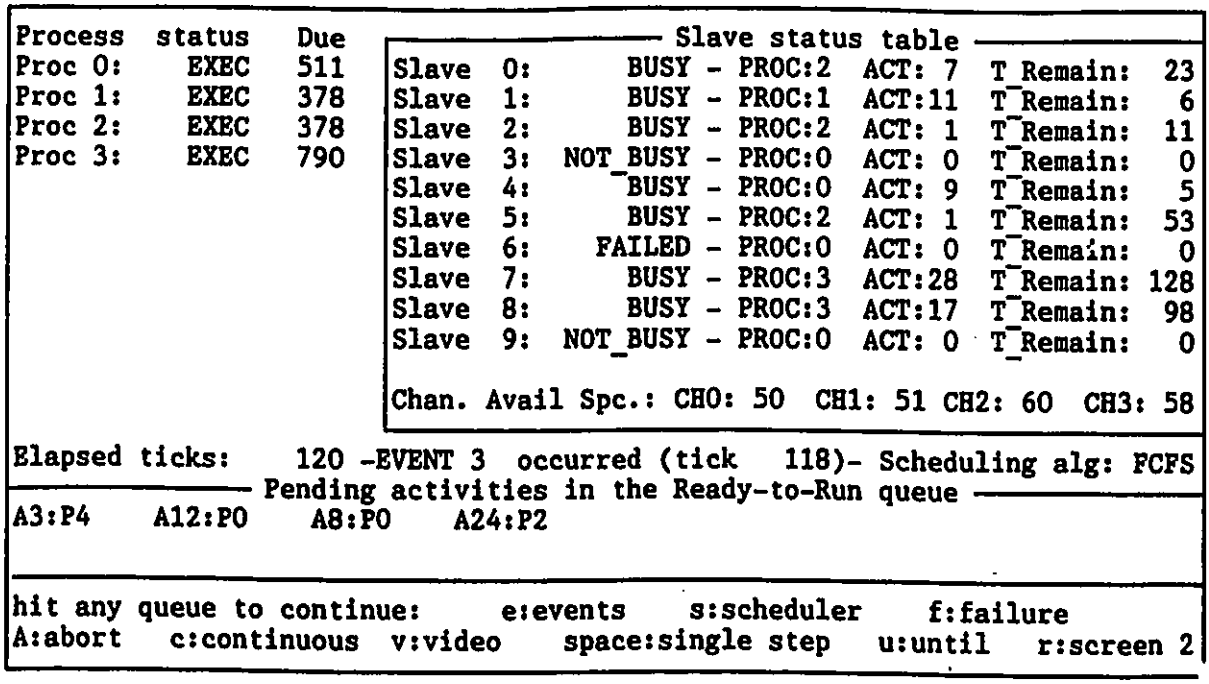

Figure A.1: SIMPAL's Main Window (DOS Version)

A.4.1 The Main Window

After a brief introductory (welcome) window, the user is presented with the main window, which is composed mainly of 7 fields. These are shown in figures A.1 and A.2 and described below.

1. *The Process Status Field*

   This field is located at the top left corner and is made of 22 columns by 16 lines. At the top line you can see the message 'Process Status Due'. *The information given in this area concerns the status of the processes and their due date (when they are supposed to finish execution). The possible statuses are:*

   - INACTIVE: *The process either has not begun execution yet or has already finished execution.*
   - READY: *The process is ready to execute. These is usually the case at tick 0 for processes that are not triggered by events and for processes that were*

Figure A.2: SIMPAL's Main Window (Unix Version)

just triggered by events.

- ACTIVE: *The executive (operating system) is executing a primitive(s) of this process.*
- EXEC: *One or more slaves are executing activities for this process . Note that a process may be executing activities and primitives at the same time. In this case its status is ACTIVE.*
- BLOCKED: *The process is waiting for the execution of a blocking element, such as an activity, WTM or DLY primitives.*

2. *The Slave Status Table Field*

   This field is located at the top right corner and is composed of 12 lines by 50 columns. Its purpose is to show the current status of the slaves, which activity they are executing, for which process and which is the execution time left of the activity.

   At the top line you can see the message 'Slave Status Table' *and at each line corresponding to a slave, the message 'Slave* id *PROC:* proc *ACT:* act *T_Remain: ttt'.*

   The three possible status of the slaves are:

   - BUSY - *When the slave is executing an activity*
   - NOT_BUSY - *When the slave is idle*
   - FAILED - *When the slave has failed*

3. *The Communication Channels Field*

   This field is composed of a single line located at the bottom of the Slaves Status Table field and shows the available space of each of the first four channels in the system (channels 0 to 3).

   The message seen is 'Channel Avail. Space: size'

4. *Elapsed Ticks, Events Occurred and Scheduling Field*

This field is composed also of a single line (line 18) and displays three information: The current clock tick (message 'Elapsed Ticks: tt'*), the last event that occurred and its tick of occurrence (message* '-Event #e occurred (tick #tt)'*) and the current scheduling algorithm (message* 'Sch. alg. ALG'*).*

5. *The Ready-To-Run Queue Field*

   This field is located between lines 19 and 21 and displays the contents of the ready-to-run queue in terms of activities and processes. Up to 16 elements of the queue can be shown in the form P#:ACT#*, i.e. process number and activity id.*

6. *The Options Field*

   The last 2 lines at the bottom show the current options available and display the message 'Press any key' *when the simulator needs a key stroke to proceed.*

7. *The Processes Elements Field*

   Available only on Unix systems, this field is located at the half right side of the screen. It shows all the processes elements and their current status (active, pending, blocked). Using this feature, the user is able to trace the processes and also produce an animation of the whole system.

A.4.2 On-Line Commands

Once the simulator is running the following options are available on-line (these options are invoked by pressing a single key):

1. *'A' - abort the simulation:*

   The simulation is stopped and no statistics are generated. This option is mainly used when an error is found at run time by the user and the results would be meaningless. Note that a capital 'A' must be pressed.

2. *'c' - continuous mode:*

The simulation enters the "continuous mode", in which each cycle (a single clock tick) is executed without interference of the user, as opposite of the single step mode, where a key stroke from the user is required at each cycle of the simulator. All previous parameters are maintained, with the exception of "single-step".

If the 'video-output' option is on, all data will be updated and displayed on the screen at each cycle. If it is off, only the current tick number and the event occurrences will be displayed. In the latter case, the simulation runs much faster (roughly 20 times) since all video output routines are performed through slow Bios calls in order to preserve compatibility among machines with very different video cards.

3. *'e' - events*

This option calls a pop-up menu and a window. The window shows the status (number of ticks to occur) of all (up to a maximum of 7)'cyclic', 'time-interval' and 'delayed events'. This is particularly useful when you need to know when a specific event will occur.

The probabilistic events are not show since it is impossible to know when they will occur (they are evaluated randomly at each cycle).

To return to simulation press any key different from 'c'.

Being in this event window, a second menu can be called by pressing the key 'c' (configure events):

- *'c' - configure events:*

  This option calls a second pop-up menu which allows the user to configure and modify the parameters of the existent events in the system (those that were declared in the SIM_ENV.n file).

  When this menu is entered the user is asked for the generation method for each of the events in the system (from event #0 to event #n). If you do not want to change the current method and its parameters all you have

to do is press Return and then you will be asked again for the generation method of the next event to configure.

The methods available are chosen by pressing:

- *'0' - Cyclic event: The simulator then asks for the period and the tick of the first occurrence of the event. The period must be positive and the tick of first occurrence must be greater than the current tick.*
- *'1'- Time-Interval event: No more questions are asked. The parameters (minimum and maximum times) cannot be modified once the simulator started execution.*
- *'2'- Probabilistic event: The simulator then asks for the probability of occurrence of the event, which must be a floating point number between 0.0 and 1.0.*

4. *'f' - failures:*

This option creates a new window in which parameters related to the reliability of the system can be modified. Basically, it is possible to modify the probability of a slave failure, to provoke the immediate failure of a slave and to change the parameters of failures frequency and recovery time.

The major objectives of having this option is to allow the study of the effects of reliability and redundancy in PAL based systems.

Two sub-options are available:

- *'m' - make slave fail now:*

  By pressing 'm' the user can force the failure of the slave whose number (id) the simulator asks. The status of the slave will change to FAIL at the next clock tick. The activity this slave was executing, if any and if not a redundant activity, will be restarted by the system in another slave. In fact, the activity is back to the ready-to-run queue with the highest priority among all the activities there.

The simulator will ask for the slave's id. Enter any valid id (from 0 to the number of slaves minus 1) and press Return.

You are now able to force another slave failure. In fact, it is possible to force any number of slaves to fail simultaneously.

- *'p' - change probability of failure:*

  This sub-option allows the change of the probability of a slave failure, whose initial value is 0.0. The simulator asks for a new value. Enter any value from 0.0 to 1.0 and press return.

- *'f' - change recovery time and failure frequency (period)*

  This allows the user to specify the the frequency of failures in a specific slave. The simulator will ask for the slave's id, the period between failures and the recovery time. Basically, cyclic failures will be generated in the specified slave with the complete cycle being the specified period plus the recovery time.

  Note that this capability can be applied to only one slave. A redefinition of the parameters will invalidate a previous definition.

To go back to the main window, just press Return again.

5. *'r' - second screen:*

   The entire screen changes and the value of all variables in the system are shown. These include the environment variables, the system (global) variables and the local variables of the first four processes. Besides the variables, the priorities, due date, slack times of these processes are also shown. To go back to the main screen just press Return. While in this second screen, the user may modify the value of any environment variable by selecting the option:

   - *'v' - modify environment variable. The simulator asks for the number of the variable to be modified (one of the environment variables) and then for its new value. Note that the type entered must be compatible with the type defined for the variable in configuration file, i.e. if the variable is defined*

as an integer the value entered must be an integer (not a floating point or a long).

6. *'s' - scheduler algorithm:*

   This option creates a new window which shows the scheduler algorithms available and asks for a new algorithm.

   If you do not want to modify the current method, just press Return or Esc, otherwise press the number which corresponds to the desired algorithm:

   - *'0' - FCFS, First Come First Serve: The activities are allocated to the slaves according to the order in which they arrived in the Ready-To-Run queue.*
   - *'1' - LACTF, Longest Activity First: The activity with the longest execution time is allocated first.*
   - *'2' - SACTF, Shortest Activity First: The activity with the shortest execution time is allocated first.*
   - *'3' - EARDF, Earliest Response Due First: The activities of the process with the closest deadline are allocated first.*
   - *'4' - HPRIF, Highest Priority First: The activities of the process with the highest priority (lowest value) are allocated first.*
   - *'5' - LSSLF, Least Static Slack Time First: The activities of the process with the smallest initial slack time are allocated first.*
   - *'6' - LDSLF, Least Dynamic Slack Time First: The activities of the process with the smallest dynamic slack time are allocated first.*

7. *'u' - run until:*

   This option creates a small window and asks the user for a tick to stop at. The simulation will continue until that tick is reached.

   The mode is changed to 'continuous-mode' and the other parameters are not changed. If the 'video-output' option is on , the screen will be updated at each

cycle.

If you want to stop before the specified tick, just press any key. If you are in the 'run-until' menu but changed your mind, just press Return or Esc and you will be back at the main screen.

8. *' ' (Space bar) or any other key:*

   Pressing space bar or any other key not allocated to one of the functions listed above will put the simulation in 'single-step' mode and will run the simulator for just one more step, when the user will be asked to press any key again.

A.5 The Statistics File

At the end of each simulation, a file is generated containing relevant statistics of the simulation. These include data about the Ready to Run queue, the slave processors, the processes, etc... The file is mostly self explanatory but the most important items are explained below. The name of the file is STAT.nnn, where the extension nn depends on the configuration of the system, the number of slaves and the final scheduling algorithm used. The first digit represents the configuration loaded (from the files SIM_ENV and SIM_PROC). The second digit represents the number of slaves in the system. The last digit represents the scheduling algorithm used (for the codes see section 1.6.1.5).

For example:

The file STAT.381 *contains the data obtained in the simulation of version 3, using 8 slaves and the Largest Activity First algorithm (#1).*

A.5.1 Run Duration

After a brief explanation about the parameters used, the file shows the run time duration. This is an equivalent time in minutes and seconds to the number of ticks that the simulation lasted. It is simply obtained by multiplying the number of ticks by a constant. Currently, it is estimated that a tick is equivalent to about 2.5 ms and the multiplying factor is derived from this.

A.5.2 Event Configuration And Handling

Following the run duration there is a description of the events in the system, as they were configured at the last tick of the simulation.

The event handling data shows a set of statistics for each of the events configured in the system. The 'occurred' field shows the percentage of time that the event happened. A 1% value means that the event happened in 1% of the total number of ticks that the simulation lasted. The 'of total' field shows the percentage of occurrences of this event compared with the total number of occurrences (independent of time). The 'queued' and 'missed' fields correspond to the percentage of time (ticks) that the event was queued (unable to be used immediately when it occurred) or missed (its queue was full). Finally, the 'detected' field shows the percentage of the occurred events that were actually used for triggering processes (responses) and it will be the same as 'queued' except when the event is volatile.

Next, statistics about the queues are shown: average size, maximum size, minimum,average and maximum time that an event spent in the queue (note that incomplete responses,i.e. responses (processes) that started but did not finish, are not counted).

Finally, the file shows minimum, average and maximum complete response times, i.e. the time spent from when the event occurred until when all its triggered processes finished execution.

A.5.3 Slaves Statistics

The number of slaves and the their usage are also shown. A slave usage of 11% means that this particular slave was busy (allocated to a process, executing an activity) 11% of the total elapsed time of the simulation.

A.5.4 Communication Channels Statistics

In this section the maximum utilization of the communication channels is shown along with the respective tick in which it occurred. The maximum utilization corresponds to

the minimum buffer size. The initial buffer size is also shown.

A.5.5 Ready-To-Run Queue Handling

These statistics refer to the size of the ready-to-run queue after the scheduling is done. This queue contains the activities that are ready to be executed by a slave. For instance, a 10% value for the size 1 item means that the ready-to-run queue had 1 element queued (blocked, unable to be executed) in 10% of the total number of ticks.

Usually a good scheduling algorithm will generate large numbers (very close to 100%) in the size 0 item. In general, the more times an activity is blocked in the ready-to-run queue, the more a process will be delayed.

A.5.6 Process Statistics

These statistics are related to the processes in the system. Each process has a set of related statistics and these are shown in different forms whether the process finished execution one (or more) times or not during the simulation. In the latter case, minimum, maximum and average values are shown (the minimum and maximum are associated with the occurrence of the process in which they occurred). Please note that a process may complete execution many times during a single simulation (processes may be triggered by events, which may occur many times).

Whenever a process did finish execution at least once, its minimum and maximum execution times are shown (and in which execution of the process they occurred) as well as the average execution time. The same is presented for the expected execution time of the process and the delays achieved (which could be negative whenever the processes finishes execution earlier than expected).

The statistics are divided into two categories: The first show the total number of ticks that a process is blocked or waiting or executing a specific primitive without considering the effect of parallel threads. For example: A process may have two parallel threads, each of them having an activity to be executed by the slaves. The time spent executing activities in the slaves is shown in the 'WEX' statistic. When both activities

are being executed, the value of this statistic will be incremented by two, however only one time tick will have been elapsed.

The second category will show this statistic under the name 'CWEX'. The value of CWEX would only be incremented once in the above case, and it would correspond to the effective time that the process was blocked only executing activities. This second category of statistics is shown under the title 'waiting concurrently data'.

Please notice that you may not need these statistics. They are used by the simulator when the scheduling algorithm chosen is 'dynamic slack'.

The statistics are:

1. *WRX - The total number of ticks that a process was waiting for a message. Note that if a process was concurrently waiting for messages in parallel threads, each tick will count for more than one unit, the number equal to the total number of parallel WTM primitives being concurrently executed.*

2. *WTM - The same for the number of ticks that a process spent blocked because a channel buffer is full.*

3. *WEVT - Shows the total time that a process was blocked waiting for an event to occur (usually through the WTE primitive)*

4. *WSLV - It is the total time spent by a process while waiting for a slave when it has an activity ready to be executed. The activity may remain in the ready-to-run queue when all slaves are busy or when it can only be executed by a specific slave and it is busy.*

5. *WEX - As explained earlier, this statistic shows the total time that was spent by the slaves executing activities for a process.*

   Following these, the next set of statistics is shown, those related to the 'concurrently waiting' values. They are very similar to the previous ones:

6. *CWRX - The time spent by a process while waiting for a message. Two parallel WTM primitives will make this variable be incremented only once at each clock tick.*

7. *CWTX - The time spent by a process while waiting for the transmission of a message because the channel buffer was full.*

8. *CWEVT - The time a process spent concurrently waiting for events.*

9. *CWSLV - The time spent by a process while concurrently waiting for a slave to execute a ready-to-run activity.*

10. *CWEX - The concurrent time spent by the slaves executing the activities of the process.*

11. *CWALL - The concurrent time spent by a process while blocked by the primitives WTM, TXM, WTE, and while waiting for an slave. If the process is blocked in more than one of these cases at the same time, only one of them will count for this statistic.*

A.6 Running SIMPAL

To run the simulator itself is extremely easy, but remember that you need to prepare two files before the simulator can run, since it will read them prior to execution. These files are the file containing the environment description (SIM_ENV.n), which you can prepare using the program GEN_ENV, and the file containing the processes description (SIM_PROC), which shall be prepared using the EDITOR.

To run SIMPAL just type 'SIMPAL' followed by the options (which are optional!). The syntax is:

SIMPAL [/B:] [/C:k] [/E:k] [/F:f] [/I:] [/N:n] [/P:s:p] [/S:k] [/T:k] [/V:k] [/W:]

A.6.1 The Options

The options are optional(!) and may appear in any order. If these options are not present the default values are:

- *B-Batch mode: No*

- *C-Number of channels: 8*
- *E-Activity execution time:1*
- *F-Failure probability: 0.0*
- *I-Failure Detection at End: No*
- *N-Number of slaves: 8*
- *P-Performance factor of slaves: 1.0 (all slaves)*
- *R-Recovery and Failure Rates: None*
- *S-Scheduling algorithm: FCFS*
- *T-Simulation time: 2000*
- *V-Version to load: 0*
- *W-Wide Screen: No*

The options are:

1. *Option 'B'*

   When this option is used, the simulation will run in batch mode. No interaction at all with the user is required (although he is still able to use the on-line parameters but with partial visual display only).

   This option is very useful when performing extensive simulations on different files with different parameters. For example, it is possible to run many simulations on different versions at once or to modify the parameters such as number of slaves and scheduling algorithm on a single version.

   This requires the creation of a DOS batch file (extension bat).

   Example: Suppose you want to simulate version 5 using several scheduling algorithms. You would create a batch file, such as RUN.BAT, like this:

simpal /v:5 /s:0 /b:

simpal /v:5 /s:1 /b:

simpal /v:5 /s:2 /b:

simpal /v:5 /s:3 /b:

You will then be able to analyse the results in the files STAT.580, STAT.581, STAT.582, STAT.583.

2. *Option 'C'*

   Using this option the simulator will define the number of channels used for interprocess communication.

   Example: When the simulator is invoked with

   SIMPAL /C:4

   the number of channels will be 4.

3. *Option 'E'*

   This option informs the simulator which is the default execution time of the activities. This default value will be assigned to all those activities which were not defined in the file SIM_ENV.n.

   Example: Suppose you invoke the simulator with

   SIMPAL /E:25

   Then the default execution time of the activities becomes 25 ticks instead of 1.

4. *Option 'F'*

   Using this option the user is able to modify the failure probability of a slave (a value is randomly generated at each cycle and checked to see whether it is within the range specified by this probability). Usage:

   SIMPAL /f:0.001

   This will change the probability that a slave fails to 0.1%.

5. *Option 'I'*

   When calling the simulator with this option, the detection of failures will occur only at the end of the execution of the activities. This is a worst case condition that represents a real system which does not have any means to detect a failure except when it is expecting results from a slave and these results never arrive. Example:

   SIMPAL /i:

6. *Option 'N'*

   This option tells the simulator with how many slaves you would like to begin.

   Example: Suppose you invoke SIMPAL with

   SIMPAL /N:10

   The simulation will run with 10 slaves (initially).

7. *Option 'P'*

   The performance factor of a specific slave can changed using this option. The first number represents the slave's id and the second one its performance factor. An example:

   SIMPAL /P:0:2.5 /P:2:0.5

   This command tells the simulator that slave 0 runs 2.5 times faster than the normal slaves and that slave 2 runs at half speed of the other slaves.

8. *Option 'R'*

   This options defines a slave, its failure rate in terms of periods between failures and the recovery time after each failure. Example:

   SIMPAL /R:3:100:15

   This command defines the period between failures for slave 3 to be 100 units of time and the recovery time to be 15 units of time (so the actual period between failures will be 115 units of time). This is equivalent to having a probability of failure of 1% (1 failure each 100 units of time-not counting recovery time).

9. *Option 'S'*

   This option tells the simulator which scheduling algorithm to use. The possible algorithms codes are:

   - *0: FCFS*
   - *1: LACTF*
   - *2: SACTF*
   - *3: EARDF*
   - *4: HPRIF*
   - *5: LSSLF*
   - *6: LDSLF*

   For example, if you want to run the simulation using the Shortest Activity First algorithm, you would type:

   SIMPAL /S:2

10. *Option 'T'*

    This may be the most used option. It defines the simulation time.

    Example:

    SIMPAL /T:100000

    The simulation will run for 100000 units of time.

11. *Option 'V'*

    This option tells the simulator which version or configuration to load. They are related to the number 'n' that appears elsewhere in this manual when mentioning the environment file SIM_ENV.n and the process file SIM_PROC.n.

    Example: Suppose you invoke the simulator with

    SIMPAL /V:3

The simulator will read the files that contain the information about version 3 (Files SIM_ENV.3 and SIM_PROC.3).

12. *Option 'W'*

    This option must be used when a wide screen output is desired. Such output increases considerably the amount of information available for the user. It includes an animation of the processes execution. This option must be used **only in Unix Systems (DEC-3100)**.

A.6.2 Maximum Configuration: Sizes and Limits

In this section we list some of the current limits and maximum capacities of SIMPAL. Note that some limits may be larger than those that are actually shown by the simulator's main window. An example is the number of slaves, whose maximum is 19 but only 11 can be shown (due to lack of space on the screen).

- *Maximum number of Processes: 20*
- *Maximum number of Variables per Process: 50*
- *Maximum number of System Variables: 100*
- *Maximum number of IO Lists: 30*
- *Maximum size of each IO List: 12 variables*
- *Maximum number of Slaves: 19*
- *Maximum number of Activities: 99*
- *Maximum number of Channels: 12*
- *Maximum size of each Channel Buffer: 60 variables*
- *Maximum number of TXM issued: 32*
- *Maximum number of WTM issued: 32*

- *Maximum number of Finished Executions Storage: 400*
- *Maximum number of Events: 15*
- *Maximum number of Processes per Event: 5*
- *Maximum number of WTE issued: 10*
- *Maximum number of DLY issued: 10*
- *Maximum number of Environment Variables: 100*
- *Maximum number of Delayed Events per Event: 5*

Appendix B

Timing in PAL processes

In order to use the simulator to evaluate real-time characteristics of a system it is first necessary to understand how to calculate the execution time of a process.

For timing purposes, a process has basically three different control constructs. These are the SEQ (sequence), CON (concurrence) and ALT (alternation) constructs. They differ significantly in how their execution time is obtained.

Let us examine a few cases, assuming the activities execution times shown in Table B.1.

ACT 1:	20 units of time	ACT 5 : 10	units of time
ACT 2:	20 units of time	ACT 6 : 5	units of time
ACT 3:	25 units of time	ACT 7 : 10	units of time
ACT 4:	28 units of time	ACT 8 : 15	units of time

Table B.1: Activity Execution Times

B.1 A Sequential Process

A sequence is defined as a sequential execution of primitives or other control constructs. The execution time of a SEQ branch is therefore calculated adding the execution times of the individual components. Let us see a very simple example.

```
SEQ 4          ; sequence of 4 elements
    ACT 1      ; activity 1 (which takes 20 units of time)
    ACT 2      ; activity 2 (which takes 20 units of time)
    ACT 3      ; activity 3 (which takes 25 units of time)
    ACT 4      ; activity 4 (which takes 28 units of time)
```

Total execution time = T(act1) + T(act2) + T(act3) + T(act4)

Total execution time = 20 + 20 + 25 +28 = 93 units of time

B.2 A Concurrent Process

As opposite to the SEQ, a CON (concurrence) specifies that the elements in a set of primitives or other control constructs are to be executed in parallel and that the construct is only finished when all the participating elements are finished. So, in this case the execution time is given by the maximum execution time of the individual elements (always assuming infinite resources).

A very simple example is:

```
CON 4          ; sequence of 4 elements
    ACT 1      ; activity 1 (20 units of time)
    ACT 2      ; activity 2 (20 units of time)
    ACT 3      ; activity 3 (25 units of time)
    ACT 4      ; activity 4 (28 units of time)
```

Total execution time = MAX (T(act1), T(act2), T(act3), T(act4))

Total execution time = MAX (20, 20, 25, 28) = 28 units of time

B.3 A Mixed Process

A normal process will contains many combinations of the above. In each case, the execution time is calculated based on the execution times of the individual components. A simple example:

```
SEQ 2                   ; sequence of 2 elements
    CON 2               ; concurrence of 2 elements
        ACT 1           ; activity 1 (20 units of time)
        ACT 2           ; activity 2 (20 units of time)
    SEQ 3               ; sequence of 3 elements
        ACT 3           ; activity 3 (25 units of time)
        ACT 4           ; activity 4 (28 units of time)
        CON 3           ; a concurrence of 2 elements
            STP 0 (3)   ; set process condition
            ACT 5       ; activity 5 (10 units of time)
            ACT 6       ; activity 6 (5 units of time)
```

The total execution time is given by adding the execution time of both elements of the first control construct (SEQ 2). The first one (CON 2) will take the maximum execution time between ACT 1 (20 units of time) and ACT 2 (20 units of time), which is 20. The second one (SEQ 3) will take the sum of its three components execution times, which are 25 for ACT 3, 28 for ACT 4 and 10 for the CON 3 construct (the maximum between 0, 10 and 5).

Thus,

Execution time = T(CON 2) + T(SEQ 3)

Execution time = MAX(20,20) + (25 + 28 + T(CON 3)

Execution time = 20 + (53 + MAX(0,10,5)) = 20 + (53 + 10)

Execution time = 20 + 63 = 83 units of time

NOTE: *The last construct of the above process contained a STP primitive, which took 0 units of time be executed. As this STP, other primitives also are considered*

instantaneous such as STS, RDE, CAS, RPT, RDS.

B.4 Uncertainties Involved

The above examples work out very well and nicely because they were completely deterministic. However, in real world applications it may not be possible to know exactly when a primitive will finish execution beforehand. This is the case of the WTE, WTM and TXM primitives. It is impossible to know exactly when a specific event will occur or when the communication channels will be empty or full, in which case the WTM and TXM will be held indefinitely. In order to evaluate the execution time of a process it is necessary to estimate such times. An example:

```
CON 4        ; sequence of 4 elements
    ACT 1    ; activity 1 (20 units of time)
    ACT 2    ; activity 2 (20 units of time)
    WTE 0    ; wait for event 0 (? units of time)
    ACT 4    ; activity 4 (28 units of time)
```

The total execution time is given by the maximum of 20 (ACT 1), 20 (ACT 2), 28 (ACT 4) and the expected execution time of the WTE primitive. If the user knows that that event will occur very frequently then he can assume that the execution time of that primitive will be smaller than the others. Thus,

Execution time = MAX (20, 20, 28) = 28 units of time

However, if event 0 will not occur very often, we can expect that the longest time will be spent waiting for this event. If we estimate an average waiting time of 50 units of time, then

Execution time = MAX (20, 20, 28, 50) = 50 units of time

B.5 A Process Wit' ALT

The ALT (alternation) construct is similar to the CON (concurrence) construct in execution except that the parallel thread is considered finished as soon as one element finishes execution. Thus, the execution time of an alternation branch will be given by the minimum individual execution time of its compone 's. Note that this construct is used usually when uncertainties are involved (such as (ne of the elements being a WTE primitive) since there is little point in using it just for completely deterministic activities, although it may still be very useful when slaves are not always availalble (thus being impossible to be sure that the shortest activity will indeed finish first). Obviously, if there are enough resources, the element that will finish earlier would be known beforehand.

```
ALT 3          ;sequence of 4 elements
    ACT 1      ; activity 1 (20 units of time)
    ACT 6      ; activity 2 (5 units of time)
    WTE 0      ; (? units of time)
```

Execution time = MIN (20, 5, ?) = ?

In order to estimate the execution of a process containing such an element, the time that the WTE primitive will take must be estimated, as it was in the previous case. In this case, since an ALT thread is considered finished when the first element finishes, it can be safely assumed that the complete thread will take not more than 5 units of time to be executed.