Distributed real-time operating system (DRTOS) modeling in SpecC

by

Ziyu Zhang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Diane T. Rover (Major Professor)
Gurpur M. Prabhu
Zhao Zhang

Iowa State University

Ames, Iowa

2006

UMI Number: 1439846

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

First and foremost, I would like to take this opportunity to express my thanks to my major advisor, Dr. Diane T. Rover, for her invaluable instruction during the whole work with this thesis. It is a great honor for me to work with her. She is always there to listen and give advice, encourage and guide me to a deeper and broader understanding of the work, and show me the need to be persistent to accomplish any goal.

I would also like to thank my committee members: Dr. Zhao Zhang and Dr. Gurpur M. Prabhu, for their efforts and contributions for this work. Thank you for all your good comments and questions to help me think through my work.

I want to say 'thank you' to my friends in our embedded system group. I am fortunate to have the opportunity to meet them and I enjoy every moment that we have worked together. I want especially to thank Joseph Schneider, Mikel Bezdek, Ramon Mercado, Daniel Helvick, and Andrew Larson for sharing their technical wisdom, research ideas, and many other things about life. I appreciate all their friendships and encouragement.

A special thanks gives to my family. My parents' encouragement and unconditional support are always with me and give me endless strength. My wife Shu's love help me go through all the though time and become a better person.

To all of you, thank you.

ABSTRACT

System level design of an embedded computing system involves a multi-step process to refine the system from an abstract specification to an actual implementation by defining and modeling the system at various levels of abstraction. System level design supports evaluating and optimizing the system early in design exploration.

Embedded computing systems may consist of multiple processing elements, memories, I/O devices, sensors, and actors. The selection of processing elements includes instruction-set processors and custom hardware units, such as application specific integrated circuit (ASIC) and field programmable gate array (FPGA). Real-time operating systems (RTOS) have been used in embedded systems as an industry standard for years and can offer embedded systems the characteristics such as concurrency and time constraints. Some of the existing system level design languages, such as SpecC, provide the capability to model an embedded system including an RTOS for a single processor. However, there is a need to develop a distributed RTOS modeling mechanism as part of the system level design methodology due to the increasing number of processing elements in systems and to embedded platforms having multiple processors. A distributed RTOS (DRTOS) provides services such as multiprocessor tasks scheduling, interprocess communication, synchronization, and distributed mutual exclusion, etc.

In this thesis, we develop a DRTOS model as the extension of the existing SpecC single RTOS model to provide basic functionalities of a DRTOS implementation, and present the refinement methodology for using our DRTOS model during system level synthesis. The DRTOS model and refinement process are demonstrated in the SpecC SCE environment. The capabilities and limitations of the DRTOS modeling approach are presented.

1. INTRODUCTION

1.1 Motivation

Today, embedded system products are widely used in our everyday lives. Embedded systems can be found in aviation, marine, and automotive navigation devices. Those devices require high accuracy, high quality, high reliability, and high stability. Consumer handheld devices represent another type of product relying on embedded systems that demand multi-functionality, easy of use, energy efficiency, and low-cost.

In general, an embedded system consists of a variety of different processing elements, storage units, I/O devices, sensors, actors, etc. The selection of processing elements includes instruction-set processors, application-specific integrated circuits (ASICs), and reconfigurable hardware devices such as field-programmable gate arrays (FPGAs). Advances in processor technology and architecture have led to a near-exponential increase in processing element speed making embedded systems more and more powerful. However, the complexity of embedded systems has increased more rapidly than the performance of individual processing elements. Today's applications often require considerably more computational power than a single processor can offer. Therefore, utilizing parallel or distributed systems architecture combined with multiple processing elements has often become necessary. Additionally, distributed operating systems are becoming an important software component in embedded systems used to provide the high-level procedures for dynamically managing the tasks and resources in a multiprocessor environment. The use of multiple processing elements and distributed operating systems in embedded systems permits the execution of application tasks in a true multi-tasking manner, which can dramatically improve the performance of the embedded systems.

Unfortunately, these advancements also significantly increase the complexity of system architecture and design for embedded systems. In order to handle the extraordinary competition present in the real industrial world today, designers are driven

to shorten their development cycle and place their products on the market as quickly as possible. Thus, a new design method is needed to assist designers in producing complex multiprocessor-based embedded systems more quickly and with less cost.

1.2 Statement of the Problem

In multiprocessor embedded systems design, system functionality and timing are the two main aspects of system constraint. Many design decisions such as tasks and resources allocation, selection of task scheduling algorithms, memory requirements, etc, may affect these constraints. Overall analysis of such critical system level effects is a major challenge. In order to study these effects before any implementation has been done, designers need a system-level model capable of capturing system runtime behavior on a multiprocessor platform.

System level design is a multi-stage process for refining a system from an abstract specification to an actual implementation. In recent years, system level design languages and tools have been introduced, allowing designers to define and model systems at various levels of abstraction [1]. The main goal of system level design languages (SLDL) is to help designers in managing high complexity of embedded systems during early design exploration.

At their earlier implementations, system level design languages lacked support for modeling real-time operating systems (RTOSs). After considerable research work focused on this area, RTOS modeling is being increasingly supported in system level design. In [2], Gerstlauer, et al, introduced their RTOS model built on top of the existing SLDL – SpecC, which supports all of the key concepts in the RTOS kernels, such as task management, preemption, task synchronization, and interrupt handling. In [3], a transaction-level model (TLM) with RTOS scheduling support is developed, which allows designers to select the correct scheduling algorithm at the higher levels of abstraction so that the system performance can be improved. However, these models currently only support modeling for a single RTOS kernel and cannot be used in

multiprocessor systems.

1.3 Objective of the Research

In order to fully facilitate system level design approaches in embedded systems with multiple processing elements, a new system level OS model must be developed to capture distributed real-time operating system (DRTOS) runtime behavior. This thesis addresses this design issue by introducing a highly abstract DRTOS model into system level design. Our DRTOS model is the extension of the RTOS model introduced in [2] including a global synchronizer and a global scheduler. We named it as ERTOS-SS, for extended RTOS with global scheduling and synchronization. The ERTOS-SS DRTOS is written on top of the SpecC language and can be integrated into existing system level design flows to accurately reflect distributed real-time operating system behavior during system level synthesis.

1.4 Contribution

The following is a summary of the contributions of this research project.

- Analysis and simulation of the existing SpecC RTOS model.
- Demonstration of the OS refinement process with insertion of SpecC RTOS model into a system model using the SCE environment.
- Modeling and refinement methodology of the abstract ERTOS-SS DRTOS model.
- The ERTOS-SS DRTOS model implementation of an example multitask multiprocessor system as a case study.

1.5 Thesis Outline

The thesis is organized into 7 chapters. The introduction in chapter 1 provides motivation and the statement of problem. The objectives and the contributions of this

research work are also described in this chapter.

Chapter 2 provides background materials of operating systems, including the main components and features of real-time operating systems and distributed real-time operating systems. This chapter also presents an briefly overview of system level design languages, followed by a description of the detailed features of the SpecC language, the system level design language used in this research work.

Chapter 3 first introduces the SCE system level design environment used in this research work. Then a brief analysis of the existing SpecC RTOS model and a demonstration of the OS refinement process in the SCE environment are presented.

In Chapter 4, the ERTOS-SS DRTOS modeling and refinement methodology is presented. It starts with a summary of the DRTOS services which must be implemented in the ERTOS-SS DRTOS model. Then the implementation details of the ERTOS-SS DRTOS model and the ERTOS-SS DRTOS refinement methodology are discussed.

A case study of using the abstract ERTOS-SS DRTOS model is demonstrated in Chapter 5. The simulation results of experiments are also shown in this chapter.

Chapter 6 summarizes related work on system level design methodologies for multiprocessor embedded systems or Multiprocessor System-on-Chip (MPSoC).

Finally, a conclusion and some recommended topics for future work are presented in Chapter 7.

2. BACKGROUND

In this chapter, an introduction of operating systems is given in section 2.1.1, followed by the detailed discussion on real-time operating systems and distributed real-time operating systems in section 2.1.2 and 2.1.3, respectively. In section 2.2.1, a brief overview of system level design languages is presented. Then one of the most prominent system level design languages, SpecC [4], developed at the University of California, Irvine, is introduced in section 2.2.2.

2.1 Operating Systems

2.1.1 Overview

Operating systems provide a layer of abstraction between the user and the bare machine. Users and applications do not see the hardware directly, but view it through operating systems. There are many types of operating systems, and their complexity varies depending upon what types of functions are provided, and for what the system is being used. There is no universal definition of what an operating system consists of. Normally operating systems can provide the following two basic functions.

Perform Resource Management

This includes:

- Time management (CPU and disk scheduling)
- Space management (main and secondary storages)
- Process synchronization and deadlock handling
- Accounting and status information

Provide User Friendliness

This includes:

- Execution environment
- Error detection and handling

- Protection and security
- Fault tolerance and failure recovery

The categories of operating systems are also complex. Depending on the number of tasks that can be performed simultaneously and the number of simultaneous users that can be supported, operating systems can be categorized as single-user single-task, single-user multi-task, or multi-user multi-task. Table 2.1 below shows some examples.

Table 2.1: Examples of different numbers of users and tasks

OS	Users	Tasks	Processors
MS DOS	S	S	1
Windows 3x	S	S	1
Amiga DOS	S	M	1
Windows 9x	S	M	1
hline MTS	M	M	1
Windows NT/2000/XP	M	M	N
UNIX	M	M	N
VMS	M	M	N

Depending on the various design approaches, operating systems styles have been classified in different catalogs as: the monolithic approach, the layered approach, the kernel-based approach, and the virtual machine approach. Table 2.2 below lists some examples of these different approaches.

Table 2.2: Examples of different design approaches

Design Approach	OS
Monolithic Approach	MS-DOS, MVS
Layered Approach	THE, MULTICS
Kernel-based Approach	Linux, Unix, Windows 2000
Virtual Machine Approach	IBM VM/370

Early operating systems designers focused on the stand-alone computer with a single processor. After decades of development in computer architecture and increasing complexities in computer applications, advanced operating systems have recently

become a mainstream technology. Generally, advanced operating systems can be classified as architecture driven, application driven, and hybrids of these two approaches. Several examples of advanced operating systems are shown in Figure 2.1.

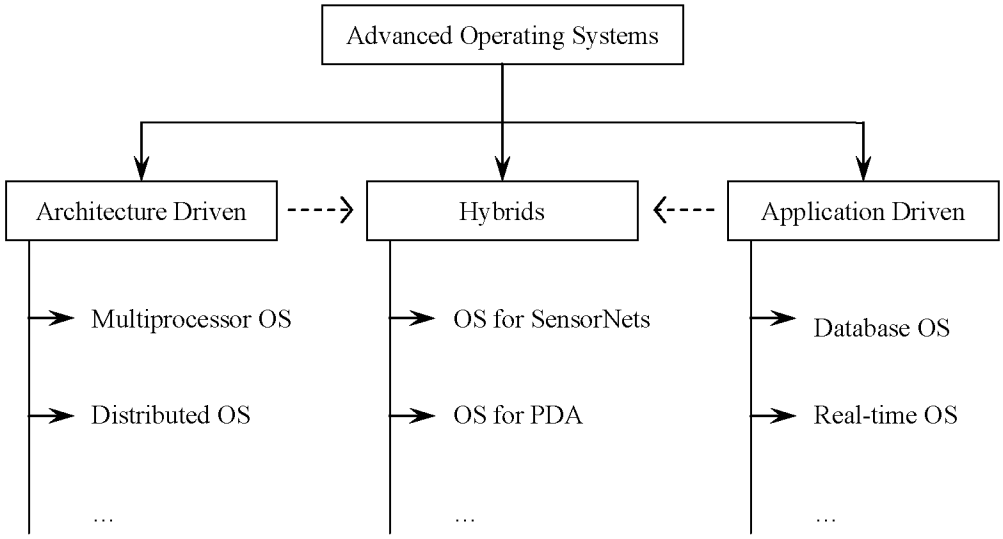


Figure 2.1: Examples of advanced operating systems

In the following two sections (2.1.2 and 2.1.3), more detailed features of real-time operating systems and distributed real-time operating systems will be presented, respectively.

2.1.2 Real-Time Operating Systems

Real-time operating systems (RTOS), such as VxWorks, pSoS, and QNX, are the operating systems used in embedded real-time systems. In order to fully understand the features of real-time operating systems, a brief overview of real-time systems will first be presented.

Real-time systems are computing systems that have both logic and timing constraints. In an ordinary system, the value of the output typically determines the correctness of the system. But in a real-time system, time issues must be considered as

well. The principal responsibility of real-time systems can be summarized as that of producing correct results within a certain time interval – a deadline. A correct output produced too late or too early could often be useless or even dangerous.

Depending on types of deadlines, a real-time system can be defined as a hard real-time system or a soft real-time system. As described in [5], hard real-time systems can be thought of as a particular subclass of real-time systems in which the lack of adherence to deadlines may result in catastrophic system failure. Examples of hard real-time systems include avionic control systems, vehicle control systems, and industrial automation systems, etc. On the other hand, soft real-time systems are those real-time systems in which the ability to meet deadlines is a high-priority requirement, but failure to do so does not necessarily cause system failure. Multimedia processing systems and internet web servers are examples of soft real-time systems. Figure 2.2 illustrates the difference between hard real-time systems and soft real-time systems.

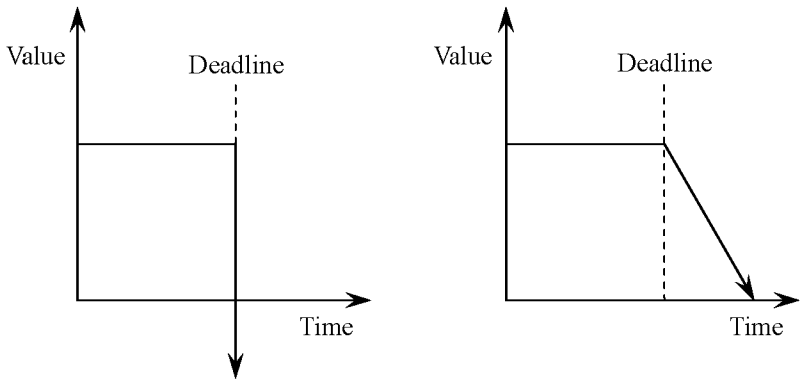


Figure 2.2: Hard real-time system and soft real-time system

The main objective of real-time operating systems is to simplify the development process of real-time systems by providing a set of interfaces with a higher abstraction level than that offered by the bare hardware architecture. Modern real-time operating systems are based on the complementary concepts of multitasking and interprocess

communications. A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources. Interprocess communication facilities allow these tasks to synchronize and communicate to coordinate their activities [6]. This section presents an overview of the main components and features used in real-time operating systems.

Tasks

A real-time application performs a set of pre-defined actions within a certain time frame. Each of these actions is typically defined as a task. A real-time task can be classified as periodic or aperiodic depending on its arrival pattern. Tasks with regular arrival times are called periodic and tasks with irregular arrival times are called aperiodic. Each task has its own context, which contains the CPU environment and system resources that the task sees each time it is scheduled to run. In a real-time operating system, the context of a task is stored in a data structure, called the task control block (TCB). In a preemptive scheduling algorithm, the context of each task is stored into, or reloaded from, the task TCB during context switching.

Real-time operating systems maintain the current state of each task in the system. The states may have different names in different operating systems, and some additional states may exist in some operating systems. Generally, based on [6], a task at any point of time can be in one of the following states: running, ready, waiting, and suspend. When first created, tasks enter the suspend state. Activation is required for a created task to enter its ready state. The ready state indicates that a task is not waiting for any resources other than the CPU. Depending on different scheduling algorithms, one task in the ready state can be executed and enter its running state. Only one task per processor can be in the running state at any instant of time: it is the task currently using the processor. If a task is blocked due to the unavailability of resources other than the CPU, it is placed in the waiting state. Figure 2.3 shows the task state transition in real-time operating systems.

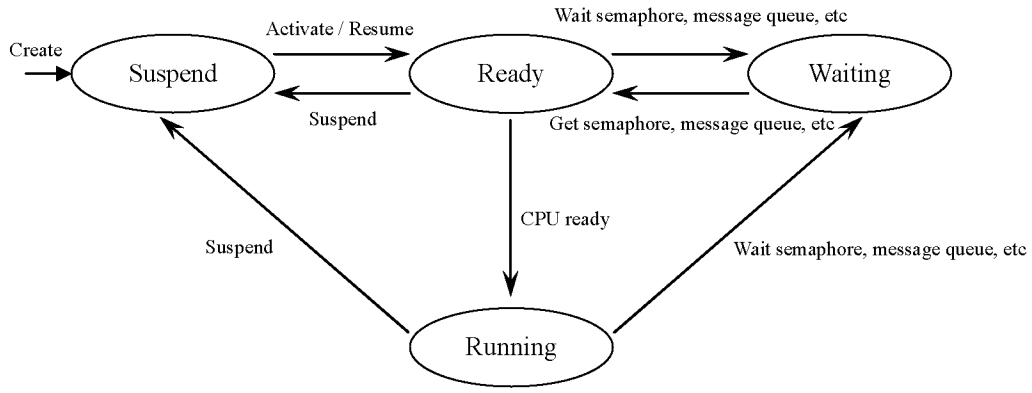


Figure 2.3: Task state transition

Scheduling

Multitasking in real-time operating systems requires a scheduling algorithm to allocate the CPU to ready tasks. The main purpose of scheduling real-time tasks is to guarantee satisfaction of temporal constraints such as deadline, release time, etc. There are two basic types of scheduling mechanisms for real-time tasks: non-preemptive scheduling and preemptive scheduling [7]. In non-preemptive scheduling, once a task has started executing, it completes its execution without interruption. The main advantage of non-preemptive scheduling is that it has less scheduling overhead because of less occurrence of context switching. However, it offers lower schedulability. Conversely, in preemptive scheduling, a task's execution can be preempted by higher priority tasks. At any point of time, the highest priority ready task is executing. Once all the higher priority ready tasks finish their execution, the preempted task can be resumed. Compared to non-preemptive scheduling, preemptive scheduling can provide a higher degree of schedulability by meeting deadlines of higher priority tasks first. The disadvantage of preemptive scheduling is that it requires higher scheduling overhead due to a greater level of context switching.

Interprocess Communication

Communication is a central component in any operating system. Co-operating

tasks (processes or threads) often communicate and synchronize. The execution of one particular task can affect another task or tasks by communication. As listed below, there are generally several types of interprocess communication (IPC) in real-time operating systems:

i. Shared Data Structure – The most obvious way for tasks to communicate is by accessing shared data structures. The instances of shared data structures include global variables, linear buffers, ring buffers, linked lists, and pointers, etc. Interprocess communication using a shared data structure is shown in Figure 2.4.

ii. Message Queue – Message queues allow a variable number of messages, each of variable length, to be queued. Tasks and interrupt service routines (ISR) can send messages to a message queue, and receive messages from a message queue. Interprocess communication using a message queue is shown in Figure 2.5.

iii. Signal – Signals are more appropriate for error and exception handling than as a general-purpose interprocess communication mechanism. Any task or ISR can raise a signal for a particular task. The task being signaled immediately suspends its current thread of execution and executes the task-specified signal handler routine the next time it is scheduled to run. The signal handler is invoked even if the task is blocked.

iv. Others – Some other communication mechanisms may exist in different real-time operating systems. *Pipes*, which are virtual I/O devices, provide an alternative interface to the message queue facility that goes through the I/O system. *Socket* is a basic network interprocess communication mechanism in which data is send from one socket to another across the network. *Remote procedure calls (RPC)* is a facility that allows a process on one machine to call a procedure that is executed by another process on either the same machine or a remote machine.

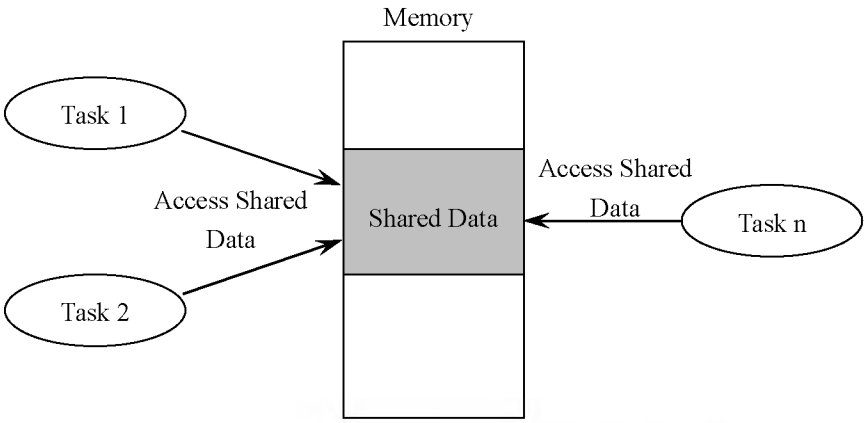


Figure 2.4: Interprocess communication via a shared data structure

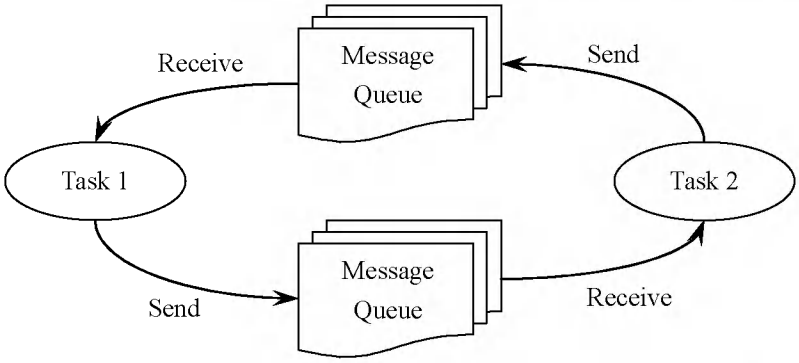


Figure 2.5: Interprocess communication via message queues

Another use of interprocess communication mechanisms is for shared resource mutual exclusion and task synchronization. A shared resource is a software structure that can be used by more than one task to advance its execution. Any operating system that supports shared resources must guarantee mutual exclusion among competing tasks. *Semaphores* are the primary method for addressing the requirements of both mutual exclusion and task synchronization. Semaphores provide mutual exclusion by interlocking access to shared resources. For synchronization, semaphores coordinate a task's execution with external events.

Interrupt

Another key facility in real-time operating systems is interrupt handling.

Interrupts are the usual mechanism for informing a system of critical external events. For the fastest possible response to interrupts, many real-time operating systems use a special context for *Interrupt service routine (ISR)* outside of any other task's context. Thus interrupt handling involves no task context switch and ISR's can be executed immediately when interrupts occur.

In this section, we first introduced real-time systems and then discussed the main components and features in real-time operating systems. As we have seen, real-time operating systems can provide efficient mechanisms and services for real-time scheduling and resource management. They are important components in embedded real-time systems design.

2.1.3 Distributed Real-Time Operating Systems

The demand for distributed or parallel hardware architectures in embedded real-time systems is due mainly to the fact that applications sometimes require more computational power than a single processor can offer [8]. Similar to real-time operating system (RTOS) kernels used for years as an industry standard in uniprocessor systems, distributed real-time operating systems can be used in a multiprocessor environment to provide high-level procedures to dynamically schedule tasks and manage resources at run-time.

Compared to conventional real-time operating systems for single processor systems, several central issues that describe almost all additional constructs in a distributed real-time operating system are presented in the following part of this section.

Multiprocessor Scheduling

The central problem in multiprocessor scheduling is to determine when and on which processor a given task is to be executed. This can be done either statically or dynamically. Static algorithms determine *a priori* the assignment of tasks to processors and the time at which each task starts execution [9]. The main advantage of static

scheduling is that all tasks' deadlines will be guaranteed if a feasible schedule is found. In dynamic scheduling, on the other hand, when a new task arrives, the scheduler dynamically determines the feasibility of scheduling the task without jeopardizing the guarantees that have been provided for previously scheduled tasks [9]. Compared to static scheduling, dynamic scheduling offers higher flexibility and is better at adapting to runtime changes such as aperiodic tasks.

As described in [8], a multiprocessor scheduling procedure in distributed real-time operating systems can be described as having three phases:

i. Allocation – the assignment of tasks and resources to appropriate nodes or processors in the system.
ii. Scheduling – ordering the execution of tasks and network communication such that timing constraints are met and consistency of recourses is maintained.
iii. Dispatching – executing the tasks in conformance with the scheduler's decisions.

There are many multiprocessor scheduling algorithms for both static and dynamic scheduling. More details about these algorithms can be found in [8] and [9].

Memory Management in Multiprocessor Systems

Three primary types of memory systems are most commonly used in multiprocessor systems:

i. Uniform Memory Access (UMA) – In UMA architectures, memory access times to the whole address space are equal for all processes. A common design technique for such systems is one in which all processors are connected to a bus, with a global shared memory connected to the same bus.
ii. Non-Uniform Memory Access (NUMA) –NUMA systems also offer a single shared address space visible to all processors, but access times to the different memory regions differs for each processor. A common design technique in this type of system is to use processor boards with on-board memory modules

attached to a shared bus.

iii. No Remote Access System (NORMA) – NORMA architectures do not offer a global shared address space. Each processor accesses only its own address space. Typically, these architectures, often referred to as clusters, consist of loosely-coupled independent computers connected through Local Area network (LAN) technologies.

Memory management of real-time operating system kernels is usually simple and primitive. Typical conventional (not real-time) operating systems mechanisms such as virtual memory, dynamic allocation and de-allocation, are avoided in real-time operating systems, since these are considered to be dangerous and unreliable features. In NORMA architecture, each processing element manages its own local memory as a single processor system. In UMA and NUMA systems, distributed real-time operating systems are responsible for providing management for the shared memory that is visible to all processors.

Interprocess Communication and Synchronization

Generally there are two types of interprocess communication (IPC) in distributed RTOS:

i. Message Passing – different processors communicate with each other by sending/receiving messages. This method is typically associated with distributed memory multiprocessors or distributed multicomputers (NORMA), such as in a network of workstations.

ii. Shared Memory – different processors communicate with each other by reading/writing data from/to shared memories. This method is typically associated with tightly-coupled shared memory multiprocessors (UMA, NUMA).

Distributed RTOS interprocess communication mechanisms are similar to those used in uniprocessor real-time operating systems in that shared data structures and message queues are used. The main difference is that in uniprocessor real-time operating

systems, shared data structures and message queues are only visible for multiple tasks located in a single processor. Conversely, shared memory and messages used in distributed operating systems can be accessed by tasks allocated to different processors.

Distributed Mutual Exclusion

In multiprocessor systems, when two or more tasks in different processors attempt to simultaneously access shared resources, distributed mutual exclusion must be provided by distributed real-time operating systems. Distributed mutual exclusion can be used to ensure the integrity of shared resources by serializing concurrent access from various sites. In general, there are two basic types of distributed mutual exclusion: assertion-based and token-based. In assertion-based distributed mutual exclusion, when a site tries to access a shared resource, two or more successive rounds of message exchanges are required among all sites to check for availability of the shared resource. The site can successfully enter the shared resource only if the local assertion variable is true. In token-based distributed mutual exclusion, each shared resource has a token that is used to control access to it. A site can enter the shared resource only if it exclusively possesses the token.

The requirements of distributed mutual exclusion algorithms include mutual exclusion, freedom from deadlock, freedom from starvation, fairness, and fault tolerance. Many assertion-based and token-based distributed mutual exclusion algorithms can be found in [7].

2.2 System Level Design Language

2.2.1 Overview

System level design is a multi-stage process in which the system specification is gradually refined from an abstract idea down to an actual implementation [2]. In order to support different approaches in system level design, system level design languages

should in general have the following two essential attributes:

First, system level design languages should support defining and modeling systems at various levels of abstraction. In a top-down methodology such as in SpecC and SystemC, a system level design begins with a highly-abstracted specification, which is purely functional without any implementation details. The designers then refine the system model to gradually reveal more details about the implementation with each step representing a different layer of abstraction. Second, the system model at each layer of abstraction can be simulated, tested, and debugged, which allows designers to validate the system functionality at each design stage.

Such a model is referred to as a model of computation. SpecC refinement methodology has four models of computation: the specification model, the architecture model, the communication model, and the implementation model, each representing a layer of abstraction in the design hierarchy. SystemC consists of five models of computation that can be applied to a top-down system level design methodology. These five models of computation are the untimed functional model, the timed functional model, the transaction-level model, the behavior-level model, and the register-transfer model. Figure 2.6 shows the hierarchy of the models of computation in SpecC and SystemC. More details of each model of computation in SpecC and SystemC can be found in [4], [10], and [11].

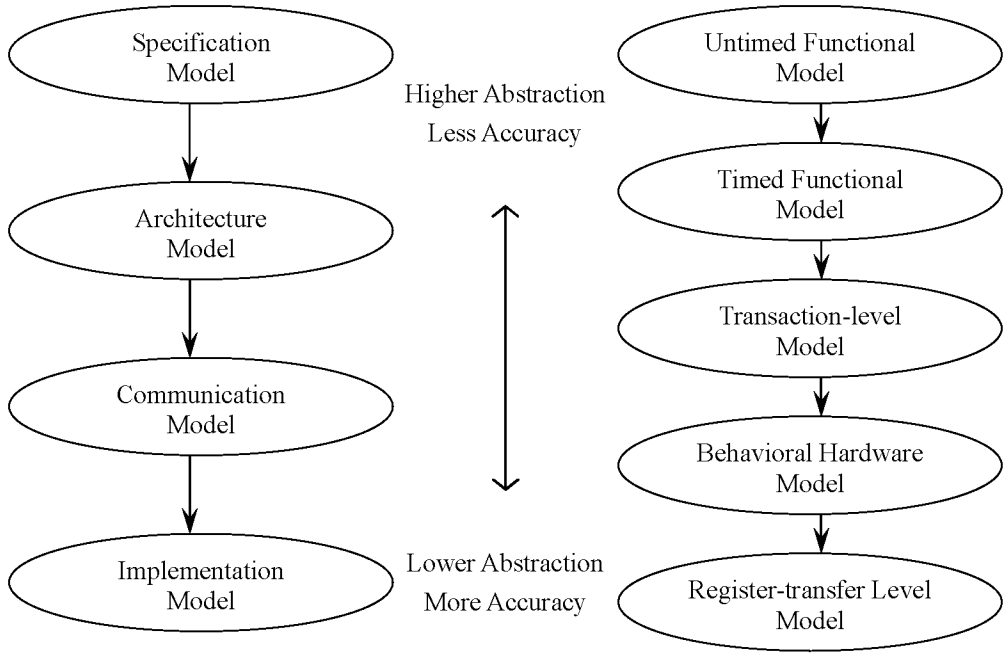


Figure 2.6: Models of computation in SpecC and SystemC

Several general requirements of system level design languages are discussed in [12]. A system level design language can be evaluated in terms of fulfillment of these requirements. These requirements are listed as follows.

- **Analyzability:** System level design languages should have the ability to analyze system models to establish their characteristics at all the levels of abstraction. SpecC, for example, supports such an analysis feature by profiling/estimation.
- **Explorability:** The syntax and semantics of system level design languages should explicitly specify the characteristics of system models at any level of abstraction. This gives the designers enhanced latitude in making implementation decisions. SpecC, for example, has `par` and `pipe` constructs for modeling parallelism and pipelined executions.
- **Refinability:** The exploration tools should allow specification of design

decisions taken in an explicit format. This allows unambiguous refinement of the model using refinement tools or through manual refinement. Also, the modeling styles of the model to be refined and the resulting model after refinement should be consistent.

- **Validability:** Models written in system level design languages should capable of validation at all the levels of abstraction. Both SpecC and SystemC allow validation by simulation.

2.2.2 SpecC Language

SpecC language is a system level design language developed at the University of California, Irvine. The first version of SpecC and its codesign methodology were introduced in late 1990s. Built on top of the ANSI-C programming language, SpecC language supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing [13]. In the remainder of this section, an overview of the main modeling components used in SpecC language is presented.

Behaviors

A SpecC behavior is an object for the specification of active functionality [14]. In general, behaviors are used to encapsulate computations. There are two types of behaviors: composite behaviors and leaf behaviors. A behavior is called a composite behavior if it contains instantiations of its child behaviors. Conversely, a leaf behavior is a behavior that describes an algorithmic program and contains no instantiations of other behaviors.

Syntactically, a behavior definition is begun with keyword `behavior`. A typical behavior consists of a set of ports, a set of local variables and methods, and a mandatory `main` function. Ports of behaviors allow for communication with other behaviors. The local variables and methods in a behavior have private attributes and can only be

accessed and called within the behavior itself. The `main` function of a behavior is the only public function and is the root of the behavior's execution. It is called whenever an instantiated behavior is executed and its completion determines the completion of the behavior [14]. If the behavior is a composite behavior, a set of child behavior instantiations is included as well.

Figure 2.7 shows an example of a leaf behavior definition and its block diagram. In this example, behavior "task" has an input port `p1`, an output port `p2`, a local variable `a` and a local function `init`. The `init` function initializes the variable `a`. The `main` function defines that the functionality of the behavior is to read the input data from port p1, increment it by the value of a, and send the result through output port p2.

```
Behavior task(in int p1, out int p2)
{
    int a;

    void init(void)
    {
        a = 1;
    }

    void main(void)
    {
        init();
        p2 = p1 + a;
    }
};
```

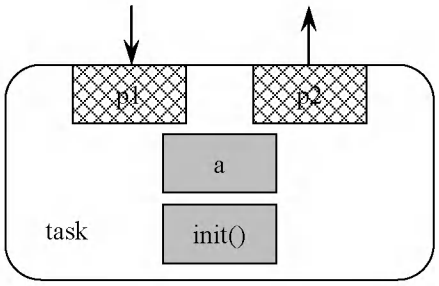
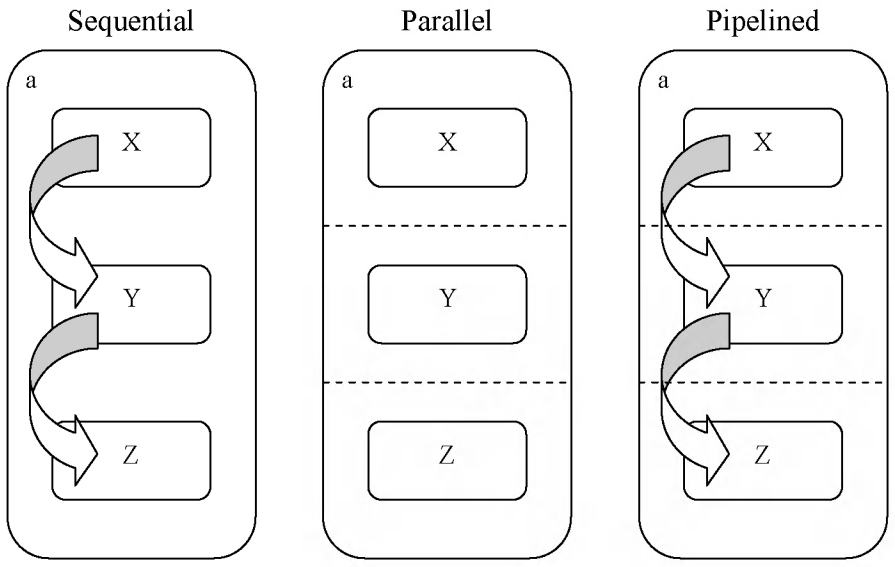


Figure 2.7: An example of leaf behavior in SpecC

In addition to components in leaf behaviors, composite behaviors may also have instances of their child behaviors. A child behavior of a composite behavior may be a leaf behavior or another composite behavior. In the `main` function of the composite behavior, the execution of a child behavior is initiated by making a function call to the child's `main` function. As shown in Figure 2.8, there are three types of execution sequences supported by SpecC language: sequential, parallel, and pipelined. In sequential execution, the default execution sequence in SpecC, one behavior starts its execution when the previous behavior finished. SpecC also supports concurrent execution for multiple

behaviors by using a `par` statement or pipelined execution by using a `pipe` statement.




```
behavior a
{
    void main(void)
    {
        X.main();
        Y.main();
        Z.main();
    }
};
```

```
behavior a
{
    void main(void)
    {
        par {
            X.main();
            Y.main();
            Z.main();
        }
    }
};
```

```
behavior a
{
    void main(void)
    {
        pipe {
            X.main();
            Y.main();
            Z.main();
        }
    }
};
```

Figure 2.8: Execution sequences in SpecC

Channels and Interfaces

A SpecC channel is an object designed for the specification of complex communication [14]. In general, a channel encapsulates the communication protocol of a communication bus. A channel can be considered to be a passive behavior. The variables and methods inside the channel are used to define the communication behaviors in the system. The channel is accessed by calling its interfaces during communication. The interface determines the set of public methods provided by the channel. It serves as a prototype of the communication protocols and is available to be used by behaviors during their communication.

Syntactically, a channel is specified by use of a keyword `channel`. A channel definition typically contains a set of private local variables and methods, and its public interfaces. An interface is specified by use of keyword `interface`. Figure 2.9 shows an example of SpecC code describing channel and interface.

```
interface I
{
   void send(int X);
   int receive(void);
};

channel C implements I
{
   int data;

   void send(int X)
   {
      data = X;
   }

   int receive(void)
   {
      return(data);
   }
};

behavior A(in int p1, I intf1)
{
   void main(void)
   {
      if(p1 > 0)
         intf1.send(p1);
      else
         intf1.send(0);
   }
};

behavior B(I intf2, out int p2)
{
   void main(void)
   {
      p2 = intf2.receive();
   }
};

behavior AB(in int p_in, out int p_out)
{
   C c1;
   A a1(p_in, c1);
   B b1(c1, p_out);

   void main(void)
   {
      a1.main();
      b1.main();
   }
};
```

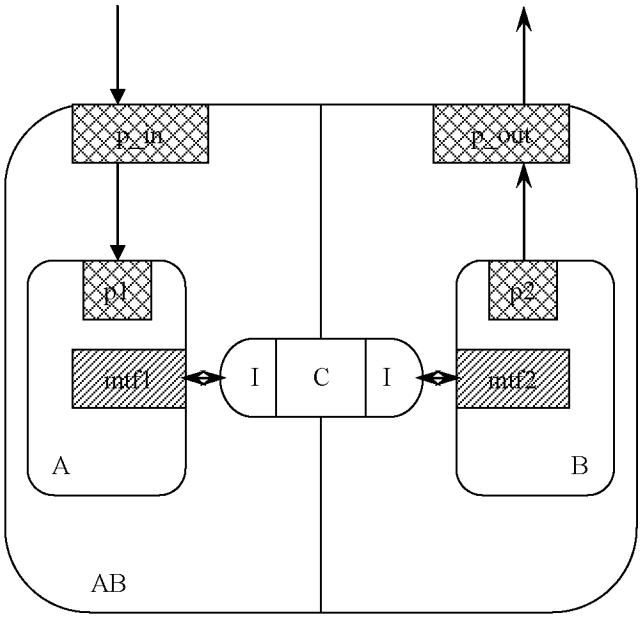


Figure 2.9: An example of channel and interface in SpecC

As we can see in the code, this example consists of one interface `I`, one channel `C` and three behaviors `A`, `B`, and `AB`. Channel `C` provides a simple communication protocol via an encapsulated integer variable `data`. The interface `I`, which the channel implements, contains the declarations of two public methods `send` and `receive`. The composite behavior `AB` contains two instances of its child behaviors `a1` and `b1`. These two behaviors are to execute concurrently by using a `par` statement and to communicate via channel `C`. The ports type of the behavior is defined explicitly. If a port is used to connect to a channel, an interface is specified as the port type in the behavior definition, such as `intf1` and `intf2`.

Synchronization

Synchronization is used to control cooperation among concurrent executing behaviors. In SpecC, synchronization is provided by using the build-in data type – event. Events can be instantiated inside behaviors or channels and bound to ports like any other data type. In order to specify synchronization, events are used as the arguments of `wait` and `notify` statements. As described in [14], the `wait` statement suspends the current behavior from execution until one of the events specified with the `wait` statement is triggered by another behavior at which point execution of the waiting behavior resumes. The `notify` statement triggers all specified events so that all behaviors waiting on one of these events can resume their execution. If no behavior is waiting on the triggered events at the time of the `notify` statement, the notification is ignored. Figure 2.10 shows an example of using event in SpecC program.

```
behavior a(out event e1)
{
    int data;
    void main(void)
    {
        data++;
        notify(e1);
    }
};

behavior b(in event e1)
{
    void main(void)
    {
        wait(e1);
        printf("Done.");
    }
};
```

Figure 2.10: An example of event in SpecC

Timing

In a SpecC program, since all other statements are executed in zero time, a `waitfor` statement is provided to model execution delays during simulations. The `waitfor` statement has a single integer argument which refers to the number of time units (nanoseconds) for which a behavior should supposed to suspend execution. Figure 2.11 shows an example on using waitfor statements in a behavior.

```
behavior task
{
    void main(void)
    {
        // code block 1
        waitfor(30);

        // code block 2
        waitfor(20);
    }
};
```

Figure 2.11: An example of using `waitfor` statement in SpecC

This section has presented an overview of SpecC language covering several basic components used in SpecC programming. Many more details about the history, features, syntax, and semantics of SpecC language can be found in [4], [13], and [14].

As we can see, SpecC language meets the requirements of system level design languages discussed in section 2.2.1. It is widely used in the system level designs of embedded systems for both industrial and academic purpose.

3. RTOS MODELING AND REFINEMENT IN SPECC

In this chapter, the methodology of RTOS modeling and refinement in system level design is presented as follows. In section 3.1, an overview of the System-on-Chip design environment (SCE) is presented. Next the SpecC RTOS model and the OS refinement methodology are introduced in section 3.2 and 3.3 respectively. Finally, a demonstration of RTOS modeling and refinement process in the SCE environment is described in section 3.4.

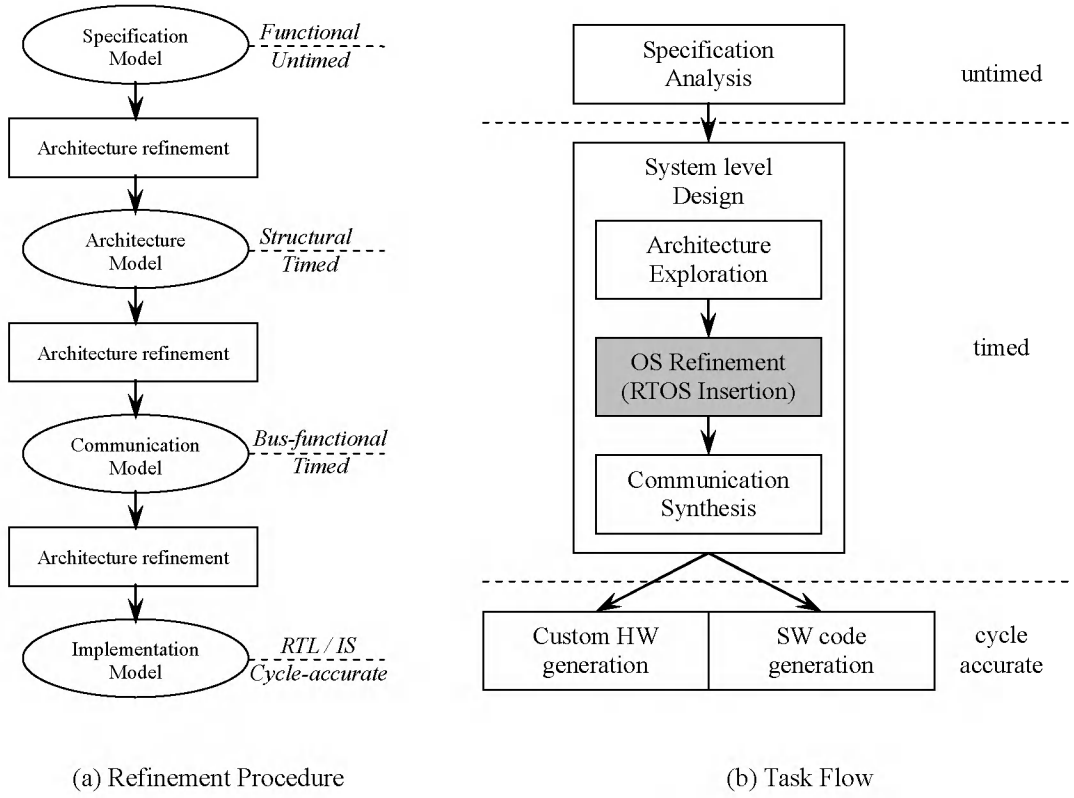
3.1 SCE Environment

The System-on-Chip design environment (SCE) is a system level design environment developed at the University of California, Irvine. The SCE environment consists of a set of tools and user interfaces to help designers refine a functional system specification to its accurate implementation with minimal effort.

The main SCE graphic user interface (GUI) contains three windows, namely, the "project management" window, the "design management" window, and the "logging" window, a set of toolbars, and various menu options. Each window maintains different information about the open project during system level design flow. The toolbars and menu options provide tools, services, and management for editing, refining, compiling, simulating, etc. For example, the *Synthesis* menu provides for launching the various refinement tools and making synthesis decisions, such as those frequently used in the demonstration described in section 3.4.

As described in [15], the SCE environment consists of four levels of model abstraction, namely, specification, architecture, communication, and implementation models. Consequently, there are three refinement steps: architecture refinement, communication refinement, and hardware/software (HW/SW) refinement. These

refinement steps are performed in top-down order beginning with a top-most abstract specification model. Figure 3.1 below shows the refinement procedures and task flow for system design with SCE.



(a) Refinement Procedure (b) Task Flow

Figure 3.1: Refinement procedure and task flow with SCE

The specification model is an untimed functional model. It describes only the desired system behaviors and has no implementation details or notion of time. Architecture refinement transforms this specification model to an architecture model by partitioning the system behaviors and mapping these partitions onto the selected components. Thus, the architecture model defines the structure of system architecture, including estimated execution times for the behaviors of each component. The next step, communication refinement, selects a set of system busses and protocols and then maps the communication functionality between components onto the system busses. Communication refinement creates the bus-functional communication model, which

reflects the system architecture consisting of busses and is timed in both computation and communication. The final step is the HW/SW refinement which transforms the communication model to an implementation model. The implementation model is a cycle-accurate structural description consisting of an RTL model for the hardware components and instruction-set-specific assembly code for the processors.

This section has presented an overview of the SCE environment. The main purpose of the SCE environment is to assist the designers in facilitating the system level design flow efficiently by providing an easy-to-use environment for modeling, synthesis, and validation. Many more details about the SCE environment can be found in [15]. The SCE environment is the main design environment used in this thesis.

3.2 SpecC RTOS Model

System level design is widely used to refine systems from abstract specifications to actual implementations by defining and modeling systems at various levels of abstraction. On the other hand, real-time operating systems become an increasingly important component in today's embedded systems implementation. Therefore, an abstract RTOS model which contains the RTOS runtime behavior is needed in system level synthesis to assist designers in evaluating the system design at the higher levels of abstraction.

In [2], a RTOS model is developed on top of the SpecC system level design language. The SpecC RTOS model provides an abstraction of the key features found in modern RTOS like task management, real-time scheduling, preemption, task synchronization, and interrupt handling. As shown in Figure 3.2, the RTOS model is implemented in the form of SpecC channels. The interface of the RTOS channel can be classified into four categories of services: operating system management, task management, event handling, and time modeling.

```
typedef int                         proc;
typedef unsigned long long int      sim_time;

interface OSAPI
{
    /* OS management */
    void init(void);
    void start(void);

    /* task management */
    proc task_create(char *name, int type,
                     sim_time period, sim_time wcet);
    void task_activate(proc tid);
    void task_sleep(void);
    void task_resume(proc tid);
    void task_terminate(void);
    void task_endcycle(void);
    void task_kill(proc tid);
    proc par_start(void);
    void par_end(proc p);

    /* event handling */
    evt event_new(void);
    void event_del(evt e);
    void event_wait(evt e);
    void event_notify(evt e);

    /* time modeling */
    void time_wait(sim_time nsec);
};
```

Figure 3.2: Interface of the SpecC RTOS model

Operating system management mainly deals with initialization of the RTOS kernel when the system starts. Two procedures, *init* and *start*, initialize the relevant data structures and start the multi-task scheduling. Task management is the main function of the RTOS model. It provides management for task creation, termination, suspension and activation via different procedures, such as *task_create*, *task_activate*, *task_terminate*, *task_kill*, *task_sleep*, *task_resume*, etc. For preemptive multi-task scheduling, two special procedures, *par_start* and *par_end*, are used to suspend the calling task for running its child task and to resume the calling task's execution when its child tasks finish. In modeling of periodic tasks, *task_endcycle* notifies the kernel that a periodic task has finished its execution in the current cycle. System calls for event handling re-implement SpecC events with RTOS events. *event_wait* and *event_notify* replace the SpecC primitives for event wait and notify. The last component in the RTOS interface is time modeling, used to model delays during simulation. In the RTOS model, the SpecC delay

primitives, such as *waitfor*, are replaced by *time_wait* calls.

3.3 OS Refinement Methodology

In this section, we will illustrate the OS refinement rules and steps via some sample examples. OS refinement inserts the RTOS model into the system architecture model and creates a scheduled model in which task execution is dynamically scheduled by the selected scheduling algorithm. Based on the discussion in [2], the OS refinement methodology is summarized in the subsections below.

3.3.1 Inserting Timing Annotations

As discussed in the previous chapter, during the system synthesis, the concept of time is introduced into the architecture model after architecture refinement by annotating delays. In the architecture model, the main purpose of annotating delays is to model the average or worst-case execution times of corresponding behaviors on the target components. In the scheduled model, these timing annotations will also serve as constraints for tasks scheduling.

In SpecC programs, the `waitfor` statement is used to model the execution delay of each behavior during simulation. Usually a behavior can be divided into several basic functional blocks. Therefore, a `waitfor` statement is inserted at the end of each basic block to represent the execution delay of code inside the block. During the OS refinement, the RTOS model `time_wait` calls replace the SpecC `waitfor` statements to model the delay of codes in the block. Alternatively, during task execution, the context switch may only happen inside the `time_wait` calls. Thus, tasks can only be preempted at the boundaries of the basis blocks. This assumption is applicable for all the scheduling algorithms supported in the RTOS model.

3.3.2 Task Refinement

During the task refinement, behaviors in the architecture model will be converted into RTOS-based tasks in the scheduled model. The conversion processes of leaf behaviors and composite behaviors are different. Leaf behaviors can be directly converted into tasks. During such conversion, an `os_task_create` method is added for creating the task. Each created task has its own task ID. The body of the task in its `main` function is enclosed in a pair of `task_activate` and `task_terminate` calls. Thus, the RTOS model can control task activation and termination. Figure 3.3 shows an example of converting a leaf behavior to a task. An aperiodic task B is created with priority equal to zero.

```
behavior B
{
    void main(void)
    {
        // code block 1
        waitfor(50);

        // code block 2
        Waitfor(10);
    }
};
```

(a) Behavior in Unscheduled Model

```
behavior task_B(OSAPI os)
{
    int tid;
    void os_task_create(void)
    {
        tid = os.task_create("B", APERIODIC, 0, 500);
    }
    void main(void)
    {
        os.task_activate(tid);

        // code block 1
        os.time_wait(50);
        // code block 2
        os.time_wait(10);

        os.task_terminate();
    }
};
```

(b) Task in Scheduled Model

Figure 3.3: Task refinement for a leaf behavior

For composite behaviors, the conversion involves the dynamic creation of child tasks within a parent task. As in the example shown in Figure 3.4, the `par` statement in a composite task is refined to fork and join the child tasks in the execution of the parent task. The parent task creates its child tasks by calling their `os_task_create`

functions. Then the parent task calls `par_start` to suspend its own execution and start the parallel execution of its child tasks in the `par` statement. After the child tasks finish their execution and the `par` statement exits, `par_end` is called to resume the execution of the parent task.

```
. . .
/* two parallel behaviors */
par
{
   b1.main();
   b2.main();
}
. . .
```

(a) Behavior in Unscheduled Model

```
. . .
task_b1.os_task_create();
task_b2.os_task_create();

/* two parallel tasks */
os_par_id = os.par_start();
par
{
   task_b1.main();
   task_b2.main();
}
os.par_end(os_par_id);
. . .
```

(b) Task in Scheduled Model

Figure 3.4: Task refinement for a composite behavior

3.3.3 Synchronization Refinement

In the system specification and architecture models, synchronization is implemented using SpecC events with the `wait` and `notify` primitives. Synchronization refinement replaces all events and event-related primitives with corresponding event handling routines of the RTOS model [2]. RTOS events can be created and deleted by `event_new` and `event_del` calls. Therefore, in the scheduled model, all SpecC event instances are replaced with RTOS events and all SpecC `wait` and `notify` statements are replaced with RTOS `event_wait` and `event_notify` calls. An example of such synchronization refinement is shown in Figure 3.5.

```
channel C
{
   event start, done;
   void send(. . .)
   {
      notify(start);
      wait(done);
      . . .
   }
};
```

(a) SpecC Events in Unscheduled Model

```
channel C(OSAPI os)
{
   evt start, done;
   void send(. . .)
   {
      os.event_notify(start);
      os.event_wait(done);
      . . .
   }
};
```

(b) RTOS Events in Scheduled Model

Figure 3.5: Synchronization refinement example

3.3.4 Task Scheduling

After task refinement and synchronization refinement, both task management and synchronization are implemented using the system calls of the RTOS model. Thus, the dynamic system behavior is completely controlled by the RTOS model layer [2]. The next step will be implementation of task scheduling.

The RTOS model library provides services for task management and scheduling. As we know in the unscheduled system model, behaviors can be executed truly in parallel by using the `par` statement. But, in the scheduled model, tasks can only be executed in an interleaved way. Thus, in order to model dynamic task scheduling, the execution of tasks must be serialized first. The RTOS model ensures that only one task can be executed on the simulation kernel at any point of time. This is achieved by blocking all other tasks on SpecC events, except the current task. During simulation, the RTOS model provides a scheduler to maintain task scheduling. The scheduler is invoked if any task state is changed in the system by a RTOS call. Each time the scheduler is invoked, it will select a task based on the current scheduling algorithm and task priorities from the ready queue, and dispatch it by releasing its SpecC event. Then the selected task becomes the current task executing on the simulation kernel, with all other tasks blocked by their SpecC events.

3.3.5 Summary

In summary, OS refinement transforms the unscheduled system architecture model to the RTOS-based scheduled model in which the RTOS model can provide the mechanisms for task management, dynamic task scheduling, communication and synchronization, etc. Thus, the simulation results of the scheduled model can be used to reflect RTOS behavior and to accurately evaluate a system design at the higher levels of abstraction.

3.4 Demonstration of OS Refinement in SCE

As discussed in section 3.1, the SCE environment provides tools and interfaces to assist designers in modeling and refining system designs at various levels of abstraction. Like other refinement steps, OS refinement can be done automatically by selecting appropriate tools and options from the SCE environment. This section shows an example to demonstrate the design flow of refining the system specification model to the RTOS-based scheduled model.

3.4.1 System Specification Model

The SCE chart of the specification model for this example is shown in Figure 3.6. The task `task_set` is the top-level behavior, which involves two parallel behaviors `proc1` and `sti1`. Behavior `proc1` has two child behaviors, `body` and `isr`. The main purpose of behavior `isr` is to function as an interrupt service routine. Behavior `body` has three concurrent child behaviors `t0`, `t1` and `t2`, each behavior having two basic execution blocks. The `waitfor` statement is used inside each basic block to model the execution delay of the behaviors. Behavior `t1` is waiting on semaphore `sem1` to start its execution, and behavior `t2` will wait on semaphore `sem2` to execute its second block. Behavior `sti1` is used to generate interruptions. There are two interrupts `e1` and `e2`. Each of these interrupts will evoke the interrupt service routine `isr`, which will release

semaphore `sem1` and `sem2` to make `t1` and `t2` continue execution. The concurrent leaf behaviors in the specification model can be executed truly in parallel by using `par` statements. Figure 3.7 shows the simulation result of the behavior execution status. We can see that during time 0 – 10, 20 – 50, and 50 – 80, at least two behaviors are executed concurrently.

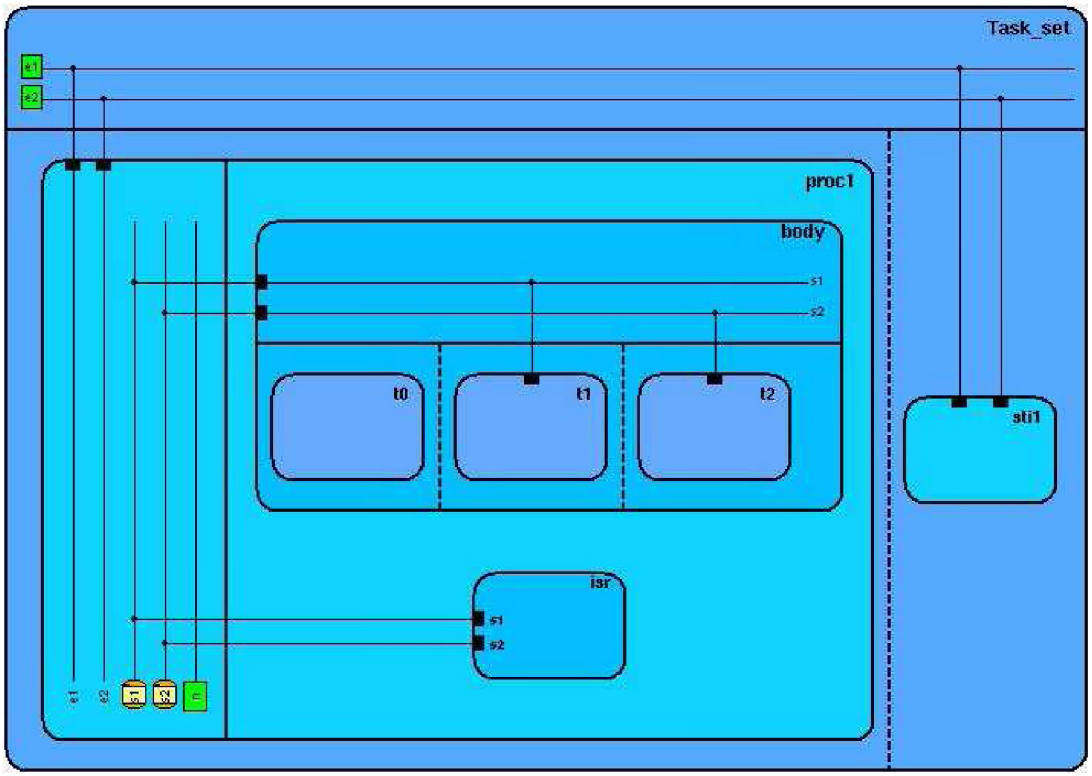


Figure 3.6: SCE chart of the system specification model

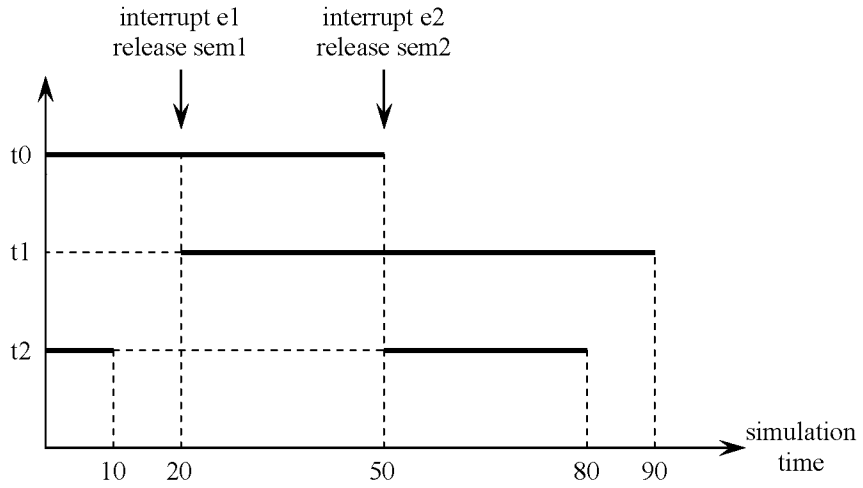


Figure 3.7: Behavior execution result in the system specification model

3.4.2 System Architecture Model

As discussed in section 3.8, the architecture refinement creates the architecture model by selecting processing elements (PE) and mapping the different partitions of the system behaviors onto each PE. In this example, we select two processing elements: a Motorola DSP 56600 and a standard custom hardware element, and name them DSP and HW respectively. The behavior `proc1` is mapped onto the DSP and `sti1` is mapped onto HW. Figure 3.8 shows the SCE chart of the architecture model. As we can see in the top-level behavior `task_set`, two PE relevant sub-behaviors, `DSP` and `HW`, are constructed and inserted during architecture refinement. Like the specification model, all the concurrent leaf behaviors `t0`, `t1` and `t2` can be executed truly in parallel. The simulation result of the behavior execution status is shown in Figure 3.9.

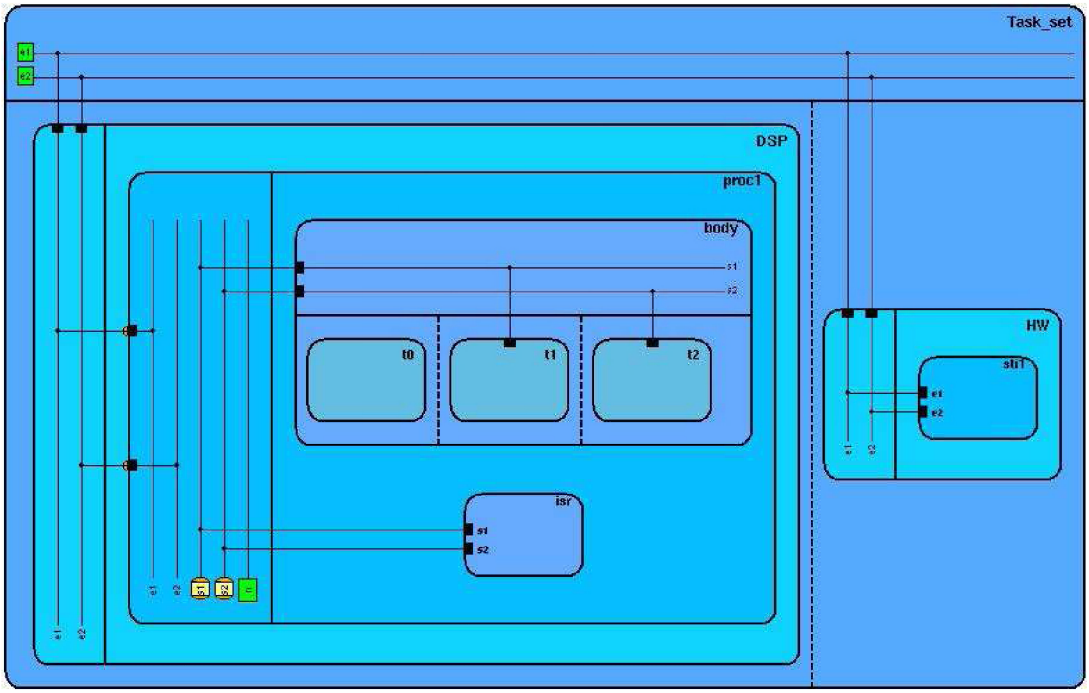


Figure 3.8: SCE chart of the system architecture model

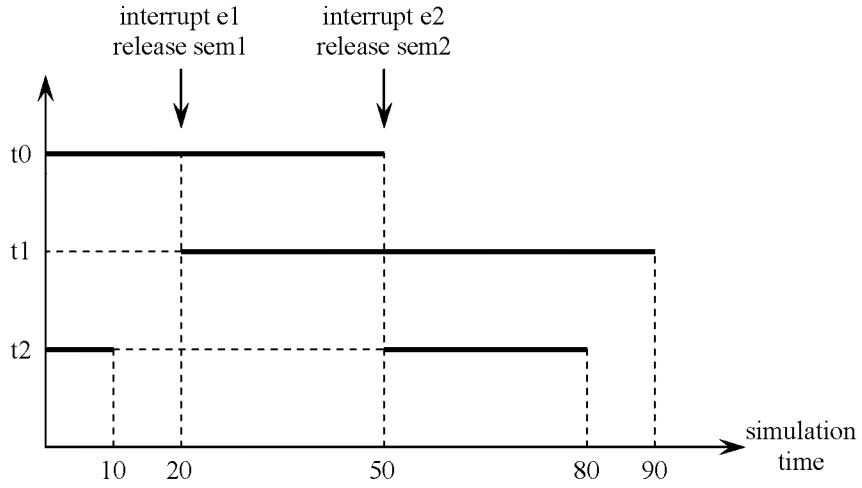


Figure 3.9: Behavior execution result in the system architecture model

3.4.3 System Scheduled Model

The next step of system level synthesis is the OS refinement, which includes inserting the RTOS model into the system and dynamically scheduling task execution

according to the selected scheduling algorithms provided by the RTOS services. During the OS refinement, the RTOS model implementing the RTOS interface is instantiated inside each PE in the form of a SpecC channel, as shown in Figure 3.10. SpecC behaviors in the architecture model are converted into RTOS-based tasks. SpecC `waitfor` statements are replaced by RTOS `time_wait` calls to model task` execution delay. The SCE environment provides convenient tools to assist the users in selecting different scheduling algorithms on each software component. In this example, we use a priority-based scheduling algorithm on DSP. The priority of each task is specified inside its `os_task_create` method, while here the priorities of task `t0`, `t1` and `t2` are 3, 2, and 1, respectively, with `t0` having the lowest priority and `t2` having the highest priority. Task execution on each PE is serialized in an interleaved way based on the scheduling algorithms and tasks priorities. The simulation result is shown in Figure 3.11.

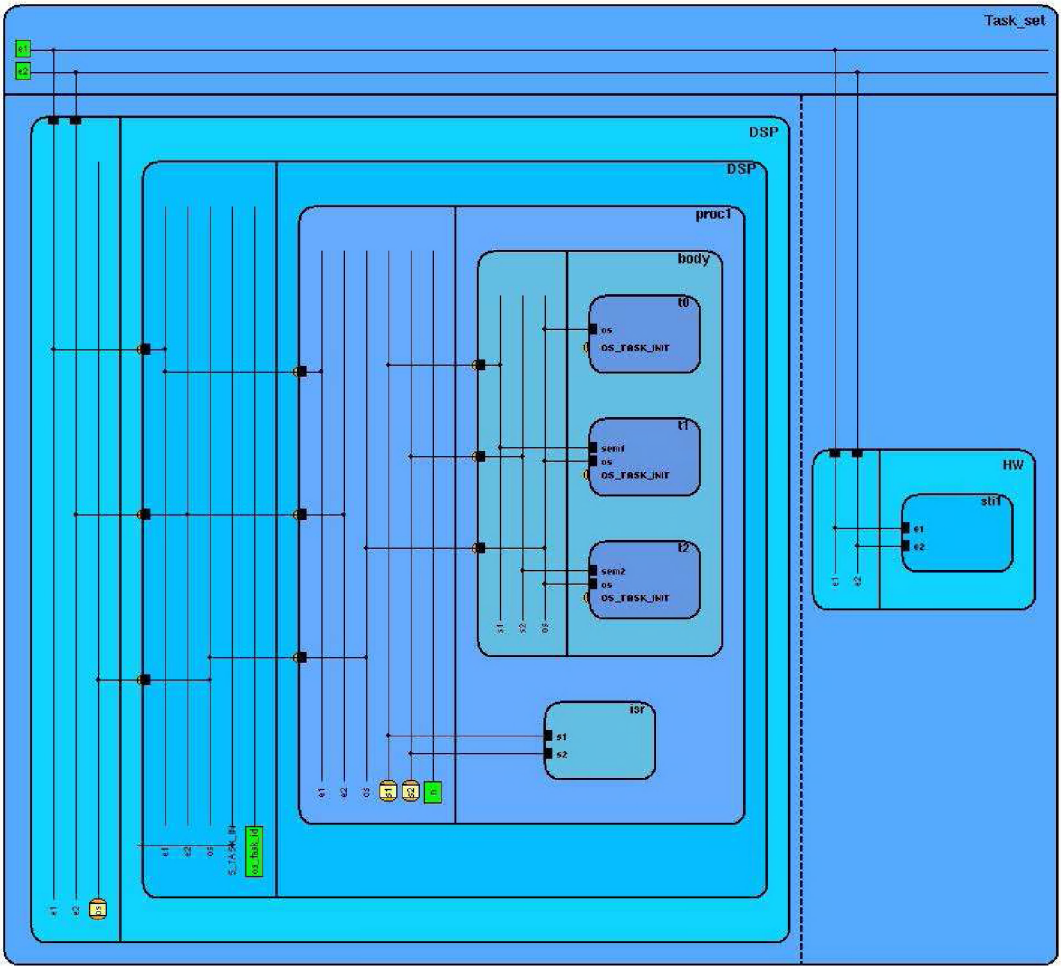


Figure 3.10: SCE chart of the system scheduled model

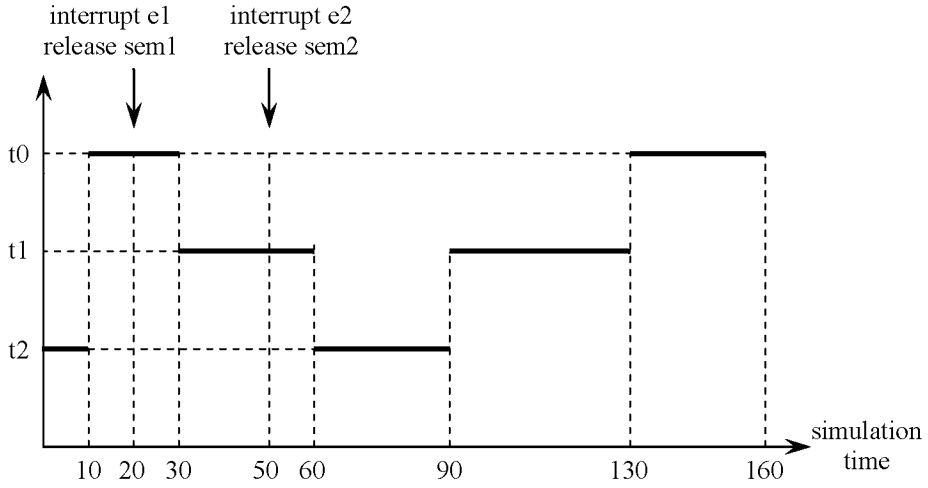


Figure 3.11: Task execution result in the system scheduled model

4. DRTOS MODELING AND REFINEMENT IN SPECC

This chapter will present the methodology of modeling and refining distributed real-time operating system (DRTOS) during system-level synthesis. Our DRTOS model is an extension of the RTOS model introduced in section 3.2, including a global synchronizer and a global scheduler. We named it as ERTOS-SS, for extended RTOS with global scheduling and synchronization. Like the RTOS model, the ERTOS-SS DRTOS model is also built on the SpecC system-level design language and can be easily integrated into the existing system-level design flow to accurately reflect DRTOS runtime behavior.

4.1 Summary of DRTOS Services

The first step in developing a DRTOS model is to determine the functionality which should be implemented in the model. In order to explicitly indicate what needs to be done, a summary of the basic services based on [7] and [8] in various commercial or research DRTOS kernels is listed below:

- **Multiprocessor Task Scheduling** – Multitask scheduling on multiprocessor systems provides mechanisms for task assignment, task scheduling, resource allocation, etc.
- **Load Balancing** – During tasks assignment, load balancing for all Processing Elements (PEs) must be fulfilled in order to best utilize system resources and maximize system performance.
- **Intra-processor Communication (IPC)** – Intra-processor communication provides mechanisms for sharing data among different processes executed on the same PE.
- **Inter-processor Communication (IPC)** – On the contrary to intra-processor

communication, inter-processor communication provides capability for sending and receiving shared data among different tasks physically located on different PEs.

- **Distributed Synchronization** – Another of the basic services provided by DRTOS kernels is distributed synchronization among cooperative tasks running in parallel on different PEs.
- **Distributed Mutual Exclusion** – Shared resources in multiprocessor systems must be protected by distributed mutual exclusion mechanisms. Concurrent accesses to shared sources from multiple PEs will be serialized by distributed mutual exclusion to secure the integrity of shared resources.

4.2 DRTOS Model Implementation Guidelines

In system level design, an abstract DRTOS model is required to perform the services listed above and to reflect DRTOS runtime behavior during system level synthesis. The methods for achieving these services in different research or commercial DRTOS kernels may vary, but at the higher level of abstraction, these services should be implemented using system-level design approaches provided in system level design languages and refinement methodology.

Based on the services listed in the previous section, the DRTOS model can be separated into three basic modules: allocator, synchronizer, and scheduler. The allocator models the mechanism of task allocation and load balancing. The synchronizer is used to model task synchronization, as well as to provide both intra- and inter-processor communications among tasks. The scheduler provides the capability to model task management and task scheduling.

4.2.1 Allocator

The main purpose of the DRTOS allocator is to provide the function of task and resource allocation for online multiprocessor task scheduling algorithms, as well as

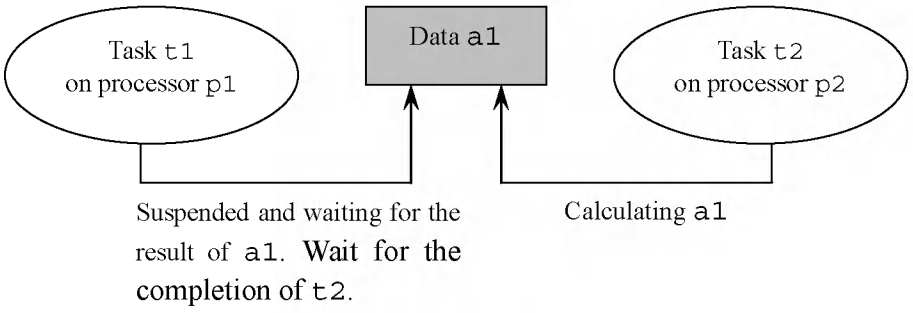
maintain load balancing. In [8], the author indicated that multiprocessor architectures combined with real-time scheduling represent a delicate problem and an ongoing research area. It is clearly easier for an off-line scheduler to be optimal. Thus, in order to simplify the scheduling algorithms, the following assumption is used throughout this research work: task allocation decisions have been made by system designers before runtime. With this assumption, task allocation and load balancing can be accomplished during architecture refinement by properly selecting PEs and mapping system behaviors onto different components.

4.2.2 Synchronizer

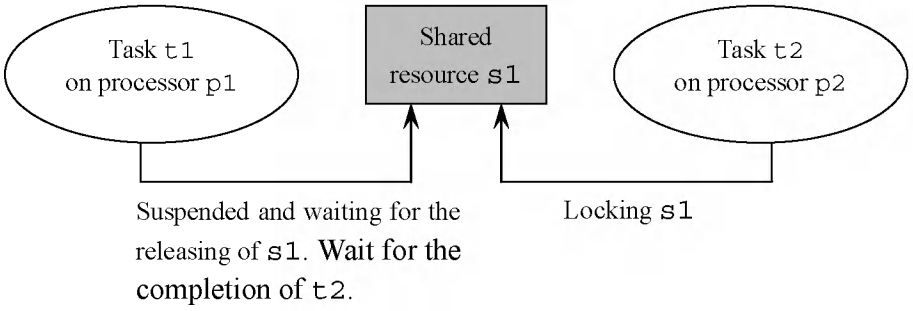
The DRTOS synchronizer can provide services for task communication, task synchronization, and mutual exclusion. Although communication, synchronization, and mutual exclusion are different types of DRTOS services, all of them are required to handle task dependency. For example suppose that two tasks `t1` and `t2` are running in parallel on two different processors `p1` and `p1`. If task `t1` needs data `a1` computed by task `t2`, the execution of task `t1` will be suspended until task `t2` is completed, at which point task `t1` will be resumed. This example of distributed synchronization is shown in Figure 4.1 (a). Figure 4.1 (b) shows an instance of distributed mutual exclusion. As can be seen, task `t1` running on processor `p1` wants to access shared resource `r1` which is currently exclusively locked by task `t2` running on processor `p2`. Thus task `t1` must wait until the completion of task `t2` in order to execute. These examples show that task dependency can be of various types, but at the higher levels of abstraction, we can ignore the nature of the dependency and simply formulate an abstraction and assert that task T_j is eligible to be released just after task T_i has finished its execution [16].

Therefore, at the higher levels of abstraction, such three types of DRTOS services can be abstracted as managing task dependencies. To clearly distinguish different types of task dependencies, we category them into two groups: internal task dependency and external task dependency. Internal task dependency indicates those dependencies among

tasks located on the same PE. On the contrary, external task dependency includes those dependencies among tasks locate on different PEs. RTOS-based events and event-handling primitives provided by the RTOS channel can be used to handle internal task dependencies within each PE. Additionally, our DRTOS model provides a synchronizer to manage external task dependencies. More detailed discussion on synchronization protocol and the implementation of the DRTOS synchronizer will be present in section 4.3.



(a) Distributed Synchronization



(b) Distributed Mutual Exclusion

Figure 4.1: Example of task dependencies

4.2.3 Scheduler

The major functionality of the DRTOS scheduler is to determine execution order of all tasks in a system. In a multiprocessor system, tasks can be classified into two

groups according to their dependencies with other tasks. If a task has no dependency with any other task or only has internal dependencies, this task can be scheduled inside its own PE without considering of other tasks on other PEs. Alternatively, if a task has external dependencies, its execution order must be synchronized with other tasks located on different PEs by a global scheduler.

As we have introduced in sections 3.2 and 3.3, the SpecC single RTOS channel has four categories of interface: OS management, task management, event handling, and time modeling. OS refinement inserts RTOS channels into system model to perform RTOS services. During simulation, the RTOS channel provides a scheduler to maintain task scheduling on each PE. Our DRTOS model is an extension of the single RTOS model and all the interfaces provided by the RTOS channel can be reused in our DRTOS modeling and refinement process. Therefore, the execution for those tasks without external dependencies can be scheduled by the scheduler provided by the single RTOS channel inside each PE.

Additionally, to determine the execution order for those tasks having external dependencies, a global scheduler must be implemented in the DRTOS model. The main responsibility of such a global scheduler is to adjust task execution order based on their external dependencies. It can be considered as an auxiliary of the scheduler provided in each RTOS channel. The implementation details of the DRTOS global scheduler will be presented in the next section.

4.3 ERTOS-SS DRTOS Model Implementation

Based on the implementation guidelines in section 4.2, the ERTOS-SS DRTOS model will be focused on functionality of the DRTOS synchronizer and the DRTOS global scheduler. More implementation details about these two modules of the ERTOS-SS DRTOS model will be presented in this section. The synchronization and scheduling protocol will be presented first, followed by an introduction of SpecC behaviors of the synchronizer and the global scheduler in the ERTOS-SS DRTOS model.

4.3.1 Synchronization Protocol

Synchronization protocol must be used to handle task dependency in a system model. More detailed discussion on different types of synchronization protocols can be found in [17]. In the remainder of this section, the synchronization protocol used in the ERTOS-SS DRTOS model will be introduced.

The DRTOS synchronizer must hold information regarding its services, such as which tasks depend on which others. In the ERTOS-SS DRTOS model implementation, management of task dependency is achieved by using the idea of semaphores. A more detailed description of semaphores can be found in [6]. Figure 4.2 shows an example of using semaphores to manage task dependencies. As we can see, the dependency between task `t1` and task `t2` can be accomplished easily by using semaphore `s1`.

```
behavior t1
{
    void main(void)
    {
        // code block 1
        waitfor(20);

        //wait semaphore
        wait semaphore s1;

        // code block 2
        waitfor(50);
    }
};
```

```
behavior t2
{
    void main(void)
    {
        // code block 1
        waitfor(40);

        //release semaphore
        release semaphore s1;

        // code block 2
        waitfor(30);
    }
};
```

(The second execution block of task1 must be executed after the completion of the first execution block of task2.)

Figure 4.2: Example of using semaphore to manage task dependency

In implementation of the DRTOS synchronizer, all dependencies among different tasks can be managed by creating a set of semaphores. Each semaphore is used to handle only one instance of task dependencies. For example, if the execution of task `t0` depends on a shared data `d1` that is calculated in task `t1`, a semaphore `sem1` is needed for the shared data `d1` to handle dependency between task `t0` and `t1`. As another example, three tasks `t2`, `t3`, and `t4` need exclusively access a shared resource `res1` during their

execution, while res1 only allows one visitor at any point of time. In such a case, a semaphore sem1 is required for the shared resource res1 to manage task dependencies between t2, t3, and t4. The number of semaphores used in the system is equal to the total number of dependencies among all tasks in the system. Each semaphore has two types of status: ready and locking. Ready status indicates that the semaphore is not locked by any task and is ready to be used. Locking status indicates that the semaphore is currently locked by a task and cannot be used by other tasks. When a semaphore is created, its initial status is locking. Each semaphore maintains a waiting list to save waiting tasks, and a locking list to save locked tasks. Notice that if a semaphore is in ready status, both its waiting list and locking list must be empty. But for locking status, a semaphore's locking list must be not empty. Since a semaphore can be exclusively accessed and locked by only one task at any point of time, the maximum length of its locking list equals to one. All the semaphore status and their corresponding lists will be saved in a "semaphore status table". The status table will be updated whenever the status of a semaphore is changed.

In the ERTOS-SS DRTOS model, the synchronizer can be seen as an event-based process that runs whenever a message (request or release) is received from a task. During the execution of a task, if a semaphore is required, the task will send out a request message. If a semaphore is released, the task will send out a release message. Each time a task issues a request or a release message, the synchronizer will receive it. When the synchronizer receives a request message, it looks at the semaphore status table to check the status of the required semaphore. If the status of the required semaphore is ready, the synchronizer will change it to locking, and invoke the DRTOS global scheduler for scheduling. If the status of the required semaphore is locking, it means the semaphore currently is being locked by other tasks. The synchronizer will not change the status of the required semaphore and directly invoke the scheduler for scheduling. Alternatively, if the synchronizer receives a release message, its reaction will depend on the status of the released semaphore as well. If the status of the released semaphore is ready, the synchronizer will do nothing. If the status of the

released semaphore is `locking`, the synchronizer will change it to `ready` and invoke the scheduler for scheduling. Examples of operations for each scenario are shown in Figure 4.3.

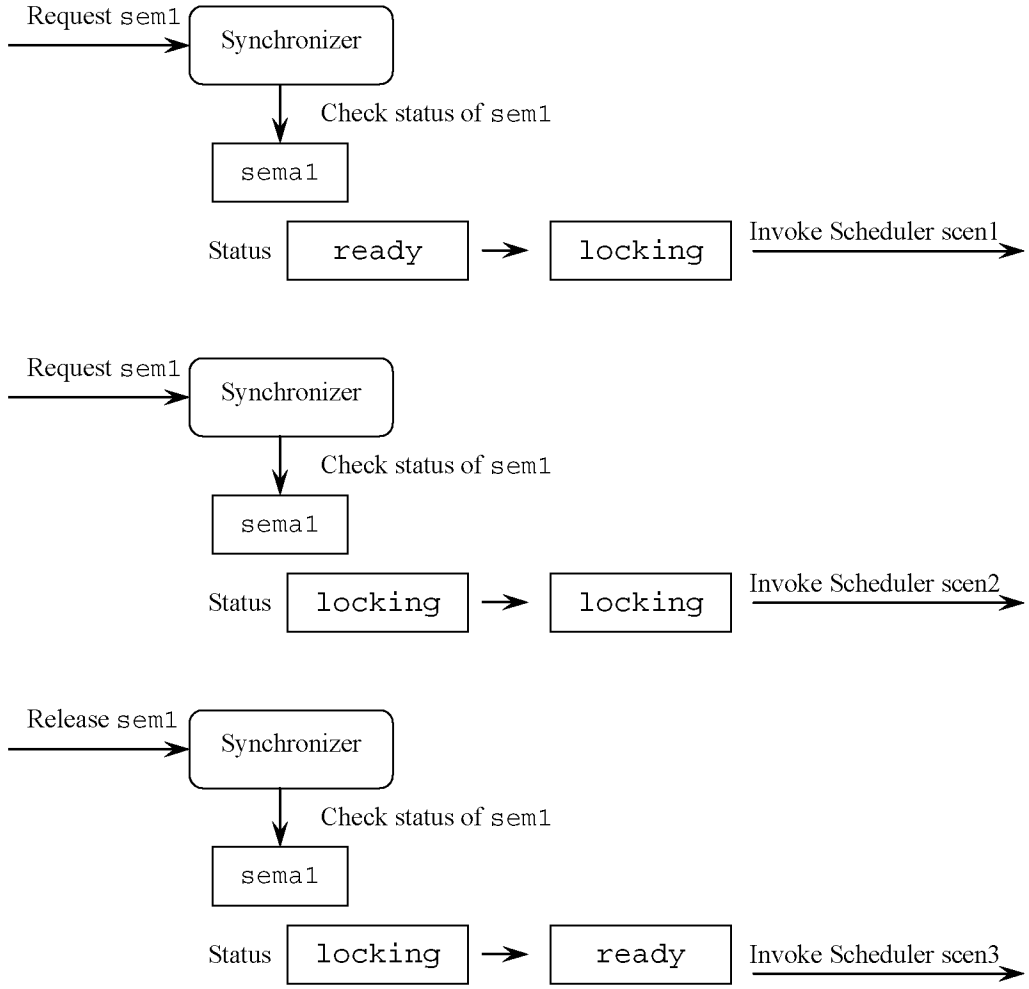


Figure 4.3: Operations of the DRTOS synchronizer

4.3.2 Scheduling Protocol

Similar to the synchronizer, the DRTOS global scheduler is modeled as an event-based process as well. The main responsibility of the scheduler is to manage semaphore's waiting and locking list and to determine which task should be executed

based on the current scheduling algorithm. The scheduler in the ERTOS-SS DRTOS model can provide two scheduling mechanisms: First-In First-Out (FIFO) and Priority-Based (PB).

Based on the synchronization protocol discussed above, there are three scenarios that the synchronizer will invoke the global scheduler for task scheduling. The first scenario is when the synchronizer changes status of the required semaphore from `ready` to `locking`. At that time the request task obtains the controls of the required semaphore and is ready to be executed. Thus, the scheduler will save the request task to the semaphore's locking list and send a `run` message back to the request task to allow it continuing its execution. The second scenario is when the status of the required semaphore is `locking`. The scheduler will add the request task to the semaphore's waiting list and send a `suspend` message back to the request task. In this case, the execution of that task will be suspended to wait for further notice. Third, when the synchronizer changes status of the released semaphore from `locking` to `ready`, the semaphore is currently ready to be used by another task. The scheduler will remove the release task from the semaphore's locking list and check the waiting list. It the waiting list is empty, no further action is needed. Otherwise, the scheduler will select a task based on the current scheduling algorithm and task priorities from the waiting list. If the current scheduling algorithm is FIFO, the first task in the waiting list will be selected. Alternatively, the task with highest priority will be selected for PB scheduling algorithm. Then the scheduler will remove the selected task from the semaphore's waiting list, add it to the locking list and send a `resume` message back to this task to resume its execution. Finally, the scheduler will properly update the semaphore's waiting list to maintain the input orders of the remaining tasks. Examples of operations for each scenario are shown in Figure 4.4.

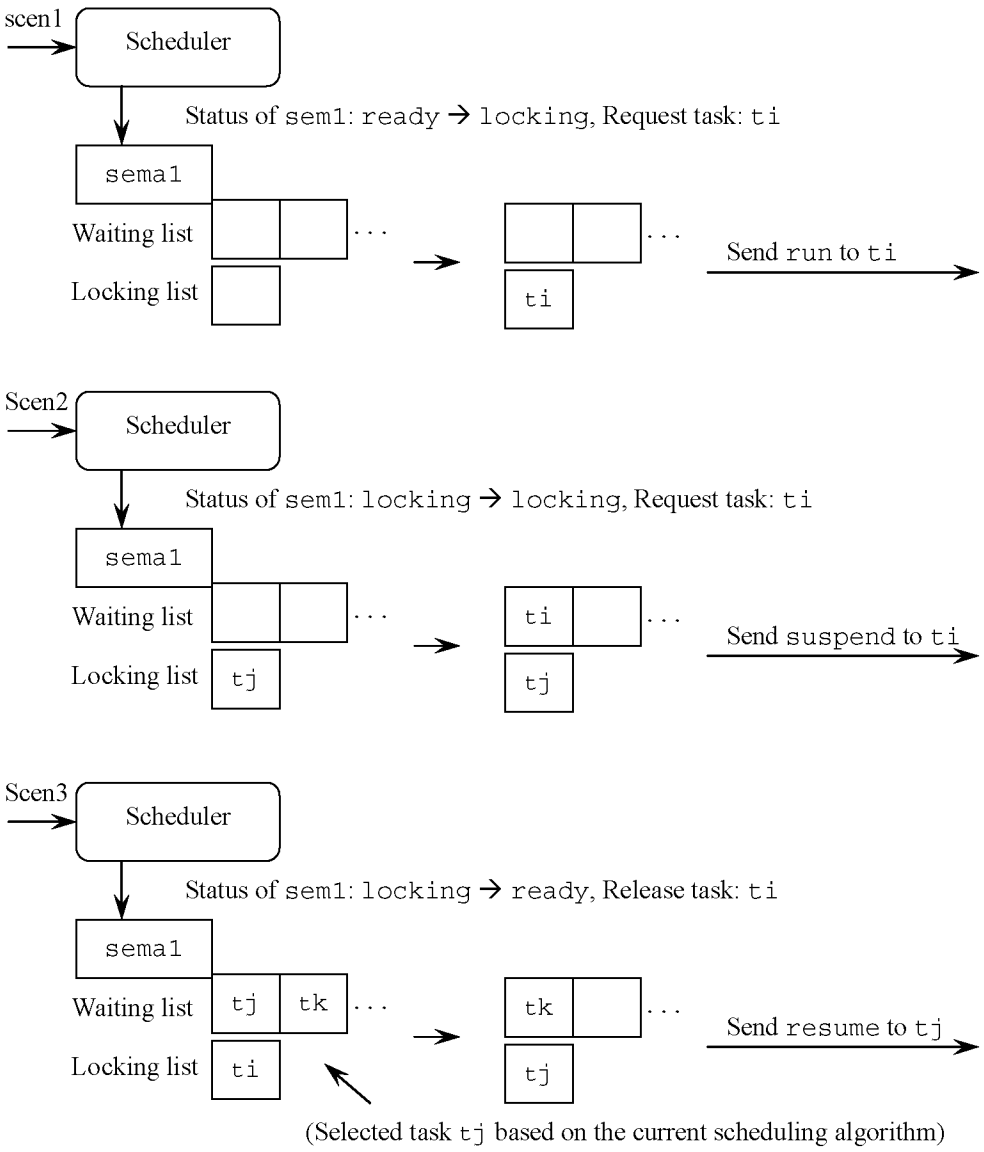


Figure 4.4: Operations of the DRTOS global scheduler

4.3.3 Implementation of the Synchronizer and the Global Scheduler

The synchronizer and the global scheduler in the ERTOS-SS DRTOS model are implemented in the form of SpecC behaviors. Recall from section 2.2.2 that SpecC behaviors are the basic unit of functionality in SpecC programs. There are two types of

behaviors: composite behaviors and leaf behaviors. Leaf behaviors may have variables and methods to define their basic functionality. Composite behaviors can have instances of child behaviors and include their functionality by calls to the child's main method. In contrast to SpecC channels, the execution of SpecC behaviors is proactive during system simulation.

The high-level functional structure of the ERTOS-SS DRTOS model is shown in Figure 4.5. In the ERTOS-SS DRTOS model implementation, the top level composite behavior `DRTOS_SERV` consists of two concurrent child behaviors: `DRTOS_SYNC` and `DRTOS_SCHD`, which functions as the DRTOS synchronizer and the global scheduler respectively. The behavior `DRTOS_SYNC` provides the functionality of receiving messages from tasks executing on each PE, updating the semaphore memory block based on the synchronization protocol, and invoking the behavior `DRTOS_SCHD` through an internal event `start`. The behavior `DRTOS_SCHD` is responsible for updating semaphores' waiting and locking lists and sending various messages back to tasks based on the scheduling protocol, and noting its completion to the behavior `DRTOS_SYNC` through another internal event `done`. The processes of both behaviors `DRTOS_SYNC` and `DRTOS_SCHD` are placed in an infinite loop and will continue to run infinitely throughout the simulation and will always be ready to receive more messages from tasks.

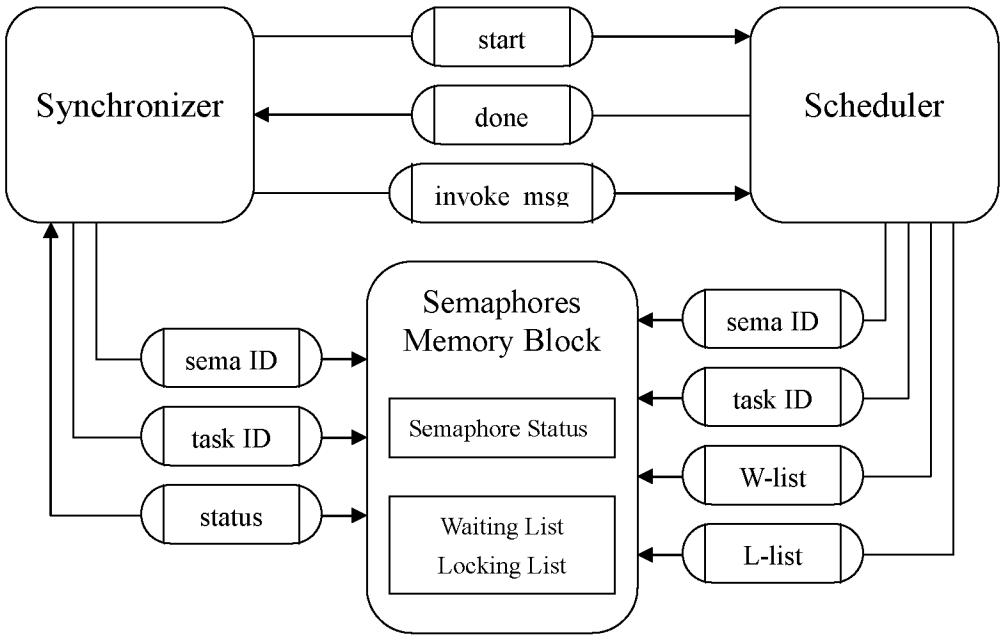


Figure 4.5: Functional model of the behavior DRTOS_SERV

Additionally, a channel `MSG_IO` is created for communication between each processing element and the ERTOS-SS DRTOS model. It has two interfaces `MSG_SEND` and `MSG_RECV` to provide a capability for message exchange between tasks and the DRTOS synchronizer.

The declarations of the behavior `DRTOS_SYNC`, the behavior `DRTOS_SCHD`, and the channel MSG_IO are shown in Figure 4.6.

```
//data type declarations

typedef structure
{
   MSG_Type         msg_type;
   int              task_id;
   int              sema_id;
} MSG_Data_Struct;

//interface declarations

interface MSG_SEND;
interface MSG_RECV;


//channel declarations

channel MSG_IO() implements MSG_SEND, MSG_RECV;


//behavior declarations

behavior DRTOS_SYNC
{
    MSG_RECV in_msg_recv,
    out event start,
    in event done,
    out MSG_Data_Struct invoke_msg
};

behavior DRTOS_SCHD
{
    MSG_SEND out_msg_send,
    in event start,
    out event done
    in MSG_Data_Struct invoke_msg
};
```

Figure 4.6: Declarations of behaviors and channels in the ERTOS-SS DRTOS model

4.3.3 ERTOS-SS DRTOS Model Implementation Summary

The ERTOS-SS DRTOS model implementation is focusing on the extension services of managing various types of task dependencies. Two SpecC behaviors `DRTOS_SYNC` and `DRTOS_SCHD` are developed to provide the ERTOS-SS DRTOS model the capability of synchronization and global scheduling. The functionality of these two behaviors, combined with the interface of the SpecC RTOS channel, may fully provide DRTOS services in system-level design. The refinement rules of inserting the

ERTOS-SS DRTOS model into the system architecture model during system level synthesis will be presented in the next section.

4.4 ERTOS-SS DRTOS Refinement Methodology

Recalling the OS refinement process in section 3.3, the RTOS channel is inserted inside each processing element to provide RTOS services to the system model. In this section, we will begin detailed discussion of the ERTOS-SS DRTOS refinement process. The ERTOS-SS DRTOS refinement process inserts the RTOS channels and the extension DRTOS behaviors into the system architecture model and creates the scheduled model in which all the DRTOS services listed in section 4.1 may be achieved.

Many refinement methodologies may be performed during system level synthesis to move the system models to the lower level of abstraction. In [18], these refinements are classified into three categories: structural reorganization, behavioral refinement, and communication refinement. Structural reorganization groups refinements that modify the hierarchical structure of the modules. Hardware/software partition, allocation and binding belong to this category. Behavioral refinement relates to modifying task descriptions, I/O primitives may be refined to comply with interface constraints and task contents may be modified to comply with semantics at different levels of abstraction. For example, when moving from the driver to the RT-level, computations executed on HW blocks must be scheduled into clock cycles, and the SW code must be adapted to the processor where it will be executed. This may involve adding specific system calls to the embedded OS. Finally, communication refinements modify the topology of ports and/or nets. Based on these three categories of refinements, the ERTOS-SS DRTOS model refinement rules and process are outlined in the subsections below.

4.4.1 Task Allocation in Architecture Refinement

The first step in the ERTOS-SS DRTOS refinement process is the task allocation during architecture refinement. Recall from section 2.2.1 and 3.1 that there are four

system models and three refinement steps in the SpecC system level design flow. Architecture refinement allocates the processing elements and maps the modules of the functional specification model to these processing elements. Therefore, the ERTOS-SS DRTOS refinement process is actually begun at architecture refinement. This architecture partitioning process is illustrated in Figure 4.7. By properly allocating and mapping system modules to the selected processing elements, task allocation and load balancing can be easily accomplished.

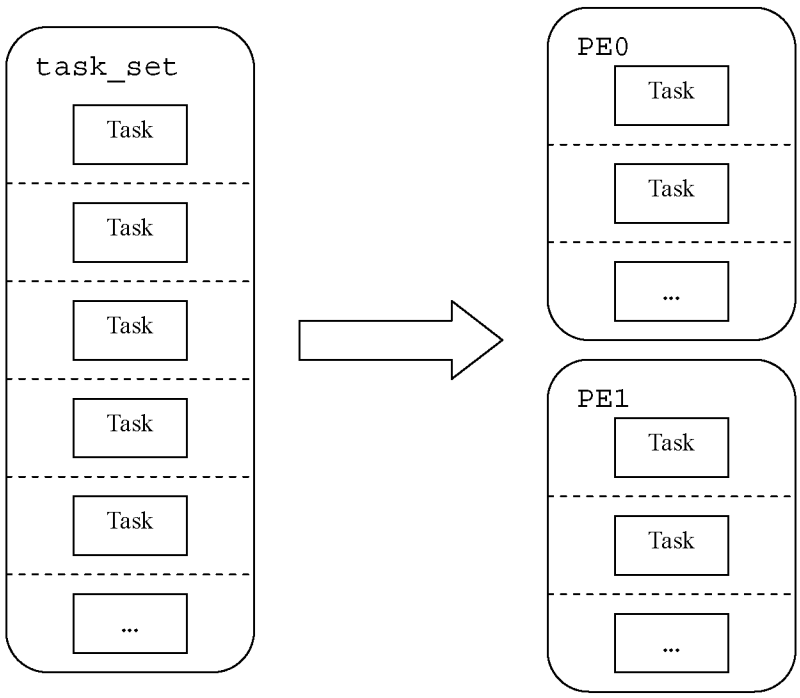


Figure 4.7: Partitioning and mapping tasks onto different PEs

4.4.2 Mapping DRTOS Extension Behaviors

The next refinement step is to map the DRTOS synchronizer and global scheduler behaviors to a new processing element. With the insertion of the ERTOS-SS DRTOS model, two extension behaviors which function as the DRTOS synchronizer and scheduler are added to the system model. Thus, after mapping existing system modules to selected PEs, a new PE to which the DRTOS extension behaviors will be mapped is

needed, as shown in Figure 4.8.

This new processing element can be a processor, a microcontroller, a DSP, or a custom hardware element. The main purpose of this component is to reflect behavior of the DRTOS synchronizer and global scheduler during simulation. In the system implementation model, these DRTOS extension behaviors along with all the relevant functionality of the RTOS channels will be exported into the real DRTOS system calls supported by target microprocessors.

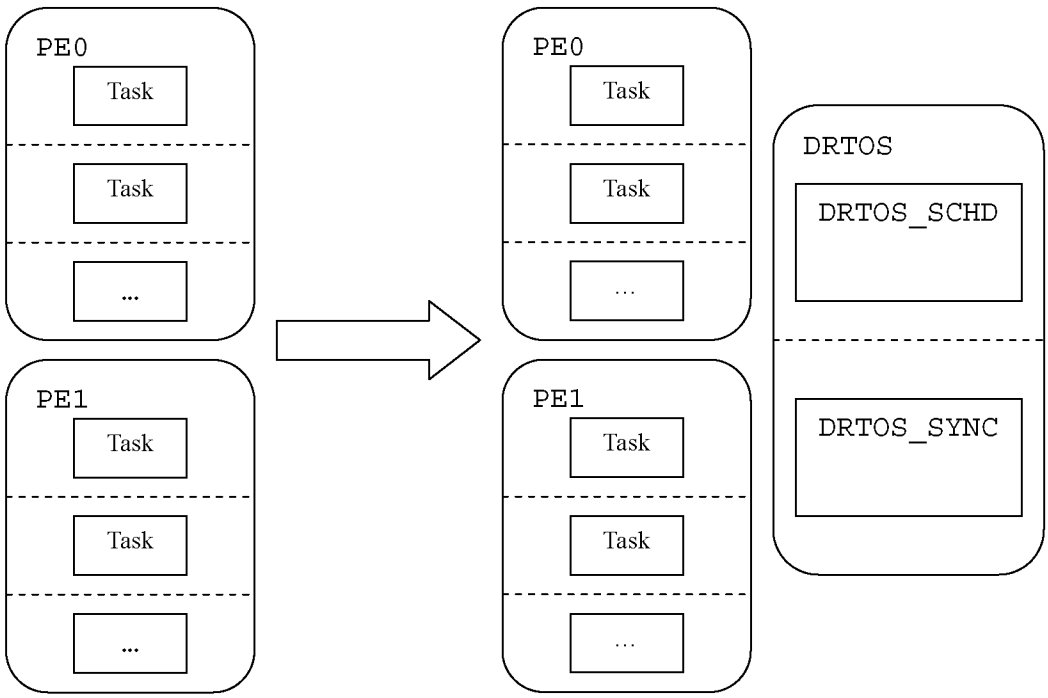


Figure 4.8: Mapping the extension DRTOS behaviors onto a new PE

4.4.3 Inserting RTOS Channel

Another major step in the ERTOS-SS DRTOS refinement process is to insert RTOS channel into each PE. The detailed rules and process of the RTOS refinement can be found in section 3.3. Figure 4.9 below shows the hierarchy of the system model with the insertion of the RTOS channels.

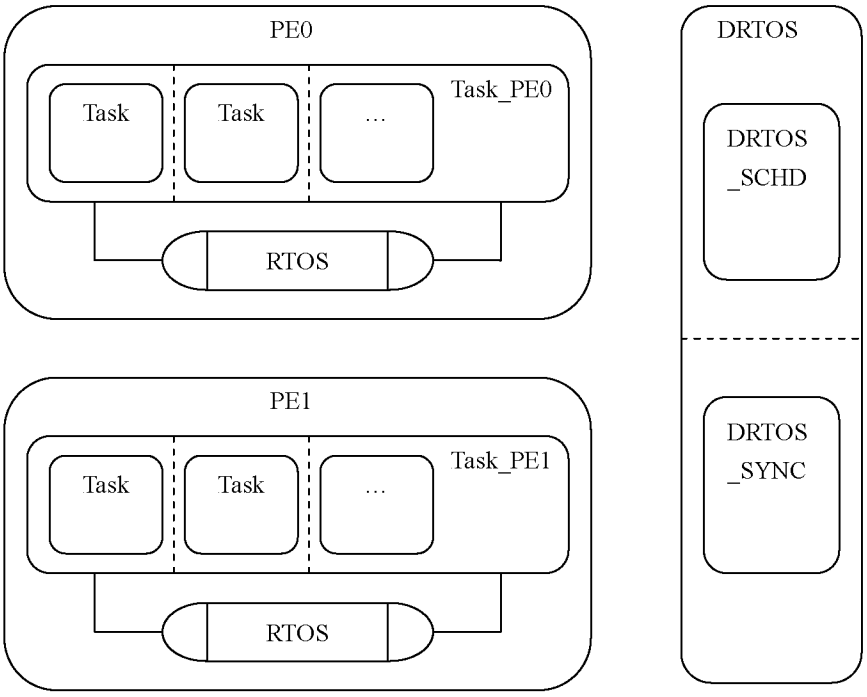


Figure 4.9: Insertion of RTOS channels into system model

4.4.4 Adding Semaphores and Gathering Synchronization Information

After architecture partitioning phase, all the synchronization-related information, such as which task is on which PE or which task depends on which others, must be generated for management of external task dependencies. A semaphore is added for each instance of dependency. The semaphore-related data types used in the ERTOS-SS DRTOS model implementation are shown in Figure 4.10.

```
typedef unsigned int SEMA_Status_Type;
enum
{
    SEMA_STATUS_READY,
    SEMA_STATUS_LOCKING,

    NUM_SEMA_STATUS
};

typedef unsigned int MSG_Type;
enum
{
    MSG_INIT,
    MSG_RUN,
    MSG_SUSPEND,
    MSG_RESUME,
    MSG_REQUEST,
    MSG_RELEASE,

    NUM_MSG_TYPE
};

typedef structure
{
    int              task_id;
    int              pe_id;
} TASK_Loca_Struct;

typedef structure
{
    SEMA_Status_Type curr_sema_sts;
    int              waiting_list[50];
    int              locking_list[1];
    TASK_Loca_Struct  assoc_tasks[50];
} SEMA_Status_Struct;
```

Figure 4.10: Semaphore-related data types used in the ERTOS-SS DRTOS model

Each semaphore has an integer component `curr_sema_sts` to save its current status during execution. Two integer arrays `waiting_list` and `locking_list` are used to save its waiting and locking task IDs. Since a semaphore can be locked only by one task at any point of time, the length of the locking list array is set to one. Here we assume that the maximum number of waiting tasks for any semaphore is fifty, thus the length of the waiting list array is set to fifty. When a semaphore is created, its initial status is `locking`. A "task" called `init` is added into its locking list to indicate that the semaphore is locked by initialization. Obviously its waiting list is empty. At that time if a `request` message is received, the status of the semaphore will be not changed and the request task will be added into the semaphore's waiting list. If a `release` message is

received, the status of the semaphore will be changed from `locking` to `ready`. The task `init` will be removed from the semaphore's locking list. All the semaphores' status, along with their waiting and locking lists, will be controlled by the DRTOS synchronizer and global scheduler behaviors during simulation. Additionally, semaphore status table has another data type `TASK_Loca_Struct` to save the task location information associated with each semaphore. Task location information can be gathered after the architecture partitioning phase. Figure 4.11 below shows an example of task dependencies in a multiprocessor system. Its initial semaphore status table is shown in Table 4.1.

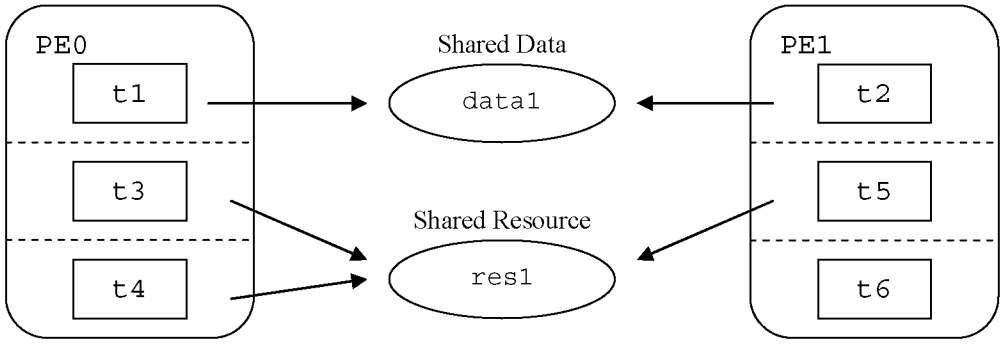


Figure 4.11: An example of external task dependencies

Table 4.1: Initial semaphore status table for the example in Figure 4.11

Semaphores		`sema1` (for `data1`)	`sema2` (for `res1`)
Current Status		`locking`	`locking`
Waiting List	Task ID	(empty)	(empty)
	Task ID		
	...	...	...
Locking List	Task ID	`init`	`init`
Task Location Information	Task ID	`t1`	`t3`
	PE ID	`PE0`	`PE0`
	Task ID	`t2`	`t4`
	PE ID	`PE1`	`PE0`
	Task ID		`t5`
	PE ID		`PE1`
	...	...	...

4.4.5 Adding Distributed Synchronization and Global Scheduling

The final step in the ERTOS-SS DRTOS refinement process is to add distributed synchronization and global scheduling by using channels and behaviors provided in the `DRTOS_SERV`. Distributed synchronization and global scheduling mainly deal with external task dependencies among different PEs and cannot be achieved using SpecC events or RTOS-based events. To handle this aspect, the semaphore status table created in the previous step and semaphore-related messages must be used. During the DRTOS refinement, a hierarchical channel `MSG_IO` is created and added into the system architecture model. `MSG_IO` provides the communication platform between each PE and the PE with the DRTOS model. It allows the exchange of semaphore-related messages between tasks and the DRTOS extension behaviors. Each time when a task requests or releases a semaphore, it sends `request` or `release` messages to the DRTOS extension behaviors through the `MSG_SEND` interface of the `MSG_IO` channel. Alternatively, the DRTOS extension behaviors receive these messages through the `MSG_RECV` interface, processes messages, and send the response messages back to task by calling `MSG_SEND`. The `MSG_IO` channels, along with the DRTOS synchronizer and scheduler behaviors, can provide the ability to model DRTOS distributed synchronization and global scheduling services in system level design.

Figure 4.12 below shows the hierarchy of the completed system scheduled model with the insertion of the ERTOS-SS DRTOS model.

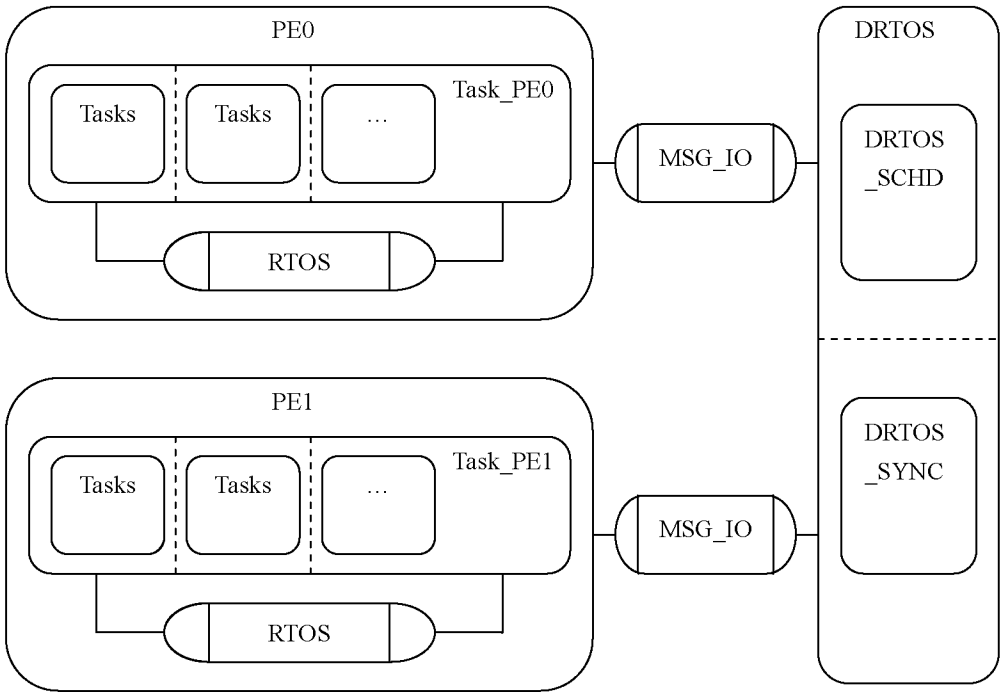


Figure 4.12: Hierarchy of complete system model with the ERTOS-SS DRTOS model

4.4.6 Summary of the ERTOS-SS DRTOS Refinement Methodology

The following is the summary of the guidelines for the ERTOS-SS DRTOS refinement methodology:

- Properly mapping system functional modules to selected processing elements to fulfill task allocation requirements and maintain load balancing.
- Allocating a new processing element and mapping onto it the DRTOS synchronizer and global scheduler behaviors.
- Inserting the RTOS channel into each processing element. This includes:
  - Inserting timing annotations into each behavior to model execution delays.
  - Refining system behaviors to RTOS-based tasks.
  - Replacing SpecC events and event-related primitives with corresponding

event-handling routines of the RTOS channel for intra-processor synchronization.

  - Performing task scheduling with scheduling algorithms supported by the RTOS channel.

- Adding semaphores to manage external task dependencies.
- Gathering synchronization related information for each semaphore based on the architecture partitioning.
- Adding distributed synchronization and global scheduling by using `MSG_IO` channels and DRTOS extension behaviors provided in `DRTOS_SERV`.

5. CASE STUDY

In chapter 4, the ERTOS-SS DRTOS modeling and refinement methodology is presented. In order to demonstrate that the ERTOS-SS DRTOS model can be easily integrated into the existing SpecC system design flow to accurately reflect the DRTOS runtime behavior, we will illustrate the ERTOS-SS DRTOS modeling and refinement methodology in this chapter by analyzing an example of multitask execution on a multiprocessor system.

5.1 Example Task Set

The example task set used in this chapter has six concurrent tasks: `t1`, `t2`, `t3`, `t4`, `t5`, and `t6`, each with two basic execution blocks. All six tasks will be started at time zero. Assume there are two task dependencies in this task set. One is that task `t1` requires the result of a shared data `data1` which will be calculated when task `t5` finishes its first execution block. The other is that task `t4` tries to access a shared resource `res1` which is currently locked by task `t3` and will be released when task `t3` finishes its first execution block. The summary of the task characterizations is listed in Table 5.1.

Table 5.1: Task characterizations of the example task set

Task	Start Time	Execution Blocks
`t1`	0	Wait for result of shared data `data1`; 20; 30;
`t2`	0	30; 40;
`t3`	0	50; Release shared resource `res1`; 10;
`t4`	0	20; Wait to access shared resource `res1`; 20;
`t5`	0	70; Calculate result of shared data `data1`; 30;
`t6`	0	40; 10;

5.2 System Specification Model

In system specification model, each task is implemented as a leaf behavior with SpecC `waitfor` statements used to model its execution blocks. All six leaf behaviors are child behaviors of top-level composite behavior `task_set`. SpecC `par` statement is used in composite behavior for concurrently executing its child behaviors. Task dependencies are implemented using SpecC events with `wait` and `notify` primitives. A high-level representation of the system specification model is shown in Figure 5.1.

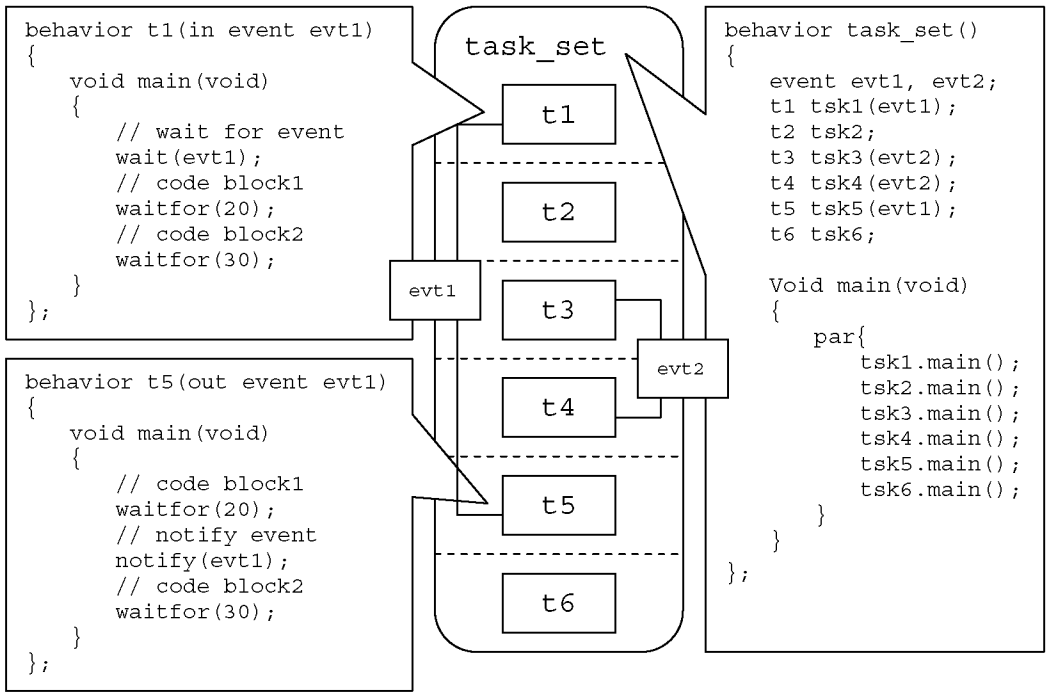


Figure 5.1: Hierarchy of the system specification model

In general, there is no notion of time in the system specification model. However, since the main purpose of this case study is to compare task execution sequences in different system models, we add simulation delays for each task started with the specification model. Figure 5.2 below shows the simulation result of task execution in the system specification model. All tasks start their execution at time 0 expect task `t1`, which is waiting for event `evt2`. At time 20, task `t4` finishes its first execution block

and starts to wait for event `evt1`. At time 50, task `t3` finishes its first execution block, notifies event `evt2`, and starts its second execution block. At the same time, task `t4` can start its second execution block. At time 70, task `t5` finishes its first execution block, notifies event `evt1`, and starts its second execution block. At the same time, task `t1` can start its execution. From the simulation result we can see that without task scheduling, concurrent leaf behaviors can be executed truly in parallel. For example, during time 0 – 20, five tasks `t2`, `t3`, `t4`, `t5`, and `t6` are executed concurrently; during time 70 – 100, two tasks `t1` and `t5` are executed concurrently.

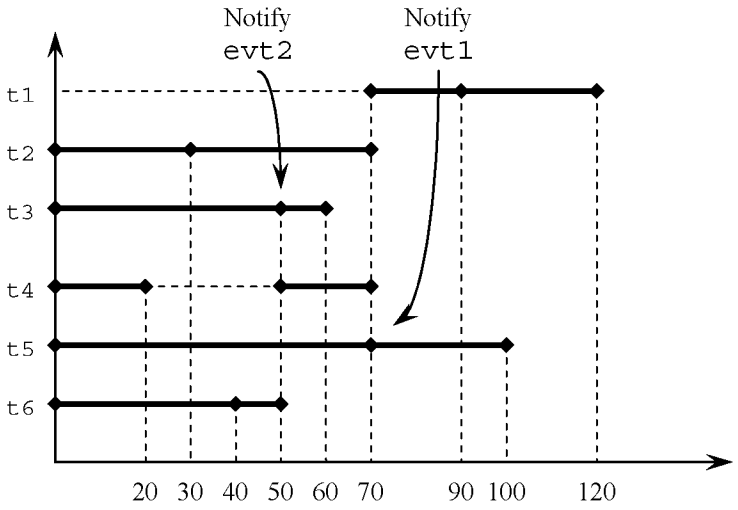


Figure 5.2: Simulation result of the system specification model

5.3 System Architecture Model

During architecture refinement, the system specification model is partitioned into different functional modules and each module is mapped onto a selected processing elements. Additionally, the ERTOS-SS DRTOS refinement methodology requires the task allocation and load balancing to be fulfilled in architecture refinement process. In this example, all six leaf behaviors in the system specification model are partitioned into two composite behaviors: `task_set1` and `task_set2`. Two processing elements, a Motorola DSP 56600 and a Motorola Coldfire processor, were selected and named as

PE1 and PE2 respectively. Behavior task_set1 with three concurrent child behaviors t1, t2, and t3, is mapped onto PE1. Behavior task_set2 with three concurrent child behaviors t4, t5, and t6, is running on PE2. The SpecC-based events used to manage task dependencies are encapsulated into SpecC channels. Figure 5.3 shows the system architecture model generated by architecture refinement.

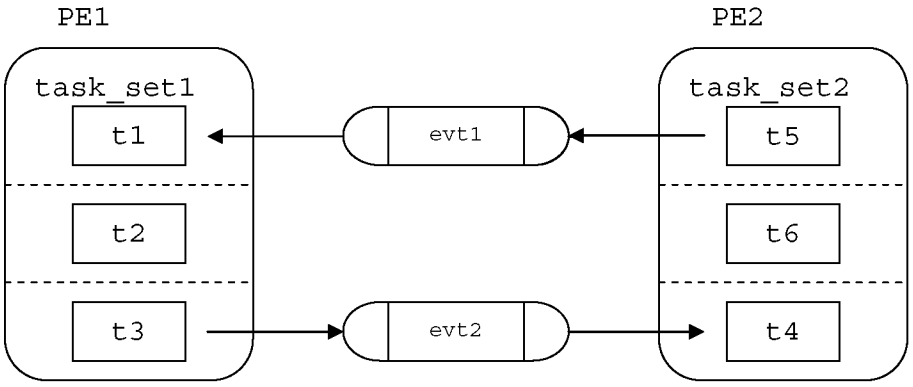


Figure 5.3: Hierarchy of the system architecture model

The simulation result of the system architecture model is shown in Figure 5.4. As we can see, task execution results are similar to those in the system specification model, with the only difference is that the task execution is allocated to two PEs. At time 0, tasks t2 and t3 start their execution on PE1, and tasks t3, t4, and t5 start to execute on PE1. Task t1 is waiting for event evt2. At time 20, task t4 finishes its first execution block on PE2 and starts to wait for event evt1. At time 50, task t3 finishes its first execution block on PE1, notifies event evt2, and starts its second execution block. At the same time, task t4 can start its second execution block on PE2. At time 70, task t5 finishes its first execution block on PE2, notifies event evt1, and starts its second execution block. At the same time, task t1 can start its execution on PE1. From the analysis of the simulation result we can get the same conclusion: because task scheduling is not added into system model, concurrent leaf behaviors can be executed truly in parallel.

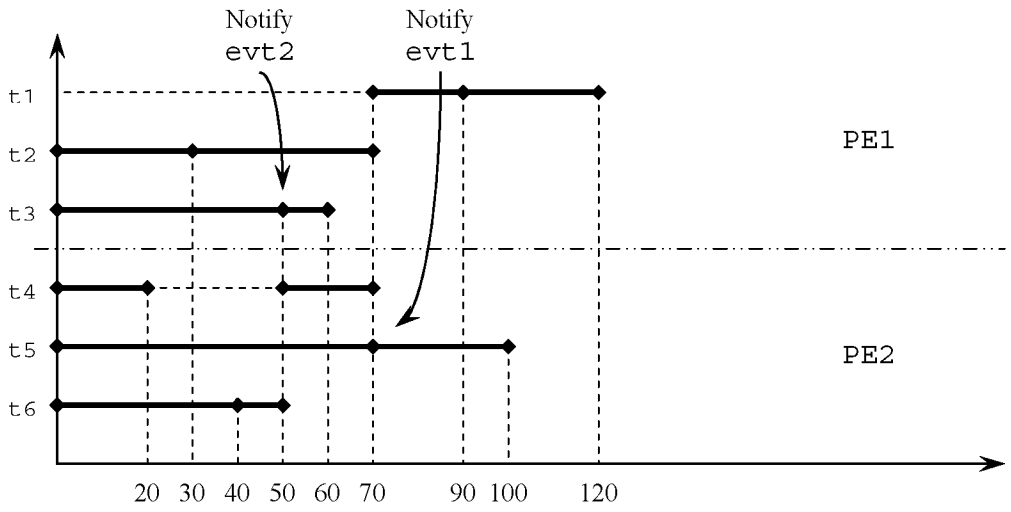


Figure 5.4: Simulation result of the system architecture model

5.4 System Scheduled Model with the RTOS Model

To compare the simulation results of the ERTOS-SS DRTOS model with the SpecC RTOS model, the next step in this case study is to add RTOS model via OS refinement and get the simulation result of the system scheduled model with the RTOS model.

The `time_wait` call is used inside each basic block of leaf behaviors to model the execution delay of the tasks. The priority of each task is specified inside its `os_task_create` method. The RTOS `par_start` and `par_end` system calls are used in composite behaviors `task_set1` and `task_set2` to fork and join the child tasks in the execution of the parent task. The SpecC events `evt1` and `evt2` used in the system architecture model are replaced by RTOS-based events. All SpecC `wait` and `notify` statements are replaced with RTOS `event_wait` and `event_notify` calls.

In the first simulation in this step, we use the same task allocation decision as in the system architecture model. On processor `PE1`, the priorities of task `t1`, `t2` and `t3` are 1, 2, and 3 with `t1` having the highest priority and `t3` having the lowest priority. On processor `PE2`, the priorities of task `t4`, `t5` and `t6` are specified as 1, 2, and 3 with `t4`

having the highest priority and t6 having the lowest priority.

However, deadlock is occurred during the simulation. After analysis we find out that the reason of deadlock is because that the RTOS model only provides scheduling and synchronization inside each PE, but it cannot manage global scheduling and synchronization among different PEs. Thus, events can get lost so specifically since the RTOS model will change the order of task execution, a task may start to wait for an event on a PE which is already released by another task on a different PE. For example, if task t0 on PE0 notifies the event evt0 at time 10 and task t1 starts to wait for event evt0 on PE1 at time 20, event evt0 is lost and deadlock is occurred.

In the second simulation, another task allocation decision is used: tasks t1, t2, and t5 are allocated to PE1, tasks t3, t4, and t6 are allocated to PE2. Thus, there is no event exchange between two different PEs. On processor PE1, the priorities of task t1, t2 and t5 are 1, 2, and 3 with t1 having the highest priority and t5 having the lowest priority. On processor PE2, the priorities of task t3, t4 and t6 are specified as 3, 2, and 1 with t6 having the highest priority and t3 having the lowest priority. The system scheduled model with insertion of the RTOS model is shown in Figure 5.5.

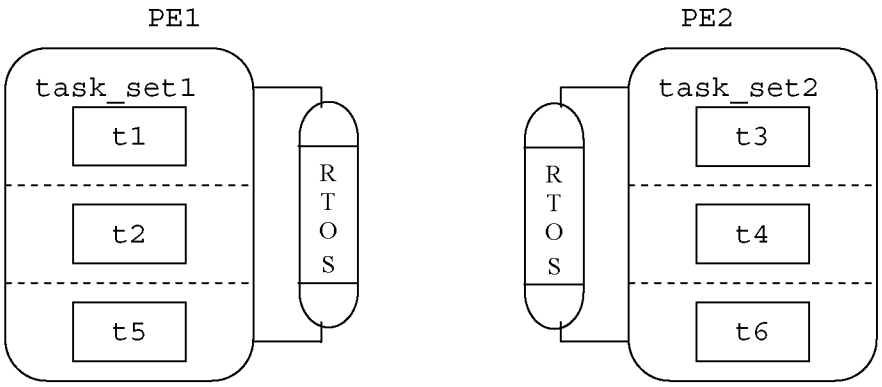


Figure 5.5: Hierarchy of the system scheduled model with the RTOS model

After insertion of RTOS channel in each PE, task management and scheduling are implemented using system calls of the RTOS channel. The priority-based scheduling algorithm is used in this example. Figure 5.6 shows the simulation results of task

execution with the insertion of the RTOS channels. As we can see, tasks are executed on each PE in an interleaved way. On PE1, task t1 has highest priority and should be executed first. But it needs to wait for event evt1. Thus, at time 0, the second highest priority task t2 starts its execution. At time 70, task t2 finishes its execution. Since task t1 is still waiting for event evt1, task t5 starts its execution. At time 140, task t5 finishes its first execution block and notifies event evt1. Then task preemption happens. Task execution of task t5 is suspended and task t1 starts its execution. At time 190, task t1 finishes its execution and task t5 starts its second execution block. At time 220, task t1 finishes its execution. On PE2, task t6 has highest priority and starts its execution at time 0. At time 50, task t6 finishes its execution and the second highest priority task t4 starts its execution. At time 70, task t4 finishes its first execution block and starts to wait for event evt2. Then task preemption happens. Task execution of task t4 is suspended and task t3 starts its execution. At time 120, task t3 finishes its first execution block and notifies event evt2. Then task preemption happens again. Task execution of task t3 is suspended and task t4 starts its second execution block. At time 140, task t4 finishes its execution and task t3 starts its second execution block. At time 150, task t3 finishes its execution.

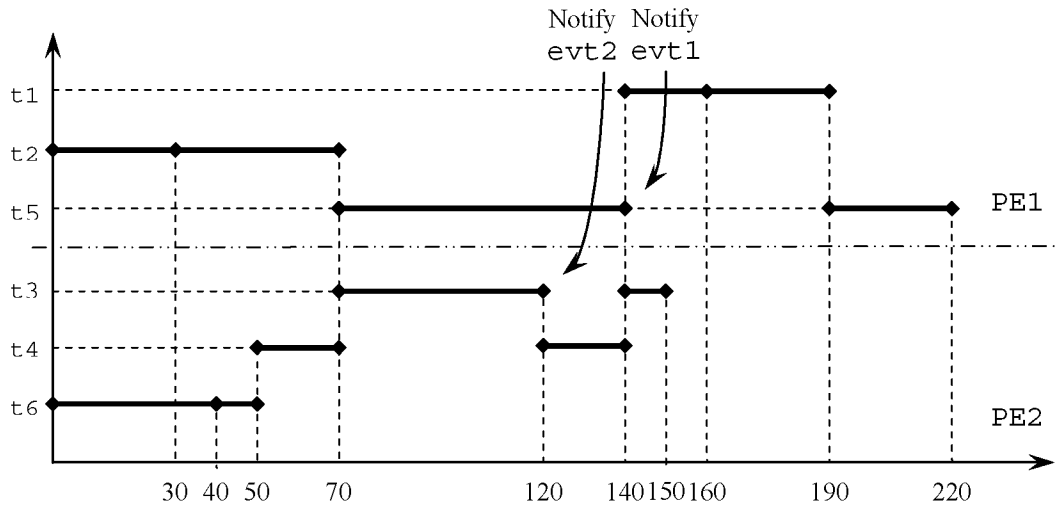


Figure 5.6: Simulation results of the system scheduled model with the RTOS model

5.5 System Scheduled Model with the ERTOS-SS DRTOS Model

The final step in this example is to add the ERTOS-SS DRTOS model into the system architecture model to provide mechanisms for multiprocessor task scheduling and manage dependencies among tasks located on different processing elements.

Recall from section 4.4.2 that the ERTOS-SS DRTOS refinement requires an additional processing element to which the DRTOS synchronizer behavior will be mapped. In this example, a standard hardware element is selected and named as `OS_PE`. The DRTOS extension behaviors `DRTOS_SYNC` and `DRTOS_SCHD` are added into system model and mapped onto `OS_PE` during the ERTOS-SS DRTOS refinement process.

Following the architecture partitioning phase of the ERTOS-SS DRTOS refinement, the next step is to add a set of semaphores to manage external task dependencies in the system. Two semaphores are added in this example: semaphore `sema1` is used to handle dependency between task `t1` and task `t5`, semaphore `sema2` is used to handle dependency between task `t3` and task `t4`. The semaphore status table is used to save semaphore status during system execution. Additionally, task location information is saved in the semaphore status table after architecture refinement. The initial semaphore status table for task set in this example is shown in table 5.2. During simulation, the DRTOS synchronizer and global scheduler behaviors can manage external task dependencies by accessing information in the semaphore status table.

Table 5.2: Semaphore status table

Semaphores		`sema1`	`sema2`
Current Status		`locking`	`locking`
Waiting List	Task ID	(empty)	(empty)
Locking List	Task ID	`init`	`init`
Task Location Information	Task ID	`t1`	`t3`
	PE ID	`PE1`	`PE1`

	Task ID	t5	t4
	PE ID	PE2	PE2

The next step is to add the synchronization channel MSG_IO into each tasks and the DRTOS synchronizer behavior. Two interfaces of the MSG_IO channel provide capability for exchange of semaphore-related messages. The modified code of task t1 that includes the interfaces of the MSG_IO channel is shown in Figure 5.7.

```
//data type declarations

typedef structure
{
   MSG_Type         msg_type;
   int              task_id;
   int              sema_id;
} MSG_Data_Struct;

//interface declarations

interface MSG_SEND;
interface MSG_RECV;


behavior t1(OSAPI os, MSG_IO msg)
{
   int tid;
   MSG_Data_Struct msg_data;

   void os_task_create(void)
   {
      tid = os.task_create("t1", APERIODIC, 0, 500);
   }
   void main(void)
   {
      os.task_activate(tid);
      msg_data.msg_type = MSG_REQUEST;   // request sema
      msg_data.task_id = 1;              // task ID is t1
      msg_data.sema_id = 1;              // sema ID is sema1

      // code block 1
      os.time_wait(50);

      // request sem1
      msg.MSG_SEND(msg_data);

      // code block 2
      os.time_wait(10);

      os.task_terminate();
   }
};
```

Figure 5.7: Updated SpecC code of task `t1`

The finished system scheduled model with insertion of the ERTOS-SS DRTOS model is shown in Figure 5.8.

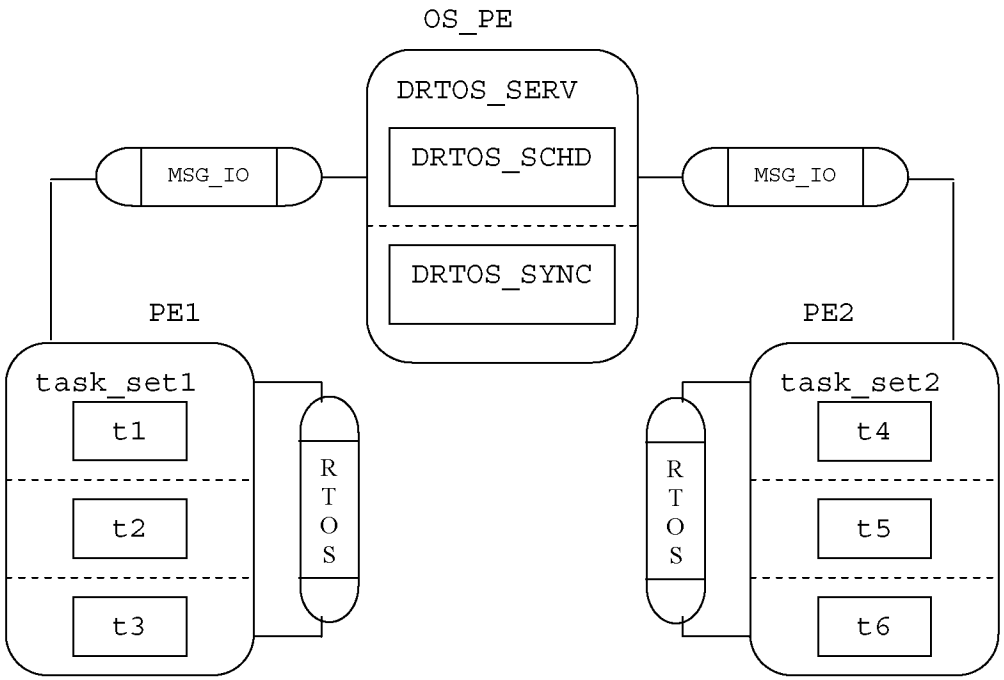


Figure 5.8: Hierarchy of the system scheduled model with the ERTOS-SS DRTOS model

Figure 5.9 shows the task execution results after the insertion of the ERTOS-SS DRTOS model. With the services of task scheduling and synchronization provided by the ERTOS-SS DRTOS model, task execution on each processor is serialized in an interleaved way based on the scheduling algorithms and task dependencies. In this example, a priority based scheduling algorithm is used. At time 0, all six tasks are ready to be executed. On processor `PE1`, task `t1` has the highest priority and should be executed first. But task `t1` is waiting for `sem1` and its execution is suspended. Thus, the task with the second highest priority `t2` will be executed first. On processor `PE2`, task `t4` has the highest priority and will be executed first. At time 20, task `t4` on processor `PE2` finishes its first execution block and will wait for `sem2` to start its second execution

block. Therefore, task preemption happens and task t5 which has the second highest priority will start to execute on processor PE2. At time 70, task t2 has finished its two execution blocks on processor PE1. Since task t1 is still waiting for sem1, task t3 will start to execute on processor PE1. At time 90, task t5 has finished its first execution blocks so it releases sem1. Since task t4 is still waiting for sem2, task t5 will continue executing its second execution block. On processor PE1, since sem1 has been released, task t1 will change its status from "suspended" to "ready". Recall that the task preemption can only happen at the boundary of the basic execution block. Since at that time task t3 is running its first execution block, it will keep executing to time 120. At time 120, task t3 finishes its first execution block and releases sem2. Then task preemption happens and task t1 begins its execution on processor PE1. On processor PE2, task t5 finishes its second execution block. Since sem2 has been released, t4 will begin to execute its second execution block. At time 140, task t4 finishes execution and task t6 starts execution. At time 170, task t1 finishes execution and task t3 starts its second execution block. At time 180, task t3 finishes execution on processor PE1 and at time 190 task t6 finishes execution on processor PE2.

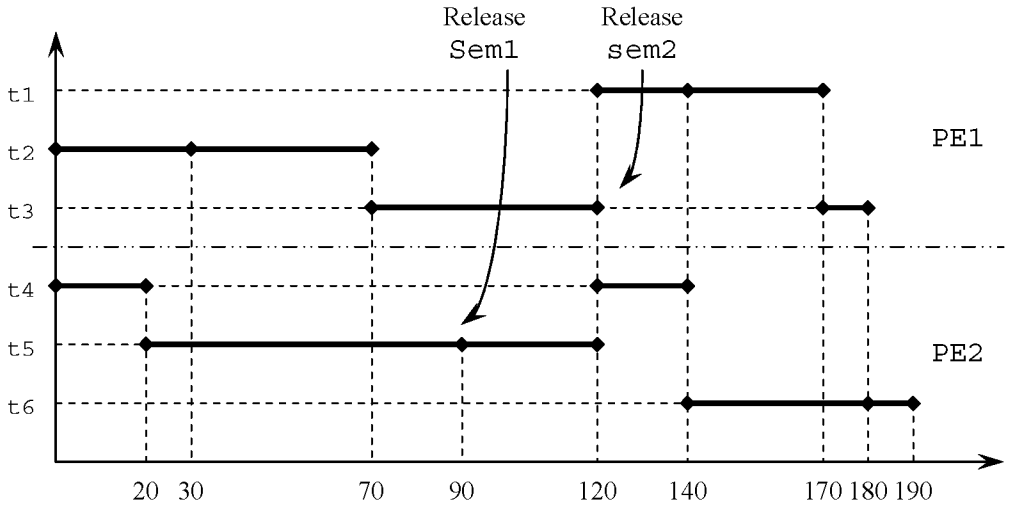


Figure 5.9: Simulation results of system scheduled model with the ERTOS-SS DRTOS model

5.6 Summary of Case Study

The result of this case study shows that in the system specification and architecture models, no task scheduling and synchronization is added. All concurrent tasks can be executed truly in parallel. At any point of time, it is possible that there are more than one tasks are being executed, which is not supported in any OS or real embedded system implementation. Thus, the system specification and architecture model cannot provide ability for modeling OS in system level design.

The SpecC RTOS model can provide basic RTOS services into system model, such as task management, event handling, etc. The insertion of the RTOS model can assistant designers in simulating runtime RTOS behavior during system level synthesis. However, the SpecC RTOS model cannot provide management for global scheduling and synchronization. It lacks support for modeling DRTOS in multiprocessor environment.

The ERTOS-SS DRTOS model is an extension of the SpecC RTOS model with a global synchronizer and a global scheduler. The ERTOS-SS DRTOS model can be inserted into system architecture model for efficient evaluation of multiprocessor task scheduling and synchronization implementation.

6. RELATED WORK

In this chapter, an overview of related research work on system level design methodologies for multiprocessor embedded systems or Multiprocessor System-on-Chip (MPSoC) is presented.

In [19], Reyes et al. introduced a tool called CASSE, what stands for CAmellia System-on-chip Simulation Environment. CASSE is a fast, flexible, and modular SystemC-based simulation environment which aims to be useful for design-space exploration and system-level design at different levels of abstraction. CASSE provides fast simulations and easy architectural modeling by using transaction-level modeling techniques. Moreover, CASSE provides a seamless KPN-derived protocol refinement to cover from application to system implementation. CASSE is being used in the CAMELLIA project, which is focuses on the mapping of innovative smart imaging applications onto an existing video encoding architecture.

The Colif project is presented by Cesario et al. in [18]. The main goal of Colif project is to provide a design representation that is able to model on-chip communication at different levels of abstraction while clearly separating component behavior from the communication infrastructure. Colif is an object-oriented intermediate model that supports multiple communication models at multiple levels of abstraction. Inside Colif model, the levels of abstraction for communication are classified into four categories: service level, message level, driver level, and register transfer (RT) level, with service level being the highest level of abstraction and RT-level being the lowest level of abstraction. The modeling and refinement methodology for using Colif to design communication model at different levels of abstraction are also presented in [18]. Currently, the Colif model is used to build a design flow for Application-Specific Multiprocessor SoC Architectures (ASMSA), and for mixed-level executable model generation.

Another system-level processor/communication co-exploration methodology for

multiprocessor System-on-Chip platforms is presented in [20]. In this co-exploration methodology, the architecture description language (ADL) LISA is used for the system-level description of processor architectures, and SystemC based CA transaction level modeling (TLM) captures the communication architecture and further peripheral devices. The major contribution of this methodology is to provide a retargetable integration of arbitrary LISA-based processor models with SystemC-based communication platform models, as well as a joint top-down refinement and an iterative profiling driven optimization of heterogeneous multiprocessor SoCs.

Other related research projects includes [21], [22], [23], and [24], presented different system level design methodologies for modeling multiprocessor system MPSoC. More generic information about metrics of MPSoC and the future of MPSoC can be found in [25] and [26].

The ERTOS-SS DRTOS modeling and refinement methodology introduced in this thesis is differentiated from other work in that it provides a DRTOS model on top of existing SpecC system level design language and it can be integrated into the existing system level synthesis flow with a minimal effort.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

System level design is one of the main technologies used in today's embedded computing system design and development. The main purpose of system level design is to assist designers in evaluating and optimizing systems early in design exploration. Due to the use of multiple processors in today's complex embedded systems, there is a need to develop a distributed real-time operating system modeling mechanism as part of system level design methodology.

In this thesis, the ERTOS-SS DRTOS modeling and refinement methodology based on SpecC system level design language is presented. The ERTOS-SS DRTOS model can provide the basic functionalities of DRTOS implementation such as multiprocessor task scheduling, inter-processor communication, distributed synchronization and mutual extension, etc. Refinement rules for inserting the ERTOS-SS DRTOS model into existing system level design flow have also been presented. These refinement rules were applied to an example task set to demonstrate the steps of the ERTOS-SS DRTOS refinement process. In summary, the ERTOS-SS DRTOS modeling and refinement methodology is mainly focused on simulating system runtime behavior at the higher levels of abstraction in order to allow designers to validate system functionality, evaluate system performance, and modify design strategies before any implementation has been done.

7.2 Future Work

Based on observations and experience gained in performing this project, several potentially fruitful possibilities for future work may be summarized as follows:

i. Our current DRTOS modeling and refinement methodology is suited for

offline multiprocessor task scheduling algorithms in which the task allocation decision is made before runtime. Since dynamic task allocation and load balancing are the critical components in many online multiprocessor task scheduling algorithms, one possible future topic could be to add support of online scheduling with dynamic mapping system behavior onto different components.

ii. Hardware reconfiguration operating systems [27] [28] are widely used in today's embedded systems to dynamically schedule hardware tasks on custom hardware processing elements such as ASICs and FPGAs. Modeling and refinement methodology for hardware reconfiguration OS is required to reflect its abstract behavior in system level design. Therefore, the extension of our DRTOS modeling and refinement with the support of both conventional OS and hardware reconfiguration OS should be an interesting topic.

iii. The SCE environment can provide convenient tools and options for automatic single RTOS modeling and refinement during system level synthesis. Currently the DRTOS modeling and refinement is manually performed by system designers. One area of future work could lie in developing tools and options for assisting automatic DRTOS modeling and refinement design flows in the SCE environment. The automation of this process would greatly increase the efficiency for system level designs of multiprocessor embedded systems with DRTOS.

iv. Compared to SpecC, the SystemC language has greater industry support and is more widely used by many major vendors. Recently, a great deal of research work has been focusing on developing a modeling framework based on SystemC to support the modeling of multiprocessor-based RTOS's and to provides system designers with a user-friendly and efficient modeling and simulation environment [16]. But to our knowledge, the lack of ability to support the DRTOS modeling on top of the SystemC language and to integrate the DRTOS refinement into the existing SystemC design flow still

need to be addressed. Thus, our DRTOS modeling and refinement methodology could be added into the SystemC language as a new modeling feature, as in SpecC. More details about SystemC language can be found in [29] and [30].

BIBLIOGRAPHY

[1] Robert Dale Walstrom, "System Level Design Refinement Using SystemC", Masters' thesis, Department of Electrical and Computer Engineering, Iowa State University, 2005.

[2] Haobo Yu and Daniel D. Gajski, "RTOS Modeling in System Level Synthesis", CECS Technical Report 02-25, Center for Embedded Computer Systems, University of California, Irvine, August 14, 2002.

[3] Haobo Yu, Andreas Gerstlauer, and Daniel Gajski, "RTOS Scheduling in Transaction Level Models", CECS Technical Report 03-12, Center for Embedded Computer Systems, University of California, Irvine, March 20, 2003.

[4] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, Shuqing Zhao, "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, 2000.

[5] F. Panzieri and R. Davoli, "Real Time Systems: A tutorial", Technical Report UBLCS-93-22, Laboratory for Computer Science, University of Bologna, Italy, October 1993.

[6] "VxWorks Programmer's Guide", Wind River Systems, Inc., available online at: http://www.windriver.com.

[7] M. Singhal and N. Shivaratri, "Advanced Concepts in Operating Systems", McGraw Hill, 2001.

[8] Mikael Åkerholm and Tobias Samuelsson, "Design and Benchmarking of Real-Time Multiprocessor Operating System Kernels", Masters' thesis, Department of Computer Science and Engineering, Mälardalen University, June 2002.

[9] C. Siva Ram Murthy and G. Manimaran, "Resource Management in Real-Time Systems and Networks", the MIT Press, 2001.

[10] Andreas Gerstlauer, "SpecC Modeling Guidelines", CECS Technical Report 02-16, Center for Embedded Computer Systems, University of California, Irvine, April 12,

2002.

[11] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan, "System Design with SystemC", Kluwer Academic Publishers, 2002.

[12] Lukai Cai, Shireesh Verma, and Daniel D. Gejski, "Comparison of SpecC and SystemC Languages for System Design", CECS Technical Report 03-11, Center for Embedded Computer Systems, University of California, Irvine, May 15, 2003.

[13] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski, "SpecC Language Reference Manual", SpecC Technology Open Consortium, December 12, 2002, available online at: http://www.specc.org.

[14] Rainer Dömer, "System-level Modeling and Design with the SpecC Language", doctoral dissertation, Department of Computer Science, University of Dortmund, Germany, 2000.

[15] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Doemer, and Daniel Gajski, "System-on-Chip Environment SCE Version 2.2.0 Beta Tutorial", Center for Embedded Computer Systems, University of California, Irvine, July 23, 2003.

[16] Jan Madsen, Kashif Virk, and Mercury Gonzales, "Abstract RTOS Modeling for Multiprocessor System-on-Chip", in proceedings of 2003 International Symposium on System-on-Chip, pages 147-150, November 2003.

[17] J. Sun and J. Liu, "Synchronization Protocols in Distributed Real-Time Systems", in proceedings of the 16th International Conference on Distributed Computing Systems, pages 38-45, May 1996.

[18] W.O. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A.A. Jerraya, "Colif: A Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design", in the 12th International Workshop on Rapid System Prototyping, pages 110-115, June 2001.

[19] V. Reyes, T. Bautista, G. Marrero, P. P. Carballo, and W. Kruijtzer, "CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip", in Euromicro Symposium on Digital System Design, pages

476-483, September 2004.

[20] A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl, "System Level Processor/Communication Co-exploration Methodology for Multiprocessor System-on-Chip Platforms", IEE Proceedings-Computers and Digital Techniques, volume 152, issue 1, pages 3-11, January 2005.

[21] Kashif Virk and Jan Madsen, "A System-Level Multiprocessor System-on-Chip Modeling Framework", in proceedings of 2004 International Symposium on System-on-Chip, pages 81-84, November 2004.

[22] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, L. Gauthier, and M. Diaz-Nava, "Multiprocessor SoC Platforms: A Component-Based Design Approach", in IEEE Design & Test of Computers, December 2002.

[23] T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, D. Bussaglia, and M. Ariyamparambath, "Virtual Architecture Mapping: A SystemC based Methodology for Architectural Exploration of System-on-Chip", in International Workshop on Systems, Architecture, Modeling and Simulation, Samos, Greece, July 2003.

[24] S. Mahadevan, M. Storgaard, J. Madsen, and K. Virk, "ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of Their Causality", in the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 480-483, September 2005.

[25] M. Issam, G. Guy, A. Mohamed, P.J. Luc, "Metrics for Multiprocessor System on Chip", in the 16th International Conference on Microelectronics, pages 787-791, December 2004.

[26] W. Wolf, "The Future of Multiprocessor Systems-on-Chip", in proceedings of the 41st Design Automation Conference, pages 681-685, June 2004.

[27] C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks", IEEE Transactions on Computers, volume 53, issue 11, pages 1393-1407, November 2004.

[28] C. Steiger, H. Walder, M. Platzner, and L. Thiele, "Online Scheduling and Placement of Real-Time Tasks to Partially Reconfigurable Devices", in proceedings of the 24th

IEEE International Real-Time Systems Symposium (RTSS'03), pages 224-235, December 2003.

[29] "SystemC Version 2.0 User's Guide", updated for SystemC 2.0.1, available online at: http://www.systemc.org.

[30] "Functional Specification for SystemC 2.0", updated for SystemC 2.0.1, version 2.0-Q, April 5, 2002.