National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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.

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.

Canada

A PROGRAMMING ENVIRONMENT FOR REAL-TIME MULTI-THREADED SYSTEMS

by
Christian Riva, B.A.Sc.

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

ISBN 0-612-11594-1

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.

Christian Riva

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.

Christian Riva

Abstract

In recent years, the rapid progress in the functionality, performance, and cost of VLSI technology has made multiprocessor computer platforms feasible and cost-effective. The key benefit of these platforms is their ability to offer true concurrency at the physical layer, through the replication of processor, memory and I/O facilities, and at the logical layer, through the support for multi-tasking and multi-threading. For many applications, such as those found in the real-time problem domain, the concurrent nature of these platforms bring numerous advantages. To this date however, because of the lack of real-time concurrent programming tools, delivering the potential benefits of multiprocessing directly to these applications has proven elusive. This thesis proposes the Multi-Activity Programming Environment (MAPE) which allows developers to exploit the concurrent capabilities of multiprocessor platforms while remaining consistent with the requirements of the real-time problem domain. MAPE provides a programming model and a set of supporting tools which facilitate the decomposition of an application into cooperating entities and allow for the specification of deadline-based time constraints. The MAPE components also support the allocation of the platform's resources and the modification of the application's functionality based on run-time urgency considerations.

Acknowledgements

I would like to acknowledge the immeasurable contribution of Dr. Moshe Krieger, my supervisor and friend whose words of guidance were instrumental in convincing me to embark on graduate work. His previous work on Multi-Activity Systems provided much of the philosophical underpinning for the research described in this thesis.

I would also like to acknowledge two groups of individuals who provided invaluable technical feedback during the course of the research and the writing of this thesis. The first group includes Declan Murphy, Scott Wilson, and George Cameron of Sun Microsystems Computer Corp. Their collective experiences with multiprocessors, operating systems, and multi-threaded software were pivotal in shaping my understanding of these topics. The second group includes my colleagues at the University of Ottawa, Voja Radonjic and Jean-Pierre Fiset. Their insightful comments questioned my approach, challenged my assertions, and cristallized my thoughts.

Lastly, I owe much to the support of family and close friends who saw me through difficult periods with their unfailing encouragement and who could always be counted upon to remind me of the reasons for pursuing this degree.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this chapter is to establish the scope of the thesis, present its motivations, and outline its goals. The first section reviews the relevant characteristics of multiprocessor platforms and describes the general properties of real-time systems. It also introduces the concepts that underlie the programming environment proposed in this thesis. The second section describes the main contribution of the research and the organization of the thesis.

1.1 Thesis Scope

In recent years, the rapid progress of VLSI technologies has led to a number of innovations which capitalize on the increasing capabilities and the decreasing costs of the resulting devices. The impact of these technological advances has been far-reaching, with the systems constructed from these components penetrating virtually every facet of society; from the medical, telecommunications, and transportation sectors to the research, finance, and consumer electronics sectors. One of the innovations which has became attractive, feasible and cost-effective as a result of VLSI technological advancement is the multiprocessor platform architecture.

Multiprocessor platforms consist of multiple microprocessors that are interconnected to share memory and I/O facilities. They are often characterized by the interconnection, interaction, and management of the constituent resources, and the degree to which they provide balanced operation [Tab90]. This thesis considers *tightly-coupled, symmetric* multiprocessors where the platform's resources are assembled around a shared bus infrastructure. In these platforms, resources share a unique image of memory, are managed by a single copy of the operating system, and communicate al-

most exclusively by reference. Data to be exchanged is written at and read from memory locations that are equally accessible by the interacting parties. Furthermore, the resources are relatively homogeneous, equidistant from one another in terms of communications delay, and suffer from no inherent bias for bus access.

The physical parallelism provided by multiprocessor architectures requires extensive support from the operating system software [Mar93]. Firstly, the operating system must coordinate the operation of the multiple resources in an orderly and efficient manner. In this respect, its goals are to maximize the utilization of the resources and to minimize the delays in accessing them. Secondly, the operating system must provide the means by which applications can access the parallel hardware. In this respect, its goal is to provide a set of abstractions that are adequate representations of the capabilities and attributes of the underlying hardware.

The operating system's traditional means of supporting application concurrency consists of the *task* abstraction [Tan92]. The task provides application developers with a single flow of control that operates out of a unique, protected address space. It is self-contained and independently schedulable on any of the processors. Tasks are individually protected by memory management hardware and as such their code and data cannot be manipulated by other tasks. Cooperating tasks must therefore communicate through the explicit movement of data from one address space to another and synchronize using the facilities of the operating system kernel. Also, the accumulated run-time state of a task must be fully removed before another task may be switched onto an available processor.

In many of today's multi-tasking operating systems, the overheads involved in performing these operations are relatively high, and in extreme cases, can reduce or nullify the performance benefits accrued by partitioning and executing the application concurrently. Consequently, although its effectiveness at supporting coarse-grain concurrency has been demonstrated, the task abstraction is considered inadequate to support the decomposition of an application into fine-grain streams of parallelism.

To provide finer-grain concurrency, present-day multiprocessor operating systems provide application developers with the *thread* abstraction [Tan92]. The thread also provides a single flow of control but operates out of an address space it shares with other threads. It is contained within a task and depending on the implementation, may be independently scheduled on the processors. The threads of an address space have no hardware protection from one another and in many instances, can communicate without explicit data movement, synchronize without kernel intervention, and be switched without removal of the task's run-time state information. Because of these

properties, the overheads incurred while coordinating threads are generally lower than those of coordinating tasks.

Multiprocessors provide application developers with an underlying hardware architecture that offers true resource concurrency, and a means to access the parallelism with minimal overheads. These characteristics, when properly exploited, can lead to heightened application performance and decreased system costs. As a consequence, the type and scope of the problems that are being addressed with these platforms is widening, and their use is becoming increasingly pervasive.

Multiprocessors are being used as replacement platforms in legacy applications for example. In this capacity, they inject a relatively inexpensive performance boost into an older system and extend the useful life of the legacy application. Usually, the transformation of these applications into concurrent entities is performed using automated tools such as parallelizing compilers and front-end source code pre-processors.

They are also being used to underlie applications addressing inherently concurrent problem domains. In these circumstances, their use is justified by the close coupling that they offer between the problem and solution spaces. A concurrent solution to a concurrent problem is more representative, and thus more easily understood by a project's design, development, and maintenance specialists.

Multiprocessors are also likely platform candidates when concurrency is imposed by the nature of the problem domain. Real-time, constraint-based systems for example, must be able to service the multiple, possibly simultaneous events generated from their immediate environment within pre-determined time constraints. In these applications, multiprocessor platforms may be necessary to meet both the physical concurrency requirements and the stringent timing requirements of the environment [Sta88]. Further, because of the potentially wide variety of operating conditions that may be faced, these systems are required to have the ability to recognize and adapt to dynamic conditions, and to ensure minimal delay in provisioning the required resources.

The real-time problem space is vast, and multiprocessors are not suitable, nor are they required in all circumstances. A general classification of real-time systems has been proposed by the Intel Corporation [Wil90]. *Event controller*s are the lowest level of the hierarchy and are responsible for controlling dedicated processes. They are typically based on microcontrollers and embedded within the environment they serve. Since it does not usually contain network, disk, or man-machine interfaces, an event controller's system software requirements are limited and often implemented by a simple ROM-based executive. This class of real-time system is found primarily in data acquisition, instrument monitoring, and component control applications.

Data controllers form the middle level and are responsible for controlling multiple processes and for providing some limited processing of the data stream. The systems are typically constructed from embedded uniprocessors that provide a higher-performance computing base. They are equipped with a restricted-functionality operating system which supports basic physical memory management, disk support, task management and networking. The use of data controllers spans a host of applications, from assembly-line control, to signal processing and data communications.

System controllers are the highest level of the hierarchy and are used to control and oversee the execution of event and data controllers. Their primary role is to process, display, store, and act on the information collected. System controllers are often equipped with sophisticated computing platforms and are usually not embedded within the immediate environment. The operating systems provide support for virtual memory and inter-task communication, multi-tasking, multi-threading, and flexible I/O facilities that provide network, disk, and man/machine interfaces to the system. This class of system is found in aerospace and avionics simulation, factory automation, power plant control, and telecommunications applications.

System controllers are the focus of this thesis. The thesis proposes the Multi-Activity Programming Environment (MAPE) for real-time, multi-threaded applications and develops the fundamental components required to support it. The goals of the environment are the following.

- to provide a programming model which addresses the execution and coordination concerns of concurrent applications
- to provide a programming model which makes provisions for the defining characteristics of the real-time problem domain
- to provide consistency with an effective approach to the development of concurrent real-time systems
- to provide the support facilities required to merge the programming model with a multiprocessor threads implementation

The programming environment developed in this thesis relies on principles that are largely analogous to those of the multi-activity embedded system development environment [Kri94]. The multi-activity approach supports the notion that proper development of embedded real-time systems requires alternative design practices. It describes a comprehensive methodology for defining the architecture of embedded systems and for designing their application and system software components.

In the multi-activity development environment, the embedded system's requirements, design, and implementation specifications are consistent and rely on the same set of common principles. The approach's guiding principles are the following:

- the system's reactions to events are formulated as *responses*, or groups of executable *activities* which are either statically or dynamically instantiated and destroyed, and allowed to synchronize and communicate.
- the system's execution and coordination concerns are separated and treated independently to permit the issues relating to the *usage* of the system resources to be considered in isolation from those relating to the *organization* of the usage of the resources.

The multi-activity approach aims to simplify embedded systems development to a level where they can be designed and implemented largely by application specialists, thus limiting the requirement for costly cross-functional teams of application and computer specialists. The design environment is prototype-based and provides abstractions which are more natural, and more representative of the entities manipulated by application specialists.

In this thesis, the basic principles of the multi-activity approach have been applied to the design and development of real-time multi-threaded applications. The hardware and system software components of a general-purpose multiprocessor platform are not modified. Instead, a multi-activity *layer* has been developed between the real-time application and the system software to permit an application to be constructed in compliance with multi-activity principles. The multi-activity layer is provided by the MAPE programming environment components which consist of the MAPE programming model, the C and CSL programming languages, and the MAPE runtime system.

1.2 Contributions and Outline

The research described in this thesis demonstrates the feasibility of developing a programming environment that simplifies real-time multi-threaded application development. It provides an analysis of the key requirements of such an environment and develops the high-level models, languages and run-time support facilities needed to realize it. The resulting programming environment facilitates the decomposition of time-sensitive applications by allowing the execution and coordination requirements of the cooperating entities to be specified independently, using dedicated high-level

abstractions. It also simplifies the treatment of the real-time considerations of these applications by allocating the underlying systems resources based on the passage of time and by permitting the functionality of the application to be modified dynamically during execution.

Such an environment has the potential of simplifying the tasks of computer specialists involved in real-time multiprocessor application development. With additional support tools, a similar environment could be constructed to permit computer-literate application specialists to not only specify the requirements of real-time applications, but to participate in their design and development.

The thesis is organized into three parts. Part I encompasses chapters 2 and 3 and aims to derive the key requirements of a real-time, multi-threaded programming environment. Part II encompasses chapters 4 and 5, and appendices A and B. Its objective is to describe the MAPE programming environment and to present the sample programs used to demonstrate its functionality. Finally, Part III encompasses chapter 6 and offers some concluding remarks on the contents of the thesis and on possible future directions.

Chapter 2 focuses on exploiting, programming, and supporting concurrency in the context of generalized applications. The chapter discusses the process of partitioning an application into cooperating work units, and the factors that influence the decomposition decisions. It also describes three approaches to programming the execution and coordination requirements of the resulting work units. The chapter also offers a discussion of the three established approaches to the implementation of threads on multiprocessor platforms. It concludes with a brief summary and the basic set of requirements for the programming environment.

Chapter 3 describes real-time systems, presenting their operational characteristics and classifying them based on the nature of their operating conditions. It discusses the various contraints that govern the development and execution of a real-time system and reviews the impacts these have on its ability to remain consistent with both the functional requirements of the application and the timing constraints of the environment. It also describes three approaches to the development of real-time systems and applications. The chapter concludes with a brief summary and lists an additional set of requirements for the programming environment.

Chapter 4 describes the Multi-Activity Programming Environment (MAPE). The chapter presents the main components of the environment and the role they play in the realization of a real-time, multi-threaded application. It discusses the programming model and its associated notation for coordination, and outlines the execution and coor-

dination languages used to specify the application's requirements. The chapter also describes the MAPE runtime system and the facilities which allow dynamic modification of the application's functionality and run-time adaptation to time constraints. The chapter concludes with a brief summary and an outline of MAPE's support for the requirements identified in Part I.

Chapter 5 presents sample programs which demonstrate the functionality of MAPE. The demonstration programs attempt to validate the concepts and principles by exercising the MAPE components with a number of response scenarios. For each example, the chapter describes the goal of the demonstration, the response or set of responses that implement it, and the results obtained during execution on a Sun Microsystems Inc. SPARCcenter 2000 Symmetric Multiprocessor System. The chapter concludes with a brief summary and a list of the MAPE capabilities that were demonstrated.

Chapter 6 concludes the thesis by offering a summary of its contents, a review of its contributions, and some potential directions for future research.

Appendix A describes the MAPE runtime system's Application Programming Interface (API). It presents both the call syntax and the semantics of the primitives that are visible to the MAPE programmer.

Appendix B includes the source code and raw results for the sample programs described in Chapter 5.

Chapter 2

Concurrent Applications

The purpose of this chapter is to investigate how to exploit, program, and support application concurrency. The first section reviews the main factors that influence the process of partitioning an application's functionality. The second section explores three broad approaches to formally describing the execution and coordination concerns of the application's cooperating entities. The third section describes three approaches to the implementation of threads as the underlying vehicle for supporting concurrency. Finally, the fourth section summarizes the chapter and lists the basic requirements of the programming environment.

2.1 Exploiting Concurrency

To exploit the multiprocessor's concurrency capabilities, an application must be organized along parallel lines and its functionality partitioned into cooperating *work units*. The goals of the partitioning process are to specify the work that comprise the individual units and to define the interactions that occur between them. The two aims are closely related, with changes in the amount and nature of the work assigned to units having an influence on the number and content of their interactions. The process is largely informal and subjective, and yields a set of work units and an interaction profile that is not unique. An optimal partitioning usually requires a detailed understanding of the application and results from an iterative approach [Blu92].

The process by which the application is partitioned into cooperating work units imposes numerous constraints on the resulting entities. Firstly, selection of the functionality encapsulated in individual work units gives rise to *precedence constraints* between those that execute in sequence. Precedence constraints determine the order

of execution and the relative time at which the functionality is performed. Secondly, the functions assigned to work units gives rise to *exclusion constraints* between those that execute in parallel. Exclusion constraints are found between units which may potentially manipulate shared data elements simultaneously, and are meant to enforce mutually exclusive access and to guard against the presence of race conditions.

Ideally, these application-level constraints should be the only ones which influence the partitioning process. In practice however, the characteristics of the programming abstractions made available through the programming model also act as constraints. The abstractions allow developers to formally describe the execution and coordination requirements of the application's work units, and depending on the programming model, exhibit characteristics that may or may not reflect the properties of the underlying vehicle of concurrency. A model offering programming abstractions that are closely coupled to the underlying execution environment for example, can force knowledge of the thread implementation's properties into the application partitioning process.

2.2 Programming Concurrency

The concurrent programming model supplies the abstractions that permit formal specification of work units and their interactions. It is the vehicle through which the application is realized and transformed into a set of entities that can be executed and coordinated. The programming model, along with the formal requirements analysis aids, programming languages, and software configuration tools, provides the foundation for the application development environment.

The execution and coordination of the application's work units are the fundamental concerns which must be represented in the programming model. These two concerns are orthogonal and distinct from one another, with each having a well-defined, non-overlapping sphere of influence [Gel92]. Because of this property, the programming model can conceptually be defined in terms of an *execution model* and a *coordination model*. The models are complimentary to one another and when jointly considered, provide the complete programming model necessary to specify concurrent applications.

The execution model allows the developer to express the computations that an individual work unit must perform on behalf of the application. Firstly, the model provides the developer with the ability to specify the structure of the data elements that

are manipulated by a work unit. It defines the set of simple data types that are allowed and the rules for creating compound structures from the basic types. Secondly, the model provides the means to specify the operations that a work unit will sequentially execute. It formalizes the syntax and semantics of the data operators and outlines the conditional and repetitive path selection mechanisms at the developer's disposal.

By contrast, the coordination model allows the developer to express the interactions that occur at various points during the execution of the work units. Firstly, the model supports the mechanics of interaction by providing means to specify the points within the work unit's processing flow at which it may initiate or receive requests for interaction. Secondly, the model provides the developer with the ability to specify the nature of the interactions, once the interruption points have been identified. It defines the mechanisms for managing the existence of work units, for synchronizing their execution, and for communicating data elements between them.

Concurrent programming models can be classified as either *integrated* or *non-integrated* [Bal90]. Although both models fully address the execution and coordination concerns of applications, they do so using different vehicles. The following sections will describe the main characteristics of the integrated and non-integrated models and contrast their differences. The integrated model will be presented first, followed by two variants of the non-integrated model; one which relies on operating system and library abstractions for coordinating work units and another which relies on a distinct language.

2.2.1 Integrated Model

In the integrated model, the execution and coordination models are embodied in a single programming language [Bal90]. The integrated language includes constructs for creating and destroying work units, for describing the computations that each must perform, for communicating information between them, and for synchronizing their execution. Some integrated languages have been developed from first principles while others such as Concurrent-C [Geh86], are variants of popular sequential execution languages. The variants usually support the execution model of the base language from which they are derived and add the formal syntactic and semantic support for the coordination model.

A favorable characteristic of the integrated model is the level of abstraction that can be offered to the application specialist. The language supporting the model can provide a rich set of abstractions that hide many of the low-level details related to the mechanics of managing an application's concurrent work units. In effect, by abstract-

ing the execution and coordination concerns, the language isolates the developer's *view* of concurrency from the underlying platform's *implementation* of concurrency.

The responsibility for reconciling the differences between the developer's views and the underlying platform's capabilities rest with the language's *runtime system*. Its main roles are to translate the language's abstraction of a work unit into the platform's vehicle for concurrency, and the language's abstraction of an interaction into the platform's mechanisms for interactions. The language abstraction for a work unit for example, may map to a task or to a thread, depending on the architecture of the underlying platform. Similarly, interactions between work units may be implemented by messages or shared memory, depending on the available facilities and their characteristics.

The relatively high-level of abstraction that is possible with the integrated model provides further advantages with respect to the portability of applications. The use of high-level abstractions and a runtime system not only permits the language to be easily ported across platforms but also frees developers from the dependencies of particular systems.

An important drawback of the integrated model however, is its inflexibility relative to the requirements of special-purpose application domains. The model is inherently static and cannot be easily extended to address the specific properties of non-mainstream application domains. Heterogeneous distributed systems for example, require special provisions from the programming model for reliability and transparency. Similarly, real-time constraint-based systems require support for monitoring and managing the passage of time. The developers of these systems not only require an effective concurrent programming model, but one which addresses the attributes of their problem space.

The rigidity of the model stems largely from the close coupling it supports between execution and coordination. By fusing the two concerns into a comprehensive indivisible model, the approach forces developers to rely on other means, which may be inconsistent with the model, in order to address the non-mainstream provisions of their application domains. The inability of the model to adequately address the special provisions that are native to individual application domains can restrict the utility of the language. In the limiting case, the model may be complimentary to a small set of concurrent programming problems.

2.2.2 Non-Integrated Model: OS and Library-based Coordination

In the non-integrated programming model, the execution and coordination models are supported by separate vehicles [Bal90]. The first is a portable high-level language which embodies the execution model. This execution language provides the constructs which allow compute, conditional, and loop operations to be specified over a set of data. The second vehicle, which in this variant consists of a collection of OS and library services, embodies the coordination model and permits formal specification of the interactions that occur between work units. Calls to the services are inserted at key points in the sequential source code to invoke the OS and library mechanism that bring work units into existence, and allow them to communicate and to synchronize.

An advantage of this approach is its flexibility with respect to the needs of a variety of application domains. In essence, by not forcibly merging the execution and coordination models into one unifying language, the approach allows an application developer to build a customized programming model from the facilities provided by the platform. The execution model for example, can be chosen based on the language (C, Fortran) and tools (compilers, debuggers) that are either most effectively supported by the platform and its vendor, or most applicable to the execution requirements of the application. Similarly, the coordination model can be chosen based on the mechanisms that are available and their characteristics, or by the degree to which they address the needs of the application.

The flexibility inherent in this model however, presents some drawbacks when application portability is taken into consideration. By relying on a combination of the OS and libraries to embody the abstractions of the coordination model, the application becomes inextricably tied to the underlying platform. For example, although C is a highly portable language, an application written in C using the syntax and semantics of the UNIX system and library calls is not directly portable to other platforms. Since the C programming language supports an execution model and makes no attempts to support a particular coordination model, only the execution component of the application is portable, the coordination component remains platform-specific.

Recent efforts by the IEEE POSIX standards committee however, have made advances in minimizing the differences between operating system platforms. By attempting to standardize the programming interface, the committee has alleviated or removed many of the portability difficulties that arise from the use of system and library services on various platforms. The POSIX 1003.1, 1003.4 and 1003.4a specifications for ex-

ample, have formalized many of the abstractions that form a coordination model's foundations.

The flexibility of the model stems largely from the low level of abstraction it offers to developers. Unlike the integrated model which supports *language* abstractions that are mapped to *operating system* abstractions using a runtime system, there is effectively no distinction between the abstractions manipulated by the programmer and those offered by the operating system. In essence, the programmer's view of concurrency is identical to the underlying platform's implementation of concurrency. This characteristic is unfavorable because it requires developers not only to identify and select the abstractions that best meet the application's needs, but also to make correct use of the mechanisms that manipulate those abstractions. When threads are used as the underlying operating system abstractions for example, interactions that involve synchronization and communication can be implemented using any of the following mechanisms.

- messages
- shared variables
- mutexes
- reader/writer locks
- condition variables
- semaphores
- pipes
- files
- signals

As a result of this array of mechanisms, depending on the expertise of the developers, the opportunity for introducing errors in the design and development phases may be considerable. Most application specialists are trained to address *application* issues and can be expected to derive a workable partitioning of the application and a corresponding profile of its coordination. Using this variant however, they must also implement the concurrent application in a platform-specific manner, requiring them to become intimately familiar with the details of the underlying system platform.

Further, the knowledge acquired by the developers must also be sufficient to debug the application. The difficulties that are inherent to concurrent programming are not masked at this low level of abstraction, they are fully exposed and must be dealt with by the application developer. The coordination profile for example, is critical to the application and is one of the main source of avoidable errors that are introduced in the development process. An improper implementation can lead to race conditions, deadlocks and other errors that can be difficult to trace.

2.2.3 Non-Integrated Model: Language-based Coordination

In this second variant of the non-integrated programming model, the coordination model is embodied in a dedicated *coordination language* [Gel92]. The sequential computations are expressed similarly to the first non-integrated approach, using a separate high-level execution language. By using two languages, each orthogonal to the other, the major concerns of the concurrent programming model can be dealt with individually. The constructs which specify the sequential computations of work units can be separated from those that specify their organization. Once the separation of concerns is established, any execution language may be coupled to any coordination language to form a comprehensive programming model.

An advantage of the approach is its flexibility relative to the requirements of various application domains. The ability to select from a variety of complimentary language pairs allows the developers to construct the programming model that is most appropriate to the needs of their application. The execution language can be chosen based on its ability to model the computational properties of the application's functionality while the coordination model can be chosen based on its ability to model the characteristics of the interactions that occur between the work units.

Special-purpose application domains are particularly well positioned to take advantage of the approach's flexibility since it permits their unique, non-mainstream characteristics to be integrated into the programming model. The PAL coordination language for example, is aimed at meeting the requirements of real-time applications and offers support for organizing their functionality into responses to the environment's events [Kri88]. Similarly, the Linda coordination language addresses the needs of large-scale, distributed processing applications by offering abstractions that facilitate the rebalancing of an application's computational load based on the availability of network links and the congestion of host systems [Ahu86][Car89].

Another favorable benefit of this approach is the high level of abstraction that can be offered to the application developer. In a manner that is similar to the integrated model, the coordination language employs a runtime system which provides platform-independent abstractions for the interactions between work units. Unlike the previous approach, which forces developers to extend their knowledge down to low-level abstractions, this approach raises the level of abstraction up to the level at which they are familiar. They are not required to become intimately acquainted with the details of the platform that will support their application, only with the abstractions supplied by

the programming languages. The developers can thus more readily concentrate on achieving the optimal partitioning and the optimal coordination profile for the application.

The high level of abstraction also has benefits with respect to the portability of the resulting concurrent application. Since the developer's view of concurrency is contained within the runtime system, a port of the runtime system makes the coordination model embodied by the language available in identical syntax and semantics across various platforms. Therefore, by describing the application in a manner that is independent of the underlying operating system abstractions, this approach isolates the developers from the dependencies of particular platforms and reduces their efforts when a port of the application is required.

2.3 Supporting Concurrency

In uniprocessor systems, cooperating tasks have predominantly been the vehicle for expressing application concurrency. These tasks contain a single thread of control and are associated with individual, protected address spaces. Under the guise of the operating system kernel, they are brought into existence, switched and allowed to communicate and synchronize. Because of performance considerations however, multi-tasking in multiprocessor environments is not the most effective means of supporting concurrency. In these environments, multiple threads which share an address space are more representative of the underlying platform architecture and more effective in their use of the underlying resources [Tan92].

Based on a survey of the related literature, approaches to the implementation of threads on multiprocessor computing platforms can be categorized into the *library, kernel,* and *balanced* implementation approaches. The following sections present the main characteristics of these approaches and investigate the execution properties of their threads, the nature of the work that they can be assigned, and the coordination overheads that they incur.

2.3.1 Library Implementation

Library implemented threads were originally developed to provide multiple threads of control for single-threaded operating system platforms [Kep85]. A single-threaded kernel architecture allows only one active thread of control to execute the kernel

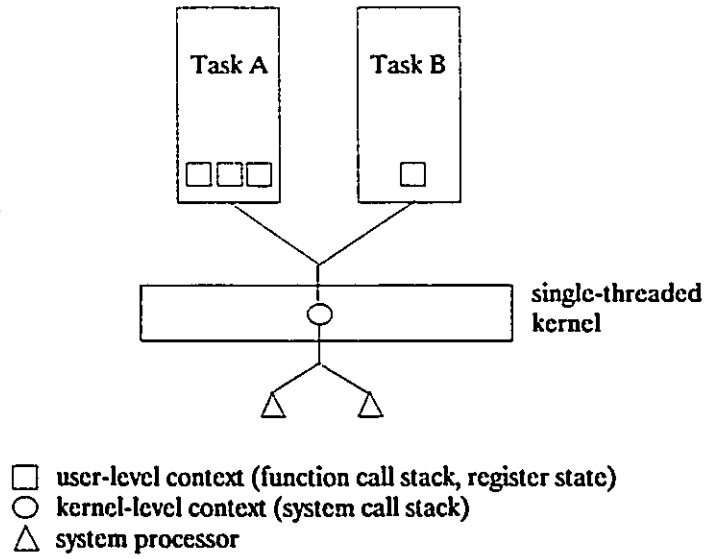


Figure 2.1: Library Implementation

routines at any one time. To counter this limitation, a runtime library linked into the virtual address space of the task was used to mimic the execution of multiple threads within the confines of the task. The mechanisms by which threads are created, destroyed, switched, and synchronized are fully implemented by library routines.

Figure 2.1 illustrates the relationship between tasks, threads and the operating system kernel. Task A is multi-threaded with three threads of control. Task B is a conventional, single-threaded task with one thread of control. Task A contains three user contexts (one for each thread) and shares the single kernel context with task B and its thread. As represented in the diagram, the threads of a library implementation exist only within a task and are unknown to the kernel.

Confinement of the existence of threads to the address space of a task is the distinguishing characteristic of library-implemented threads. The operating system kernel cannot differentiate between a multi-threaded task and its single-threaded counterpart; it provides system services to the *task* and not to individual *threads* within the task. Consequently, the threads must share the use of these services and assume the identity of the task whenever requests are made of the kernel. Since the kernel considers the task to be single-threaded, it has provided a single kernel context and reserved access to processor and I/O resources for only one thread. To provide access to all the threads of the task, the library must multiplex them onto the one thread supported by the kernel. The kernel then schedules the task containing the thread made current by the library.

This property has important ramifications for the execution characteristics of the thread and the nature of the work that it can be assigned. Firstly, the priority of the task within the system is shared by all of its threads. The lack of a local priority mechanism translates into homogeneous threads that cannot be differentiated and cannot pre-empt one another. In the library model, the current thread remains current until it expires or voluntarily suspends itself. This is due largely to the absence of clock interrupt handling routines in the library that could serve as the basis for higher-order scheduling functions. Secondly, the application is incapable of specifying the extent of the concurrency that it requires from the underlying platform. It may specify its logical concurrency requirements by creating multiple threads, but it is restricted in the degree of physical concurrency to the one kernel context offered to the task.

Thirdly, by forcing multiple threads to share the single kernel context, the library implementation causes the threads to be dependent on one another for progress. A kernel context acts as an entry point into the kernel for system-level service (i.e. file access, etc.). A thread that performs a system call or incurs a page fault during execution, is treated by the kernel as a task that performs a similar operation. Such an occurence causes the task to be suspended until resolution of the condition, irrespective of the availability of a runnable thread within the task.

Confinement to the address space can be a favorable quality however, since it allows thread coordination to be performed with minimal overheads. The mechanisms that bring threads into existence, switch between them and allow them to synchronise are provided by the library and are executed without any kernel interaction. Creating or destroying a thread for example, involves a call to a library routine which allocates or deallocates a new user context in the address space of the task. Local placement of the threads information and the manipulating routines also permits switching operations to be performed quickly. A context switch for example, consists of saving only the current thread's volatile register state. The state that is shared with the next thread in the library's dispatch queue can remain, lessening the processor loading and unloading requirements during context switches.

Thread synchronization is performed similarly, without kernel knowledge or intervention. In most library implementations, the library supplies routines which execute atomic operations on local synchronization objects such as mutex locks, condition variables, semaphores, and reader-writer locks. Routines that result in blocking a running thread are responsible for suspending the thread until the condition is resolved. An attempt to lock a mutex that is already locked for example, causes the offending thread to be suspended and placed on a sleep queue until the lock is released. Similarly,

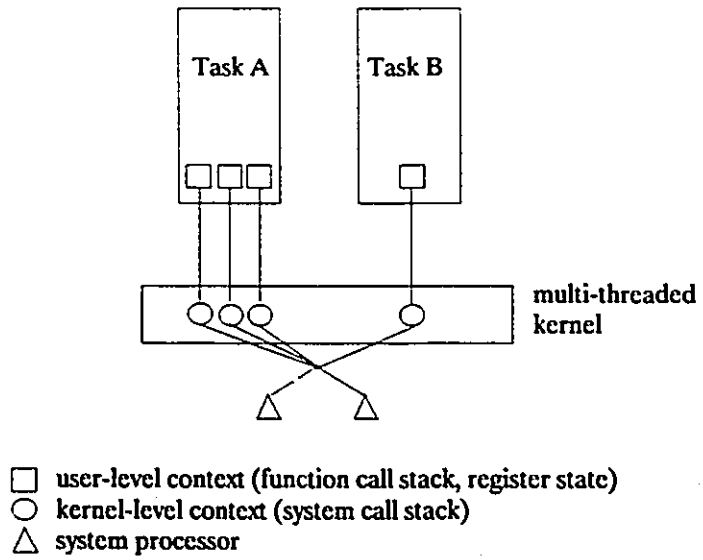


Figure 2.2: Kernel Implementation

routines that result in unblocking suspended threads are responsible for returning the blocked thread to the dispatch queue.

2.3.2 Kernel Implementation

The development of kernel-implemented threads originated with the increasing prevalence of multiprocessor architectures and multi-threaded kernels [Tev87][Coo90]. Unlike their single-threaded counterparts, multi-threaded kernels are capable of executing multiple system-level functions simultaneously and on different processors. The strategy underlying the implementation of kernel threads is to extend this kernel functionality and make it available to the applications using a simplistic library. The role of the library is essentially to extend the kernel thread facilities and provide a programming interface to the application developers.

As figure 2.2 suggests, the threads of a task are associated on a one-to-one basis with the threads of the kernel. The diagram represents the relationship between a task and its threads, the kernel and its threads, and system processors. Task A is multi-threaded with three threads of control while Task B is a traditional process with a single thread of control.

The distinguishing characteristic of kernel-implemented threads is that they are entities managed by the kernel on behalf of tasks. The kernel is aware of the existence of the multiple threads within a task and knows their state at all times. Through dedicated kernel contexts, it provides system services directly to individual threads and ensures

access to resources for each of them. The kernel maintains both the data structures which identify and describe the threads of a task and the routines which operate on them. Operations that manipulate the state of individual threads (create, destroy, suspend, resume, query, etc.) are accessed using the system call interface. The role of the library in kernel implementations is to provide higher-level operations and to map those operations onto the correct system calls.

The placement of responsibility for threads in the kernel is a favorable property when the execution characteristics of the resulting threads are considered. Since they are identified system-wide, kernel-implemented threads are assigned individual priorities which allows them to be differentiated from their peers. They can pre-empt and be pre-empted by the threads of the same task and by the threads of other tasks. They can also be assigned a time quantum. Also, the one-to-one relationship established between user threads and kernel threads results in complete independence from influencing the progress of other threads. A thread that blocks as the result of a system call or a page fault for example, is switched with another thread, making full use of the platform resources.

A disadvantage however, is that the application is incapable of specifying the extent of the concurrency that it requires from the underlying platform. Unlike library-implemented threads which are unable to control the degree of physical concurrency because they share a single kernel context, kernel-implemented threads cannot influence the degree of physical concurrency because they receive a dedicated kernel context. Much as it cannot increase its share of kernel resources in library implementations, the application cannot decrease its share in kernel implementations. Consequently, a potential drawback arises for inefficient use of kernel resources in conditions where large numbers of threads exist within the system.

Moreover, because the kernel is responsible for thread management, operations that coordinate threads suffer from high overheads. The mechanisms that create and synchronize threads are located in the kernel and require the library to generate time-consuming system calls to effect state changes. A task that creates or destroys a thread for example, invokes library routines to allocate or deallocate the user context. These routines subsequently invoke the corresponding *thread_create()* or *thread_destroy()* system call to allocate or deallocate the kernel context.

The library performs a similar function with respect to synchronizing the multiple threads of a task. Using a set of basic system calls as building blocks, the library implements atomic operations that permit synchronization and guarantee mutually exclusive access. An attempt to lock a mutex held by another thread for example, causes the

library to suspend the thread by generating a *thread_suspend()* call to the kernel while an attempt to release the mutex results in the blocked thread being returned to the dispatch queue by a *thread_resume()* system call.

High coordination overheads also stem from surrendering the responsibility for switching threads to the kernel. Because of shared memory serialization, pre-emption, and time quantum expiration, switching is a frequently occuring operation in kernel implementations, and time spent unnecessarily in executing the switch may have a negative effect on application performance. Unlike library implementations which perform partial context switches because of shared state between threads, kernel implementations must perform full context switches because the next thread on the kernel's dispatch queue may not be from the same task. Generally, optimizing assumptions cannot be made and the cost of saving and restoring full state must be absorbed by the application.

2.3.3 Balanced Implementation

The balanced thread implementation originated largely from the observations and the experiences of developing applications using the library and kernel implementations [Gol90][Pow91]. Firstly, the balanced approach attempts to retain the low coordination overheads found in library implementations by placing the core routines related to thread coordination in a library where they can be accessed very rapidly. This design characteristic results directly from an observation that the threads of a task tend to interact primarily with others within the task and therefore should not require the kernel's intervention for most of their coordination functions.

Secondly, the balanced implementation attempts to limit the restrictions it places on the nature of the work that can be assigned to threads because of their execution characteristics by ensuring kernel support for threads. This property results from experiences which show that the inability to make concurrent system calls and handle simultaneous page faults eliminates the advantages of multi-threading and decreases its utility in executing large, I/O intensive applications.

Figure 2.3 is a representation of the possible relationships that may exist between tasks, the threads of a task, and the threads of the kernel. Tasks A and C are multi-threaded with three threads of control while task B is a conventional task with a single thread of control. In task A, each thread is associated with a user context and can be multiplexed onto one of three kernel contexts. In task C, two of the threads are similarly *unbound* and may be multiplexed onto one kernel context. The third thread is *bound*

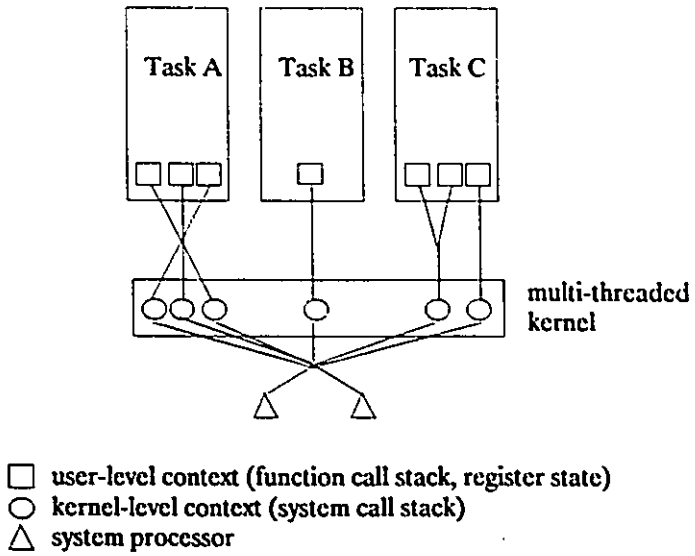


Figure 2.3: Balanced Implementation

and its relationship with the underlying kernel context is fixed. Task B contains a single thread, a corresponding user context, and is bound to a dedicated kernel context.

The defining characteristic of balanced implementations is that the association between the threads of a task and the threads of the kernel is configurable. By supporting both bound and unbound threads, the balanced implementation permits a high degree of flexibility in matching the type of thread to the needs of the application. Configurability stems from the collaborative management of threads at both the task and kernel levels. At the task level, the library is responsible for managing the user contexts, for multiplexing unbound threads onto the pool of kernel threads, and for synchronizing both types of threads. The kernel is responsible for managing the kernel contexts, for scheduling the kernel threads on the underlying system's processors, and for manipulating the state of kernel threads on behalf of the task.

The execution characteristics of balanced threads depend on the associations established between the task and kernel threads. Bound threads for example, exhibit characteristics that are analogous to kernel-implemented threads. Unbound threads however, have execution characteristics that are largely distinctive. Firstly, they are assigned local, configurable priorities that permit differentiation from their peers within the task. The priorities are used to guide the library in its multiplexing function, and to determine the task thread to be dispatched when a kernel thread becomes available. Other, global and system-wide priorities are assigned to the kernel thread and are used

by the kernel to determine the kernel thread to be dispatched when a processor becomes available.

Secondly, unlike both library- and kernel-implemented threads, unbound-type balanced threads permit control over the extent of the physical concurrency that is offered to the application. The size of the underlying pool of kernel threads can be either configured by the application or regulated automatically by the library, as dynamic conditions warrant. In the first case, the kernel is informed of the number of kernel contexts required to support the set of unbound threads. An attempt to set the number of kernel threads for example, causes the library to verify the present allocation and to generate a *thread_create()* system call for each missing thread and a *thread_destroy()* system call for each excess thread. In the second case, kernel contexts are created and removed as the execution of the task dictates.

Thirdly, unbound threads may or may not be dependent on their peers for progress. In circumstances where the degree of physical concurrency is automatically managed, unbound threads are guaranteed independent progress because the availability of kernel contexts is ensured by the implementation. However, if the application controls the extent of physical concurrency and forces a relationship where the pool of kernel threads is insufficient to service the number of unbound threads, a dependent progress condition may arise. In this case, a number of threads awaiting resolution of system calls or page faults equal to the number of kernel threads would halt all progress within the task, irrespective of the availability of ready threads.

The ability to configure the associations between task and kernel threads also has important influences on the overheads incurred to coordinate bound and unbound threads. The overheads involved in creating, destroying, switching and synchronizing bound threads for example, are high and largely equivalent to those absorbed for kernel-implemented threads. Conversely, since most of the management functions can be performed without kernel intervention, the overheads involved in coordinating unbound threads are minimal and comparable to those of library implementations.

2.4 Relevance

The purpose of this chapter was to investigate how to exploit, program, and support concurrency in generalized applications. The chapter reviewed the application decomposition process and the key factors which affect the partitioning decisions. It also reviewed the role of the programming model and the importance of its fundamental

characteristics. Lastly, the chapter discussed the accepted approaches to the implementation of threads and the impact these have on the properties of the abstractions supported by the programming model.

From the discussions of this chapter, some basic requirements can be derived relative to the capabilities of a concurrent programming environment. To be effective in its support of concurrent applications, the programming environment should address the following requirements.

- provide a programming model which allows an application to be decomposed into cooperating work units
- provide a programming model which allows the execution and coordination concerns of the application to be treated separately
- provide a programming model with a set of high-level abstractions for work units and their interactions
- provide development-time support facilities to formally specify the execution and coordination requirements of the application
- provide run-time support facilities which allow priorities to be assigned to individual threads, the degree of physical concurrency assigned to the application to be controlled, and the nature of the work assigned to the underlying threads to be unrestricted
- provide run-time support facilities which minimize the coordination overheads

The first three requirements are judged to be necessary if the programming model is to effectively address the execution and coordination concerns of concurrent applications. The last three requirements are necessary to provide the support for effectively merging the programming model with the underlying threads implementation.

Chapter 3

Real-Time Systems

The purpose of this chapter is to investigate the key characteristics of real-time systems and to describe approaches to their development. The first section explores the relevant properties of real-time systems, and the general attributes of the operating conditions in which they execute. The second section outlines the structure of real-time applications. The third section reviews the constraints that may be imposed on applications, and the effects these have on the ability of the application's intended functionality to be performed. The fourth section describes three approaches for developing real-time systems and applications. Finally, the fifth section briefly summarizes the chapter and lists additional requirements of the programming environment.

3.1 General Characteristics

Real-time systems are becoming increasingly commonplace in today's technology-driven society. Our reliance on these systems is evidenced by the vital role they play in modern avionic, manufacturing, process control, and telecommunications applications. Real-time systems are generally characterized by the ability to interact with their immediate environment [SR93]. They monitor changes in their surroundings, react, and exert a controlling influence on them. The reactions must be predictable, reliable and completed within constraints that are imposed by the application's functional requirements, the environment, and the underlying computing platform.

Changes of interest that occur in the environment propagate through the system by means of *events*. As figure 3.1 suggests, when communicated to the application, these events cause pre-determined actions, or *responses* to be performed. The events serve to inform the application about the state of the environment. They have individual

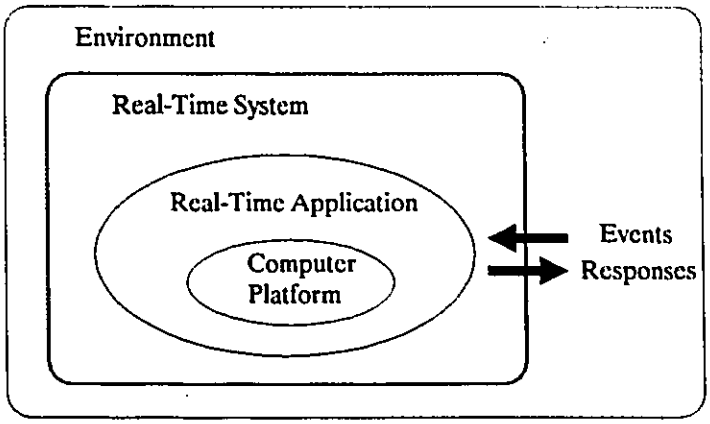


Figure 3.1: Real-Time Systems

characteristics, exhibit different occurence profiles, and place different demands on the system. To successfully address the requirements of the application, all generated events, including simultaneous ones, should be recognized and serviced by the system.

Based on the nature of the operating conditions, real-time systems can be characterized as either *static* or *dynamic* [NZ92]. In static systems, the operating conditions are either non-changing or slow to change. They can be well-understood by the designers during development and anticipated during execution. The predictable nature of the conditions permits the application's requirements to be fully determined, simplifying its construction, and minimizing the likelihood of violating the imposed constraints. In dynamic systems however, the conditions are either fast-changing or cannot be fully determined a priori. They are usually not as well-understood by the designers and cannot always be foreseen during execution. The unpredictable nature of the conditions complicates the development of the application, and compromizes its ability to meet the imposed constraints.

The work described in this thesis is aimed at dynamic systems. It investigates the ability of dynamic systems to perform their intended functionality in the presence of disturbances in their operating conditions. Some of the disturbances emanate from external sources and have global influence on the overall system while others emanate from within the system and have local impact on particular components. The potentially varied nature of the conditions that dynamic systems face within their sphere of operation can be traced to two primary sources; the unpredictability that is present in the environment and the non-determinism that is exhibited by the underlying computing platform.

Unpredictability in the environment, as perceived by the application, can result from a variety of factors. Firstly, the environment itself may be unpredictable. Secondly, the accuracy and finite number of sensors that monitor the environment and capture the events may limit the application's view of its surroundings. The sampling frequency of platform sensors for example, may not track the environment often enough to provide an unambiguous, consistent view of its state. Since exhaustive instrumentation with sensors of infinite precision is largely infeasible in practice, a finite probability remains that the application's view of its surroundings diverges from their true properties.

Thirdly, the perceived behavior of the environment may violate the design limits of the application. In essence, an application design that does not offer consideration to a particular condition may not be capable of reacting properly when faced with that condition. The condition may be unrecognized by the system logic and an indeterminate, possibly incorrect response may be performed. Many real-time systems for example, operate in volatile environments where many unforeseen conditions exist. Depending on the thoroughness of the analysis and design, the system may operate unpredictably in unfamiliar situations.

The sources of the non-deterministic properties in the underlying computing platform are equally varied. At the hardware layer, the organization of physical resources and their particular attributes may generate unpredictable, and in the limiting case, unbounded delays. The multiplicity of resources and the use of multiple-access network technologies for example, exposes some elements of the applications to the contention for shared facilities. Similarly, at the operating system layer, the availability of operations that do not execute in constant time and the use of nested interrupt schemes contributes to non-deterministic platform behavior. Operations that manipulate I/O devices for example, are dependent on the quantity of information transferred, and may suspend the calling work unit for unpredictable periods of time.

3.2 Application Structure

In a manner similar to general-purpose concurrent programming, to exploit the concurrency of the underlying multiprocessor platform, real-time applications are partitioned into cooperating entities. The process of decomposition is guided by layers of abstraction which provide structure and allow the application's functionality to be described at various levels of detail [KJ86]. The layers aim to manage the potential

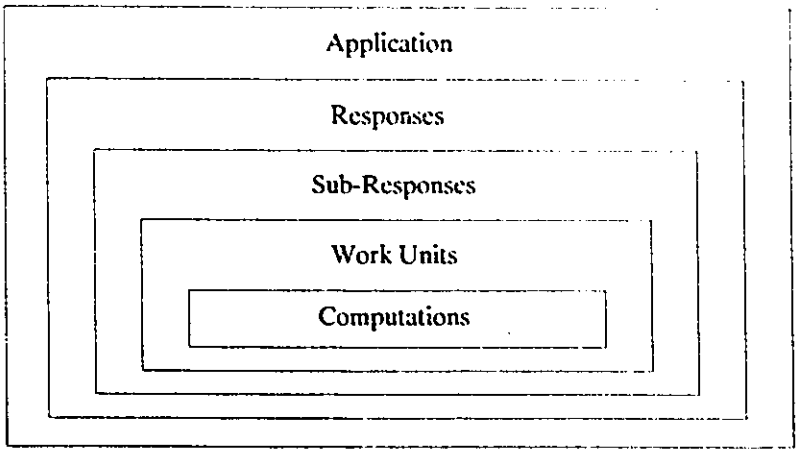


Figure 3.2: Levels of Abstraction

complexity of the application by successively refining the description of its functionality. Figure 3.2 outlines such a structure. It is based on a response-oriented view of real-time software design [KJ86].

At the first level, the functionality is organized into self-contained *responses* that address the requirements of each event generated by the environment. Responses may be either independent or dependent. An independent response requires no direct interaction with the other responses in the application while a dependent one requires one or more interactions to complete its assigned functionality.

At the second level of abstraction, the functionality of the response is grouped into *sub-responses*. The sub-response abstraction allows the developer to organize the internal functions of the response based on the characteristics they exhibit. The response's functionality can be organized based on its *functional* characteristics for example. In this circumstance, a sub-response may be created to initialize the response and wait for the external event, while another may be built to process the event once it is captured.

The response's functionality can also be decomposed according to its *temporal* characteristics. In some applications, parallel paths may exist where one path is subject to a time constraint for example, and another is not. In this case, a sub-response may be constructed to contain the response's time-sensitive functionality while one or more others contains the remaining functions.

Also, the sub-response abstraction level allows the response to be organized based on its *operational* characteristics. Decomposition of the response according to this

property allows the functionality of individual sub-responses to be determined based on which processor they execute, or on which I/O channel they access.

At the third level of abstraction, the functionality of individual sub-responses is grouped into *work units*. The work unit collects a portion of the functions that must be performed in service of an event. Since it can execute in parallel with other work units in the sub-response and with those in other sub-responses, the work unit is subject to precedence and exclusion constraints. The order in which the work units are executed is determined by both the control dependencies inherent in the sub-response's assigned functionality and the data dependencies that exist between them.

Lastly, at the fourth level the functionality assigned to the work units is organized into *computations*. A computation is the lowest level of abstraction that is of interest to the developer. It is a meaningful grouping of operations that are sequentially performed over a prescribed set of data elements.

3.3 Application Constraints

Concurrent real-time applications are generally governed by constraints that are more stringent than those of general-purpose applications. The nature of the system and the role it is intended to fulfill determines the stringency of the individual constraints and the degree to which the application is sensitive to them. As illustrated in figure 3.3, the application is constrained primarily by its internal structure, the environment with which it interacts, and the computing platform on which it executes.

The first type of constraint is imposed by the environment and requires the system to respond to event occurences within pre-determined time periods [Sta88]. Time constraints specify a window of opportunity or a deadline within which the response to an event is valid and useful to the environment. A *hard* constraint on a response should be satisfied by the system under *all* operating conditions. Failure to meet such a constraint adversely affects the integrity of the system and may result in irreparable damage to life or property. By contrast, a *soft* constraint on a response should be satisfied by the system under *most* operating conditions. Failure to meet such a constraint is tolerated and may result in a noticeable but acceptable reduction in system performance.

The second type of constraint is imposed by the underlying computing platform and requires the system to deliver the intended functionality using resources of finite availability and of limited utility. The number of physical resources such as processors, memory, and I/O devices in the platform for example, is fixed and their performance

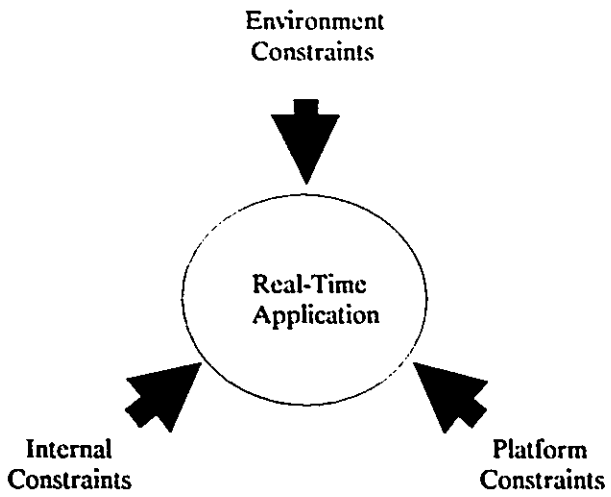


Figure 3.3: Application Constraints

is bounded. Similarly, logical resources such as threads, buffers, and communication channels, are limited in quantity and in utility relative to the requirements of the application. Failure to meet these constraints can adversely affect system operation, causing events to be unrecognized, preventing urgent responses from completion and causing time constraints to be missed.

Lastly, the third type of constraint results from the internal characteristics of the application. Firstly, the application constraint requires the system to perform functionality that is consistent with the requirements established by the system's functional specification. Secondly, it requires the system to adhere to the precedence and exclusion relationships established during the partitioning process. These constraints guide the execution of the application by imposing sequential or concurrent ordering between the responses and their constituent work units. The system must respect the relationships that exist and reflect them in the application's processing flow. Failure to meet these constraints compromizes the integrity of the application's intended functionality, and may bring about indeterminate system behavior.

3.4 System Development

The primary goal of real-time system development is to engineer a solution to the application's functional requirements. The solution must be resilient to the dynamic conditions in which the system will operate, ensure conformance to the environment's constraints, and offer a level of functionality that is consistent with the goals of the ap-

plication. To achieve this aim, the relationships that exist between the three types of constraints restricting the application should be considered. As a consequence, the relative influence of the constraints on the overall system should be determined, and the tradeoffs should be evaluated.

The first tradeoff balances the internal constraints of the application against the constraints of the environment. The relationship between these constraints restricts the application from performing the functionality it desires at the time it wants to. The alternatives for resolving the tradeoff are to maintain the intended functionality, thereby jeopardizing the time constraint, or to modify its functionality, thereby jeopardizing the internal constraint. The purpose of the tradeoff is to establish the application's demand for resources as a function of time.

The second tradeoff which guides the development of the real-time system balances the constraints of the platform against the constraints of the environment. The relationship between these constraints restricts the application from receiving resources at the time they are required. The possibilities for resolving the tradeoff are to force the application to accept the level of resource availability, thereby jeopardizing the time constraint, or to increase the availability of resources, thereby violating the platform constraint. In effect, the choices made as a consequence of the tradeoff establish the supply of resources as a function of time.

The basis for considering the tradeoffs lies in the importance attributed to the application's functional and temporal correctness. As a result of the constraints that govern their execution, real-time applications are not only concerned with achieving correct *functional* results as dictated by the requirements, but also correct *temporal* results as dictated by the environment. The functional correctness of a computation is proportional to the amount of functionality performed in order to achieve a result and is an indication of the quality of the result. The temporal correctness of a computation is proportional to the amount of time taken to achieve a result and is an indication of the timeliness of the result. An application is functionally correct if all of its assigned functionality is completed and temporally correct if the completion occurs within the time constraint.

The relationship between an application's temporal and functional correctness is treated differently depending on the problem being addressed [Wil90]. In general-purpose applications for example, the emphasis tends to be placed on functional correctness at the expense of temporal correctness. The temporal correctness is usually not enforced providing the delay in obtaining a result is within reason. The functional requirements of an application are sacrificed only when performance gains can be made

to justify the loss in functionality. In effect, the goal of general-purpose applications is to produce *correct results in time*, and not necessarily *correct results on time*.

In real-time applications however, the relationship is dependent on the stringency of the constraints. For responses that are governed by hard real-time constraints, the emphasis is placed on temporal correctness at the expense of functional correctness. Since the primary objective of hard-constrained responses is to meet the time constraints under all operating conditions, functional correctness is relegated to a secondary objective. A result obtained with partial functionality is satisfactory providing it was obtained within the specified time constraint. In essence, a certain amount of functional incorrectness may be tolerated to ensure timely procurement of a result. The goal of these responses is to produce *results on time,* and not necessarily the *correct results on time.*

For responses that are governed by soft real-time constraints, the relationship is biased towards either temporal or functional correctness, depending on the extent to which the time constraint can be violated. A response with a time constraint that is desirable but not necessary for example, has characteristics similar to general-purpose applications, with functional correctness being the primary consideration. Conversely, one that has an inviolable time constraint has characteristics more closely associated with hard real-time responses, where temporal correctness is emphasized.

From a survey of the relevant literature, three broad approaches to developing dynamic, concurrent real-time systems can be identified. Firstly, the *platform-centric* approach attempts to modify the capabilities of the underlying platform to ensure that the supply of resources is not unduly constraining the application. Secondly, the *application-centric* approach attempts to modify the functionality of the application to guarantee that the demand for resources can be met by the platform's capabilities. Thirdly, the *constraint-centric* approach attempts to dynamically manage both the supply and the demand for resources to maximize the functionality that is performed by the application, to minimize the number of resources required to adequately service the functionality, and to ensure conformance to the time constraint.

3.4.1 Platform-Centric Approach

The traditional, and most pervasive approach to developing real-time systems focuses on selecting a computing platform that exhibits a level of functionality and performance consistent with meeting the constraints [Sta88]. As depicted in figure 3.4, the emphasis during development is placed on the application's platform constraints. The

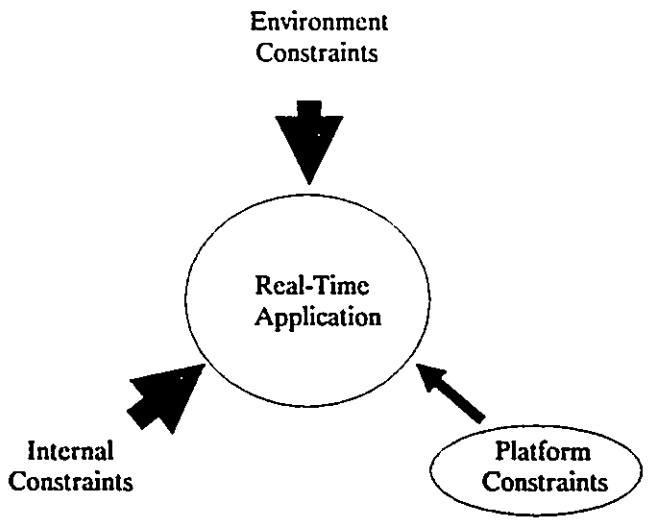


Figure 3.4: Platform-Centric Approach

approach aims to minimize the effects of platform constraints by matching the availability and utility of resources to the requirements of the application. In the tradeoff between internal and environment constraints for example, the approach favors maintaining the intended functionality while in the tradeoff between platform and environment constraints, it favors modifying the availability of resources.

Allowing the environment and internal application constraints to dictate the composition of the underlying platform permits a margin of safety to be introduced in the system. This over-capacity allows delay-causing conditions to be tolerated without impacting the ability to meet constraints and allows applications to be developed with greater confidence that their execution will conform to the imposed time constraints.

The performance and functionality of the platform can be augmented in a variety of ways. At the IC level for example, the continuous advances made in semiconductor technology and VLSI design have enabled higher clock rates and greater on-chip functionality, resulting in faster devices that are specialized to particular application domains. Similarly, at the hardware level, the proliferation of these devices has precipitated a decrease in their costs, making high-performance multiprocessor architectures feasible and attractive. Finally, at the system software level, the availability of real-time operating systems for example, has permitted optimized management of physical and logical resources without sacrificing mainstream OS features, allowing key system operations to be performed at speeds approaching the limits of the hardware.

A disadvantage of this approach is that the system developer has no means of directly specifying an application's time constraints to the underlying platform. The platform is thus unaware of the time-dependent requirements of the responses it executes. Without the explicit incorporation of time into the development process of an application, the urgency of constituent computations cannot be determined and acted upon. The application has no means by which it can re-balance its functionality based on the passage of time and ensure that its time constraints are met. The computations that result from the approach contain largely static functionality that is rigidly defined and executed.

As a consequence of not integrating the temporal requirements of the application, the approach's ability to consistently meet constraints is compromized. This unfavorable characteristic manifests itself in the inability of the system to offer guarantees that the application's constraints will be met under all operating conditions. In fact, it does not guarantee that they will be met under *any* conditions. Conditions of transient overload for example, can adversely affect the application's ability to meet constraints. A sudden burst of events can cause many of them to be unrecognized by the system. By extension, those events that are recognized may not meet their time constraints because of the system's attempts to process the flood of incoming events. In effect, over-capacity is relied upon to handle unusual operating conditions.

Similarly, the computing platform's extra capacity is relied upon to support the evolution of the application. The inevitable modifications that the system will be subjected to may cause constraints that were met before the modifications, to be missed after the changes are made. The margin of safety provided by over-specification of the computing platform may be reduced considerably by the addition of functionality and the resolution of bugs. In the limiting case, an existing application which has evolved and no longer meets constraints because it has outgrown the underlying platform must be ported to a faster, more powerful computing platform.

3.4.2 Application-Centric Approach

The application-centric approach to developing concurrent real-time systems focuses on the internals of the application and the interface it shares with the underlying computing platform. As illustrated in figure 3.5, the emphasis during development is placed on the application's internal constraints. The approach aims to minimize the effects of the application's internal constraints by manipulating its resource demands to fit within the platform's resource availability. In the tradeoff between internal and

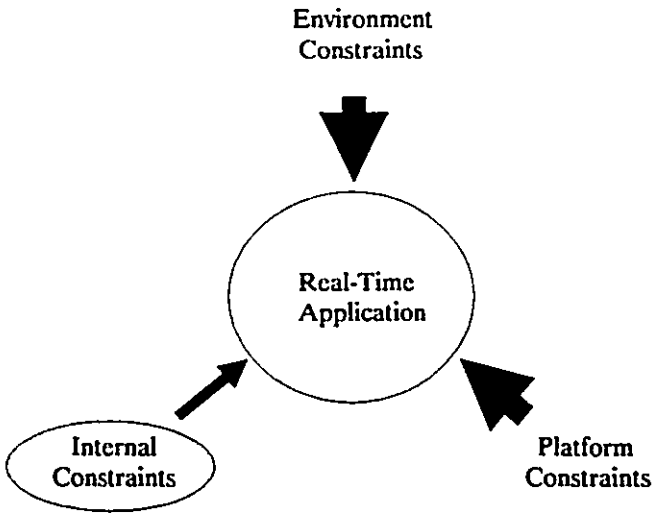


Figure 3.5: Application-Centric Approach

environment constraints for example, the approach favors modifying the intended functionality while in the tradeoff between platform and environment constraints, it favors maintaining the level of resource availability.

To optimize the application's responses and their constituent work units such that they consistently and reliably execute within their time constraints, the approach makes use of a *modelling* process which investigates the behavior of the system under different operating conditions. The basis for this process is a functional representation, or *model* of the system, and its environment. The model is used to obtain statistical measures of the execution behavior of responses and to verify the effects of alternative design strategies on this behavior. It is essentially a skeleton of the critical features of the application, the environment, and the computing platform. It reflects the functionality of these interacting components in general, and contains the characteristics that affect constraints in particular.

The modelling process can be classified as either *analytical* or *experimental*. Analytical modelling constructs models from a set of mathematical equations that relate the behavior of functional entities to a set of input parameters [Sta91]. The models are particularly well-suited to analyzing systems in which customers enter a service area, receive service for some period of time and then depart the service area. In a real-time system for example, customers can be used to represent events that are generated by the environment, while services can represent responses to those events. By extension,

input parameters such as customer arrival rates correspond to the frequency of event generation and mean service periods correspond to average response execution times.

By contrast, experimental modelling techniques such as simulation and prototyping construct models that can be executed. Simulation relates artificial input parameters to the performance metrics by executing the model on an *abstract* representation of the target computing platform [AP77]. Prototyping achieves a similar objective by executing the model on a *concrete* representation of the target platform [LB88]. Simulation is used primarily to investigate design alternatives when the target computing platform is non-existent or has little or no statistics reporting facility. Prototyping is usually considered an integral part of the development process and used both for proof-of-concept and what-if type of analysis. Prototype models are implemented on either the actual target computing platform if it is complete or on one that reflects its key characteristics.

The analytical modelling strategy essentially provides a *snapshot* view of the system. The behavior that is manifested as a result of the input parameters is largely lost in the resulting static mathematical relationships. By contrast, the experimental modelling strategy provides an *evolutionary* view of the system. It allows the designer to gauge the run-time behavior and determine the cause of fluctuations in performance metrics. It also allows the designer to detect transient characteristics in the system and monitor their migration to either steady-state or bottleneck conditions. During the development process, both strategies may be utilized depending on the coarseness of the data that is required for decision-making, the error that can be tolerated in the results, and the effort which must be expended in constructing and maintaining the model.

A favorable property of the modelling approach is that it integrates the application's time constraint during analysis and design. With the aid of modelling methods and tools, the behavior that is manifested by the model as a result of controlled stimuli can be observed for anomalies, measured to yield statistical profiles for metrics, and altered relative to the constraints. Computational algorithms that introduce unbounded and unnecessary delays for example, can be identified and modified for time-boundedness and reduced execution times. However, the functionality of each response remains static once the process is complete. The response functionality is determined through iterative techniques that aim to identify the maximum functionality that will enable the application to consistently meet its constraint. It thus cannot vary based on the application's run-time urgency considerations.

An unfavorable characteristic of the application-centric approach is that it causes designers to rely on the integrity of the model to ensure conformance to constraints.

Consequently, the modelling approach guarantees that an application's constraints will be met under *some*, but not all operating conditions. Guarantees are extended primarily for operating conditions that were identified during the modelling activity and evaluated for possible negative effects on response timeliness. The occurence of an unforeseen condition for example, causes indeterminate behavior in an implementation based on the model and allows for events to be missed and time constraints to be violated.

Furthermore, reliance on a representative model of the system has some drawbacks with respect to the maintenance and evolution of the application. Since the design produced by the modelling activity is effectively tuned to a particular workload and configuration, the integration of voluntary or involuntary modifications to the application's functionality may have effects on the model's manifested behavior. In this circumstance, previous modelling efforts are obsolete, and the modelling process must be re-visited to observe the effects of the changes on the model's behavior. The iterations may add considerable cost and delay to the deployment of the system.

3.4.3 Constraint-Centric Approach

The constraint-centric approach focuses on matching the resource requirements of the application with the availability of the underlying platform's resources as they vary in time. As depicted in figure 3.6, the emphasis during execution is placed on both the application's internal and platform constraints. The approach aims to minimize the effects of these constraints by managing the allocation of existing platform resources dynamically. It controls the application's demand for resources and regulates their availability based on its urgency requirements. During development, the approach does not favor particular alternatives in either the internal/environment constraint or platform/environment constraint tradeoffs, preferring instead to resolve these at run-time.

The basis for determining the allocation of resources according to urgency lies in the *value* each computation imparts to the response that produces the result [Wil90][SR93]. The value informally represents the importance of a computation relative to the others in the application. It is initially assigned according to application semantics, and depending on the urgency precipitated by the time constraint, may fluctuate during the execution of the response. As a result, the goal of the constraint-centric approach is to maximize the value imparted by each computation to the overall application. By doing so, the system is guaranteed to deliver the maximum amount of functionality within the pre-determined time constraints.

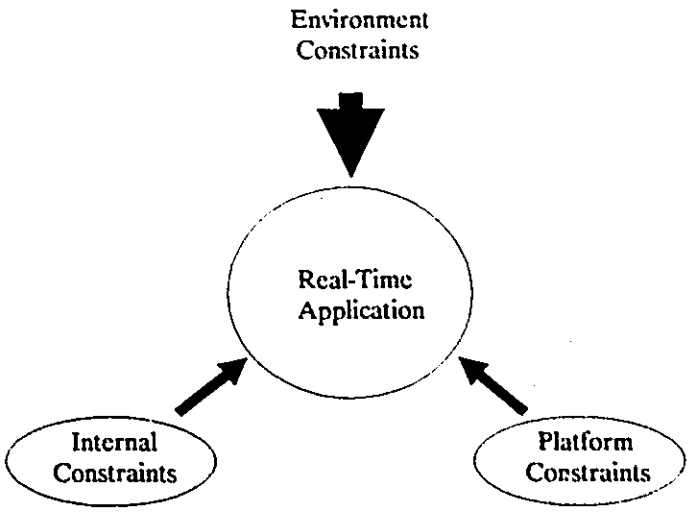


Figure 3.6: Constraint-Centric Approach

A hard constrained response which has insufficient time to complete its assigned functionality for example, will forego computations that contribute small or negligible value to the application and will concentrate on performing those that impart a high or infinite value. Similarly, a soft constrained response with urgency concerns may elect to either forego some computations in favor of others in order to meet the time constraint or may pursue all computations irrespective of the time constraint.

To resolve the time-dependent conflicts between the application's demands for resources and the platform's availability of resources, the approach makes use of on-line scheduling facilities. The facilities may consist of multiple layers of scheduling with the lowest layer being contained within the platform's operating system. The facilities collect requests for the resources, arbitrate their use based on the origins of the outstanding requests and the state of the requested resource, and maintain notion of the passage of time. Their role is to mediate and control the demand for resources and to regulate the availability of resources.

Figure 3.7 illustrates two possibilities for controlling the demand for resources [NZ92]. In the *scheduler-controlled* variant, the responsibility for determining the application's resource demand as a function of time rests with the scheduling facilities. The variant effectively requires the application to surrender control of its functionality to the facilities. The application supplies the facilities with its time constraints and with allowable options to varying its functionality. Based on the passage of time and the

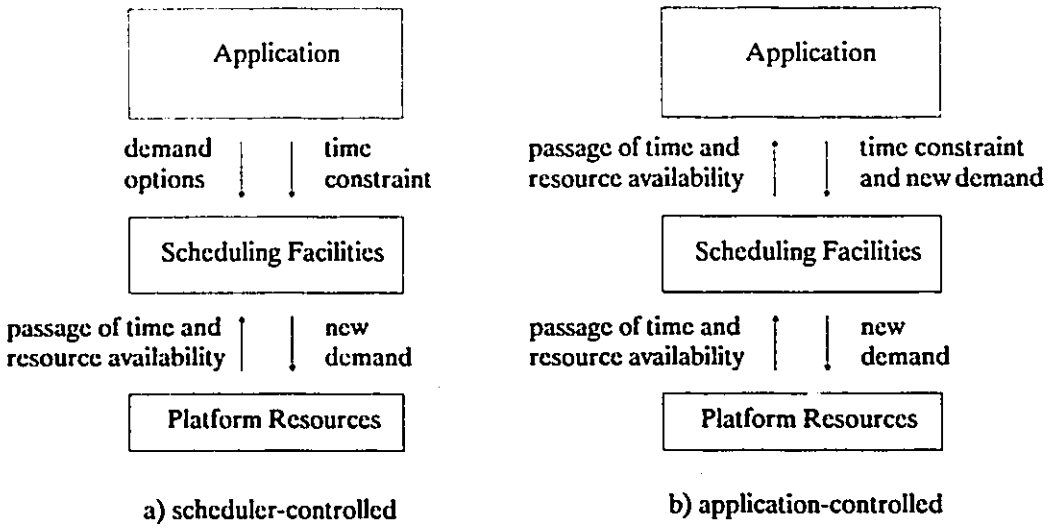


Figure 3.7: Variants for Controlling Demand

availability of resources, the scheduling facilities select alternative resource demand profiles based on the options that were supplied by the application.

The application specifies the functionality options through either the *multiple version* or *imprecise computation* techniques [Liu91]. The multiple version technique for example, allows a developer to specify multiple copies of the same response. Each version prepared has a particular level of functionality and a different execution time. To select a copy at run-time, the scheduler focuses on the estimated execution time of the versions relative to the time left before the time constraint expires. This method is applicable to responses with execution times that are bounded and can be specified from the outset.

Alternately, the imprecise computation technique allows a developer to partition each response into its mandatory and optional components. The mandatory component embodies an algorithm that produces an intermediate, approximate result which the optional component extends and refines into a complete result. At run-time, the scheduler executes the mandatory component in its entirety and the optional component if sufficient time remains. The optional component may be preempted by the mandatory component of another computation or suspended by the expiration of the time constraints.

In the *application-controlled* variant, the responsibility for determining resource demand as a function of time lies with the application. The variant requires the developer to imbed specific logic in the application's logic flows to deal with the time

constraint and the scheduler's status information. At initialization, the application informs the scheduling facilities of its time constraints. During execution however, based on the time lapse and resource availability information supplied by the scheduling facilities, the application modifies its functionality, and makes new demands on the platform. In effect, through the exchange of status information, the platform and the application negotiate the demand for resources on the basis of urgency.

Once it has interpreted the status information, the application modifies its functionality using either the *milestone* or *sieve* techniques [Liu91]. The *milestone* technique allows a developer to record a response's intermediate results at specified instances during its execution. As time progresses, the algorithm embodied in the response refines the result and periodically updates the recorded copy. The response either terminates with a correct result or is aborted at the expiration of the time constraint with the last update serving as the final result. Because of its reliance on incremental refinement, the milestone technique is particularly well-suited to *monotone* computations, where the quality of intermediate results increases as the algorithm progresses in its execution.

For computations which are not strictly monotone, the *sieve* technique allows for variable functionality by skipping some elements of the response's algorithm. Based on this run-time status information, alternate paths through the algorithm can be selected, and the functionality optimized with respect to the time remaining and the resources available. Through its ability to adjust its functionality on a fine-grain basis, the technique allows the application to adapt to potentially rapid changes in the operating conditions. This ability translates to greater maximization of the value of individual computations, a closer coupling with the environment and more efficient use of the underlying platform's finite resources.

The mechanism used by the scheduling facilities to regulate the availability of resources consists of manipulating the priorities of the responses and of their constituent work units. Under this policy, each work unit is associated with a priority that distinguishes its importance relative to its peers [SR93]. To arbitrate access to the resource, the scheduling facilities compare the priorities of the work units that have outstanding requests with the priority of that which is currently making use of the resource. The work unit with the highest priority is allowed to access the resource. Depending on the fixed or variable nature of the criteria that is used, priorities can be either *static* or *dynamic*. Static priorities remain fixed during the lifecycle (invocation to completion) of the work unit while dynamic priorities may fluctuate.

Static priority-based algorithms such as those employed in *rate-monotonic scheduling* use the repetition rate of periodic events as the defining criteria for determining the priority of response computations [KLR94]. The period between triggering events is fixed over the life of the response and thus gives a fair indication of its importance. In rate-monotonic scheduling, priorities are assigned in inverse relation to the inter-arrival period. In effect, the response which must be invoked with the highest frequency (or smallest period between event occurences) receives the highest priority. Similarly, in *deadline-monotonic scheduling*, the response's deadline is used as the defining criteria. Priorities are assigned in inverse relation to the deadline, with the highest priority being assigned to the response with the strictest time constraint.

Conversely, dynamic priority-based algorithms use determining criteria that varies over the lifecycle of the response [RSS90]. In *earliest-deadline scheduling*, priorities are re-balanced dynamically by the scheduling facilities to reflect impending deadlines. In effect, the response which has the greatest urgency requirement (nearest deadline) is assigned the highest priority and given immediate access to its requested resources. Similarly, in *least-slack-time scheduling* (also known as *least-laxity scheduling*), the difference between the projected response execution time and the deadline is used as the defining criteria for priority assignment. In this scheme, priorities are varied in inverse relation with the difference, associating the highest priority with the most time-constrained response (smallest difference).

A favorable property of this approach is that it offers support for the application's time constraints during execution. The developer has direct means of both specifying the time constraints and the options to be considered when they cannot otherwise be met. It permits the functionality of the application to be controlled by its urgency considerations as they vary in time. This property is particularly advantageous in systems where the outcome of previous computations causes the resource requirements of current ones to vary and where dynamic operating conditions cause the availability of resources to fluctuate.

As a consequence, the constraint-centric approach allows applications to adapt to changing operating conditions and to selectively guarantee conformance to time constraints. The approach effectively grants an option to the application for either meeting the constraint by varying its functionality or violating it. It has the singular ability to ensure that the application's time constraints will be met under all operating conditions. Conditions of overload for example, which could adversely affect the system's ability to meet time constraints can be alleviated by compromizing the executed functionality, in order to remain conformant to the time constraint.

The approach's flexibility and adaptability characteristics are also beneficial with respect to the maintenance of the system since it permits its main components to evolve independently of each other. Unlike the platform-centric approach, the application does not rely on a margin of safety to meet constraints and does not necessarily require changes to the application when the platform is modified. Further, unlike the application-centric approach, the application is not tuned to a particular workload and configuration, and does not necessarily require changes to the platform when the application is augmented with greater functionality and bugs are resolved.

3.5 Relevance

The purpose of this chapter was to investigate the characteristics of real-time systems and the accepted approaches to their development. The chapter described the general properties of time-sensitive systems, the nature of the conditions in which they operate, the basic structure of the application software they contain, and the constraints that they face. The chapter also discussed the main approaches to the development of these systems and the effects these have on the characteristics of the resulting systems. Finally, the chapter described the ramifications of the tradeoffs which are favored by each approach and the basis on which they are made.

The discussions of this chapter derived additional requirements relative to the capabilities of a real-time programming environment. To be effective in its support of real-time systems, the programming environment should address the following requirements.

- provide a programming model which allows the application to be structured as responses to the environment's events
- provide a programming model which allows the complexity within a response to be managed through layers of abstraction
- provide a programming model which allows applicable time constraints to be specified
- provide a programming model which allows the application to dynamically control the functionality it performs
- provide the run-time support facilities which inform the application of its consumption of time and negotiate the demand for platform resources
- provide the run-time support facilities which manage the availability of platform resources based on the application's urgency considerations

The first four requirements are judged to be necessary if the programming model is to make adequate provisions for the particular characteristics of the real-time problem domain. The last two requirements are necessary to provide consistency with the constraint-centric approach to the development of real-time systems.

Chapter 4

The Multi-Activity Programming Environment

The purpose of this chapter is to present the characteristics of the Multi-Activity Programming Environment (MAPE). The first section describes the three main components of MAPE and the role they play in the realization of an application. The second and third sections outline how the MAPE programming model addresses the execution and coordination concerns of real-time applications. The second section describes the execution model and the language that was selected for this thesis. The third section outlines the coordination model, the specification notation used to support it, and the language developed to formalize it. The fourth section describes the MAPE runtime system. Finally, the fifth section briefly summarizes the chapter and outlines MAPE's support for the requirements identified in Part I.

4.1 MAPE Components

MAPE aims to provide a simplified, flexible programming environment customized for the real-time application domain. The three main components include:

- an execution model and supporting language for translating the application's execution requirements into machine form
- a coordination model, formal notation, and supporting language for translating the application's coordination requirements into machine form
- a runtime system for controlling the application's execution

These components are used at various stages in the development of an application. The generalized process for realizing an application is depicted in figure 4.1. The ini-

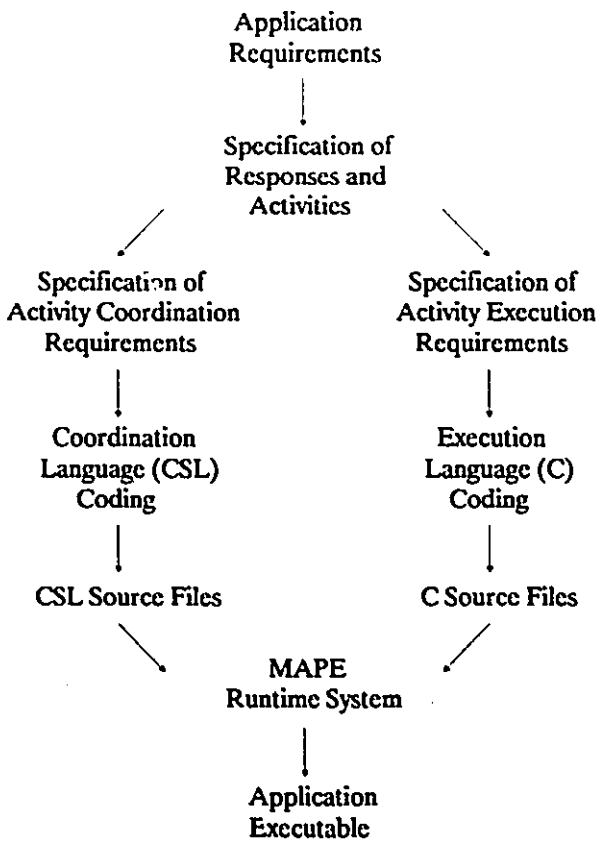


Figure 4.1: Application Realization

tial step in the process is an analysis of the application's functional and temporal requirements. The analysis serves to outline the external events to be serviced, to evaluate the nature of the work to be performed in response to the events, and to identify the applicable time constraints. The requirements information is then used to partition the overall functionality into event responses, sub-responses, and work units.

In MAPE, the response's cooperating work units are abstracted using *activities*. An activity is a meaningful sequence of computations that implements a portion of the response's functionality. It is defined by the relationships it shares with others in the application, the functionality it implements, and its maximal execution time. The maximal execution time is the estimated time to complete all the functionality that it is assigned and is referred to as the Activity Estimate (AE).

Once the specification of functionality is complete, the execution and coordination requirements of the activities are extracted and dealt with separately. The execution requirements can be specified in a variety of methods, using either free-form text or a

formal notation for expressing sequential operations. The next step in the process is to translate the requirements for each activity into the formal constructs of the execution language. For this thesis, the C programming language was chosen as the target execution language. The result of this step is a collection of C source files that unambiguously specify the computations that are to be performed by each activity.

Similar steps are performed with the activities' coordination requirements. Firstly, the requirements are specified using the graphical Process Activity Diagram (PAD) notation. PAD is a formal notation for expressing the precedence relationships in a natural, control flow format and the timing constraints in an unambiguous manner [KJ86]. Secondly, the requirements are translated into machine-manipulable form using the Coordination Specification Language (CSL). The result of this step is a set of CSL source files that specify the extent and nature of the interactions that are to occur between the activities. The CSL files also specify the timing constraints that apply to the time-constrained sub-responses.

The final step integrates the application-specific C and CSL source files with the application-generic runtime system. The runtime system supports the MAPE programming model by providing an additional layer of abstraction between the application and the underlying computer platform. Its primary role is to implement the high-level abstractions offered in the programming model and to make the platform-specific concurrency details transparent to the developer. The result of the integration is an executable program containing the code that implements the functionality of the application, the rules that guide the coordination of its activities, and the code that performs the sequencing.

4.2 Execution Model

The MAPE execution model provides the constructs that enable the activities' computations to be specified in an unambiguous manner. It supplies the developer with a set of operators with which to manipulate data elements, evaluate and act on conditions, and control repeatable operations. The execution model also supplies the developer with a set of basic data types that can be used to create compound types and satisfy the application's data requirements.

The MAPE execution model has two important characteristics. Firstly, the model makes use of consistent shared memory as its underlying memory paradigm. In MAPE, the data elements that can be manipulated by an activity's computations are categorized

```
BEGIN ActivityName (StatusCode, InputArguments)
IF (StatusCode = NotLate)
    perform computations that implement full
    functionality since sufficient time remains
ELSE
    perform computations that implement partial
    functionality since insufficient time remains
ENDIF
END
```

Figure 4.2: Generalized Activity Structure

as either *unrestricted* or *restricted*. Unrestricted data elements are shared at all times with the other activities in the application and any changes made to them are immediately recognized by the others. Their scope is global to the application. Restricted data elements however, are not shared by the application's activities and changes made to them are seen only by the activity performing the changes. Their scope is local to the activity that has defined them. For the change to be recognized by the others, the updated data element must be communicated to the other activities.

Consistency issues can arise in models that rely on the shared memory paradigm. However, in the MAPE execution model, these issues are limited largely to the manipulation of unrestricted data elements by concurrent activities. Since a computations' operations on an unrestricted data element are made in full view of other activities, multiple activities may operate on the same data element simultaneously. For this reason, any provisions for dealing with this possibility must be handled by the application using a mutual exclusion guarantee mechanism.

Another important characteristic of the MAPE execution model is its support for variable functionality within an activity. Upon invocation, the first input argument to the activity is a status code that indicates its lateness relative to its run-time estimate. Based on this status code, the activity determines the urgency of its constituent computations and selects the extent to which they will be performed. Figure 4.2 shows the structure of a generalized activity. The variable functionality method supported by the execution model is based largely on the sieve method, where alternate functionality paths are determined by the activity during execution (see section 3.4.3). In conditions of urgency for example, the method supported in MAPE allows an activity to skip or sieve computations that impart low value in an attempt to gain time and meet the time constraint.

4.2.1 The C Execution Language

In MAPE, the C programming language embodies the execution model and facilitates a formal implementation of the application's execution requirements specification. It serves primarily to provide the developer with a rich set of compute, condition, and loop operators that can be used to specify the sequence of operations that individual computations must perform.

The intrinsic properties of the C programming language lend themselves well to the requirements of the MAPE execution model. The language's modular code structure for example, supports the assembly of computations into functions which can be mapped directly to the activities. Similarly, its interface specification mechanism allows the transfer of status codes and the communication of arguments to be performed in a formal, typed manner. Furthermore, the language's scope rules enforce the separation defined between restricted and unrestricted data elements and protect activities from adversely affecting others.

Figure 4.3 shows the C language code for an activity that processes 256x256 uncompressed digital images at a prescribed 1:1 resolution. During conditions of urgency, the activity processes the image at half the prescribed resolution by skipping every second pixel. This produces a lower quality image but requires half the processing time. The example assumes that a previous activity has placed the image's digitized information in the shared memory area before the activity is invoked.

The *ProcessImage* activity is coded as a conventional C function, with both an argument vector and argument count parameter. The argument vector's only element contains the status code identifying the urgency condition (early, on time or late). Its data requirements are satisfied by two unrestricted data elements which are in the shared memory area and three restricted data elements which are not. The unrestricted *current_image* array holds the 64K pixels to be processed while the *enhanced_image* array holds the 64K pixels of the modified image. The restricted *i* and *j* data elements hold the indexes used to cycle through the *current_image* array for the *enhance* algorithm while the status code holds the urgency condition.

```
/* Unrestricted Data Element Definition */
char current_image[65536];
char enhanced_image[65536];

/* Activity Definition */
int ProcessImage (argv, argc)
char *argv[];
int argc;
{
/* Restricted Data Element Definition */
int status_code, i, j;

status_code = argv[0];

if (status_code = = 0)
    /* early or on time condition */
    for (j=0; j<256; ) {
        for (i=0; i<256; ) {
            enhance (&current_image[i,j], &enhanced_image[i,j]);
            i = i + 1;
        }
        j = j + 1;
    }
else
    /* late condition */
    for (j=0; j<256; ) {
        for (i=0; i<256; ) {
            enhance (&current_image[i,j], &enhanced_image[i,j];
            i = i + 2;
        }
        j = j + 1;
    }
} /* end ProcessImage */
```

Figure 4.3: ProcessImage Activity C Code

4.3 Coordination Model

The MAPE coordination model allows the developer to unambiguously specify the interactions that arise as a result of partitioning the real-time application into concurrent activities. Using a coordination notation and formal language, the developer can specify both the points within the application at which interactions are initiated and received, and the nature of the interaction that is to be performed at that point. The model also allows specification of the time constraints that govern the execution of an application's time-dependent sub-responses.

Depending on the application's semantics, the interactions that arise between activities may span structure boundaries. Interactions may occur for example, between

activities that belong to the same sub-response, between those that belong to the same response, and between those that belong to different responses. As a consequence, the model supplies a set of high-level abstractions that mask the details of each type of interaction and allow the developer to describe the interactions in a uniform manner.

The main abstractions supported by the model consist of the *node* and the *edge*. The nodes abstract the activities of the application. They also abstract the management of the control flows within the application. The edges abstract the interactions between activities and act as *triggers* which must be activated to effect a transfer of control.

The MAPE coordination model supports three forms of interactions. The first form involves the communication of data elements between activities. Because of the underlying shared memory paradigm however, not all data elements are subject to being communicated. Unrestricted data elements for example, are shared by all activities at all times, and need not be communicated. Restricted data elements however, are subject to being communicated. They are owned by one manipulating activity at a time and must be transferred to others before they may be operated upon.

The second form of activity interaction involves the synchronization of cooperating activities. The MAPE coordination model supports synchronization at the boundaries of activities, and not within activities. An implicit assumption is made that the computations within an activity do not require synchronization with the other activities in order to perform their functionality. The primary role of synchronization is to preserve the order of execution by enforcing the precedence relationships. In sequential precedence profiles, synchronization interactions regulate the flow of control between the activities and indirectly arbitrate the manipulation of unrestricted data elements. In parallel precedence profiles, they create and collect multiple flows of control, but do not arbitrate the manipulation of unrestricted data elements.

The third form of interaction involves the management of the state of the application's activities. The MAPE coordination model supports the dynamic creation and destruction of the activities as well as their conditional suspension and resumption. Activities may be suspended based on conditions for example, and resumed when those conditions are satisfied. Furthermore, the model isolates the developer from the details of claiming and returning the underlying computer platform resources that are required by the activities.

Lastly, the MAPE coordination model allows the developer to specify the time constraint for sub-responses in the form of a deadline. The deadline corresponds to the time alotted to the sub-response, relative to its start, for performing the assigned

functionality. The deadline serves as the basis for determining the urgency of the activities of a sub-response.

4.3.1 The PAD Graphical Notation

The PAD graphical notation is a MAPE tool for specifying the coordination requirements of concurrent, real-time applications [KJ86]. The constructs it supports facilitate application construction by permitting the developer to specify interactions within and across responses in a natural, graphical format. Using a series of nodes that are interconnected by directed edges, the notation represents the precedence, time and state management requirements of cooperating activities.

Figure 4.4 illustrates the nine types of nodes. An *Activity* node is used to represent the execution of an activity. It contains a variable number of input and output ports. The edges converging onto the node represent the conditions which must be satisfied before the activity can be executed. These conditions may relate to the availability of data needed by the activity, or may represent the precedence constraints that must be maintained. Once all input triggers are activated, the node executes the associated activity. Upon completion, the node activates all output triggers and transfers control to its downstream peers.

A *Gate* node implements a state-based mechanism for blocking the propagation of triggers. It contains a variable number of input and output ports as well as one enable port and one disable port. Initially, the node is in a disabled state, unable to activate its output triggers irrespective of the activation state of the input triggers. Once the enable input is activated, the node enters the enable state and waits until all of its input triggers are activated. When the condition is met, the input triggers are cleared, the output triggers are activated, control is transferred to its downstream peers and the node reverts to the disabled state. In the disabled state, the node ignores all inputs except for the enable input.

A *Counter* node implements a state-based mechanism for counting the occurence of activated input triggers. It has a single input port, a variable number of output ports, one enable port, one disable port and an associated count value. Initially, the node is in a disabled state, oblivious to the activation state of its input trigger. Once the enable input is activated, the node enters the enabled state and the counter is loaded with the count value. Any subsequent activation of the input trigger decrements the counter value. When the counter value reaches zero, all output triggers are activated, control is passed to the downstream peers and the node reverts to the disabled state. The Gate

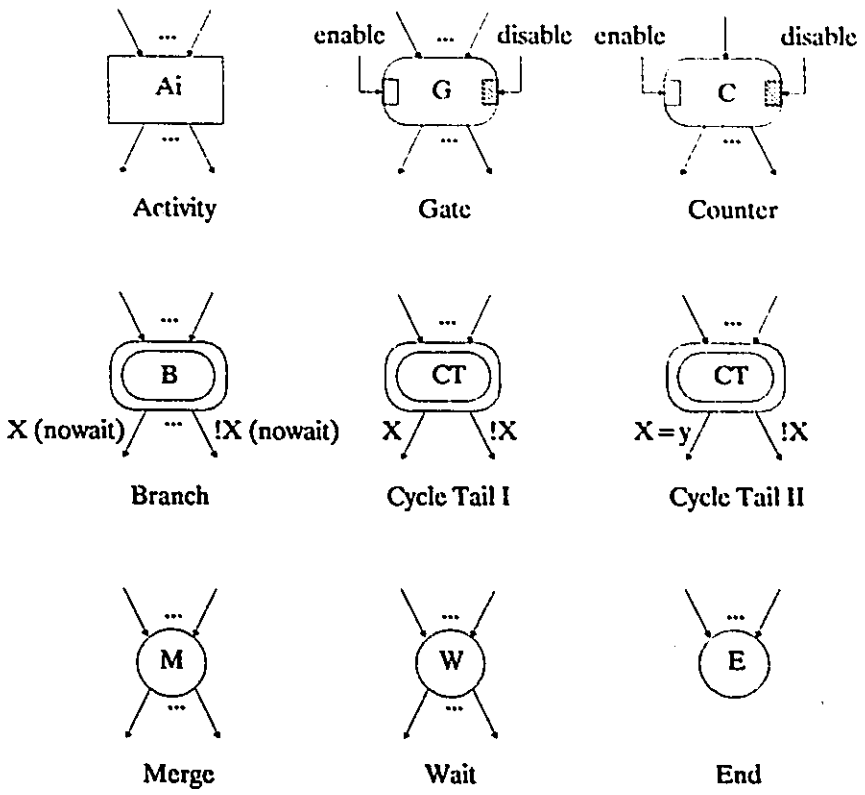


Figure 4.4: PAD Node Types

and Counter nodes can be combined to provide control structures for cycles. However, the resulting cycles are necessarily static.

A *Branch* node is used to select alternate execution paths depending on the outcome of an explicit condition. It contains a variable number of input and output ports. The output edges are labeled with either X or !X and an associated qualifier (WAIT, NOWAIT). Once all input triggers are activated, the node evaluates the condition. If the condition is non-zero, all output triggers labeled X are activated irrespective of their qualifier. If the condition is zero, output triggers labeled X and qualified as WAIT are suspended while those qualified as NOWAIT are ignored. A similar analysis is performed with edges that are constructed from negative logic !X conditions. A Branch node transfers control only along activated triggers and not along suspended triggers. Control is transferred along suspended triggers only when the condition that causes their suspension is overturned.

The *Cycle-Tail* node is both a generalization of the function provided by Gate and Counter nodes for controlling cycle iterations and an extension to these functions to

support dynamic cycles. The first type of Cycle-Tail node (type I) contains a variable number of input ports, and two output ports. It is used to control dynamic cycles, where the number of iterations of the cycle is dependent on the execution of activities and is not known at the outset. The first output edge is labeled X and serves as the feedback path while the second is labeled !X and serves as the path out of the cycle. Once all input triggers are activated, the loop condition is evaluated. If the condition evaluates to non-zero, the output trigger labeled X is activated, and control is transferred to the head of the cycle. If the condition evaluates to zero, the output trigger labeled !X is activated, and control is transferred to the subsequent node.

The second type of Cycle-Tail node (type II) similarly contains a variable number of input ports, and only two output ports. It is used to control static cycles, where the number of cycle iterations is non-changing and known a priori. The first output edge is labeled X=y to indicate y iterations of the cycle. The second output edge is labeled !X. Once all input triggers are activated, the node decrements an associated iteration counter. If the count is greater than zero, the output trigger labeled X is activated, and control is transferred to the head of the cycle. If the count is equal to zero, the output trigger labeled !X is activated, the cycle is exited and control is transferred to the downstream node.

A *Merge* node reconciles OR-type paths in the execution flow of the application. It contains a variable number of input and output ports. OR-type paths occur primarily as a result of branching, where a choice is made and some execution paths are not pursued. Once any one of its input triggers is activated, the node activates all of its output triggers, surrendering control to its downstream peers.

A *Wait* node reconciles AND-type paths in the application's processing flow. It contains a variable number of input and output ports. The node performs a joining function for multiple execution paths. Once all input triggers are activated, the node activates all of its output triggers, transferring control to subsequent nodes.

An *end* node is the PAD termination construct. It contains a variable number of input ports and no output ports. The node performs a join function similar to the Wait node and terminates the response when all of its input triggers are activated.

Based on the coordination requirements of the application, a logical grouping of these nodes are assembled into an Activity Sequence Network (ASN). The ASN serves to integrate the control flow requirements of a set of activities. It specifies the creation and destruction of constituent activities, their ordering (concurrent or sequential), as well as the conditional selection of their execution paths. Figure 4.5a shows the ASN for response 1. The response is comprised of a sub-response which waits for an exter-

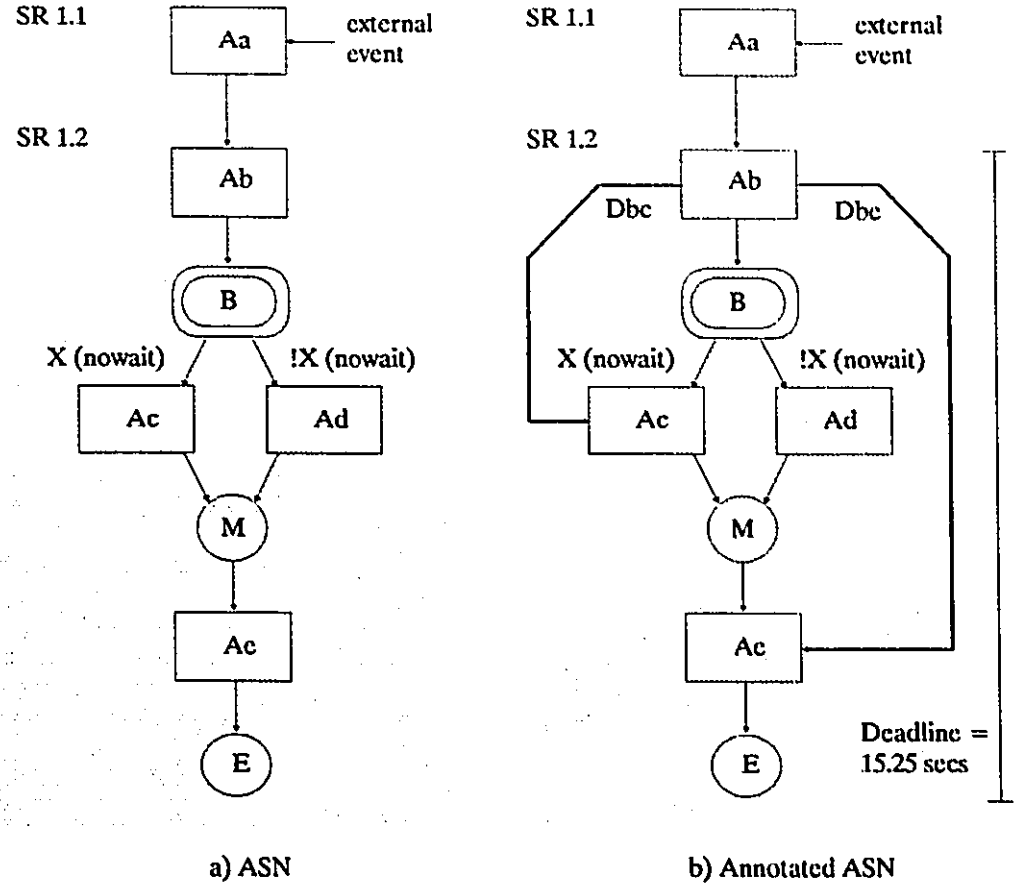


Figure 4.5: ASN and Annotated ASN for Response 1

nal event (sub-response 1.1) and another which processes the event (sub-response 1.2). In activity organizations that are not governed by time constraints, the ASN may span sub-response and response boundaries. In those that are restricted by time, the ASN can only represent one time-constrained sub-response.

The application's data flow and time constraint requirements are integrated into the specification by annotating the ASN with the relevant information. Figure 4.5b depicts this type of information for response 1. An explicit flow of data is defined between activities *Ab* and *Ac* (*Dbc*) and between *Ab* and *Ae* (*Dbe*). The data need not necessarily be the same in both instances. In addition, the annotated ASN specifies a deadline of 15.25 seconds imposed on sub-response 1.2, from the start of activity *Ab* through the *E* end node.

```
asn_graph : = functional_section temporal_section

functional_section : = FUNCTIONAL edge [edge ...]
edge : = source_name destination_name destination_port edge_qualifier
edge_qualifier : = counter_node | condition_edge | cycle_node | NONE
counter_node : = COUNT count
condition_edge : = condition_name condition_type
cycle_node : = condition_name cycle_qualifier
cycle_qualifier : = STATIC iteration_count | DYNAMIC

temporal_section : = TEMPORAL deadline estimate [estimate ...]
estimate : = source_name activity_estimate

source_name : = ascii_string
destination_name : = ascii_string
destination_port : = INPUT | ENABLE | DISABLE
count : = integer_value
condition_name : = ascii_string
condition_type : = WAIT | NOWAIT
iteration_count : = integer_value
deadline : = real_value
activity_estimate : = real_value
```

Figure 4.6: CSL Backus-Naur Form Grammar

4.3.2 The CSL Coordination Language

The Coordination Specification Language (CSL) is a special-purpose coordination language for the real-time application domain. The language embodies the MAPE coordination model and is used to produce a formal implementation of the application's coordination requirements. Its role is to supply the constructs required to translate the synchronization, communication and lifecycle management requirements formalized by ASN graphs into a form that can be interpreted by the MAPE runtime system.

The Backus-Naur Form (BNF) grammar of the CSL language is shown in figure 4.6. A formal ASN specification of a PAD-specified graph consists of two main sections. The *functional* section specifies the dependency and precedence relationships identified by the PAD nodes and their connecting edges. For every edge in the ASN, an entry is generated to specify its source node, its destination node, its destination port, and any outstanding qualifiers. Qualifiers are required to define the initial count for counter nodes, the name and nature of the condition for conditional edges, and the cycle type for cycle-tail nodes. The *temporal* section specifies the temporal requirements of one time-constrained sub-responses. The section defines the deadline for the

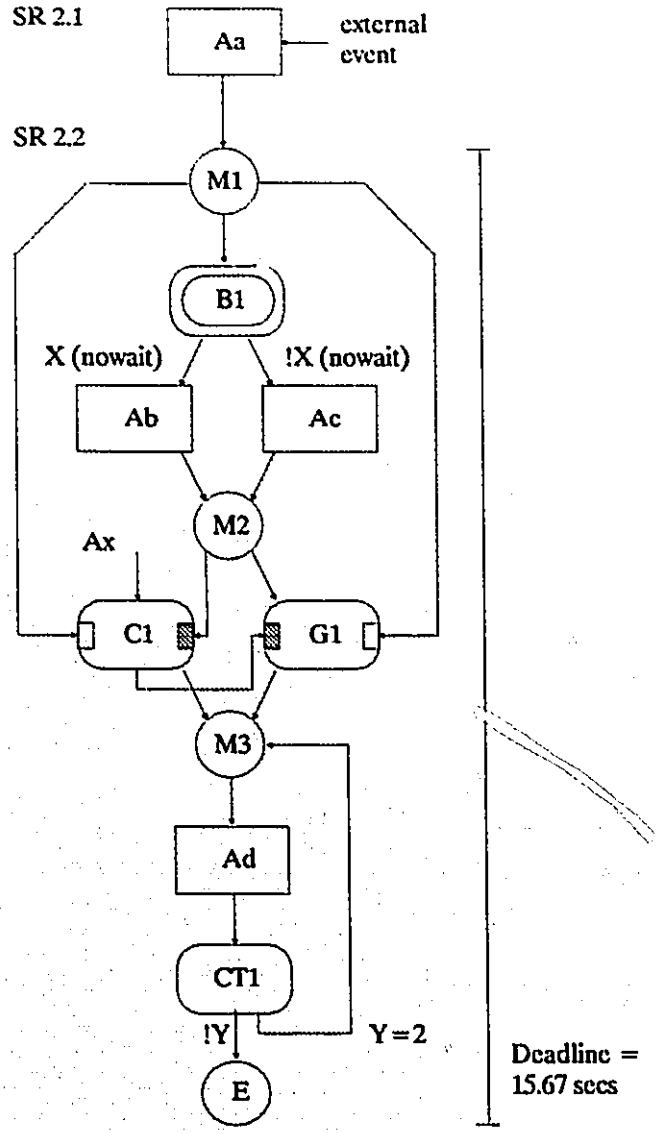


Figure 4.7: Annotated ASN for Response 2

sub-response in units of seconds and the activity execution time estimate in units of seconds, for every constituent activity.

Figure 4.7 shows the annotated ASN graph for response 2 while figure 4.8 shows its corresponding CSL language code. The first sub-response waits for an external event and is not constrained by time. The second processes the event and is constrained by a 15.67 seconds deadline. In service of the event, sub-response 2.2 performs either the

```
FUNCTIONAL
Aa      M1      INPUT       NONE
M1      C1      ENABLE      NONE
M1      B1      INPUT       NONE
M1      G1      ENABLE      NONE
Ax      C1      INPUT       COUNT    1
C1      M3      INPUT       NONE
C1      G1      DISABLE     NONE
B1      Ab      INPUT       X        NOWAIT
B1      Ac      INPUT       !X       NOWAIT
G1      M3      INPUT       NONE
Ab      M2      INPUT       NONE
Ac      M2      INPUT       NONE
M2      C1      DISABLE     NONE
M2      G1      INPUT       NONE
M3      Ad      INPUT       NONE
Ad      CT1     INPUT       NONE
CT1     M3      INPUT       Y        STATIC 2
CT1     E       INPUT       !Y
TEMPORAL 15.67
Ab      5.30
Ac      7.12
Ad      2.51
```

Figure 4.8: CSL Language Code for Response 2

Ab or *Ac* activities, based on condition *X*. The sub-response also monitors for an internal event from the *Ax* activity. The *Ax* activity belongs to another response and the example assumes that its input triggers and possibly other output triggers are specified using another ASN. The ASN of figure 4.7 only specifies the edge that serves as input to the *C1* counter node. The transfer of control into the static cycle is done when either the *Ab* or *Ac* activity completes or when the internal event is triggered. The response terminates after two iterations of the cycle.

4.4 Runtime System

The Runtime System (RS) acts as the intermediary between the real-time application and the underlying computer platform. Its primary function is to interpret the CSL source files that describe the application's coordination requirements. In this capacity, the RS offers an Application Programming Interface (API) through which the developer defines and starts the application, specifies the location of the activities code, retrieves the activity input arguments, and sets conditions for the selection of alterna-

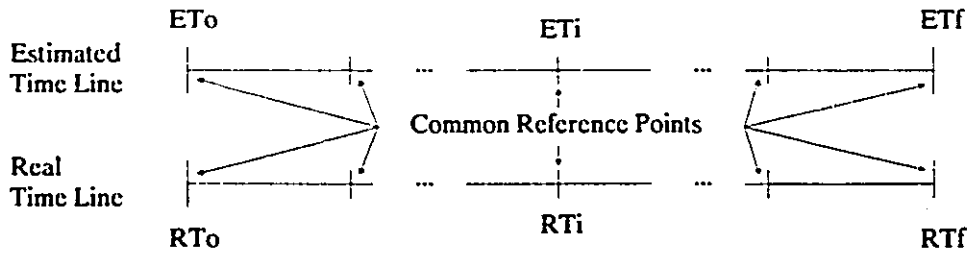
tive execution paths. Appendix A outlines the RS API in detail, including the call syntax and parameter lists for the RS primitives.

Its primary role however, is to implement a translation of the high-level abstractions offered by the programming model into the low-level abstractions offered by the computer platform. Firstly, it maps the execution model's unit of concurrency (activity) into the operating system's vehicle of concurrency (thread). Secondly, it translates the coordination model's high-level abstraction for an interaction into the operating system's mechanisms for interaction. Depending on the characteristics of the edge for example, the interaction is transformed into calls to the thread management subsystem, to the thread synchronization subsystem, or to local shared memory operations.

The RS also fulfills the role of managing the allocation of resources as a function of time. In a manner analogous to that of the scheduling facilities described in section 3.4.3, the RS attempts to dynamically match the application's resource demands with the platform's resource availability. Firstly, it supplies the application with status information concerning its consumption of time (early, on time, or late) and negotiates its demand for resources on the basis of urgency. Secondly, it regulates the availability of resources by manipulating the priorities of the application's activities based on the time constraint information.

The runtime system's ability to dynamically allocate resources is predicated on monitoring the progression of the sub-response's functionality with respect to the passage of time. To facilitate the analysis, the RS makes use of the time lines illustrated in figure 4.9. The *estimated* time line represents the anticipated time requirements of the sub-response's functionality and is used to measure the progression of work. The line begins at the start of the sub-response and ends at its estimated completion time. The *real* time line represents the actual execution of the functionality and is used to gauge the progression of time. It begins at the start of the sub-response and ends at its imposed deadline.

Figure 4.9 also shows a set of common reference points that are shared between the two time lines. The points correspond to the actual and expected start and end times of the constituent activities and act as *signposts* for monitoring the sub-response's execution. At various times during the execution, the RS analyses the current position on the real time line relative to the expected position on the estimated time line to determine the extent of the progress that has been made. In early or on time conditions, the reference points on the real time line occupy positions that are either behind or in alignment with their counterparts on the estimated time line. In late conditions



ETo: start of sub-response
ETi: estimated completion time of activity i
ETf: estimated completion time of sub-response
RTo: start of sub-response
RTi: completion time of activity i
RTf: deadline of sub-response

Figure 4.9: Sub-response Time Lines

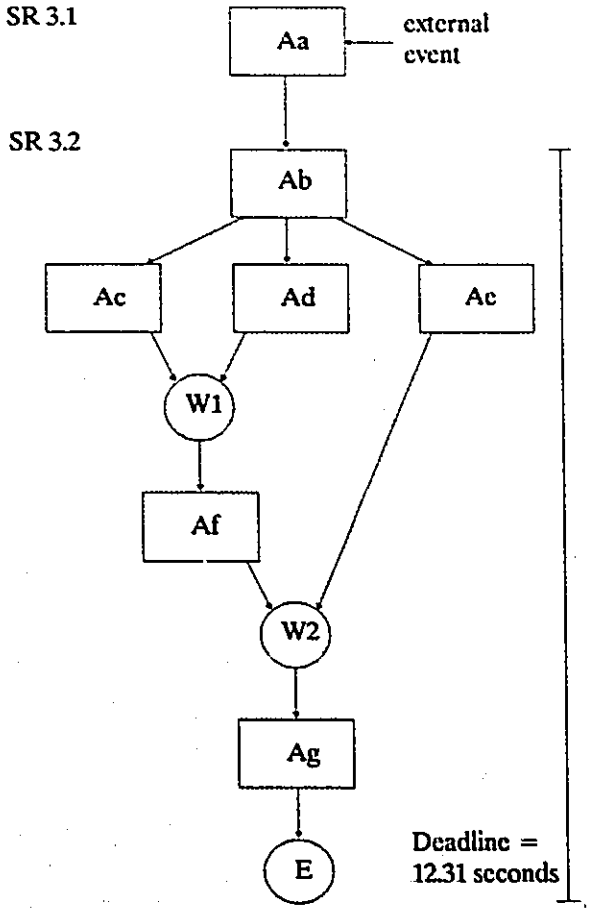
however, the reference points on the real time line occupy positions that are ahead of the corresponding points on the estimated time line.

The following sections outline the main responsibilities of the RS. The first section investigates the method used for determining the reference points on the estimated time line. The second section presents the logic embedded within the RS that derives the status codes that are transferred to the application. Lastly, the third section outlines the priority manipulation mechanisms used by the RS to influence resource availability.

4.4.1 Cumulative Run Time Estimates

The role of the Cumulative Run Time Estimate (CRTE) is to guide the runtime system in the assignment of priorities and status codes for the activities of time-constrained sub-responses. The CRTE provides a measure of the progression of work within the sub-response and forms the basis for determining the lateness of constituent activities. In essence, it gives an indication of *where* the sub-response should be relative to the passage of time by establishing the signposts on the estimated time line that are used to validate its progress.

The CRTE is based on the Activity Estimates (AEs) provided by the developer and is computed for every invocation of an activity. An AE represents the full functionality version of the activity and the maximum time it can consume. The CRTE is derived automatically by the runtime system during the ASN definition phase, when the ASN is initially loaded and processed. The computation is a cumulative sum representing



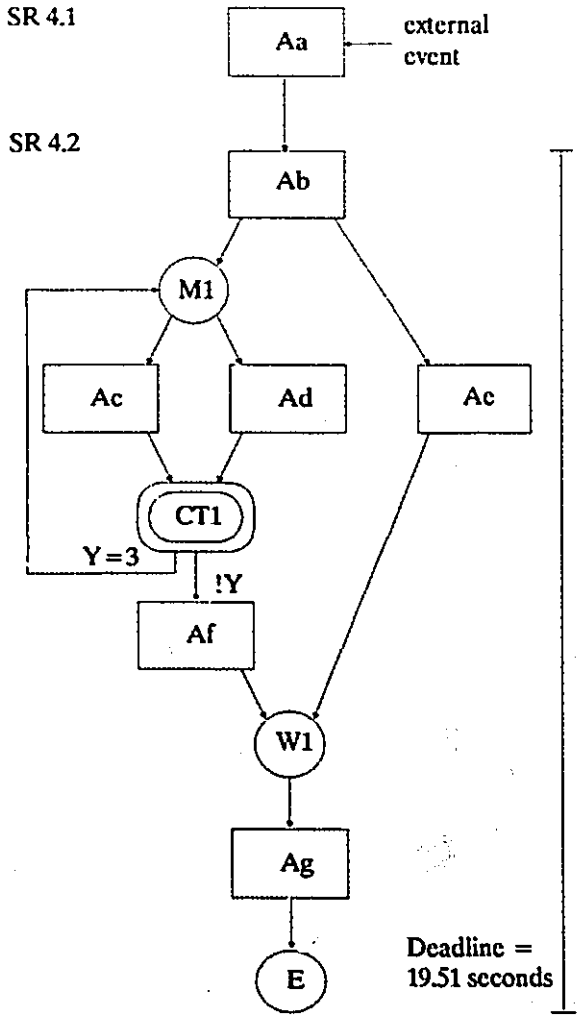
Activity	AE	CRTE
Ab	2	2
Ac	1	3
Ad	3	5
Ae	6	8
Af	2	7
Ag	3	11

Figure 4.10: Annotated ASN, AE, and CRTE for Response 3

the maximum time that could be taken to traverse the sub-response assuming that necessary resources are available. The CRTE for a particular activity represents the estimated time of its completion, relative to the start of the sub-response.

Figure 4.10 illustrates the ASN for a response that contains two sub-responses; one which is not governed by a deadline (sub-response 3.1) and one which is (sub-response 3.2). The pre-determined AEs and the CRTE computations for the time-constrained sub-response are also shown. The general algorithm for computing the CRTE is the following:

CRTE = greatest_predecessor_CRTE + AE



Activity	AE	CRTE
Ab	2	2
Ac	1	3
Ac	1	6
Ac	1	9
Ad	3	5
Ad	3	8
Ad	3	11
Ac	6	8
Af	2	13
Ag	3	16

Figure 4.11: Annotated ASN, AE, and CRTE for Response 4

Since activity *Ab* is the first in the sub-response for example, and no preceding CRTE exists, its CRTE is equal to its AE. The remaining activities however accumulate the time spent in the preceding activities in their CRTE. The CRTE for activity *Af* for example, is computed by first determining the greater of the *Ac* and *Ad* CRTEs and adding the AE.

Cycle constructs are considered in a similar manner, with the CRTE computations following the general flow of control established by the sub-response's precedence relationships. The ASN of figure 4.11 is an example of a sub-response which includes a three-iteration cycle construct. During the first iteration, the CRTE for activities *Ac*

and *Ad* are computed relative to the CRTE of Activity *Ab*, by adding the preceding node's CRTE to their individual AEs. For the second and third iterations however, the CRTE for activities *Ac* and *Ad* are computed relative to the CRTEs obtained for the previous iteration of the cycle. Activity *Af*, the first activity encountered at the exit of the cycle is treated similarly. Its CRTE is derived by determining the greater of the *Ac* and *Ad* CRTEs in the last iteration of the cycle and adding the AE.

4.4.2 Status Codes

Status codes are the means by which the runtime system communicates lateness information to time-constrained sub-responses and their constituent activities. Lateness conditions manifest themselves primarily as a result of misalignment in the positions of the reference points on the estimated and real time lines. The RS identifies these conditions at both the sub-response and activity levels and generates one of three status codes:

Code	Meaning
0	sub-response or activity is early or on time
1	sub-response is late
2	activity is late

The role of the status code is to *inform* the application of lateness. Once the information is communicated, the application may either act on it by varying its operations or it may disregard it completely. In the case of a status code equal to 0 for example, the functionality implemented by the sub-response or the individual activity need not be altered while in the cases of a status code equal to 1 or 2, it may need to be if the imposed constraint is to be met.

At the start of every time-constrained sub-response and before the first activity is launched, the RS evaluates the alignment of the *final* reference points. For the estimated time line, this point corresponds to the sub-response's estimated completion time while for the real time line, it corresponds to its pre-defined deadline. Figure 4.12 depicts the possible conditions faced by the RS.

Case *a)* presents the condition where the estimated completion time (CRTE of the final activity) is less than or equal to its imposed deadline. In this scenario, The RS would launch the first activity with a status code equal to 0, indicating a normal condition which does not require the sub-response to alter its functionality. By contrast, case

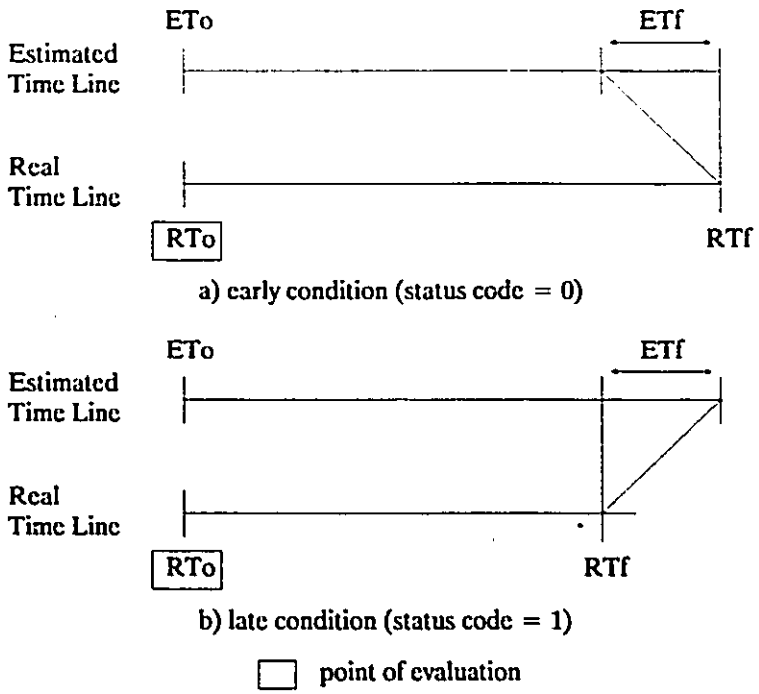


Figure 4.12: Response-Level Status Codes

b) presents the condition where the estimated completion time is greater than the predefined deadline. In this scenario, the status code for the first activity would be set equal to 1, indicating that the deadline cannot be met with full sub-response functionality and that the internal operations must be modified if the sub-response is to conform to its time constraint.

The sub-response level status codes are not automatically propagated by the RS within the sub-response. A status code equal to 1 for example, is delivered only to the first activity. The responsibility for communicating sub-response level lateness information to the other constituent activities lies with the first activity.

For subsequent activities, a similar analysis is performed by the RS before each activity is launched. The analysis consists of evaluating the alignment of *current* reference points along the sub-response's time lines. For the estimated time line, the current reference point corresponds to the previously completed activity's estimated completion time, which gives a measure of the work done at this stage in the progress of the sub-response. For the real time line, the reference point corresponds to the current real time relative to the start of the sub-response. It gives a measure of the time used to complete the previous activity and all of its preceding activities. Figure 4.13 illustrates the possible conditions that can be encountered by the RS.

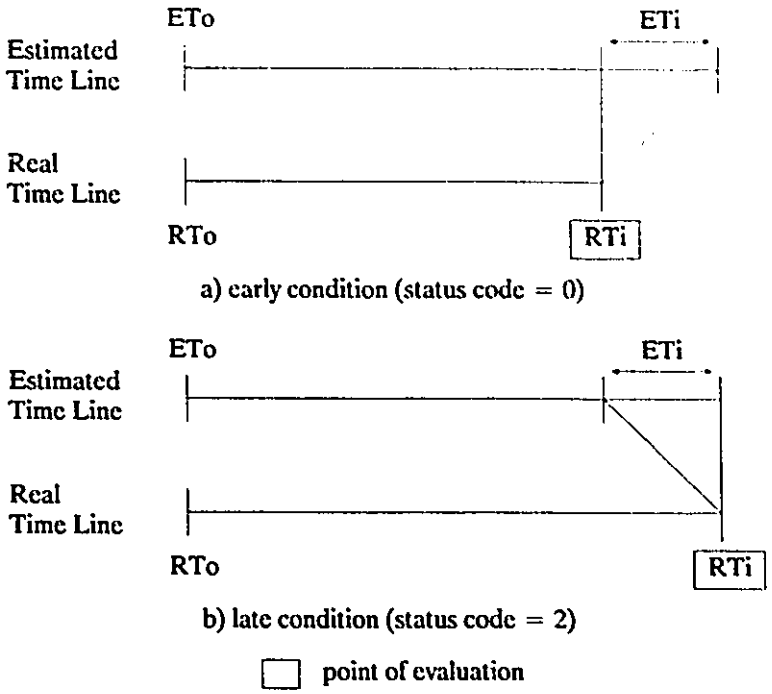


Figure 4.13: Activity-Level Status Codes

Case *a)* presents the condition where the estimated completion time of the previous activity (its CRTE) is greater than or equal to the current real time. Since this scenario indicates a condition which does not require the activity to alter its functionality, the RS would launch the current activity with a status code equal to 0. At this stage in the progress of the sub-response, it is proceeding as anticipated (or better than anticipated) and no adjustments are necessary. Alternately, case *b)* presents the condition where the estimated completion time of the previous activity is less than the current real time. In this scenario, the status code for the current activity would be set equal to 2, indicating that the previous activities have consumed more time than estimated and that the current activity must alter its functionality if it is to help the sub-response to meet its deadline.

4.4.3 Priority Manipulation

Priorities form the basis of the MAPE resource allocation policy. They are the means by which the RS dynamically constructs allocation schedules that enable conformance to time constraints. The RS is governed by a design assumption that considers all sub-response deadlines as hard time constraints which should be met under all

operating conditions. This assumption is consistent with the principle that final responsibility for maintaining conformance to the time constraint lies with the application.

The MAPE strategy for manipulating priorities consists of active control at both the sub-response and activity levels. The strategy is implemented using the *priority assignment* and *priority refinement* algorithms. The priority assignment algorithm is responsible for assigning unique base priorities to each of the application's time-constrained sub-responses before execution while the priority refinement algorithm is responsible for tuning the priorities during execution. The criteria for determining and altering priorities is necessarily different in the two algorithms, as is the frequency and time at which they execute.

At the sub-response level, the priority assignment algorithm executes once during application initialization. Its aim is to assign a unique base priority to each time-constrained sub-response. The base priority serves to categorize the sub-responses with respect to importance and reflects their rank within the application. To fulfill its priority assignment objectives, the algorithm implements *earliest deadline* and *least slack time* scheduling, using the sub-response's deadline and slack time as the defining criteria. Slack time is defined by the following:

Slack Time = Deadline - Estimated Completion Time

The importance, or *weight* attributed to each of the criteria in the priority assignment process is configurable by the developer and identical for each sub-response (PA_Weight). It is possible for example, to assign base priorities based entirely on earliest deadline, entirely on least slack time, or using a mixture of the two. The weighted value is defined by the following:

Weighted = PA_Weight * Deadline + (1 - PA_Weight) * Slack Time
Value

The base priorities for individual sub-responses are assigned in inverse relation to their weighted values; the sub-response with the lowest value receives the highest base priority. To preserve the relative positioning of the sub-responses with respect to the available priority levels, base priorities are assigned in increments. These increments are configurable and under developer control. They serve to establish stratas within the range of available priorities and to isolate highly time-critical sub-responses from their

less critical counterparts. The increments also isolate the group of time-constrained sub-responses from those that are not governed by time.

At the activity level, the priority refinement algorithm executes before each activity is launched. Its objective is to refine the priorities of the individual activities as warranted by the changing operating conditions, and as lateness considerations dictate. The adjustments are made at discrete points during the execution of the sub-response and are inherited by the activities that follow the changes. They cause the importance of activities to be altered relative to the others in the sub-response and in the rest of the application. The algorithm implements *least registered progress* and *least anticipated progress* scheduling, using the registered progress and anticipated progress as the defining criteria. Registered progress is defined by the following:

Registered Progress = Time Alotted - Time Consumed

The registered progress criteria gives a measure of current lateness by quantifying the progress accumulated before the point of evaluation. A positive result identifies an early condition (sufficient time alotted) while a negative result reflects an existing late condition. Anticipated progress is defined by the following:

Anticipated Progress = Time Remaining - Time Needed

The anticipated progress criteria gives a measure of future lateness by quantifying the progress expected after the point of evaluation. A positive result identifies an early condition (sufficient time remains) while a negative result reflects a projected late condition.

In a manner that is similar to the priority assignment algorithm, the refinement algorithm makes use of a configurable weight to control the importance attributed to the two defining criteria. The weight is identical for every activity in the application (PR_Weight). The weighted value is defined by the following:

Weighted Value = PR_Weight * Registered Progress + (1 - PR_Weight) * Anticipated Progress

The weighted value determines if an increase in the priority of the current activity is necessary. A priority increase for the current activity is granted in inverse relation to

the sign of the weighted value; a negative result receives an increase in priority. The amount by which the priority is increased is controlled by a configurable increment. Since the goal of priority manipulation at the activity level is to refine the base priority, this increment is necessarily smaller than that used by the priority assignment algorithm.

4.5 Relevance

The purpose of this chapter was to present the Multi-Activity Programming Environment (MAPE). The chapter introduced the three main components of MAPE and explained the role each plays in the development and execution of a concurrent real-time application. It described the characteristics of the execution and coordination models, and the activity, node and edge programming abstractions that are offered to the application developer. It also described the notation and language facilities that allow developers to specify both the computations that must be performed in order to fulfill the application's functionality and the interactions that arise as a result of partitioning the functionality. Lastly, the chapter explained the design of the run-time support facilities and described the methods used to facilitate dynamic adaptation of the application's functionality.

The discussions of this chapter attempted to outline MAPE's support for the requirements identified in Part I (see sections 2.4 and 3.5). They have shown the following areas of support.

- support for the decomposition of functionality into cooperating entities
- support for separate treatment of the execution and coordination concerns
- support for structuring the functionality of a response into layers of abstraction
- support for specification of execution and coordination requirements
- support for specification of deadline-based time constraints
- support for linear (sequential and parallel) coordination control structures
- support for conditional (instantaneous and deferred) coordination control structures
- support for cyclic (static and dynamic) coordination control structures
- support for dynamic modification of application functionality
- support for urgency-based allocation of platform resources

Chapter 5

MAPE Functionality

The purpose of this chapter is to demonstrate the features of MAPE using a collection of sample programs. The first section describes the multiprocessor system used to perform the tests and the elements that are common to each demonstration program. The following six sections each outline the goals of a demonstration, the functional characteristics of the response(s) that constitute the program, and the results collected during test runs. Lastly, the final section offers a brief summary and outlines the features that were demonstrated.

5.1 Test Details

The example programs were executed on a high-end, general-purpose Sun Microsystems Inc. SPARCcenter 2000 Symmetric Multiprocessor System. The SPARCcenter is designed using the tightly-coupled shared-memory architecture where the processors, memory and I/O system are connected to a high capacity system backplane. A single copy of the SunOS 5.2 multi-threaded operating system coordinates the system processor activities and allows prioritized access to the system's I/O and memory facilities. The system's backplane is composed of two 250 MB/s packet-switched interconnecting busses which operate in parallel and carry the traffic between the processor subsystems, the memory banks and the I/O channels. The test configuration consisted of eight 50 MHz SuperSPARC processors and 512 MB of memory.

The examples presented in this chapter were constructed using the process described in section 4.1, by specifying the execution requirements using the C language and the coordination requirements using the PAD notation and the CSL language. The C and CSL source code for each example can be found in Appendix B. The results are

derived from the raw statistics generated by the runtime system (see section A.1) and have been abstracted for clarity and understanding. For each example, a timing chart outlining the start and end times for the individual activities is shown to illustrate the run-time progress of the response(s) which comprise the example. The raw results are also reproduced in Appendix B.

The sample programs were designed to exercise the MAPE components, and as such are not necessarily representative of actual real-time application functionality. The activities they contain simulate real functionality by executing a time-bounded, CPU-intensive C function. The function consumes processor time by repeating a series of floating-point computations. Its call syntax is the following.

```
int asn_delay(seconds_to_delay)
int seconds_to_delay;
```

5.2 Example 1: Basic Functionality

The primary purpose of this example is to demonstrate the use of the various nodes incorporated in the PAD notation and supported in the CSL language. Of the nine nodes supported by the notation and language, six are represented in this example. They include the activity, merge, counter, wait, gate, and end nodes. The remaining three (branch, cycle tail types I and II) are demonstrated in examples 2 and 3. A secondary purpose of this example is to demonstrate the runtime system's ability to dynamically interpret a response's coordination requirements using multiple processors. The ASN for the basic functionality example is shown in figure 5.1.

The response implements a three iteration static cycle that is controlled by gate and counter nodes. *Aa*, the first activity in the response simulates a period of initialization and a period of waiting for the starting event. Once this simulated function is complete, the activity terminates and transfers control along its output edges. This enables the *C1* counter node and activates the input trigger to the *M1* merge node. The *M1* merge node disables the *G1* gate node to block trigger propagation out of the cycle and enables the *G2* gate node to allow propagatation along the cycle's feedback path. *M1* also activates the input trigger of the counter node to decrement the counter value. Finally, the node transfers control to the body of the cycle.

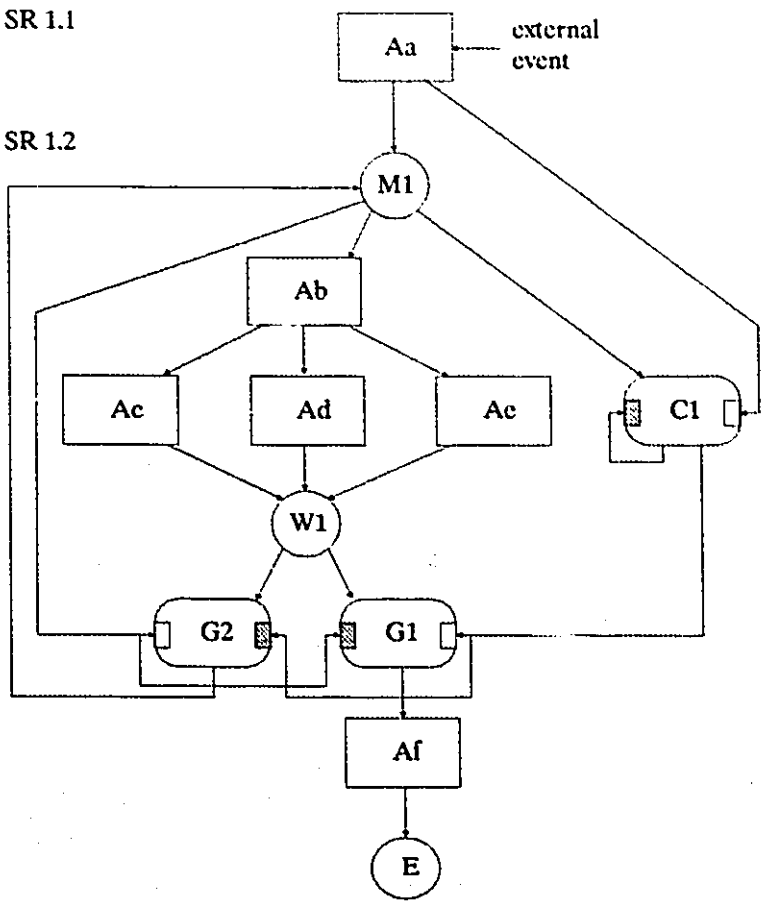


Figure 5.1: ASN for Example 1

Each iteration of the cycle consists of the execution of four activities. Firstly, the *Ab* activity is executed. Upon completion, the activity delegates control to the *Ac, Ad,* and *Ae* activities, which can execute simultaneously. The multiple paths are joined using the *W1* wait node. At the bottom of the iteration, the *W1* wait node activates the input triggers of the *G1* and *G2* gate nodes. These nodes remain in their present state until the counter value reaches zero, which causes their state to reverse. At that point, the *G1* gate node is enabled to allow the *W1* wait node trigger to propagate to the *Af* activity and the *G2* gate node is disabled to block the trigger from propagating to the head of the cycle. The *Af* activity then executes and the response terminates.

5.2.1 Test Results

The basic functionality example was run in one, two, and three processor configurations. The results obtained from the test runs are illustrated using the timing charts of figure 5.2.

The timing charts of the three test cases depict similar coordination profiles; the *Aa* activity is executed once, followed by three iterations of the *Ab, Ac, Ad,* and *Ae* activities, and finally, the *Af* activity. The simulated delays times in seconds are the following.

Activity	Simulated Delay
Aa	2
Ab	3
Ac	3
Ad	8
Ae	5
Af	3

The charts illustrate the operation of the runtime system in a multiprocessor context and the effects of parallel hardware on a concurrent response. In the one processor configuration, the activities within the body of the cycle are interleaved onto the processor and executed serially, thereby extending the response's execution time by a relatively wide margin. The activities of the response are assigned the same base priority and compete amongst themselves and the runtime system for the lone processor. As a result, the difference between the start and end times of concurrently executing activities is greater than the simulated delay times. In the two and three processor cases, this condition is alleviated by the other processors which permit the *Ac, Ad,* and *Ae* activities to execute simultaneously and reduce the overall time required to complete the response.

Furthermore, the timing charts show the effects of executing the runtime system on the same platform resources. During the execution of the response, the runtime system is active, determining and performing the control transfers between the activities and managing the map of activities to threads. The time spent in performing these support function are reflected in the scattered start times of the individual activities. In the two and three processor case, this overhead is relatively the same, but because the load is distributed across more processors, its impact is less apparent.

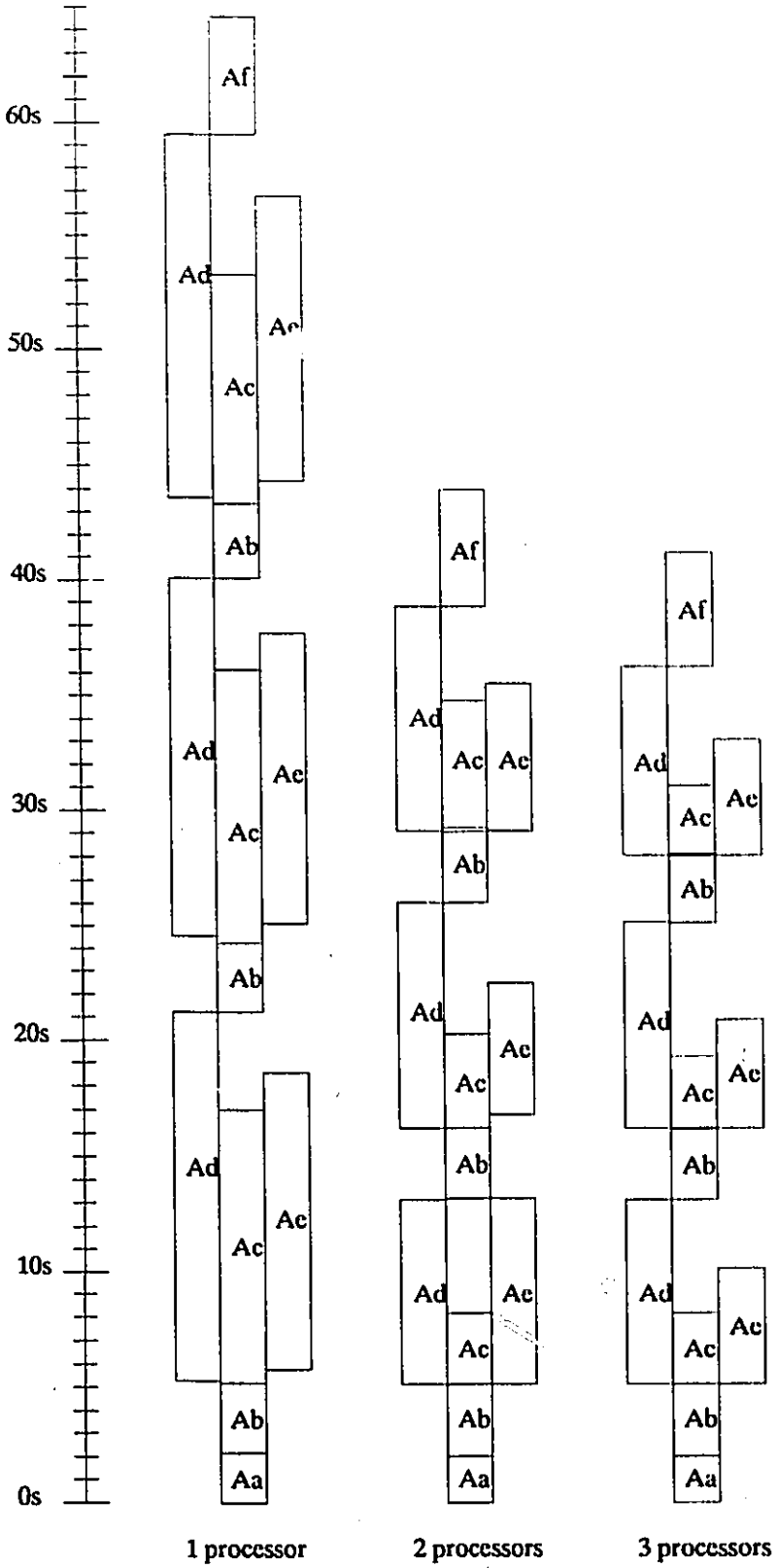


Figure 5.2: Timing Chart for Example 1

5.3 Example 2: Conditional Path

The aim of this example is to demonstrate the use of the branch node with respect to conditions and their qualifiers. The ASN for the conditional path response is shown in figure 5.3.

The response implements a decision tree containing all combinations of condition values and qualifiers. The example relies on a set of condition variables which are defined as unrestricted data elements. These elements have global scope, are shared by all activities in the response, and can be manipulated by any one of them at any time. The condition variables are initialized in the following manner.

```
c1 = -1
c2 = 0
c3 = 0
c4 = 1
c5 = 0
c6 = 2
```

The *c2* condition variable is the only one which changes as a result of executing the response. In its initialized state, the value of 0 causes the input triggers to the *Ae* and *Ag* activities to be suspended pending transition of *c2* to a non-zero value. After completing half of its assigned workload, the *Ad* activity invokes the *set_cond* primitive (see A.7) to set *c2* equal to 1. This operation signals the change required of both input triggers and releases them from their suspended state.

5.3.1 Test Results

The conditional path example was run in a four processor configuration. The results obtained from the test runs are illustrated using a timing graph in figure 5.4.

The timing graph outlines the coordination profile of the executed activities. The *Aa* and *Ab* activities are executed in sequence, followed by the *Ac* and *Ad* activities which are executed in parallel. At the halfway point in the *Ad* activity's execution, the call to the *set_cond* primitive is made to release the input triggers for the *Ae* and *Ag* activities. The three activities then execute concurrently and are merged at the *Ah* activity. The *Ah* activity is followed by the simultaneous execution of the *Ai, Aj,* and *Ak* activities which are in turn followed by the parallel execution of the *Al, Ao* and *Ap,* and

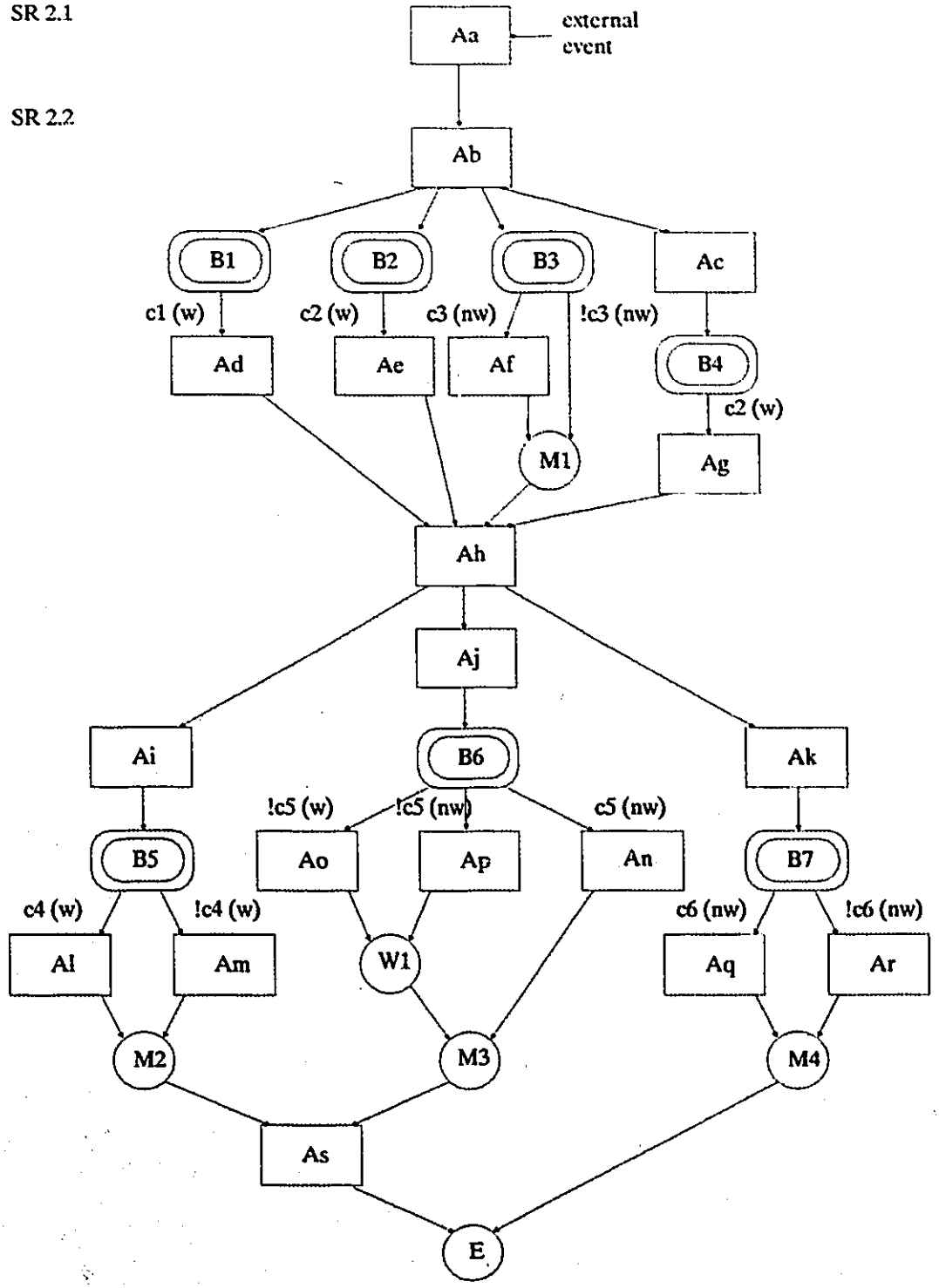


Figure 5.3: ASN for Example 2

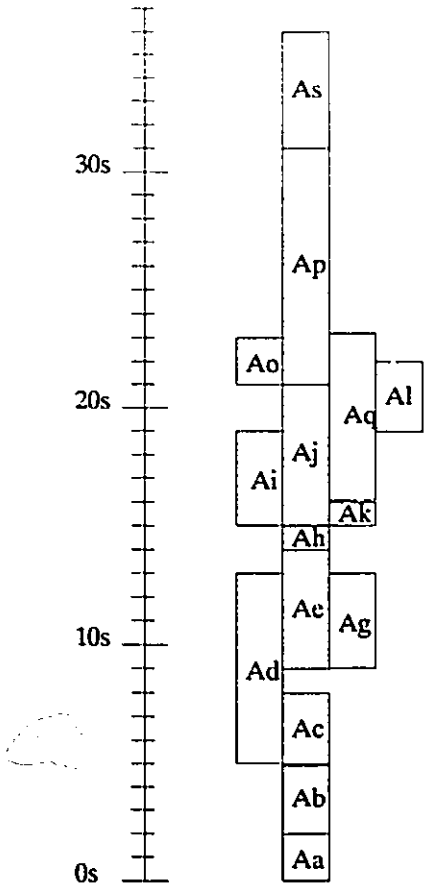


Figure 5.4: Timing Chart for Example 2

Aq activities, respectively. Finally, the *As* activity is executed and the response terminates. The simulated delay times for the activities are the following.

Activity	Simulated Delay
Aa	2
Ab	3
Ac	3
Ad	8
Ae	5
Ag	4
Ah	1
Ai	4
Aj	6
Ak	1
Al	3

```
% > example_2 -s -t 1 asn-cp
0.003: Thread id = 4 Started activity Aa at prio 0
2.005: Thread id = 4 Finished activity Aa
2.007: Thread id = 5 Started activity Ab at prio 0
5.007: Thread id = 5 Finished activity Ab
5.015: Thread id = 10 Started activity Ad at prio 0
5.016: Thread id = 11 Started activity Ac at prio 0
8.036: Thread id = 11 Finished activity Ac
9.013: SET_COND(c2, 1) < = id = 10
9.016: Thread id = 9 Started activity Ae at prio 0
9.016: Thread id = 15 Started activity Ag at prio 0
13.010: Thread id = 10 Finished activity Ad
13.042: Thread id = 15 Finished activity Ag
14.011: Thread id = 9 Finished activity Ae
14.014: Thread id = 18 Started activity Ah at prio 0
15.015: Thread id = 18 Finished activity Ah
15.018: Thread id = 19 Started activity Ai at prio 0
15.019: Thread id = 21 Started activity Ak at prio 0
15.018: Thread id = 20 Started activity Aj at prio 0
16.019: Thread id = 21 Finished activity Ak
16.022: Thread id = 23 Started activity Aq at prio 0
19.014: Thread id = 19 Finished activity Ai
19.018: Thread id = 25 Started activity Al at prio 0
21.013: Thread id = 20 Finished activity Aj
21.016: Thread id = 27 Started activity Ap at prio 0
21.017: Thread id = 28 Started activity Ao at prio 0
22.015: Thread id = 25 Finished activity Al
23.015: Thread id = 28 Finished activity Ao
23.075: Thread id = 23 Finished activity Aq
31.007: Thread id = 27 Finished activity Ap
31.015: Thread id = 36 Started activity As at prio 0
36.012: Thread id = 36 Finished activity As
```

Figure 5.5: Partial Output for Example 2

Ao	2
Ap	10
Aq	7
As	5

A copy of the output generated by the run is presented in figure 5.5. The first line consists of the shell prompt and the command issued to the system to invoke the *example_2* sample program (see A.1). The *s* argument instructs the runtime system that a summary report of the statistics is required (not shown) while the *t* argument instructs it that level 1 tracing (activity control information) should be performed. Lastly, the *asn-cp* argument corresponds to the CSL source file which contains the example's coor-

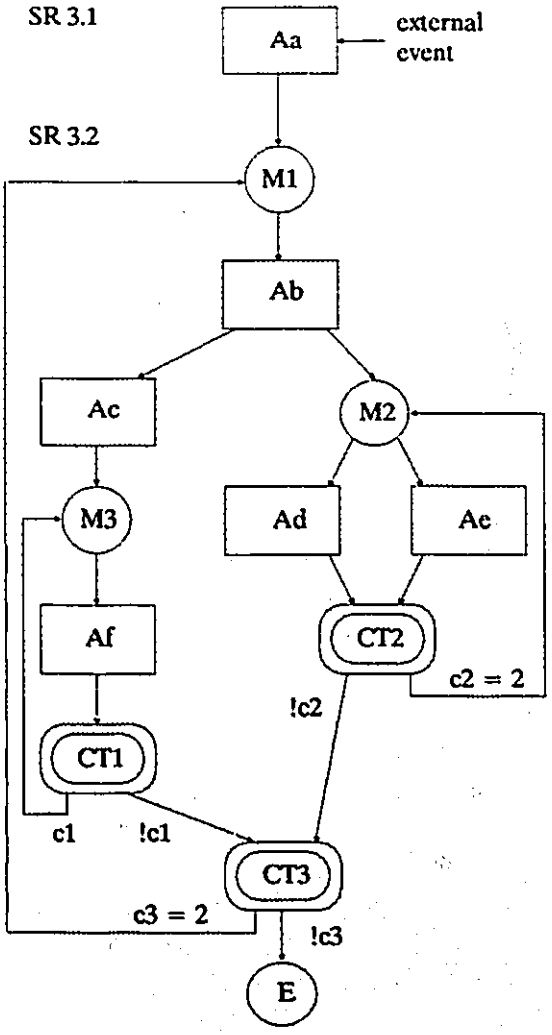


Figure 5.6: ASN for Example 3

dination information. Of particular interest in this figure is the call made by the *Ad* activity (underlying thread id equal to 10) to the *set_cond* primitive at time 9.013 seconds into the response, and how it immediately (within 0.003 seconds) released the suspended input triggers to the *Ae* and *Ag* activities.

5.4 Example 3: Nested Cycles

The purpose of this example is to demonstrate MAPE's support for static and dynamic cycles. The ASN for the nested cycle response is shown in figure 5.6.

```
int Af (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);

    loop_count--;
    if (loopcount = = 0)
    {
        loop_count = 2;
        set_cond(&c1, 0);
    } else
        set_cond(&c1, 1);
    fprintf(stdout, "c1 %d loop_count %d\n", c1, loop_count);
}
```

Figure 5.7: Activity Af for Example 3

The response implements two cycles within a main, two iteration static cycle. The inner cycles consist of a two-iteration dynamic cycle and a two-iteration static cycle. The three cycles are controlled by cycle-tail nodes. For the two static cycles, no additional control mechanism beyond specifying the number of iterations in the CSL source code is required of the activities. For the dynamic cycle however, the *Af* activity within the body of the cycle is required to interact with the runtime system to control the number of iterations.

The static outer cycle is activated by the completion of the *Aa* activity and controlled by the *CT3* cycle-tail node. To ensure the required number of iterations, the *CT3* node decrements the *c3* condition variable (which was initialized to 2) every time its input triggers are activated. It then evaluates the resulting value. If it is non-zero, the node transfers control to the *M1* merge node along the *c3* feedback path. Alternately, the node activates the trigger associated with the *!c3* output edge and passes control to the *E* node, which terminates the response. The static inner cycle, which is bounded by the *M2* and *CT2* nodes is controlled similarly.

The dynamic inner cycle is activated by the completion of the *Ac* activity and controlled by the *CT1* node. However, unlike the static cycles, the number of iterations in this cycle is maintained by the activity within its body. The *Af* activity manipulates the number of iterations using the *loop_count* global variable and the *c1* condition variable. The initial values for these variables are the following:

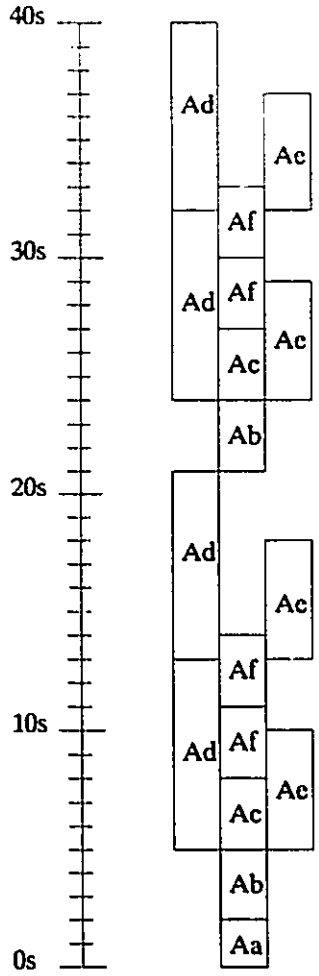


Figure 5.8: Timing Chart for Example 3

```
c1 = 1
loop_count = 2
```

Figure 5.7 shows the code within the *Af* activity for maintaining the cycle's loop count and for setting the condition variable. On every invocation, *loop_count* is decremented and evaluated to see if it is zero. If it is, *loop_count* is reset and the *c1* condition variable is set to zero by a call to the *set_cond* primitive. This action indicates to the runtime system that the cycle has completed its last iteration. Alternately, the *c1* condition is set to 1. In both instances, when the *Af* activity completes, the *CT1* node evaluates the *c1* condition. If it is zero, the node transfers control along the *!c1* output edge. If the *c1* condition value is non-zero, the node transfers control along the *c1* feedback path, which transfers control back to the head of the cycle.

```
%> example_3 -s asn-fb
c1 1 loop_count 1
c1 0 loop_count 2
c1 1 loop_count 1
c1 0 loop_count 2
```

Figure 5.9: Partial Output for Example 3

5.4.1 Test Results

The nested cycles example was run in a four processor configuration. The results obtained from the test runs are illustrated using a timing chart in figure 5.8.

The timing chart shows the execution sequence of the response's activities. The *Aa* activity is executed once followed by two iterations of the outer cycle. The body of the outer cycle consists of executing the *Ab and Ac* activities, two iterations of the static inner cycle, and two iterations of the dynamic inner cycle. The static cycle is formed by the simultaneous execution of the *Ad* and *Ae* activities while the dynamic cycle is composed simply of the *Af* activity. The simulated delay times are the following.

Activity	Simulated Delay
Aa	2
Ab	3
Ac	3
Ad	8
Ae	5
Af	3

The example was run without tracing. However, a *printf* statement was included in the *Af* activity to monitor the changes in the *loop_count* and the *c1* cycle control condition variable. A copy of the output generated by the run is presented in figure 5.9. A line of output corresponds to an invocation of the *Af* activity. The activity is invoked four times, twice in the first iteration of the outer cycle and twice in the second. The first two lines result from the first two iterations of the inner cycle. In the first iteration, *loop_count* has been decremented by 1 and *c1* has been set to 1 using the *set_cond* primitive. In the second iteration, *loop_count* has been set to 2 (re-initialized for the next invocation of the cycle) and the *c1* condition has been set to 0, indicating that the last

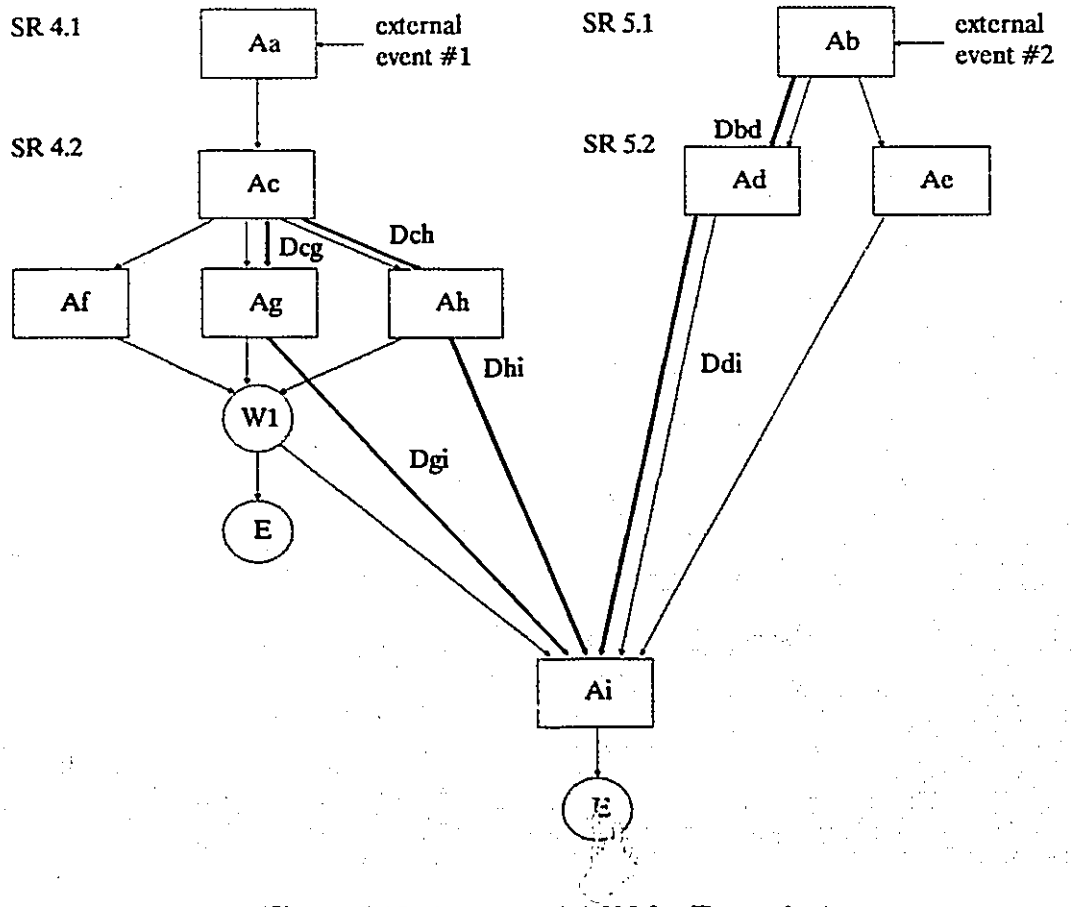


Figure 5.10: Annotated ASN for Example 4

iteration has been completed. The last two lines occur under similar circumstances and correspond to the second iteration of the outer cycle.

5.5 Example 4: Data Flow

The purpose of this example is to demonstrate the runtime system's support for the communication of restricted data elements between activities. The annotated ASNs for the responses used in the example are shown in figure 5.10.

The example consists of two dependent responses which interact by transferring data and control between their constituent activities. As discussed in section 4.2, data flow in MAPE is limited to restricted data elements. These elements are declared by individual activities and have local scope. As a result, these data elements must be prepared by the sending activity, placed into sets, and explicitly transmitted to the des-

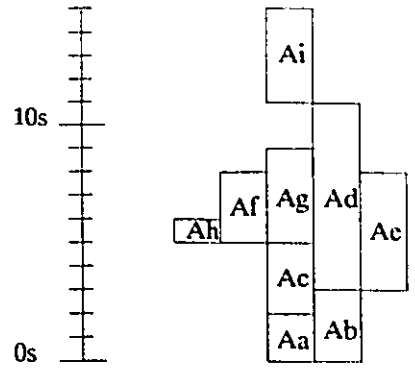


Figure 5.11: Timing Chart for Example 4

tination by a call to the *put_args* primitive (see A.5). The data flows captured by the example are the following.

Source	Destination	Contents of Data Set
Ac	Ag	-7, 7
Ac	Ah	Test_String
Ab	Ad	-5, 0, 5
Ag	Ai	Ag, -7, 7
Ah	Ai	Ah, Test_String
Ad	Ai	Ad, -5, 0, 5

Retrieval by the destination activity is automatic for the first set of data and manual thereafter. The first set of data is communicated to the activity along with its status code. Subsequent data sets are retrieved by a call to the *get_args* primitive (see A.6). Since the order by which the runtime system delivers an activity's outstanding data is indeterminate, the receiving activity is responsible for processing each set of incoming data to determine its source. To facilitate identification of the source, this example makes use of a convention whereby the source activity identifier is included as the first argument in the set.

5.5.1 Test Results

The data flow example was run in a five processor configuration. The timing chart in figure 5.11 illustrates the results gathered from the test run while figure 5.12 shows the output generated by the executing activities.

```
%> example_4 -s asn-df-1 asn-df-2
Ab received 5 arg(s), status 0 : example_4, -s, asn-df-1, asn-df-2,
Aa received 5 arg(s), status 0 : example_4, -s, asn-df-1, asn-df-2,
Ac received 1 arg(s), status 0 :
Ad received 4 arg(s), status 0 : -5, 0, 5,
Ae received 1 arg(s), status 0 :
Af received 1 arg(s), status 0 :
Ag received 3 arg(s), status 0  -7, 7,
Ah received 2 arg(s), status 0 : Test_String,
Ai received 5 arg(s), status 0 from Ad : -5, 0, 5,
Ai received 3 arg(s) from Ag : -7, 7,
Ai received 2 arg(s) from Ah : Test_String,
```

Figure 5.12: Partial Output for Example 4

The execution profile depicted in the timing chart shows the concurrent execution of the two responses. Activities *Aa* and *Ab* for example, reflect the simultaneous initialization of their responses and the wait for their respective starting events. Moreover, the chart shows the arbitrary choice of the relative arrival times of the starting event; the event for response 4 arrives before that of response 5. Also, the profile illustrates the sequence of activities performed by each response. The simulated delay times for the executed activities are the following.

Activity	Simulated Delay
Aa	2
Ab	3
Ac	3
Ad	8
Ae	5
Af	3
Ag	4
Ah	1
Ai	4

The output displayed in figure 5.12 is the result of *printf* statements inserted in the activities' code to display the arguments they received. In the example, activities such as *Ac, Ae,* and *Af,* are not destinations for data flows and therefore receive only the status code from the runtime system (a status code of 0 indicates a normal invocation condition). In addition to the status code, the remaining activities receive the data elements that were communicated to them. In the case of activities *Aa* and *Ab*, since they

are the first ones launched as part of their respective responses, they receive the command line arguments used to invoke the sample program. In all other cases, the data elements that were received originated from activities within the two responses. In the case of the *Ai* activity, the incoming data originated from three such sources. The first data set, received as part of the input arguments originated from the *Ad* activity while the others (from *Ag* and *Ah*) were received as a result of repeated calls to the *get_args* primitive.

5.6 Example 5: Run-Time Control - Sub-Response Level

The purpose of this example is to demonstrate the runtime system's priority and status code assignment algorithms for time-constrained sub-responses. A discussion of the control capabilities supported by the runtime system was presented in sections 4.4.2 and 4.4.3. Figure 5.13 depicts the ASNs for the example's responses.

The example consists of two independent responses, which are each composed of an initialization sub-response that is not constrained by time and an event processing sub-response that is. Both responses make use of unrestricted condition variables to guide their respective execution flows. The values for these conditions are not changed during the course of execution and are initialized in the following manner.

```
c1 = 1
c2 = -1
```

The deadline of sub-response 6.2 has been fixed at 18.10 seconds while that of sub-response 7.2 has been fixed at 15.68 seconds. These figures are relative to the start of their respective sub-responses.

5.6.1 Test Results

The example was run in a four processor configuration for two test cases. The test cases differ in the criteria used by the runtime system's priority assignment algorithm to determine base priorities; the first one uses the slack time while the second uses the deadline. The figures which support the discussion consist of figures 5.14 and 5.15 which show the timing chart and partial run output for the first test case, and figures 5.16 and 5.17 which show identical information for the second test case.

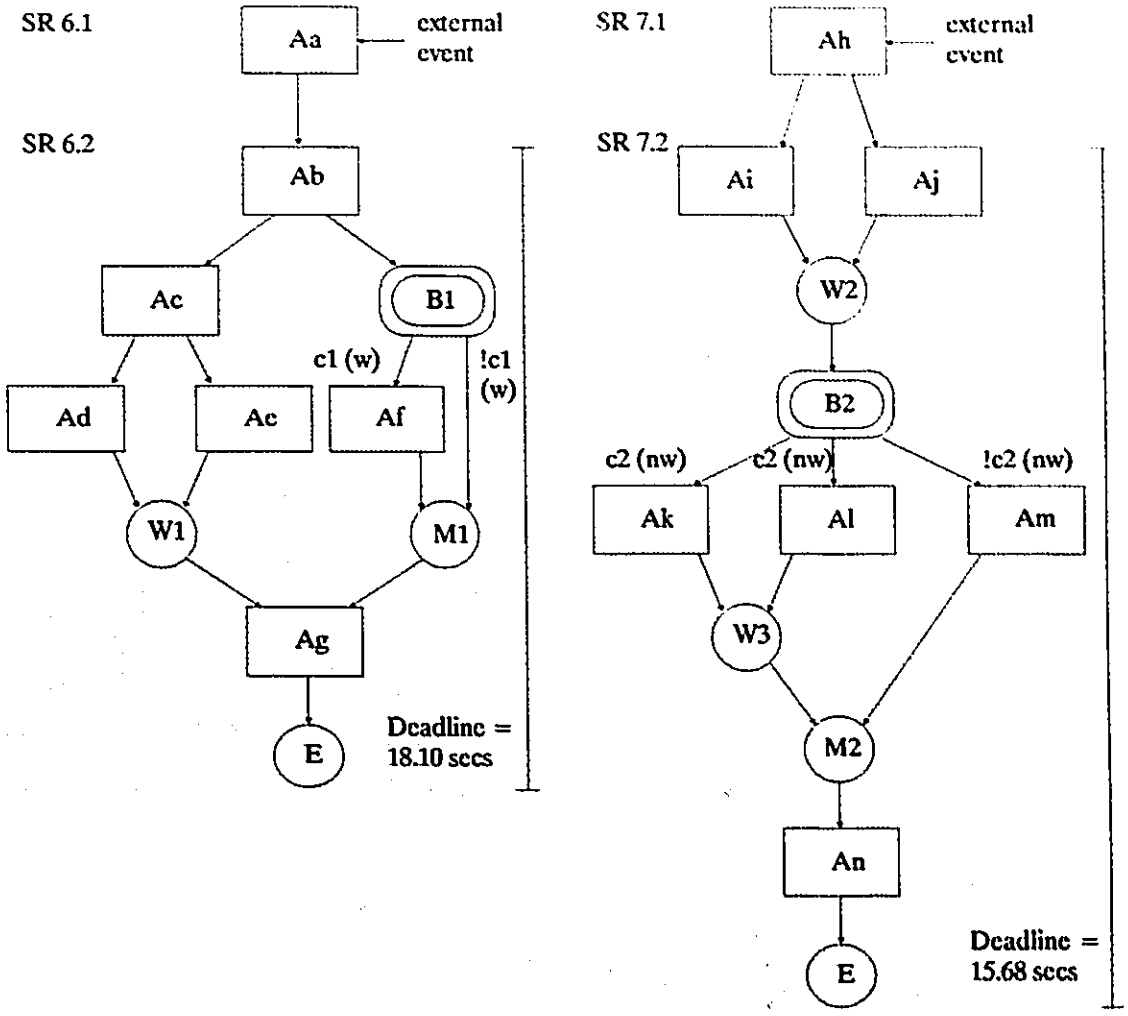


Figure 5.13: Annotated ASN for Example 5

The timing charts show the execution profiles for the two responses and the base priority assigned to each sub-response. The simulated delay times in seconds for the activities are the following.

Activity	AE	Simulated Delay
Aa	None	2
Ab	3.3	3
Ac	3.3	3
Ad	8.8	8
Ae	5.5	5

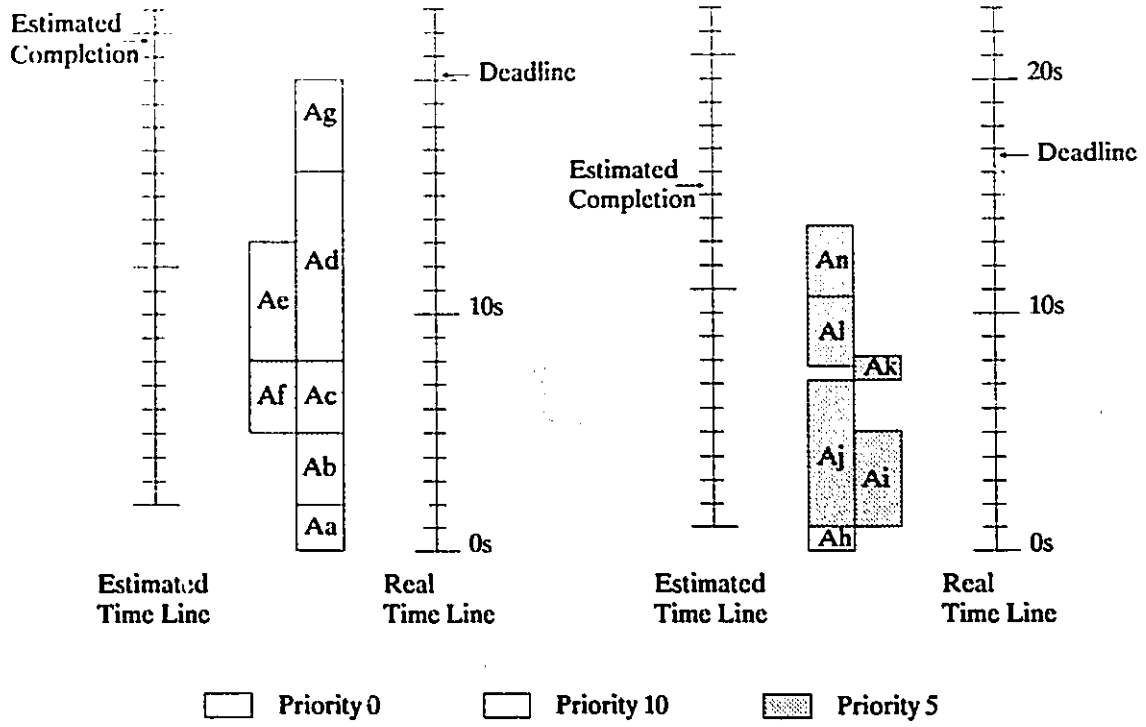


Figure 5.14: Timing Chart for Example 5 (test case 1)

Af	3.3	3
Ag	4.4	4
Ah	None	1
Ai	4.4	4
Aj	6.6	6
Ak	1.1	1
Al	3.3	3
An	3.3	3

The timing charts also show the time lines used to determine the status codes and priorities, indicating the estimated completion times for the sub-responses on the estimated time lines and their deadlines on the real time lines. The deadlines are positioned relative to the start of the respective sub-responses.

The partial output from the test runs shows the command line arguments that were passed to the runtime system to control its operation and the status output it generated as a result of level 4 tracing (deadline evaluation information). As in the previous example, each activity prints a line of output describing its identity, the number of arguments it received, and its starting status code.

```
% > example_5 -s -t 4 -d 1 0 asn-ddlcrse asn-ddlcrse-1 asn-ddl-crse-2
Aa received 11 arg(s), status 0
Ah received 11 arg(s), status 0
1.047: Deadline_1 evaluated Ai - deadline 15.68, slack 1.38, wgtd 1.38
1.048: Deadline_1 evaluated Aj - deadline 15.68, slack 1.38, wgtd 1.38
Ai received 1 arg(s), status 0
Aj received 1 arg(s), status 0
2.001: Deadline_1 evaluated Ab - deadline 18.10, slack -1.70, wgtd -1.70
Ab received 1 arg(s), status 1
Ac received 1 arg(s), status 0
Af received 1 arg(s), status 0
Ak received 1 arg(s), status 0
Al received 1 arg(s), status 0
Ad received 1 arg(s), status 0
Ae received 1 arg(s), status 0
An received 1 arg(s), status 0
Ag received 1 arg(s), status 0
```

Figure 5.15: Partial Output for Example 5 (test case 1)

The starting status codes for the sub-responses are determined by the relative positioning of their estimated completion times and their deadlines. The test cases illustrate the two possibilities for positioning of the reference points. Firstly, sub-response 6.2 represents a late condition where the estimated completion time is greater than the deadline. As shown in figures 5.15 and 5.17, the runtime system launches the first activity in the sub-response (*Ab*) with a status code of 1 to indicate that it may be required to vary its functionality during execution if it is to meet its deadline. Secondly, sub-response 7.2 represents an early or on time condition where the deadline is greater than the estimated completion time. As a result, the runtime system launches its first activities (*Ai* and *Aj*) with a status code of 0 to indicate its assumption that sufficient time is available to execute the full functionality.

The base priorities assigned to the sub-responses are obtained through similar analysis of the reference point positioning. As shown in the invocation command of figure 5.15, in the first test case, the priority assignment weight (PA_Weight) was set to 0 to direct the runtime system to determine the relative importance of the sub-responses based solely on the slack time criteria. Slack time is defined as the difference between the deadline and the estimated completion time. For sub-response 6.2, this value is -1.70 seconds while for sub-response 7.2, it is 1.38 seconds. Consequently, as shown in figure 5.14, the sub-response with the least slack time receives the highest priority (sub-response 6.2). In the second test case, the weight was set to 100 to direct

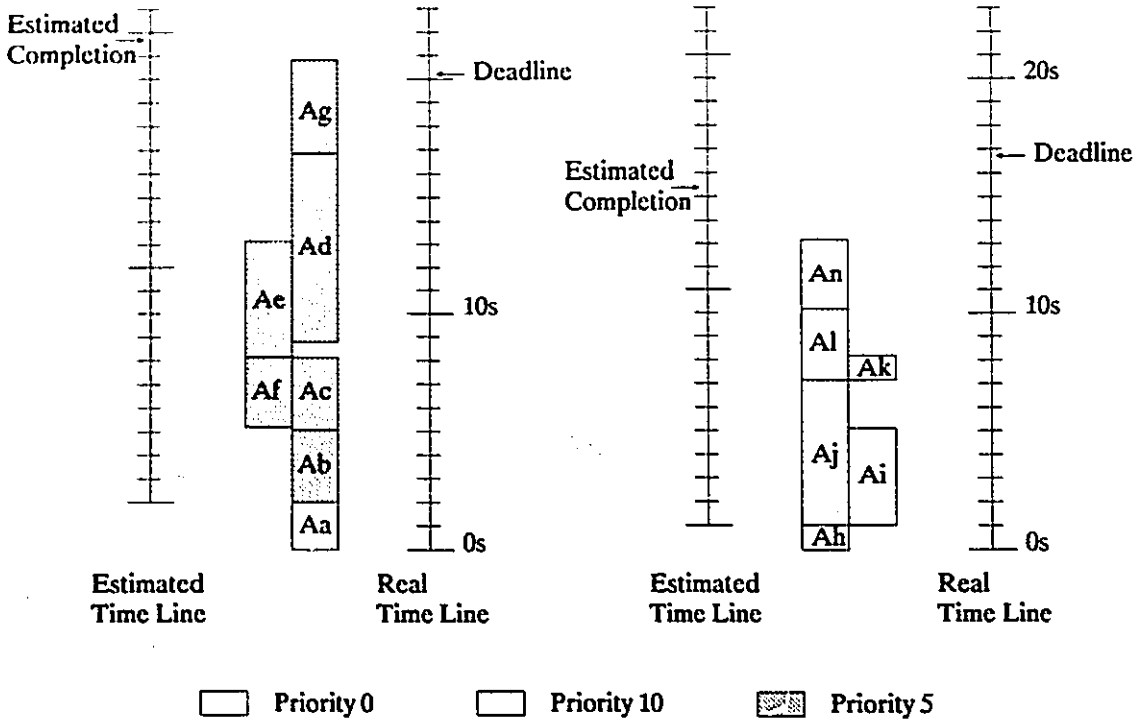


Figure 5.16: Timing Chart for Example 5 (test case 2)

the runtime system to derive base priorities based solely on the deadline criteria. The deadline for sub-response 6.2 is 18.10 seconds while for sub-response 7.2, it is 15.68 seconds. Hence, as shown in figure 5.16, the sub-response with the earliest deadline receives the highest priority (sub-response 7.2). In both test cases, since sub-responses 6.1 and 7.1 are not constrained by time, they are executed at the lowest priority.

A comparison of the results of figures 5.14 and 5.16 shows that the coarse-grain management of the priorities of sub-response 6.2 does not in itself guarantee conformance to the time constraint. In the first test case, a relative increase in priority allows it to execute within the time alotted without modifying its functionality. Relative to the start of the sub-response, its deadline is at time 20.10 seconds and it completes at time 20.00 seconds. In the second test case however, a relative decrease in priority causes it to miss its time constraint, and terminate at time 20.84 seconds. If sub-response 6.2 implemented functionality that was soft-constrained, this condition could have proven acceptable. However, if it was hard-constrained, the sub-response would have been required to act on the late status code and reduce or *sieve* its functionality.

```
%> example_5 -s -t 4 -d 1 100 asn-ddlcrse asn-ddlcrse-1 asn-ddl-crse-2
Aa received 11 arg(s), status 0
Ah received 11 arg(s), status 0
1.048: Deadline_1 evaluated Ai - deadline 15.68, slack 1.38, wgtd 15.68
1.048: Deadline_1 evaluated Aj - deadline 15.68, slack 1.38, wgtd 15.68
Ai received 1 arg(s), status 0
Aj received 1 arg(s), status 0
2.003: Deadline_1 evaluated Ab - deadline 18.10, slack -1.70, wgtd 18.10
Ab received 1 arg(s), status 1
Ac received 1 arg(s), status 0
Af received 1 arg(s), status 0
Ak received 1 arg(s), status 0
Al received 1 arg(s), status 0
Ad received 1 arg(s), status 0
Ae received 1 arg(s), status 0
An received 1 arg(s), status 0
Ag received 1 arg(s), status 0
```

Figure 5.17: Partial Output for Example 5 (test case 2)

5.7 Example 6: Run-Time Control - Activity Level

The purpose of this example is to demonstrate the runtime system's priority refinement and status code assignment algorithms for the activities of time-constrained sub-responses. The annotated ASN for the response used in the example is shown in figure 5.18.

The example consists of a single response. The first sub-response is not time-constrained and simulates a sequence which initializes the response and waits for the starting event. The second sub-response simulates processing of the event and is constrained by a deadline of 30.00 seconds. It executes activities in both sequential and parallel structures and contains a branch node with a condition variable. The condition value is initialized according to the following.

c1 = -1

Since the c1 condition variable is not changed during the execution of the response, the execution path through the ASN is static.

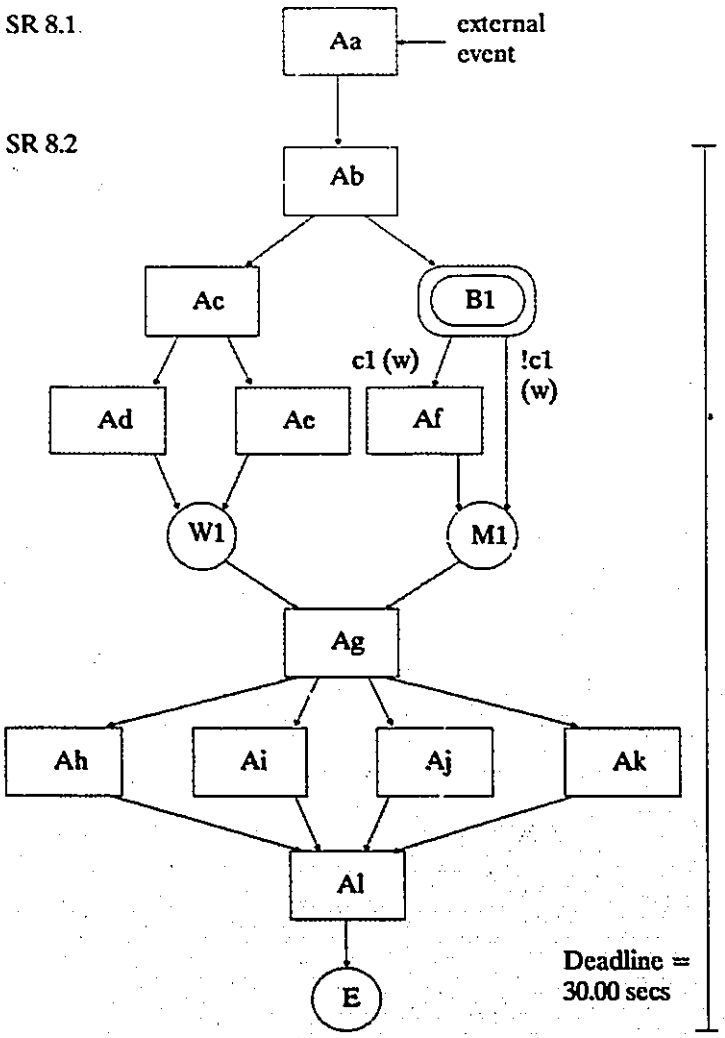


Figure 5.18: Annotated ASN for Example 6

5.7.1 Test Results

The test runs were performed in a two processor configuration for two test cases. The difference between the test cases lies in the criteria used by the runtime system's priority refinement algorithm to determine priority increases for activities. In the first test case, increases are determined based on anticipated progress while in the second, they are determined based on registered progress. The discussion of this example is supported by figures 5.19 and 5.20 which show the timing chart and the partial run out-

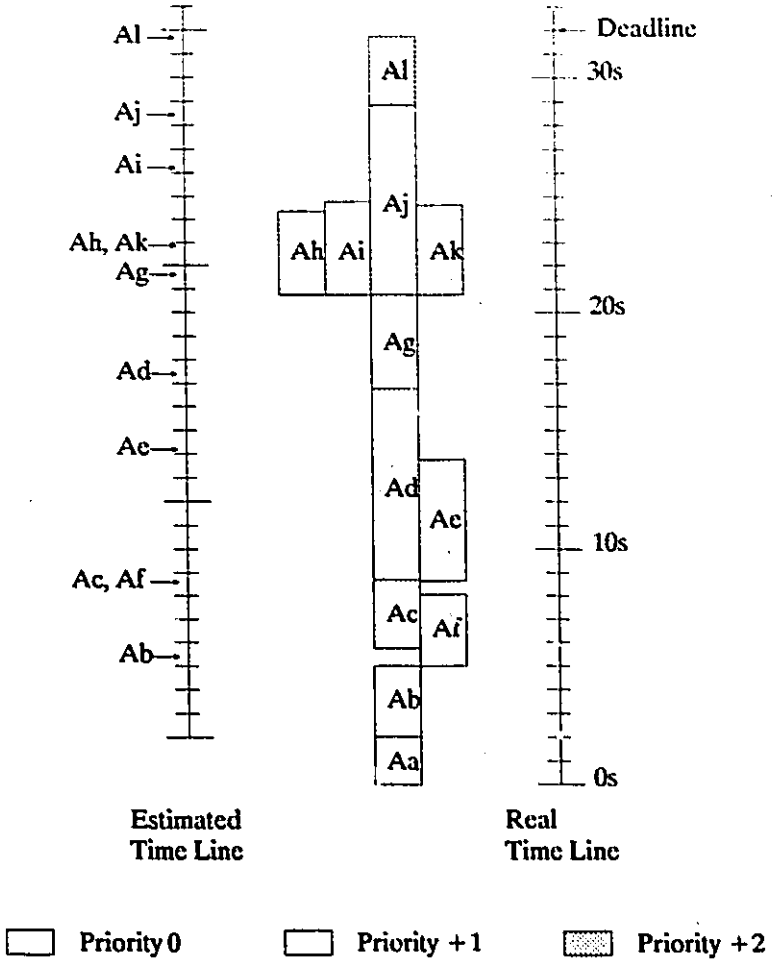


Figure 5.19: Timing Chart for Example 6 (test case 1)

put for the first test case, and figures 5.21 and 5.22 which show similar information for the second.

The timing charts show the activity execution sequences for the two responses and the priorities at which they executed. The simulated delay times for the activities depicted in the profiles are the following.

Activity	AE	Simulated Delay
Aa	None	2
Ab	3.3	3
Ac	3.3	3
Ad	8.8	8
Ae	5.5	5

```
% > example_6 -s -t 4 -d 2 0 asn-ddlfine asn-ddlfine-1
Aa received 10 arg(s), status 0
2.005: Deadline_2 evaluated Ab - reg_prog 0.00, ant_prog 0.30, wgtd 0.30
Ab received 1 arg(s), status 0
5.023: Deadline_2 evaluated Af - reg_prog 0.28, ant_prog 0.58, wgtd 0.58
Af received 1 arg(s), status 0
5.701: Deadline_2 evaluated Ac - reg_prog -0.40, ant_prog -0.10, wgtd -0.10
Ac received 1 arg(s), status 2
8.704: Deadline_2 evaluated Ad - reg_prog -0.10, ant_prog 0.20, wgtd 0.20
8.705: Deadline_2 evaluated Ae - reg_prog -0.10, ant_prog 0.20, wgtd 0.20
Ad received 1 arg(s), status 2
Ae received 1 arg(s), status 2
16.759: Deadline_2 evaluated Ag - reg_prog 0.65, ant_prog 0.95, wgtd 0.95
Ag received 1 arg(s), status 0
20.760: Deadline_2 evaluated Ah - reg_prog 1.05, ant_prog 1.35, wgtd 1.35
20.762: Deadline_2 evaluated Aj - reg_prog 1.04, ant_prog 1.34, wgtd 1.34
20.762: Deadline_2 evaluated Ai - reg_prog 1.04, ant_prog 1.34, wgtd 1.34
Ai received 1 arg(s), status 0
20.764: Deadline_2 evaluated Ak - reg_prog 1.04, ant_prog 1.34, wgtd 1.34
Ak received 1 arg(s), status 0
Aj received 1 arg(s), status 0
Ah received 1 arg(s), status 0
28.779: Deadline_2 evaluated Al - reg_prog -0.37, ant_prog -0.07, wgtd -0.07
Al received 1 arg(s), status 2
```

Figure 5.20: Partial Output for Example 6 (test case 1)

Af	3.3	3
Ag	4.4	4
Ah	1.1	1
Ai	4.4	4
Aj	6.6	6
Ak	1.1	1
Al	3.3	3

The charts also show the time lines used to determine the individual activity status codes and priorities. The estimated completion times for activities are indicated on the estimated time lines and the sub-response deadlines are shown on the real time lines. The positioning of the deadlines is relative to the start of the sub-responses.

The partial output from the test runs shows the command used to invoke the *example_6* sample program and the level 4 (deadline evaluation information) tracing output generated by the runtime system. As in the previous example, activities print their identity, the number of arguments they received, and their starting status code.

As discussed in section 4.4.2, the status codes for individual activities are derived before each is launched based on the estimated completion time for the previous ac-

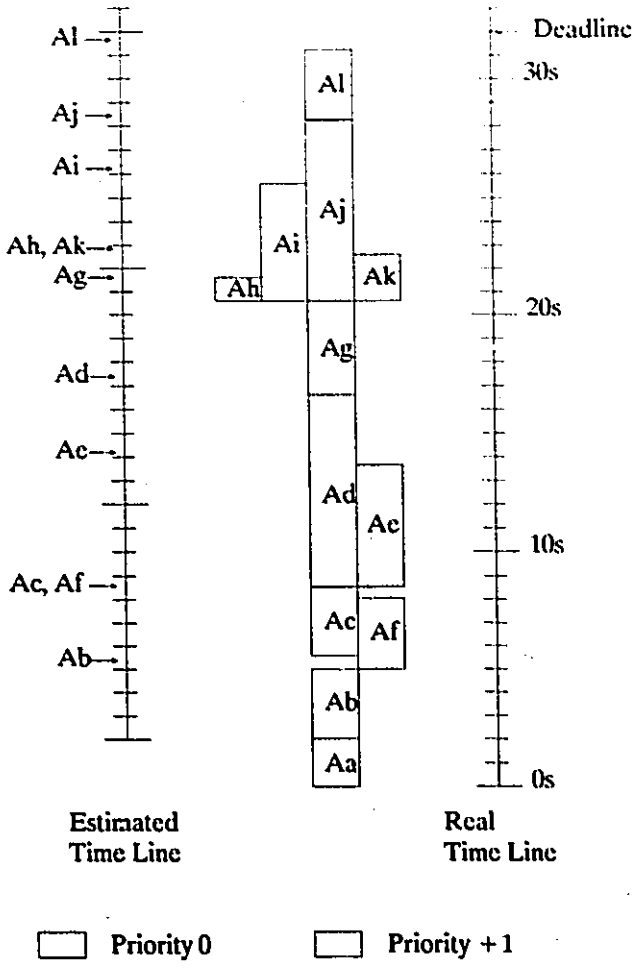


Figure 5.21: Timing Chart for Example 6 (test case 2)

tivity and the current time. The current time is the time immediately preceding an activity's launch.

Figures 5.19 and 5.20 show the two possibilities for the relative positioning of the reference points. The first possibility, as represented by the *Ac* activity, is a late condition where the current time (5.701 seconds) is greater than the estimated completion time of the previous activity (5.30 seconds). In such a situation, the runtime system determines that the work performed before the *Ac* activity has consumed more time than expected. As a result, *Ac* is launched with a status code of 2 to indicate that it is currently late and may be required to alter its functionality. The second possibility, as represented by the *Ag* activity, is an early or on time condition where the estimated completion time of the previous activity (17.40 seconds) is greater than the current time

```
% > example_6 -s -t 4 -d 2 100 asn-ddlfine asn-ddlfine-1
Aa received 10 arg(s), status 0
2.005: Deadline_2 evaluated Ab - reg_prog 0.00, ant_prog 0.30, wgtd 0.00
Ab received 1 arg(s), status 0
5.009: Deadline_2 evaluated Af - reg_prog 0.30, ant_prog 0.60, wgtd 0.30
Af received 1 arg(s), status 0
5.007: Deadline_2 evaluated Ac - reg_prog 0.30, ant_prog 0.60, wgtd 0.30
Ac received 1 arg(s), status 0
8.624: Deadline_2 evaluated Ad - reg_prog -0.02, ant_prog 0.28, wgtd -0.02
Ad received 1 arg(s), status 2
8.625: Deadline_2 evaluated Ae - reg_prog -0.02, ant_prog 0.28, wgtd -0.02
Ae received 1 arg(s), status 2
16.658: Deadline_2 evaluated Ag - reg_prog 0.75, ant_prog 1.05, wgtd 0.75
Ag received 1 arg(s), status 0
20.679: Deadline_2 evaluated Ai - reg_prog 1.13, ant_prog 1.43, wgtd 1.13
20.681: Deadline_2 evaluated Ah - reg_prog 1.13, ant_prog 1.43, wgtd 1.13
Ah received 1 arg(s), status 0
20.680: Deadline_2 evaluated Aj - reg_prog 1.13, ant_prog 1.43, wgtd 1.13
Aj received 1 arg(s), status 0
20.682: Deadline_2 evaluated Ak - reg_prog 1.12, ant_prog 1.42, wgtd 1.12
Ak received 1 arg(s), status 0
Ai received 1 arg(s), status 0
28.269: Deadline_2 evaluated Al - reg_prog 0.14, ant_prog 0.44, wgtd 0.14
Al received 1 arg(s), status 0
```

Figure 5.22: Partial Output for Example 6 (test case 2)

(16.759 seconds). Under these conditions, the runtime system determines that the work performed since the start of the sub-response has consumed less time that expected and the activity is launched with a status code of 0.

The priorities of individual activities are also derived based on the positioning of the reference points and refined before each is launched. As outlined in section 4.4.3, the priorities of activities are inherited from their predecessor(s) and refined upwards in configurable increments. In this example, the increment is 1.

In the first test case, the priority refinement weight (PR_Weight) was set to 0 to direct the runtime system to determine priority increases based entirely on the anticipated progress criteria. Anticipated progress is defined as the difference between the time remaining before the deadline and the estimated time required to complete the outstanding work. A negative anticipated progress, as shown in figure 5.20 for activities *Ac* (-0.10 seconds) and *Al* (-0.07 seconds), reflects a late condition where the remaining time is judged insufficient to complete the sub-response. Figure 5.19 illustrates the resulting manipulation by the runtime system; the priorities for these activities are increased by the configured increment.

As shown in the invocation command of figure 5.22, in the second test case, the weight was set to 100 to direct the runtime system to derive base priorities based entirely on the registered progress criteria. Registered progress is defined as the difference between the time alotted to perform the completed work and the time consumed to complete it. A negative registered progress, as shown for activities *Ad* (-0.02 seconds) and *Ae* (-0.02 seconds), constitutes a late condition where the time consumed in performing the previous activities was greater than the time alotted for their completion. Hence, as depicted by figure 5.21, the priorities for these activities are increased by the configured increment and inherited by their successors.

5.8 Relevance

The purpose of this chapter was to demonstrate the features of MAPE by exercising its components in an actual execution environment. The chapter reviewed the details of the testing and described the functional characteristics and test results for six sample programs. For purposes of clarity and to facilitate understanding, the sample programs were designed to isolate and exercise a manageable subset of MAPE's features. As a result, they are largely synthetic and are not necessarily representative of an actual multi-threaded, real-time application. Although not presented here, the C and CSL code for each of the sample programs, as well as the corresponding raw results, have also been provided (see Appendix B).

The sample programs have shown the following main features of the environment.

- ability to interpret and execute linear, conditional, and cyclic coordination control structures
- ability to communicate data elements between activities
- ability to interpret CSL source files using multiple processors
- ability to dispatch individual activities onto multiple processors
- ability to assign status codes for time-constrained sub-responses based on early, on time, or late conditions
- ability to assign priorities for time-constrained sub-responses based on slack time and deadline
- ability to assign status codes for individual activities based on on early, on time, or late conditions
- ability to refine priorities of individual activities based on registered progress and anticipated progress

Chapter 6

Conclusion

The purpose of this chapter is to offer some concluding remarks on the thesis and the research it describes. The first section reviews the goals of the thesis and provides a brief summary of its contents. The second section discusses its contributions and identifies some potential areas for future study.

6.1 Thesis Summary

The goals of the thesis were to define an environment for programming real-time multi-threaded applications, to develop the basic components required to support it, and to demonstrate its operation. The focus of Part I (Chapters 2 and 3) was to investigate and derive the capabilities that were required of the programming environment in order to adequately address the problem domain. The detailed requirements of sections 2.4 and 3.5 can be aggregated into the following key capabilities.

- to provide a programming model which addresses the execution and coordination concerns of concurrent applications
- to provide a programming model which makes provisions for the defining characteristics of the real-time problem domain
- to provide consistency with an effective approach to the development of concurrent real-time systems
- to provide the support facilities required to merge the programming model with a multiprocessor threads implementation

The first capability required that the environment support a programming model that addressed the fundamental concerns of concurrent applications. The role of the model is to provide a set of abstractions which allow developers to describe the execu-

tion and coordination requirements of applications. In Chapter 2, various types of concurrent programming models were discussed using the following points of comparison.

- ability of the model to facilitate description of the functionality that is contained within the work units and the interactions that occur between them
- ability of the model to support provisions for the characteristics of specialized problem domains
- ability of the model to support application portability

Of the three types of concurrent programming models reviewed, the non-integrated model with dedicated languages for execution and coordination seemed most capable of providing the level of abstraction, flexibility, and portability that is required by special-purpose concurrent applications.

The second capability required that the programming model address the particular characteristics of the real-time problem domain. Chapter 3 discussed these characteristics and identified four classes of provisions which required support in the model.

- provisions to allow an application to be specified as a set of responses to events
- provisions to enable management of complexity within the application by structuring responses using layers of abstraction.
- provisions for continued operation in the presence of dynamic operating conditions
- provisions for accomodating the stringent environment, platform, and internal application constraints that are representative of the problem domain

The third capability required that the environment be consistent with an effective real-time system development approach. The development approach's goal is to engineer a solution that conforms to the application's functional requirements and that embodies the tradeoffs made between the various constraints. Also, by determining the degree to which functionally correct results may be exchanged for temporally correct results, the solution outlines the importance placed on the functional and temporal correctness of the system. In Chapter 3, various approaches to the development of real-time systems were evaluated using three points of comparison.

- ability of the resulting system to ensure conformance to time constraints under various operating conditions
- ability of the approach to facilitate maintenance and evolution of the system
- ability of the approach to support the urgency-based modification of the system's responses

Of the three approaches considered, the constraint-centric development approach seemed most capable of ensuring conformance to time constraints under unusual and possibly unanticipated operating conditions, of effectively supporting the maintenance and evolution of the system, and of controlling application functionality based on the passage of time.

The fourth capability required that the environment provide the support facilities necessary to integrate the programming model with existing thread implementations. The thread implementation provides the multiprocessor platform's logical view of concurrency and contains a set of abstractions for representing threads and their interactions. The characteristics of the thread implementation and the abstractions it supports influence the degree of added functionality that must be implemented by the support facilities to remain consistent with the abstractions offered in the programming model. In Chapter 2, various strategies for the implementation of threads were considered using two basic points of comparison.

- the execution properties of the supported threads and their influence on the nature of the work which may be assigned
- the thread coordination overheads and their influence on the amount of work that may be encapsulated

Of the three thread implementation strategies reviewed, the balanced implementation appeared most capable of limiting restrictions on the nature and the amount of work assigned to threads and of simplifying the responsabilities of the support facilities.

The focus of Part II (Chapters 4 and 5, Appendices A and B) was to describe and demonstrate the capabilities of MAPE. The environment defines orthogonal execution and coordination models which are independently tailored to the non-mainstream requirements of the real-time problem domain. In both models, the abstractions are sufficiently high-level as to isolate the application from the underlying view of concurrency and to limit its dependency on the platform. In Chapter 4, the three main components of MAPE were described.

The first component is the execution model and it provides the data definition and manipulation constructs required to fully describe the application's execution requirements. Its main abstraction is the activity, which encapsulates portions of the application's functionality and serves as the main response building block. Through the starting status code, the model allows an activity to be made aware of conditions relat-

ing to the passage of time, and to forego some computations based on its urgency considerations. As a result, the activity may adapt its functionality to offer continued operation in the presence of dynamic conditions and to ensure conformance to time constraints. The execution model is supported by the C execution language.

The second component is the coordination model and it provides the constructs required to specify the coordination requirements of activities. Its main abstractions are the node, which represent activities and control points, and the edge, which represents the interactions between nodes in general and between activities in particular. These abstractions permit activities to be assembled into event-driven response structures. The model also makes provisions for the specification of deadline-based time constraints and for the organization of response complexity. The complexity is managed by progressive levels of refinement which are based on the functional, temporal, or operational attributes of the assigned functionality. The model is supported by the graphical PAD notation and the specialized CSL language.

The third MAPE component is the runtime system and it provides the logic which dynamically interprets the application's coordination specification and translates the abstractions offered by the programming model into those supplied by the thread implementation. The MAPE runtime system also attempts to match the application's demand for resources with the platform's availability of resources. It monitors the application to derive its temporal status, communicates it by way of activity status codes, and acts on it by manipulating the priorities of sub-responses and activities.

Lastly, in Chapter 5, some sample programs were used to demonstrate the key features of the environment. Examples 1 through 3 showed the environment's ability to specify and execute the transfer of control between activities in linear, conditional, and cyclic structures respectively. Example 4 showed its support for the specification and the execution-time exchange of locally-defined restricted data elements. The chapter also included examples 5 and 6 which showed the programming environment's support for the specification of estimated activity execution times and support for adaptability to time constraints. These examples also demonstrated MAPE's ability to inform the application (both at the sub-response and activity levels, respectively) of its consumption of time during execution and to influence resource allocation based on urgency conditions.

6.2 Contributions and Future Research

In summary, the research described in this thesis demonstrated the feasibility of developing a programming environment that simplifies the task of developing time-sensitive multiprocessor applications. The research has contributed an analysis of the requirements of such an environment, in terms of the capabilities that it requires to satisfy the concerns of concurrent, real-time application development and execution. It has also contributed the high-level models and languages necessary to partition an application into cooperating entities, to structure these as responses to events, and to specify the applicable time constraints. Furthermore, it has developed and demonstrated the support facilities necessary to monitor the passage of time, to recognize outstanding urgency conditions, to inform the application of its consumption of time, and to manage the availability of platform resources.

In its current state, MAPE lacks some relatively important features and possible improvements can be found in its three main components, as well as in its general capabilities. If pursued, these improvemements would serve to complete the environment and effect a migration towards a more robust and comprehensive application *development* environment.

Firstly, the execution model could be extended to support the management of errors during the execution of the application. As described, the model makes no reservations for errors that occur during execution and provides no means of recovering from these failures. The model is optimistic as it assumes that errors will either not occur, or that they will occur and be managed locally without impact to the other activities. Hierarchical error management for example, could be implemented whereby errors occuring at the activity level are propagated to the response level which are in turn propagated to the application level as required.

Secondly, the coordination model could be extended to include support for the specification and implementation of exclusion constraints between activities that execute in parallel. In its present form, the environment places the responsibility for handling mutual exclusion in parallel execution paths with the developer. Activities that execute concurrently must declare thread synchronization objects and invoke the corresponding operations during activity execution. This requirement has the potential of restricting the portability of MAPE-compliant applications.

The coordination model could also be extended to support the specification of different forms of time constraints. The model embodied in the PAD notation and CSL language described in this thesis accepts time constraints as deadlines relative to the start of the sub-response. In future versions, the constraint could be specified as a deadline with a tolerance on the acceptable jitter, or as a time window with the minimum and maximum completion times explicitly stated by the developer.

The runtime system could be improved by increasing the number of status codes that are communicated to the activities of a time-constrained response. The current runtime system support for two status codes may be restricting and may not convey sufficient *useful* information to the application. The number of return codes for example, could be augmented to better reflect the degree of lateness. In such a scheme, codes could be defined to represent conditions where the activity is late by 10%, 20%, 30%, etc. in addition to the early and on time conditions.

The runtime system could also be studied to gauge the nature of the overhead it brings to an executing application. The assumptions made in the sample programs (relatively coarse-grain activities and deterministic execution times) may mask the true impact on application performance. In support of this for example, the *asn_delay()* primitive could be modified to simulate delay periods of a finer granularity (less than a second) and to include random delays.

Further study of the runtime system could also investigate the relationship between the layers of scheduling present in MAPE-compliant applications. These layers consist of the runtime system layer, which dispatches activities onto user threads, the threads package layer, which dispatches user threads onto kernel threads, and the kernel layer, which dispatches kernel threads onto processors.

In a broader scope, the programming environment could be equipped with a collection of productivity tools that facilitate application development. A graphical user interface (GUI) tool could be constructed for example, to support the PAD notation and allow graphical input of the application's coordination requirements. The tool could include a PAD editor, verifier, activity execution time estimator, and automatic CSL source file generator.

Finally, a case study resulting from the development of a representative application using the programming environment could yield considerable insight into its utility and completeness. The example responses described in the thesis were constructed to demonstrate the main features of the PAD notation, the CSL language, and the MAPE runtime system and as such may not necessarily reflect the characteristics of actual real-time applications.

Bibliography

[AP77] Adkins G. and Pooch U., *Computer Simulation: a Tutorial*, IEEE Computer, April 1977.

[Ahu86] Ahudja, S., Gelernter, D. and Carriero, N., *Linda and Friends*, IEEE Computer, August 1986.

[Bal90] Bal, H., *Programming Distributed Systems*, Silicon Press and Prentice-Hall, 1990.

[Blu92] Blum, B.I., *Software Engineering: A Holistic View*, Oxford University Press, 1992.

[Car89] Carriero, N. and Gelernter, D., *Linda in Context*, Comm. of the ACM, April 1989.

[Coo90] Cooper E. and Draves R., *C-Threads*, Carnegie Mellon University, School of Computer Science, September 1990.

[Geh86] Gehani, N.H. and Roome, W.D., *Concurrent C*, Software Practice and Experience, September 1986.

[Gel92] Gelernter, D. and Carriero, N., *Coordination Languages and their Significance*, Comm. of the ACM, February 1992.

[Gol90] Golub D. et al., *Unix as an Application Program*, USENIX Summer 1990, June 1990.

[Kep85] Kepecs J., *Lightweight Processes for UNIX Implementation and Applications*, USENIX Summer 1985.

[KJ86] Krieger M. and Joannis R., *Process Activity Diagrams: A Tool for the Specification and Design of Multiple Microprocessor Systems*, Proceedings of the 1986 ISMM Symposium, February 1986.

[KLR94] Klein M. and Lehoczky J. and Rajkumar R., *Rate-Monotonic Analysis for Real-Time Industrial Computing*, IEEE Computer, January 1994.

[Kri88] Krieger M. et al., *Process Activity Language (PAL) for Planning and Coordination of FMS*, Proceedings of the 1988 Symposium on Manufacturing Application Languages, June 1988.

[Kri94] Krieger M., *The Multi-Activity Paradigm: An Approach for the Design of Embedded Systems by Application Specialists*, Submitted for Publication, June 1994.

[LB88] Luqi and Berzins V., *Rapidly Prototyping Real-Time Systems*, IEEE Software, September 1988.

[Liu91] Liu J.W.S. et al., *Algorithms for Scheduling Imprecise Computations*, IEEE Computer, May 1991.

[NZ92] Natarajan S. and Zhao W., *Issues in Building Dynamic Real-Time Systems*, IEEE Software, September 1992.

[Mar93] Marrin, K., *Multithreading Support Grows Among Real-Time Operating Systems*, Computer Design, March 1993.

[Pow91] Powell M. et al., *SunOS Multi-Thread Architecture*, USENIX Winter 1991, January 1991.

[RSS90] Ramamritham, K. et al., *Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems*, IEEE Trans. on Parallel and Distributed Systems, April 1990.

[SR93] Stankovic J.A. and Ramamritham, K. (Editors), *Advances in Real-Time Systems*, IEEE Computer Press, 1993.

[Sta88] Stankovic J.A., *Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems*, IEEE Computer, October 1988.

[Sta91] Stallings W., *A Practical Guide to Queuing Analysis*, Byte Magazine, February 1991.

[Tab90] Tabak D., *Multiprocessors*, Prentice Hall, Englewood Cliffs N.J., 1990.

[Tan92] Tanenbaum, A., *Modern Operating Systems*, Prentice Hall, Englewood Cliffs N.J., 1992.

[Tev87] Tevanian A. et al., *Mach Threads and the Unix Kernel: The Battle for Control*, TR CMU-CS-87-149, Carnegie Mellon University, August 1987.

[Wil90] Williams T., *Real-time Operating Systems Struggle with Multiple Tasks*, Computer Design, October 1990.

[XP93] Xu J. and Parnas D.L., *On Satisfying Timing Constraints in Hard Real-Time Systems*, IEEE Trans. on Software Engineering, January 1993.

Appendix A

Runtime System API

The application programming interface (API) for the MAPE runtime system consists of three types of primitives. The first type includes the primitives that are invoked once during initialization. They relate to the definition of the ASN graph for each response and the invocation of the runtime system. The second type includes those that are invoked on multiple occasions but only during initialization. These primitives provide functionality for communicating the address for C functions and for condition variables. Finally, the third type of primitive includes those that may be invoked on many occasions during the execution of the application. They relate to the transfer of restricted data elements between activities and the manipulation of conditions.

A.1 define_asn

```
int define_asn(argument_count, argument_vector)
int argument_count;
char *argument_vector[];
```

The *define_asn* primitive is used to define an ASN to the RS and to specify the parameters of operation. The *argument_vector* is a pointer to an argument set. The argument set, in turn is structured as an array of pointers to the actual arguments. The *argument_count* specifies the total number of arguments contained in the set pointed to by *argument_vector* (size of the array). The argument set may contain pointers to any of the following arguments:

asn_file ..

The *asn_file* value must correspond to an existing file that contains a valid CSL definition for an ASN. Multiple files may be specified.

-t trace_level

The *trace_level* value consists of a bitmap that is used to control the trace output's level of detail. Only the three least significant bits of the value are considered.

000 - no trace output (default)
xx1 - activity control information
x1x - thread control information
1xx - deadline evaluation information

The trace facility precedes all output information with a timestamp.

-o optim_level

The *optim_level* value must be in the set (0, 1). It is used to specify the degree of optimization to be exercised over the ASN input graph. The goal of optimization is to speed up run-time interpretation of the graph.

0 - no optimization (default)
1 - collapse branch, merge, and wait nodes

-d deadline_level deadline_weight ..

The *deadline_level* must have a value in the set (0, 1, 2). It is used to control the level at which priority manipulation is performed.

0 - no manipulation (default)
1 - sub-response level manipulation
2 - activity level manipulation

At the sub-response level, the deadline and slack time are the supported criteria while at the activity level, the registered progress and anticipated progress are supported. The sub-response and activity levels may be used simultaneously.

The *deadline_weight* must have a value in the set (0 .. 100). It is used to control the relative importance of the individual criteria during priority assignment. In the case of sub-response level manipulation, the weight (PA_Weight) reflects the importance of the deadline criteria while in the case of activity level manipulation, the weight (PR_Weight) reflects the importance of the registered progress criteria.

-s

The argument controls the collection of statistics during the execution of the application and the generation of a summary report upon termination. An activity execution report consists of the following statistics:

Node : the name of the activity.

Priority : the priority at which the activity executed (0 is the lowest priority level supported).

Start-W : the wall clock time elapsed between the invocation of the application and the start of the activity.

End-W : the wall clock time elapsed between the invocation of the application and the end of the activity.

Elapsed-R : the virtual clock time registered by the activity during execution. Unlike Start-W and End-W, which are samples of a free running clock, Elapsd-R is the sum of all time periods spent executing on the system processor(s).

AE (as % of Elapsd-R) : the relationship between the estimated activity run time (AE) and the actual, recorded run time. A value greater than 100% corresponds to an overestimation of the actual run-time while a value less than 100% corresponds to an underestimation.

CRTE : the cumulative run time estimate computed relative to the start of the event. The value is based on both the ASN graph and the individual activity run time estimate (AE).

Elapsed-W (as % of CRTE) : the relationship between the elapsed wall clock time and the CRTE. A value greater than 100% corresponds to a condition where the activity completed after its alotted time. A value less than 100% corresponds to a condition where the activity completed before it was expected to.

Deadline : deadline specified in units of seconds relative to the starting event of the sub-response to which this activity belongs.

Slack : the time difference between the deadline and the estimated completion time for the sub-response (CRTE of final activity) in units of seconds.

PA_Value : the weighted value computed by the priority assignment algorithm and used to determine the base priority for the sub-response. The value integrates the deadline and slack time criterias when a *deadline_level* equal to 1 is specified for the *-d* argument.

Registered Progress : the difference in units of seconds between the time alotted for completion of the activity's predecessors and the time consumed for completion of the activity's predecessors.

Anticipated Progress : the difference in units of seconds between the time remaining for completion of the activity and its successors and the time needed for completion of the activity and its successors.

PR_Value : the weighted value computed by the priority refinement algorithm and used to determine priority increases for the activity. The value integrates the registered and anticipated progress criteria when a *deadline_level* equal to 2 is specified for the *-d* argument.

A.2 put_act_ptr

```
int put_act_ptr(CSL_activity_name, C_activity_pointer)
char *CSL_activity_name;
char *C_activity_pointer;
```

The *put_act_ptr* primitive is used to create a link between the CSL source file and the C source file for the activities. The *CSL_activity_name* is the string representation for the activity. The name must be identical to that used in the CSL source file. The *C_activity_pointer* is the pointer representation (starting address) for the C function associated with the activity. The *C_activity_pointer* must point to a valid C function in the C source file.

A.3 put_cond_ptr

```
int put_cond_ptr(CSL_condition_name, C_condition_pointer)
char *CSL_condition_name;
int *C_condition_pointer;
```

The *put_cond_ptr* primitive is used to create a link between the CSL source file and the C source file for the condition variables. The *CSL_condition_name* is the string representation for the condition. The name must be identical to that used in the CSL source file. The *C_condition_pointer* is the pointer representation (virtual address) for the condition variable associated with the condition. The *C_condition_pointer* must point to the variable name that is used and referenced in the C source file.

A.4 start_asn

```
int start_asn(argument_count, argument_vector)
int argument_count;
char *argument_vector[];
```

The *start_asn* primitive is used to delegate control to the RS upon completion of the application's initialization activity (in main). It is also used to communicate arguments to the response's initialization activities. The *argument_vector* is a pointer to an argument set. The argument set, in turn is structured as an array of pointers to the actual arguments. The *argument_count* specifies the total number of arguments contained in the set pointed to by *argument_vector*.

A.5 put_args

```
int put_args(C_activity_pointer, argument_count,
argument_vector)
char *C_activity_pointer;
int argument_count;
int *argument_vector[];
```

The *put_args* primitive is used to communicate restricted data from storage that is local to an activity to the shared memory space where it will be saved until it is required by the destination activity. The *C_activity_pointer* is the pointer representation (starting address) for the C function associated with the destination activity. The *argument_vector* is a pointer to an argument set which is in turn structured as an array of pointers to the actual arguments. The *argument_count* corresponds to the number of

arguments contained in the set. The put_args primitive uses the *argument_vector* to copy the argument pointers contained in the set, not the data to which they point.

A.6 get_args

```
argument_count get_args(C_activity_pointer,
argument_vector)
int argument_count;
char *C_activity_pointer;
int **argument_vector;
```

The *get_args* primitive is used to communicate restricted data from the shared memory space to storage that is local to an activity. The *C_activity_pointer* is the pointer representation (starting address) for the C function associated with the destination activity. The *argument_vector* is a pointer to a local argument set structured as an array of pointers. The *get_args* primitive copies the argument pointers contained in the set, not the data to which they point. The *argument_count* corresponds to the number of arguments copied into the set.

A.7 set_cond

```
int set_cond(C_condition_pointer, condition_value)
int *C_condition_pointer;
int condition_value;
```

The *set_cond* primitive is used to update the value of a condition variable. *The C_condition_pointer* is the pointer representation (virtual address) for the condition variable associated with the condition. The *C_condition_pointer* must point to the variable name that is used and referenced in the C source file. The *condition_value* is the new value to be placed at the address of the condition variable. The *set_cond* primitive also verifies if any activity triggers are suspended, waiting for the condition to be updated. If one exists, and the new *condition_value* satisfies the condition for release, the activity trigger will be activated and the activity will proceed with its operation.

Appendix B

Sample Source Code and Results

```
int Aa (argc, argv)
int argc;
char *argv[];
{
    asn_delay(2);
}
int Ab (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}
int Ac (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}
int Ad (argc, argv)
int argc;
char *argv[];
{
    asn_delay(8);
}
int Ae (argc, argv)
int argc;
char *argv[];
{
    asn_delay(5);
}
int Af (argc, argv)
int argc;
char *argv[];
{
    asn_delay(5);
}
main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);

    start_asn(argc, argv);
}
```

Figure B.1: C Source for Example 1

```
FUNCTIONAL
Aa    M1    INPUT     NONE
Aa    C1    ENABLE    NONE
M1    C1    INPUT     COUNT   3
M1    Ab    INPUT     NONE
M1    G2    ENABLE    NONE
M1    G1    DISABLE   NONE
Ab    Ac    INPUT     NONE
Ab    Ad    INPUT     NONE
Ab    Ae    INPUT     NONE
Ac    W1    INPUT     NONE
Ad    W1    INPUT     NONE
Ae    W1    INPUT     NONE
W1    G1    INPUT     NONE
W1    G2    INPUT     NONE
G1    Af    INPUT     NONE
G2    M1    INPUT     NONE
C1    G2    DISABLE   NONE
C1    G1    ENABLE    NONE
Af    E     INPUT     NONE
```

Figure B.2: CSL Source for Example 1

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.00	2.07	2.03
Ab	0	2.08	5.11	3.00
Ab	0	21.24	24.27	3.00
Ab	0	40.44	43.47	3.00
Ac	0	5.11	17.01	3.00
Ac	0	24.27	36.12	3.00
Ac	0	43.48	53.32	3.00
Ad	0	5.17	21.23	7.99
Ad	0	24.45	40.44	8.00
Ad	0	43.70	59.57	7.99
Ae	0	5.77	18.62	4.99
Ae	0	25.05	37.80	4.99
Ae	0	44.30	56.93	4.99
Af	0	59.58	64.61	5.00

Figure B.3: Results for Example 1 (1 processor)

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.01	2.01	2.00
Ab	0	2.02	5.05	3.02
Ab	0	13.17	16.19	3.01
Ab	0	26.01	29.03	3.02
Ac	0	5.06	8.09	3.02
Ac	0	16.20	20.32	3.00
Ac	0	29.12	34.89	3.02
Ad	0	5.05	13.08	8.03
Ad	0	16.20	26.01	8.02
Ad	0	29.04	38.97	8.01
Ae	0	5.05	13.16	5.00
Ae	0	16.79	22.53	5.02
Ae	0	29.04	35.60	5.03
Af	0	38.98	44.00	5.02

Figure B.4: Results for Example 1 (2 processors)

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.01	2.00	2.00
Ab	0	2.02	5.03	3.02
Ab	0	13.10	16.11	3.00
Ab	0	25.15	28.17	3.02
Ac	0	5.04	8.04	3.00
Ac	0	16.11	19.23	3.00
Ac	0	28.17	31.17	3.00
Ad	0	5.04	13.09	8.04
Ad	0	16.11	25.14	8.02
Ad	0	28.17	36.22	8.02
Ae	0	5.04	10.04	4.99
Ae	0	16.11	21.11	4.99
Ae	0	28.17	33.17	5.00
Af	0	36.23	41.26	5.02

Figure B.5: Results for Example 1 (3 processors)

```
int c1 = -1;
int c2 = 0;
int c3 = 0;
int c4 = 1;
int c5 = 0;
int c6 = 2;

int Aa (argc, argv)
int argc;
char *argv[];
{
    asn_delay(2);
}

int Ab (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}

int Ac (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}

int Ad (argc, argv)
int argc;
char *argv[];
{
    asn_delay(4);
    set_cond(&c2, 1);
    asn_delay(4);
}

int Ae (argc, argv)
int argc;
char *argv[];
{
    asn_delay(5);
}

int Af (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}

int Ag (argc, argv)
int argc;
char *argv[];
{
    asn_delay(4);
}

int Ah (argc, argv)
int argc;
char *argv[];
{
    asn_delay(1);
}

int Ai (argc, argv)
int argc;
char *argv[];
{
    asn_delay(4);
}

int Aj (argc, argv)
int argc;
char *argv[];
{
    asn_delay(6);
}

int Ak (argc, argv)
int argc;
char *argv[];
{
    asn_delay(1);
}

int Al (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}
```

Figure B.6: C Source for Example 2

```
int Am (argc, argv)
int argc;
char *argv[];
{
    asn_delay(4);
}

int An (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}

int Ao (argc, argv)
int argc;
char *argv[];
{
    asn_delay(2);
}

int Ap (argc, argv)
int argc;
char *argv[];
{
    asn_delay(10);
}

int Aq (argc, argv)
int argc;
char *argv[];
{
    asn_delay(7);
}

int Ar (argc, argv)
int argc;
char *argv[];
{
    asn_delay(1);
}

int As (argc, argv)
int argc;
char *argv[];
{
    asn_delay(5);
}

main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);
    put_act_ptr("Ag", Ag);
    put_act_ptr("Ah", Ah);
    put_act_ptr("Ai", Ai);
    put_act_ptr("Aj", Aj);
    put_act_ptr("Ak", Ak);
    put_act_ptr("Al", Al);
    put_act_ptr("Am", Am);
    put_act_ptr("An", An);
    put_act_ptr("Ao", Ao);
    put_act_ptr("Ap", Ap);
    put_act_ptr("Aq", Aq);
    put_act_ptr("Ar", Ar);
    put_act_ptr("As", As);

    put_cond_ptr("c1", &c1);
    put_cond_ptr("c2", &c2);
    put_cond_ptr("c3", &c3);
    put_cond_ptr("c4", &c4);
    put_cond_ptr("c5", &c5);
    put_cond_ptr("c6", &c6);

    start_asn(argc, argv);
}
```

Figure B.7: C Source for Example 2 (continued)

```
FUNCTIONAL
Aa    Ab    INPUT    NONE
Ab    B1    INPUT    NONE
Ab    B2    INPUT    NONE
Ab    B3    INPUT    NONE
Ab    Ac    INPUT    NONE
B1    Ad    INPUT    c1      WAIT
B2    Ae    INPUT    c2      WAIT
B3    Af    INPUT    c3      NOWAIT
B3    M1    INPUT    !c3     NOWAIT
Ac    B4    INPUT    NONE
Ad    Ah    INPUT    NONE
Ae    Ah    INPUT    NONE
Af    M1    INPUT    NONE
B4    Ag    INPUT    c2      WAIT
M1    Ah    INPUT    NONE
Ag    Ah    INPUT    NONE
Ah    Ai    INPUT    NONE
Ah    Aj    INPUT    NONE
Ah    Ak    INPUT    NONE
Ai    B5    INPUT    NONE
Aj    B6    INPUT    NONE
Ak    B7    INPUT    NONE
B5    Al    INPUT    c4      WAIT
B5    Am    INPUT    !c4     WAIT
Al    M2    INPUT    NONE
Am    M2    INPUT    NONE
M2    As    INPUT    NONE
B6    An    INPUT    c5      NOWAIT
B6    Ap    INPUT    !c5     NOWAIT
B6    Ao    INPUT    !c5     WAIT
Ao    W1    INPUT    NONE
An    M3    INPUT    NONE
M3    As    INPUT    NONE
Ap    W1    INPUT    NONE
W1    M3    INPUT    NONE
As    E     INPUT    NONE
B7    Aq    INPUT    c6      NOWAIT
B7    Ar    INPUT    !c6     NOWAIT
Aq    M4    INPUT    NONE
Ar    M4    INPUT    NONE
M4    E     INPUT    NONE
```

Figure B.8: CSL Source for Example 2

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.01	2.00	2.00
Ab	0	2.01	5.01	3.00
Ac	0	5.02	8.04	3.02
Ad	0	5.02	13.01	7.99
Ae	0	9.02	14.01	4.99
Ah	0	14.02	15.01	1.00
Ag	0	9.02	13.04	4.02
Ai	0	15.02	19.01	4.00
Aj	0	15.02	21.01	5.99
Ak	0	15.02	16.02	1.00
Al	0	19.02	22.02	3.00
As	0	31.02	36.01	5.00
Ap	0	21.02	31.01	9.99
Ao	0	21.02	23.02	2.00
Aq	0	16.02	23.07	7.05

Figure B.9: Results for Example 2 (4 processors)

```
int c1 = 1;
int loop_count = 2;

int Aa (argc, argv)
int argc;
char *argv[];
{
    asn_delay(2);
}
int Ab (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}
int Ac (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
}
int Ad (argc, argv)
int argc;
char *argv[];
{
    asn_delay(8);
}
int Ae (argc, argv)
int argc;
char *argv[];
{
    asn_delay(5);
}
```

```
int Af (argc, argv)
int argc;
char *argv[];
{
    asn_delay(3);
    loop_count--;
    if (loopcount = = 0)
    {
        loop_count = 2;
        set_cond(&c1, 0);
    } else
        set_cond(&c1, 1);
    fprintf(stdout, "c1 %d loop_count
    %d\n", c1, loop_count);
}
main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);

    put_cond_ptr("c1", &c1);

    start_asn(argc, argv);
}
```

Figure B.10: C Source for Example 3

```
FUNCTIONAL
Aa    M1    INPUT    NONE
M1    Ab    INPUT    NONE
Ab    M2    INPUT    NONE
Ab    Ac    INPUT    NONE
M2    Ad    INPUT    NONE
M2    Ae    INPUT    NONE
Ad    CT1   INPUT    NONE
Ae    CT1   INPUT    NONE
CT1   M2    INPUT    c2     STATIC 2
CT1   CT3   INPUT    !c2    NOWAIT
Ac    M3    INPUT    NONE
M3    Af    INPUT    NONE
Af    CT2   INPUT    NONE
CT2   M3    INPUT    c1     DYNAMIC
CT2   CT3   INPUT    !c1    NOWAIT
CT3   M1    INPUT    c3     STATIC 2
CT3   E     INPUT    !c3    NOWAIT
```

Figure B.11: CSL Source for Example 3

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.00	2.00	2.00
Ab	0	2.01	5.00	3.00
Ab	0	21.08	24.07	3.00
Ac	0	5.01	8.01	3.00
Ac	0	24.08	27.07	3.00
Ad	0	5.01	13.03	8.02
Ad	0	13.04	21.07	8.03
Ad	0	24.08	32.07	7.99
Ad	0	32.07	40.09	8.01
Ae	0	5.01	10.02	5.01
Ae	0	13.04	18.03	5.00
Ae	0	24.08	29.11	5.03
Ae	0	32.07	37.07	5.00
Af	0	8.01	11.01	3.00
Af	0	11.01	14.01	3.00
Af	0	27.08	30.07	3.00
Af	0	30.08	33.09	3.01

Figure B.12: Results for Example 3 (4 processors)

```
#define ACT_D 0
#define ACT_G 1
#define ACT_H 2
int integer_1 = -7;
int integer_2 = 7;
char string_1[] = {'T', 'e', 's', 't', '_', 'S', 't', 'r', 'i', 'n', 'g', '\0'};

int Aa (argc, argv)
int argc;
char *argv[];
{
    int i;

    fprintf(stdout, "Aa received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    asn_delay(2);
}
int Ab (argc, argv)
int argc;
char *argv[];
{
    int i;
    int int_args[3];

    fprintf(stdout, "Ab received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    int_args[0] = -5;
    int_args[1] = 0;
    int_args[2] = 5;
    put_args(Ad, 3, int_args);

    asn_delay(3);
}
```

Figure B.13: C Source for Example 4

```
int Ac (argc, argv)
int argc;
char *argv[];
{
    int i;
    int int_args[2];

    fprintf(stdout, "Ac received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    int_args[0] = (int) &integer_1;
    int_args[1] = (int) &integer_2;
    put_args(Ag, 2, int_args);

    int_args[0] = (int) &string_1;
    put_args(Ah, 1, int_args);

    asn_delay(3);
}
int Ad (argc, argv)
int argc;
char *argv[];
{
    int i;
    int int_args[4];

    fprintf(stdout, "Ad received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    int_args[0] = ACT_D;
    int_args[1] = (int) argv[1];
    int_args[2] = (int) argv[2];
    int_args[3] = (int) argv[3];
    put_args(Ai, 4, int_args);

    asn_delay(8);
}
int Ae (argc, argv)
int argc;
char *argv[];
{
    int i;

    fprintf(stdout, "Ae received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    asn_delay(5);
}
```

Figure B.15: C Source for Example 4 (continued)

```
int Af (argc, argv)
int argc;
char *argv[];
{
    int i;

    fprintf(stdout, "Af received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    asn_delay(3);
}
int Ag (argc, argv)
int argc;
char *argv[];
{
    int i;
    int int_args[3];

    fprintf(stdout, "Ag received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    int_args[0] = ACT_G;
    int_args[1] = (int) argv[1];
    int_args[2] = (int) argv[2];
    put_args(Ai, 3, int_args);

    asn_delay(4);
}
int Ah (argc, argv)
int argc;
char *argv[];
{
    int i;
    int int_args[2];

    fprintf(stdout, "Ah received %d arg(s), status %d : ", argc, argv[0]);
    for (i=1; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
    fprintf(stdout, "\n");

    int_args[0] = ACT_H;
    int_args[1] = (int) argv[1];
    put_args(Ai, 2, int_args);

    asn_delay(1);
}
```

Figure B.16: C Source for Example 4 (continued)

```
int Ai (argc, argv)
int argc;
char *argv[];
{
    int i, alt_argc, *alt_argv;

    fprintf(stdout, "Ai received %d arg(s), status %d : ", argc, argv[0]);
    switch ((int) argv[1]) {
        case ACT_D:
        fprintf(stdout, " from Ad : ");
        for (i=2; i<argc; i++) fprintf(stdout, "%d, ", argv[i]);
        break;
        case ACT_G:
        fprintf(stdout, " from Ag : ");
        for (i=2; i<argc; i++) fprintf(stdout, "%d, ", *((int *)argv[i]));
        break;
        case ACT_H:
        fprintf(stdout, " from Ah : ");
        for (i=2; i<argc; i++) fprintf(stdout, "%s, ", argv[i]);
        break;

        default:
        fprintf(stdout, " from unknown source : ");
        break;
    }
    fprintf(stdout, "\n");

    while ((alt_argc = get_args(Ai, &alt_argv)) != -1) {
        fprintf(stdout, "Ai received %d arg(s)", alt_argc);
        switch ((int)*alt_argv) {
            case ACT_D:
            fprintf(stdout, " from Ad : ");
            for (i=1; i<alt_argc; i++) fprintf(stdout, "%d, ", *(alt_argv+i));
            break;
            case ACT_G:
            fprintf(stdout, " from Ag : ");
            for (i=1; i<alt_argc; i++) fprintf(stdout, "%d, ", *((int *)*(alt_argv+i)));
            break;
            case ACT_H:
            fprintf(stdout, " from Ah : ");
            for (i=1; i<alt_argc; i++) fprintf(stdout, "%s, ", *(alt_argv+i));
            break;

            default:
            fprintf(stdout, " from unknown source : ");
            break;
        }
        fprintf(stdout, "\n");
    }

    asn_delay(4);
}
```

Figure B.17: C Source for Example 4 (continued)

```
main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);
    put_act_ptr("Ag", Ag);
    put_act_ptr("Ah", Ah);
    put_act_ptr("Ai", Ai);

    start_asn(argc, argv);
}
```

Figure B.18: C Source for Example 4 (continued)

```
FUNCTIONAL
Aa      Ac      INPUT       NONE
Ac      Af      INPUT       NONE
Ac      Ag      INPUT       NONE
Ac      Ah      INPUT       NONE
Af      W1      INPUT       NONE
Ag      W1      INPUT       NONE
Ah      W1      INPUT       NONE
W1      Ai      INPUT       NONE
W1      E       INPUT       NONE

FUNCTIONAL
Ab      Ad      INPUT       NONE
Ab      Ae      INPUT       NONE
Ad      Ai      INPUT       NONE
Ae      Ai      INPUT       NONE
Ai      E       INPUT       NONE
```

Figure B.19: CSL Source for Example 4

Node	Prio Used	Start (W)	End (W)	Elapsed (R)
Aa	0	0.00	2.01	2.00
Ac	0	2.02	5.02	3.00
Af	0	5.02	8.02	3.00
Ag	0	5.02	9.03	4.01
Ah	0	5.02	6.03	1.00
Ai	0	11.01	15.01	4.00
Ab	0	0.00	3.01	3.00
Ad	0	3.01	11.01	7.99
Ae	0	3.01	8.02	5.00

Figure B.20: Results for Example 4 (5 processors)

```
int c1 = 1;
int c2 = -1;

int Aa (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Aa received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(2);
}

int Ab (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ab received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ac (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ac received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ad (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ad received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(8);
}

int Ae (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ae received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(5);
}
```

Figure B.21: C Source for Example 5

```
int Af (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Af received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ag (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ag received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(4);
}

int Ah (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ah received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(1);
}

int Ai (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ai received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(4);
}

int Aj (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Aj received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(6);
}

int Ak (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ak received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(1);
}
```

Figure B.21: C Source for Example 5 (continued)

```
int Al (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Al received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Am (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Am received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(4);
}

int An (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "An received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);
    put_act_ptr("Ag", Ag);
    put_act_ptr("Ah", Ah);
    put_act_ptr("Ai", Ai);
    put_act_ptr("Aj", Aj);
    put_act_ptr("Ak", Ak);
    put_act_ptr("Al", Al);
    put_act_ptr("Am", Am);
    put_act_ptr("An", An);

    put_cond_ptr("c1", &c1);
    put_cond_ptr("c2", &c2);

    start_asn(argc, argv);
}
```

Figure B.21: C Source for Example 5 (continued)

```
FUNCTIONAL
Aa      Ab      INPUT       NONE
Ah      Ai      INPUT       NONE
Ah      Aj      INPUT       NONE
Ah      E       INPUT       NONE
Ag      E       INPUT       NONE
```

a) asn-ddlcrse

```
FUNCTIONAL
Ab      Ac      INPUT       NONE
Ab      B1      INPUT       NONE
Ac      Ad      INPUT       NONE
Ac      Ae      INPUT       NONE
Ad      W1      INPUT       NONE
Ae      W1      INPUT       NONE
W1      Ag      INPUT       NONE
B1      Af      INPUT       c1          WAIT
B1      M1      INPUT       !c1         WAIT
Af      M1      INPUT       NONE
M1      Ag      INPUT       NONE
TEMPORAL 18.10
Ab      3.3
Ac      3.3
Ad      8.8
Ae      5.5
Af      3.3
Ag      4.4
```

b) asn-ddlcrse-1

```
FUNCTIONAL
Ai      W2      INPUT       NONE
Aj      W2      INPUT       NONE
W2      B2      INPUT       NONE
B2      Ak      INPUT       c2          NOWAIT
B2      Al      INPUT       c2          NOWAIT
B2      Am      INPUT       !c2         NOWAIT
Ak      W3      INPUT       NONE
Al      W3      INPUT       NONE
W3      M2      INPUT       NONE
Am      M2      INPUT       NONE
M2      An      INPUT       NONE
TEMPORAL 15.68
Ai      4.4
Aj      6.6
Ak      1.1
Al      3.3
Am      4.4
An      3.3
```

c) asn-ddlcrse-2

Figure B.22: CSL Source for Example 5

Node	Prio Used	Start (W)	End (W)	Elapsed (R)	CRTE	Dead-line	Slack	PA_ Weight
Aa	0	0.00	2.00	2.03	NONE	NONE	NONE	NONE
Ab	10	2.00	5.00	3.00	3.30	18.10	-1.70	-1.70
Ah	0	0.00	1.05	1.04	NONE	NONE	NONE	NONE
Ai	5	1.05	5.05	4.00	4.40	15.68	1.38	1.38
Aj	5	1.05	7.10	6.05	6.60	15.68	1.38	1.38
An	5	10.71	13.71	3.00	14.30	NONE	NONE	NONE
Ag	10	16.01	20.00	4.00	19.80	NONE	NONE	NONE
Ac	10	5.00	8.00	3.00	6.60	NONE	NONE	NONE
Ad	10	8.00	16.00	8.00	15.40	NONE	NONE	NONE
Ae	10	8.00	13.00	4.99	12.10	NONE	NONE	NONE
Af	10	5.00	8.00	3.00	6.60	NONE	NONE	NONE
Ak	5	7.11	8.11	1.00	7.70	NONE	NONE	NONE
Al	5	7.70	10.70	3.00	9.90	NONE	NONE	NONE

Figure B.23: Results for Example 5 (4 processors - test case 1)

Node	Prio Used	Start (W)	End (W)	Elapsed (R)	CRTE	Dead-line	Slack	PA_ Weight
Aa	0	0.00	2.00	2.00	NONE	NONE	NONE	NONE
Ab	5	2.01	5.03	3.02	3.30	18.10	-1.70	18.10
Ah	0	0.00	1.05	1.04	NONE	NONE	NONE	NONE
Ai	10	1.05	5.05	4.00	4.40	15.68	1.38	15.68
Aj	10	1.05	7.07	6.01	6.60	15.68	1.38	15.68
An	10	10.07	13.07	3.00	14.30	NONE	NONE	NONE
Ag	5	16.85	20.84	4.00	19.80	NONE	NONE	NONE
Ac	5	5.03	8.03	3.00	6.60	NONE	NONE	NONE
Ad	5	8.85	16.84	7.99	15.40	NONE	NONE	NONE
Ae	5	8.03	13.06	5.03	12.10	NONE	NONE	NONE
Af	5	5.04	8.06	3.02	6.60	NONE	NONE	NONE
Ak	10	7.07	8.07	1.00	7.70	NONE	NONE	NONE
Al	10	7.07	10.07	3.00	9.90	NONE	NONE	NONE

Figure B.24: Results for Example 5 (4 processors - test case 2)

```
int c1 = 1;

int Aa (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Aa received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(2);
}

int Ab (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ab received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ac (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ac received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ad (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ad received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(8);
}

int Ae (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ae received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(5);
}
```

Figure B.25: C Source for Example 6

```
int Af (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Af received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int Ag (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ag received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(4);
}

int Ah (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ah received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(1);
}

int Ai (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ai received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(4);
}

int Aj (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Aj received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(6);
}

int Ak (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Ak received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(1);
}
```

Figure B.25: C Source for Example 6 (continued)

```
int Al (argc, argv)
int argc;
char *argv[];
{
    fprintf(stdout, "Al received %d arg(s), status %d : ", argc, argv[0]);

    asn_delay(3);
}

int main (argc, argv)
int argc;
char *argv[];
{
    define_asn(argc, argv);

    put_act_ptr("Aa", Aa);
    put_act_ptr("Ab", Ab);
    put_act_ptr("Ac", Ac);
    put_act_ptr("Ad", Ad);
    put_act_ptr("Ae", Ae);
    put_act_ptr("Af", Af);
    put_act_ptr("Ag", Ag);
    put_act_ptr("Ah", Ah);
    put_act_ptr("Ai", Ai);
    put_act_ptr("Aj", Aj);
    put_act_ptr("Ak", Ak);
    put_act_ptr("Al", Al);

    put_cond_ptr("c1", &c1);

    start_asn(argc, argv);
}
```

Figure B.25: C Source for Example 6 (continued)

```
FUNCTIONAL
Aa      Ab      INPUT       NONE
Al      E       INPUT       NONE
```

a) asn-ddlfinc

```
FUNCTIONAL
Ab      Ac      INPUT       NONE
Ab      B1      INPUT       NONE
Ac      Ad      INPUT       NONE
Ac      Ac      INPUT       NONE
Ad      W1      INPUT       NONE
Ac      W1      INPUT       NONE
W1      Ag      INPUT       NONE
B1      Af      INPUT       c1          WAIT
B1      M1      INPUT       !c1         WAIT
Af      M1      INPUT       NONE
M1      Ag      INPUT       NONE
Ag      Ah      INPUT       NONE
Ag      Ai      INPUT       NONE
Ag      Aj      INPUT       NONE
Ag      Ak      INPUT       NONE
Ah      Al      INPUT       NONE
Ai      Al      INPUT       NONE
Aj      Al      INPUT       NONE
Ak      Al      INPUT       NONE
TEMPORAL 30.00
Ab      3.3
Ac      3.3
Ad      8.8
Ae      5.5
Af      3.3
Ag      4.4
Ah      1.1
Ai      4.4
Aj      6.6
Ak      1.1
Al      3.3
```

b) asn-ddlfinc-1

Figure B.26: CSL Source for Example 6

Node	Prio Used	Start (W)	End (W)	Elapsed (R)	CRTE	Reg. Progress	Ant. Progress	PR_ Weight
Aa	0	0.00	2.00	2.00	NONE	NONE	NONE	NONE
Ab	0	2.01	5.02	3.01	3.30	0.00	0.30	0.30
Al	2	28.78	31.80	3.02	29.70	-0.37	-0.07	-0.07
Ac	1	5.70	8.70	3.00	6.60	-0.40	-0.10	-0.10
Ad	1	8.71	16.76	8.00	15.40	-0.10	0.20	0.20
Ae	1	8.71	13.73	5.02	12.10	-0.10	0.20	0.20
Ag	1	16.76	20.76	4.00	19.80	0.65	0.95	0.95
Af	0	5.02	8.04	3.01	6.60	0.28	0.58	0.58
Ah	1	20.76	24.39	1.00	20.90	1.05	1.35	1.35
Ai	1	20.76	24.82	4.03	24.20	1.04	1.34	1.34
Aj	1	20.76	28.78	6.01	26.40	1.04	1.34	1.34
Ak	1	20.76	24.76	1.00	20.90	1.04	1.34	1.34

Figure B.27: Results for Example 6 (2 processors - test case 1)

Node	Prio Used	Start (W)	End (W)	Elapsed (R)	CRTE	Reg. Progress	Ant. Progress	PR_ Weight
Aa	0	0.00	2.00	2.00	NONE	NONE	NONE	NONE
Ab	0	2.01	5.01	3.00	3.30	0.00	0.30	0.00
Al	1	28.27	31.28	3.01	29.70	0.14	0.44	0.14
Ac	0	5.61	8.62	3.01	6.60	0.30	0.60	0.30
Ad	1	8.62	16.65	8.03	15.40	-0.02	0.28	-0.02
Ae	1	8.63	13.65	4.99	12.10	-0.02	0.28	-0.02
Ag	1	16.66	20.68	4.02	19.80	0.75	1.05	0.75
Af	0	5.01	8.02	3.01	6.60	0.30	0.60	0.30
Ah	1	20.68	21.68	1.00	20.90	1.13	1.43	1.13
Ai	1	20.68	25.69	4.00	24.20	1.13	1.43	1.13
Aj	1	20.68	28.27	6.02	26.40	1.13	1.43	1.13
Ak	1	20.69	22.69	1.00	20.90	1.12	1.42	1.12

Figure B.28: Results for Example 6 (2 processors - test case 2)