Master's Thesis

INTEGRATING FORMAL VERIFICATION INTO THE SOFTWARE DEVELOPMENT PROCESS

by
Rui, WANG

August, 2004

INTEGRATING FORMAL VERIFICATION INTO THE SOFTWARE DEVELOPMENT PROCESS

by

Rui, WANG

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Delft University of Technology

Faculty of Electrical Engineering,
Mathematics and Computer Science

2004

Approved by Prof. Dr. Arie van Deursen
Ir. Hylke W. van Dijk
Ir. F. Ververs
Dr. Ir. A. J. C. van gemnud

Date August 10th, 2004

Delft University of Technology

INTEGRATING FORMAL VERIFICATION INTO THE SOFTWARE DEVELOPMENT PROCESS

by Rui, WANG

Supervisor
Ir. Hylke W. van Dijk

Faculty of Electrical Engineering, Mathematics and Computer Science
Software Engineering Group

A thesis presented on integrating the formal verification into implementation level and architecture level of software development process.

TABLE OF CONTENT

INTRODUCTION 1

CHAPTER 1 BACKGROUND 3

1. Software Development Process 3
2. Software Life Cycle 4
3. Formal Verification 5
4. Model Checking 5

CHAPTER 2 INTEGRATING PROCESS 7

CHAPTER 3 CASE STUDY 9

1. Case Description 9
2. Concerns 10
3. Scheduling 10
4. Scheduling Algorithm 11
    - 4.1 First Come First Service 11
    - 4.2 Round Robin 12
    - 4.3 Fixed Priority Scheduling 13
5. Implementative Preemptive FPS with RR Fashion 15
6. Summary of Scheduling Algorithm 16

CHAPTER 4 VERIFICATION WORK 19

1. Characterizing The Task Automata 19
    - 1.1 Task Model 19
    - 1.2 Generic Timed Automata 20
    - 1.3 Characterizing The Task Template 21
        - 1.3.1 Task States and Their Relationship 21
        - 1.3.2 Task Template 22
    - 1.4 Instantiating Task Automata from The Task Template 23
2. Characterizing Arbiter Automaton 24
3. Verification Model 27
4. Defining Properties for Verification 27
5. Choosing Verification Tool 27

CHAPTER 5 EXPERIMENTS IN IMPELEMENTATION LEVEL 31

1. Implementation Concerns 31
2. Verification Model in Implementation Level 32
    - 2.1 Implementation Task Model 32
        - 2.1.1 Uniforming Implementation Parameters 32

2.1.2 Upgrading Task Model to Implementation Task Model 33
2.2 Implementation Task Template 33
2.2.1 Upgrading Task Template to Implementation Task Template......33
2.2.2 Instantiating Implementation Task Automata 34
3. Implementation Verification Model and Property 34
4. Implementation Verification Result 35

CHAPTER 6 EXPERIMENTS IN ARCHITECTURE LEVEL 36

1. Architecture Concerns 36
2. Feasibility Checking..36
2.1 Feasibility 36
2.2 Liu and Layland Formula..36
2.3 Checking Feasibility of Architecture Concerns 37
3. Verifying Feasibility 37
3.1 Verifying FCFS Scheduling 38
3.1.1 Task Template and Arbiter Automaton for FCFS...........................38
3.1.2 FCFS Verification Model and Property..41
3.1.3 FCFS Verification Result..42
3.2 Verifying RR Scheduling..43
3.2.1 Task Template and Arbiter Automaton for RR 43
3.2.2 RR Verification Model and Property 46
3.2.3 RR Verification Result 46
3.3 Verifying Preemptive FPS Scheduling 48
3.3.1 Task Template and Arbiter Automaton for Preemptive FPS 48
3.3.2 FPS Verification Model and Property..50
3.2.3 RR Verification Result 51
4. Summary 52

CHAPTER 7 CONCLUSIONS AND FUTURE WORK 53

Bibilography 55
Other Resourses 57
Appendix A RR Veification Results 59

LIST OF FIGURES

Figure 1.1 V-Model 4

Figure 2.1 Integrating Formal Verification into the SDLC of V-Model 7

Figure 2.2 Integrating Process 8

Figure 3.1 An Image with Spiral Dots 9

Figure 3.2 Scheduling Illustration for FCFS 12

Figure 3.3 Scheduling Illustration for RR 13

Figure 3.4 Scheduling Illustration for FPS in Non-Preemptive Mode 14

Figure 3.5 Scheduling Illustration for FPS in Preemptive Mode 14

Figure 3.6 Scheduling Illustration for FPS in Periodic Preemptive Mode 15

Figure 3.7 Scheduling Illustration for FPS with Round Robin Fashion 16

Figure 4.1 Relationships of Task Model, Task Template, and Task Automata 20

Figure 4.2 Task State Relationship Model 21

Figure 4.3 Task States and Parameters 21

Figure 4.4 Task Template (*task_template*) 22

Figure 4.5 Structure of *Select* 23

Figure 4.6 Arbiter Automaton (*arbiter*) 24

Figure 4.7 Structure of *Qtable* 25

Figure 51 Execution Time and Sleeping Time of **TransMotor** 32

Figure 52 Task Template in Implementation Level (*imp_task_template*) 34

Figure 6.1 Task Template for FCFS (*FCFS_task_template*) 39

Figure 6.2 Arbiter Automaton for FCFS (*FCFS_arbiter*) 39

Figure 6.3 Structure of *queue* for FCFS 40

Figure 6.4 Task Template for RR (*RR_task_template*) 44

Figure 6.5 Arbiter Automaton for RR (*RR_arbite*) 44

Figure 6.6 Structure of *ready_flag* for RR 45

Figure 6.7 Structure of *queue* for RR 45

Figure 6.8 Task Template for Preemptive FPS (*FPS_task_template*)...................48

Figure 6.9 Arbiter Automaton for Preemptive FPS (*FPS_arbiter*) 49

Figure 6.10 Structure of *priority* and *pri* for Preemptive FPS...............................49

LIST OF TABLES

Table 3.1 Parameters of Small Example 11

Table 3.2 Parameters for FCFS 12

Table 3.3 Parameters for RR 13

Table 3.4 Parameters for FPS 13

Table 3.5 Parameters for FPS with Round Robin Fashion 16

Table 4.1 Task Models Table 19

Table 4.2 Tools Basic Comparisons 28

Table 4.3 Tools Applied System Comparisons 28

Table 4.4 Tools Integrating Environment Comparisons 28

Table 5.1 Original Task Parameters in Implementation Level 31

Table 5.2 Uniformed Task Parameters in Implementation Level 33

Table 5.3 Implementation Task Models Table 33

Table 6.1 Original Task Parameters in Architecture Level 36

Table 6.2 Architecture Task Models Table 38

Table 6.3 FCFS Verification Results Table 42

Table 6.4 RR Verification Results Table (Quantum=2ms) 47

Table 6.5 Preemptive FPS Verification Results Table 51

Abstract

This thesis aims at integrating Formal Verification into the software development process. Pursuing exhaustive verification, we select Model Checking, one of the Formal Verification methods, as our method for the verification work. We introduce a conceptual model to realize the process of integrating. Later on, a case study, the ELM-2 project, is chosen to study the practicability and details of the conceptual model.

In the analysis, the integrating work in the implementation level of the case study is not satisfied, because we don't get any verification results from UPPAAL, a Model Checking tool applied in the verification work. Contrarily, we get some experiments from the integrating work in the architecture level. Therefore, we conclude here that the conceptual model is practical, but the resulting verification models and available verification tools limit the scope of the applicability of formal verification methods.

INTRODUCTION

With the rapidly increasing demand for the large scale and complex embedded software, system complexity and hence the likely number of design errors, grow exponentially with the number of interacting system components. At the same time, time-to-market constraints are tight due to competition and profit driven market so that the time left for testing is not enough to validate whole system thoroughly. Moreover, traditional debugging and validation techniques based on Simulation and Testing, are inadequate for detecting errors and ensuring system reliability. Therefore, it is necessary to involve a new technique to facilitate checking correctness and reliability of embedded software systems and detecting errors.

Comparing with Testing, we select Formal Verification [VSS95] as the technique of ensuring the correctness of embedded systems. Formal Verification is the process of checking whether or not a design satisfies the requirements of system performance and behavior. For example, the coffee machine will pour a cup of coffee, after you pay the money. Formal Verification guarantees system correctness by exhaustively checking all possible circumstances. We apply Model Checking [CES86], one of the Formal Verification methods, to the development phases of V-Model [7]. Model Checking is an algorithmic method of verifying the system correctness automatically. Given a model of a system and a property stated in some appropriate logical formalism such as Temporal Logic [Pnu77], Model Checking automatically checks the property. The Model Checking results reflect whether the model satisfies the property, namely whether the designed system satisfies system performance or behavior requirements. Findings derived from the results could be fed back to the software development engineers, so that the software development engineers could modify, adjust, or even redesign the system in order to meet the requirements.

This thesis aims at integrating Formal Verification into the software development process. The thesis is separated into seven Chapters. In Chapter 1, a brief introduction about the thesis background is presented, including the software development processes, software development life cycle, V-model, Formal Verification, and Model Checking. In Chapter 2, we raise questions on the integration of Formal Verification. Later on, a solution is given. In Chapter 3, we choose a case study to check the practicability and details of the solution. We present the concerns of the case study and also give a brief introduction of information that the case study involves. In Chapter 4, we characterize the verification model from the concerns. The verification model will be used to study the solution. In Chapter 5 and 6, we present experiments gained from

implementation level and architecture level of the case study. In Chapter 7, the last Chapter, we draw our conclusions from the experiments and list future works that we ought to do in the future studies.

CHAPTER 1

BACKGROUND

1. Software Development Process

Software development is a complicated and error-prone process. It requires careful planning and execution to meet the goals [Fug00]. As Fuggetta said "Developing software is not just a matter of creating effective programming languages and tools. Software development is a collective, complex, and creative effort." This is particularly true for large and complex systems, especially for embedded systems, which are notorious for their harsh requirements. For instance, embedded systems require strong resource constrains, complicated environment, limited behaviors, and software dedicated to proprietary hardware. Most of modern software are developed under a structured process, which is "a coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product."

Commonly, the software development process can be structured into five phases, which are

- Requirement Analysis,
- Architecture Design,
- Implementation,
- Testing, and
- Maintenance.

The activities of the Requirement Analysis focus on providing description of users' needs, and documenting all functions, performances, and interfacing requirements of the software. The Architecture Design activities not only specify the system architecture and modules based on the documented requirements, but also outline data structures, procedural details and interface characterizations. In the work of Architecture Design, the requirements are transformed into a representation of the software before the actual implementation begins. After determining the architecture of the system, the software development process goes into the Implementation phase. In the Implementation phase, the source codes will be written. In the Testing phase, the system will be tested in order to make sure that the developed system is in accordance with the requirements documented in the requirement analysis phase. Meanwhile, the errors will be detected and fixed as many as possible. The activities of the Maintenance phase are to modify the product, to correct errors,

to improve performance or other attributes after delivery, or to adapt the product to a changed environment.

2. Software Life Cycle

Software life cycle [ER03] is a general model of the software development process, such as Waterfall Model [Roy70], Spiral Model [Boe88], and V-Model [7]. The software life cycle not only classifies all the activities and required work products to several different phases, but also defines the principles and guidelines according to which these different phases have to be carried out.

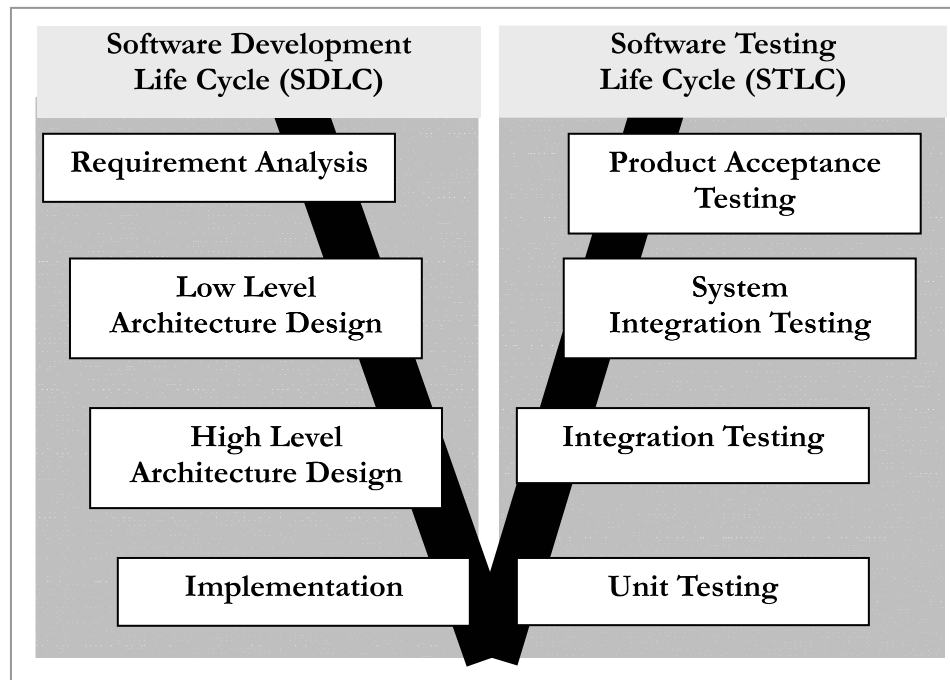

Figure 1.1 V-Model

The V-Model, shown in Figure 1.1, is a variant of the Waterfall Model. It is developed by the German Ministry of Defense and has been widely used. The V-Model consists of eight phases. They are Requirement Analysis, Low Level and High Level Architecture Design, Implementation, Unit Testing, Integration Testing, System Integration Testing, and Product Acceptance Testing. These phases could be separated into two parts. The phases of left side are software development life cycle (SDLC), and the phases of right side are software testing life cycle (STLC). The V-Model checks the system correctness by Testing. When problems are found by Testing in the STLC, they will be fed back to the SDLC to be fixed and improved.

3. Formal Verification

In the V-Model, testing is an important way to detect errors and to validate the correctness of a system. However, testing is generally used after completing the implementation. The coverage of testing is limited, which means testing only checks parts of possibilities of system behaviors. It is the reason why some undesired or unexpected behaviors still may happen, when a presumably correct system proven by the testing is began to use. For example, the bug in Intel's Pentium-II floating-point division unit in the early 1990's was caused by multiplier table that was not thoroughly verified [JPK99].

As the embedded system is notoriously difficult to implement, and an error free development could help to make the development process efficient. It is necessary to involve a technique to guarantee system correctness. Formal Verification is a good choice for such a purpose.

Formal Verification is the process of checking whether or not a design satisfies the requirements of system performance and behavior. For example, the coffee machine will pour a cup of coffee, after you pay the money. Formal Verification guarantees system correctness by exhaustively checking all possible circumstances. Therefore, we are going to apply a Formal Verification method, Model Checking, to the SDLC of the V-Model.

4. Model Checking

Model Checking was introduced in the early eighties. It was first developed by academic research teams and awarded the 1998 ACM Paris Kanellakis Award for Theory and Practice. Model Checking has proven to be a successful method and has frequently used to uncover well-hidden bugs in sizeable industrial cases. For instance, Intel uses Model Checking to verify their new chip designs. Model Checking checks specifications under all possibilities automatically. When a specification is found not to hold, model checking will result in a counterexample (e.g., an evidence of the offending behavior of the system).

Model Checking is an algorithmic method automatically verifying the correctness of software and reactive systems. A reactive system is a system consisting of several components and these components are designed to interact with one another and with the system's environment. The output of one component can be used as the input for another component. For example, a traffic light controller is a typical reactive system.

C h a p t e r   2

INTEGRATING PROCESS

In Chapter 1, we pointed out the weakness of Testing, and decided to integrate Formal Verification into the SDLV of the V-Model, namely Requirement Analysis, Architecture Design, and Implementation. Figure 2.1 presents an overview of the integration.

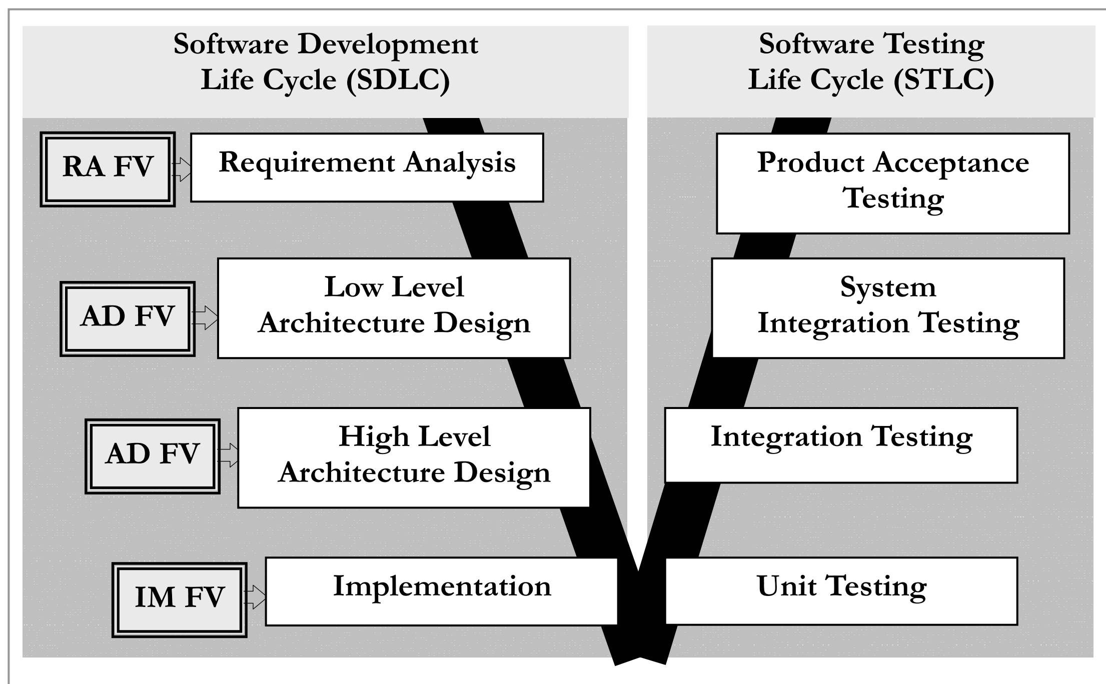

Figure 2.1 Integrating Formal Verification into SDLC of V-Model

After we decided to integrate Formal Verification, mainly Model Checking, into the SDLC, we face two problems:

1. How to integrate formal verification into the software development process?
2. Whether the formal verification is able to assist the software development?

For resolving these two problems, we first construct a conceptual model. Figure 2.2 is the conceptual model of integrating, named Integrating Process.

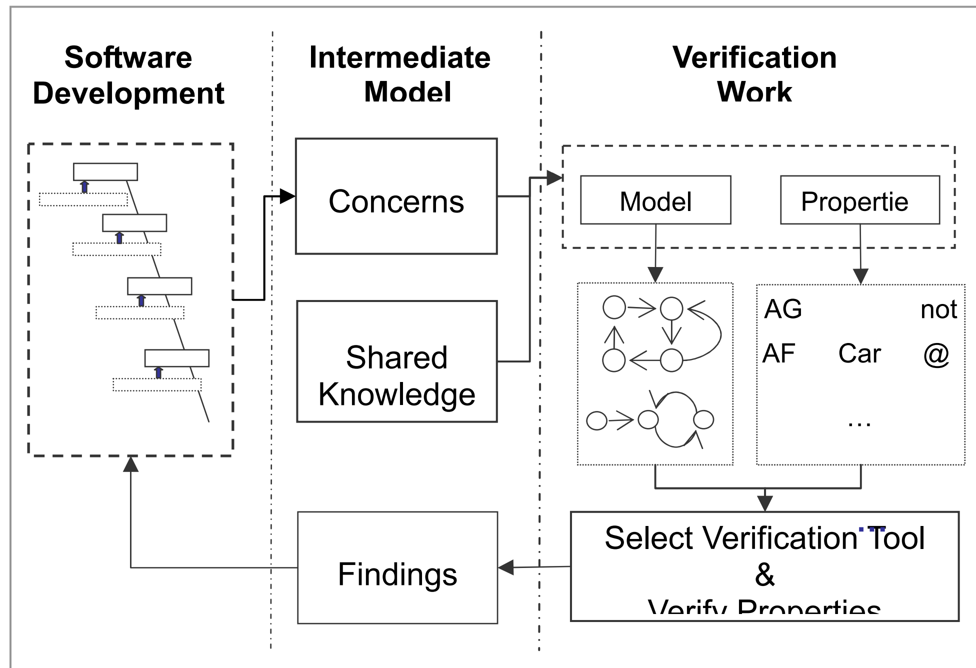

Figure 2.2 Integrating Process – Conceptual Model

The Integrating Process includes three parts.

- The left part is the software development. It is the SDLC of the V-Model.
- The right part is the verification work. In this part, the verification engineers use tools of Model Checking to verify the properties.
- The middle part is intermediate model. This part connects the left part and the right part together.

When an Integrating Process begins, the software development engineers in the left part provide their concerns to the verification engineers. Combining with necessary shared knowledge, the verification engineers characterize a model and a set of properties from the concerns. According to the model and the properties, the verification engineers select a Model Checking tool. Using the tool, the properties are be verified. After the verification engineers get results of verifying, they feed back their findings to the software development engineers.

In following Chapters, we will use a case study to study the practicability and the details of the Integrating Process.

C H A P T E R 3

CASE STUDY

1. Case Description

For checking the practicability and the details of the Integrating Process, we choose the ELM-2 project as our case study. ELM is short for Experiments with Lego MindStorm. The first version of the ELM (or ELM-1) project started in January 2003 and completed before the summer of 2003. The ELM-1 project developed a copier machine using Lego MindStorm technology.

The second version of the ELM (or ELM-2) project is the continuation of the ELM-1 project. It focuses on developing a spinning image scanner that is considered as the upgrade of the scanning part of the copier artifact as developed in the ELM-1 project. The spinning image scanner is a device to read information from an image with spiral dots, shown as Figure 3.1.

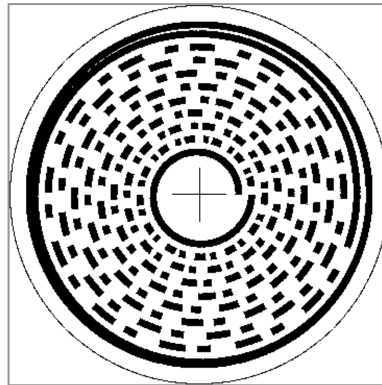

Figure 3.1 An Image with Spiral Dots

In the ELM-2 project, the system of the spinning image scanner is realized with four independent periodical tasks, including

- controlling a motor to move along with the spiral dots,
- tracking positions via sensors,
- reading information from the dots, and
- playing the information.

In our case study, the implementation engineers care whether they can be scheduled in a single CPU. The architecture engineers care two questions:

- How many CPUs does the spinning image scanner require to

accomplish these tasks?
- How to schedule these tasks using what policy?

2. Concerns

The three questions asked by the implementation engineers and the architecture engineers service as one part of the concerns in the Integrating Process. The four tasks mentioned in the case description service as the other part of concerns. They are typically periodical tasks.

A periodical task can be completely described with four parameters, including priority, execution time (or running time), period, and deadline. The priority, denoted by p, is used to measure the importance or urgency of a task. The priority usually is expressed in a non-negative integer value. The greater the integer is, the higher the priority is. One task with higher priority means that the task is more urgent than the other tasks. The execution time, denoted by C, is the amount of time required to complete the task's execution. The period, denoted by T, is the amount of time of an interval taken to complete one execution cycle of a periodical task. The deadline, denoted by D, is the constraint on the latest time at which the task's execution must complete within a period. In addition, we specify that within a period the amount of time that the task is not running is called sleeping time, denoted by S.

In the ELM-2 project, the software development engineers specify that the deadline of each task is always equal to the period. Thus, three parameters is enough to describe a task.

3. Scheduling

The scheduling services as the shared knowledge in the Integrating Process, because all of the questions in the ELM-2 project with respect to it.

The scheduling [KZ95] refers to assign priorities to each task via a policy (or scheduler) and to select next task to execute via a mechanism (or arbiter) according to the priorities.

The scheduler assigns the priority to a task established on the factors that effect importance or urgency of the task, such as the deadline and the period. The priority assignment could be static (or fixed) or dynamic. [KZ95]

- In the static mode, the scheduler assigns a priority to a task only one time, and never changes it. Examples include First Come First Service, Round Robin, Rate monotonic Scheduling, Deadline Monotonic

Scheduling, and Fixed Priority Scheduling.
- In the dynamic mode, the scheduler might change the priorities from request to request, such as Earliest Deadline First, and Least Laxity First.

The arbiter selects the next task with the highest priority from a set of ready-to-run (or runnable) tasks to execute. The selection can be preemptive or non-preemptive.
- In the preemptive mode, the arbiter may preempt the control of the CPU from a task with lower priority and grant the control of the CPU to a task with a higher priority. Whenever a task becomes ready to run, if its priority is higher than the running task's priority, the preemption occurs.
- In the non-preemptive mode, the arbiter will never preempt the control of the CPU from any tasks, no matter which task becomes ready to run. The running task releases the CPU voluntarily.

For example, the Round Robin works in the preemptive mode. The First Come First Service works in the non-preemptive mode. And the Fixed Priority Scheduling may work in either the preemptive or the non-preemptive mode.

4. Scheduling Algorithm

In the architecture level verification, we will use three scheduling algorithms to check the scheduling questions in the ELM-2 project, including First Come First Service, Round Robin, and Fixed Priority Scheduling. We illustrate these scheduling algorithms with a small example.

In the example, there are three tasks, named **A**, **B**, and **C**, respectively. Let the unit of time be millisecond (ms). Their execution time, period, and deadline are list in the Table 3.1. Their priorities will be determined later.

Table 3.1 Parameters of Small Example

Task	Priority	Execution Time	Period	Deadline
A		8 ms	50 ms	50 ms
B		4 ms	50 ms	50 ms
C		9 ms	50 ms	50 ms

4.1 First Come First Service

First Come First Service (FCFS) scheduling algorithm works in static and non-preemptive mode. The task that arrives first gets the highest priority. Once the task is running, it will continue to run until it releases the CPU voluntarily. If two or more tasks have the same arrival time, FCFS is non-deterministic,

which means FCFS selects a task from them randomly.

In the small example, suppose that the time *t* of the first arrived task, **A**, is recoded as *t=0ms*. **B** arrives when *t=1ms*, and **C** arrives when *t=7ms*. According to FCFS, the scheduler assigns the highest priority to **A**, the medium priority to **B**, and the lowest priority to **C**, listed in Table 3.2. The scheduling is illustrated in Figure 3.2.

Table 3.2 Parameters for FCFS

Task	Priority	Execution Time	Period	Deadline
A	3	8 ms	50 ms	50 ms
B	2	4 ms	50 ms	50 ms
C	1	9 ms	50 ms	50 ms

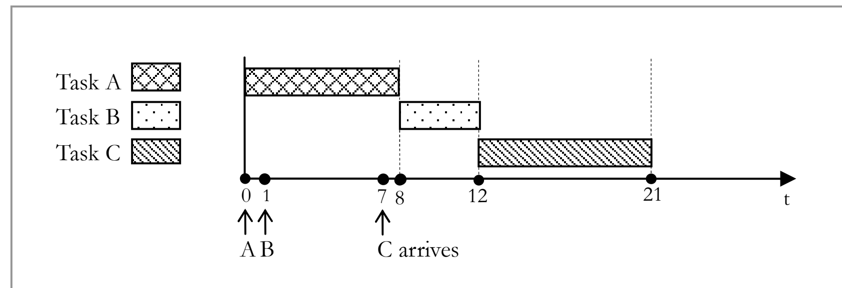

Figure 3.2 Scheduling Illustration for FCFS

4.2 Round Robin

Round Robin (RR) scheduling is a simple and widely used scheduling algorithm. It works in static and preemptive mode. All ready-to-run tasks are kept in a circular queue. They are assigned an identical priority by the scheduler. The arbiter goes around the circular queue and allocates the CPU to each task for a small unit of time, called timeslice or quantum. The arbiter schedules tasks at regular intervals, i.e. scheduling points. Every scheduling point, the arbiter picks the task from the head of the circular queue and sets a timer to interrupt after one quantum. If a task is still running at the end of the quantum, the arbiter interrupts execution of running task and adds the task to the tail of the circular queue. If the task completes execution before the end of the quantum, the task releases the control of the CPU voluntarily.

In the small example, suppose that the initial order of all ready-to-run tasks in the circular queue is **CBA**. **C** is at the head of the circular queue, and **A** is at the tail of the circular queue. Let the quantum be 2ms. Their priorities are 1. See Table 3.3. The scheduling is shown in Figure 3.3.

Table 3.3 Parameters for RR

Task	Priority	Execution Time	Period	Deadline
A	1	8 ms	50 ms	50 ms
B	1	4 ms	50 ms	50 ms
C	1	9 ms	50 ms	50 ms

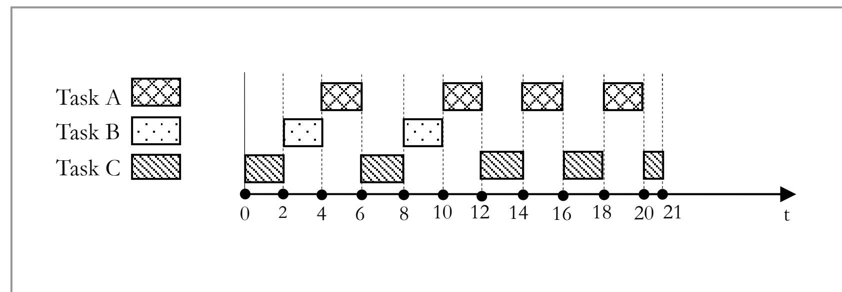

Figure 3.3 Scheduling Illustration for RR

4.3 Fixed Priority Scheduling

Fixed Priority Scheduling (FPS) is a static scheduling. The arbiter can be used in three modes, non-preemptive, preemptive, or periodical preemptive.

- In non-preemptive mode, the running task will not release the CPU until it completes execution.
- In preemptive mode, the arbiter will preempt a running task whenever a higher priority task becomes ready to run.
- In periodical preemptive mode, tasks are scheduled at periodic intervals (so-called slides).

In FPS, the arbiter only needs to know about priorities and it always chooses the highest priority task when a task selection is made.

In the small example, suppose that **A**'s priority is 1, **B**'s priority is 2, and **C**'s priority is 3. See Table 3.4.

Table 3.4 Parameters for FPS

Task	Priority	Execution Time	Period	Deadline
A	1	8 ms	50 ms	50 ms
B	2	4 ms	50 ms	50 ms
C	3	9 ms	50 ms	50 ms

FPS: Non-Preemptive Mode

According to FPS, if the three tasks are scheduled in non-preemptive mode, **C** runs firstly due to its highest priority. When **C** finishes execution and releases the control of the CPU voluntarily, **B** begins to run. After **B** releases the control of the CPU voluntarily, **A**. runs. The scheduling is illustrated in Figure 3.4.

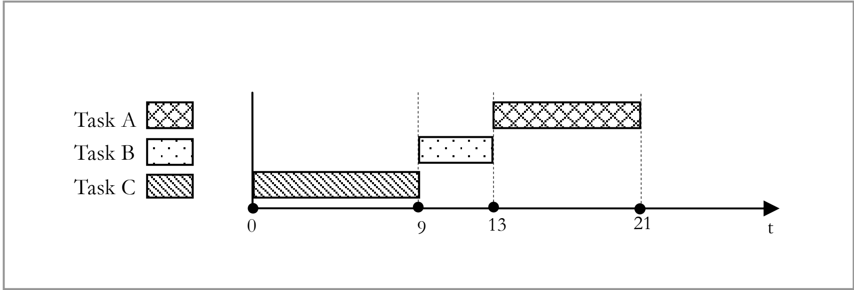

Figure 3.4 Scheduling Illustration for FPS
in Non-Preemptive Mode

FPS: Preemptive Mode

When FPS works in preemptive mode, the arbiter allows preemption. The scheduling of the three tasks is different from that in non-preemptive mode. In order to illustrate the preemption, we assume that **A** and **B** are ready to run when *t=0ms*, and **C** becomes ready when *t=7ms*, shown in Figure 3.5.

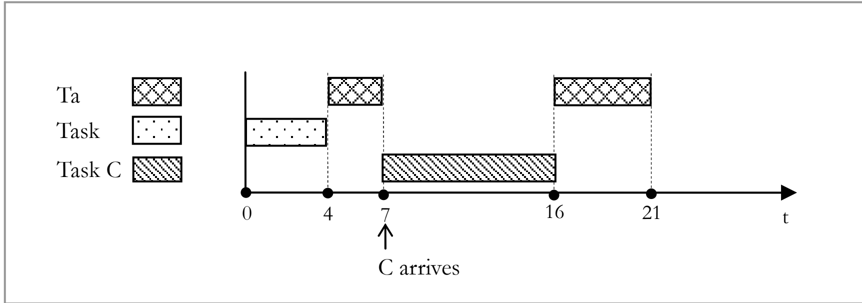

Figure 3.5 Scheduling Illustration for FPS
in Preemptive Mode

At the beginning, there are two ready-to-run tasks, **A** and **B**. Because the priority of **B** is higher than **A**, the arbiter selects **B** to run first. When *t=4ms*, **B** finishes execution and the arbiter gives the control of the CPU to **A**. Unfortunately, when *t=7ms*, **C** becomes ready to run. Because **C**'s priority is higher than **A**'s, the arbiter preempts the CPU from **A** and lets **C** run. When *t=16ms*, **C** finishes execution and **A** gets back the control of the CPU. **A** completes execution when

t=21ms.

FPS: Periodic Preemptive Mode

When FPS works in periodic preemptive mode, tasks are scheduled at periodic intervals. Reconsider the small example of three tasks. Suppose the periodic interval is 2ms, which means the scheduling points are 0, 2, 4ms, and so on. In order to illustrate the preemption, we assume that **A** and **B** are ready to run when *t=0ms*, and **C** becomes ready when *t=7ms*. Hence, the scheduling can be shown in Figure 3.6.

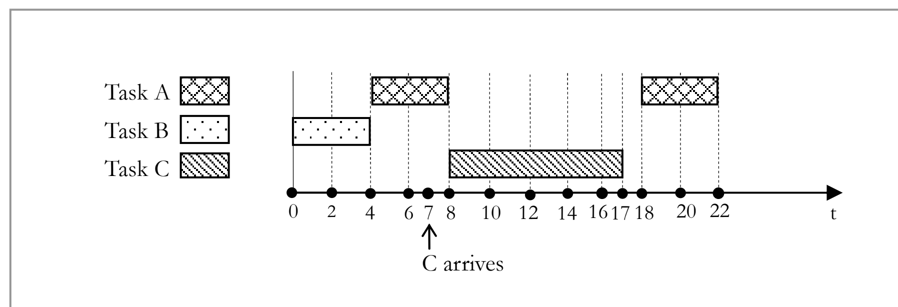

Figure 3.6 Scheduling Illustration for FPS in Periodic Preemptive Mode

At the beginning, **A** and **B** are two ready-to-run tasks. The arbiter selects **B** to run due to its higher priority. When *t=2ms*, the first scheduling point occurs. **B** is still the highest priority tasks in the system, and hence continues to run. When *t=4ms*, **B**'s execution is over and the second scheduling point occurs. The arbiter selects **A** to run, because **A** is the only ready-to-run task. When *t=7ms*, **C** arrives. However, *t=7ms* is not a scheduling point, so that **C** cannot get the control of the CPU immediately. When *t=8ms*, a scheduling point, the arbiter grants the control of the CPU to C due to its highest priority. When *t=17ms*, **C**'s execution is over and **C** releases the control of the CPU voluntarily. When *t=18ms* **A** obtains the CPU again, and when *t=22ms* **A** completes its execution.

5. Implementative Preemptive FPS with RR Fashion

In the implementation level, we verify the scheduling problem by using the preemptive FPS with round robin fashion algorithm.

The preemptive FPS with round robin fashion is the combination of FPS in preemptive mode and RR scheduling algorithm. The preemptive FPS is applied to schedule tasks with different priorities, whereas RR is applied to schedule tasks with identical priorities. The arbiter keeps a round robin fashion queue for

each possible priority, denoted as *Queue(priority)*. The queues are also called ready-queue. For example, if the priority is ranging from 1 to 10, there are 10 ready-queues in the arbiter. Whenever, a task becomes ready to run, the arbiter inserts the task into the tail of *Queue(priority)*. When the arbiter tries to select a task to run, it finds the non-empty ready-queue with the highest priority and picks up the task from the ready-queue's head. Thus, preemptive FPS with round robin fashion can schedule tasks with arbitrary priorities.

Once more, we consider the small example (Table 3.1) that used in previous subsection. Assume the priority of **A** is 2, and the priorities of **B** and **C** are 1. See Table 3.5. The quantum for RR scheduling algorithm is 2ms.

Table 3.5 Parameters for Preemptive FPS with Round Robin Fashion

Task	Priority	Execution Time	Period	Deadline
A	2	8 ms	50 ms	50 ms
B	1	4 ms	50 ms	50 ms
C	1	9 ms	50 ms	50 ms

The scheduling is shown in Figure 3.7. We present the ready-queue of priority 1, namely *Queue(1)*. Initially, **B** is stored at the head of *Queue(1)*, and **C** is at the tail. At the beginning, the arbiter selects **A** to run due to its highest priority. When *t=8ms*, **A**'s execution is over and the arbiter selects **B** from the head of *Queue(1)* to run. From *t=8ms* to *t=14ms,* the arbiter selects **B** and **C** alternately and lets each of them run for a slice. After *t=14ms*, **B** is the only ready-to-run task in the system and it runs until it completes execution.

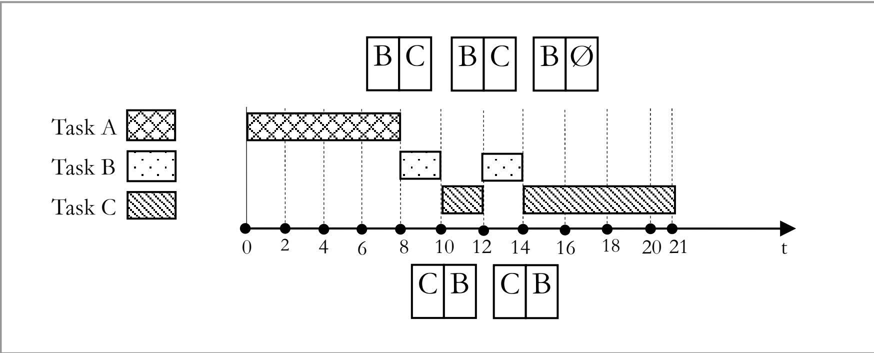

Figure 3.7 Scheduling Illustration for preemptive FPS with Round Robin Fashion

6. Summary of Scheduling Algorithm

Comparing FCFS, RR, FPS, and preemptive FPS with round robin fashion scheduling algorithm, we know that, FCFS schedules tasks only according to

their arriving time. In this manner, FCFS considers that importance or urgency of a task is related to the arrival time of the task, which is not actual situation of most cases. RR can only handle tasks with an identical priority. In contrast, FPS can only handle tasks with different priorities. Preemptive FPS with round robin fashion algorithm, the combination of FPS and RR, is a more advanced scheduling algorithm that can schedule a set of tasks with arbitrary priorities.

In FCFS and non-preemptive FPS, preemptions are not allowed. Hence, the resource and time used to changes the context of task in the CPU is less than the scheduling allowed preemptions, such as RR, preemptive FPS, periodic preemptive FPS, and preemptive FPS with round robin fashion.

CHAPTER 4

VERIFICATION WORK

We are going to apply FCFS, RR, and preemptive FPS in the architecture level and apply preemptive FPS with round robin fashion algorithm in the implementation level. All of these four scheduling algorithm are static. Once the priority of each task is decided, the arbiter is enough for scheduling. In this Chapter, we introduce a model and a set of properties that are characterized from the concerns and the shared knowledge of the ELM-2 project.

The model for doing the verification work in the Integrating Process consists of a set of automata modeling from periodical tasks (Task Automata) and an automaton modeling from the arbiter (Arbiter Automaton). The Task Automata are characterized based on periodical tasks in the ELM-2 project, which are discussed in the second section of Chapter 3. Meanwhile, the Arbiter Automaton is characterized according to preemptive FPS with round robin fashion algorithm.

1. Characterizing The Task Automata

1.1 Task Model

As mentioned in the second subsection of Chapter 3, we may describe an ELM-2 task with three parameters. They are priority, execution time, and period. Each task can be simply distinguished by its task name. However, from the point of view of verification tools, we must give each task an integer identity. Thus, every ELM-2 task may be described by a Task Model, shown as the follows.

```
task (identity, priority, execution time, period)
```

Suppose that there are three Task Models, **A**, **B**, and **C**. We can easily use a table to express them, as shown in Table 4.1.

Table 4.1 Task Models Table

Task	Identity	Priority	Execution Time	Period(Deadline)
A	$A_{identity}$	$A_{priority}$	$A_{execution\ time}$	A_{period}
B	$B_{identity}$	$B_{priority}$	$B_{execution\ time}$	B_{period}

C	$C_{identity}$	$C_{priority}$	$C_{execution\ time}$	C_{period}

Because the Task Models are uniform, a better way to model Task Automata is to define a template for automata firstly. Later on, we instantiate the Task Automata from the template by providing it with the Task Models. See Figure 4.1 The template is called Task Automata Template, Task Template for short.

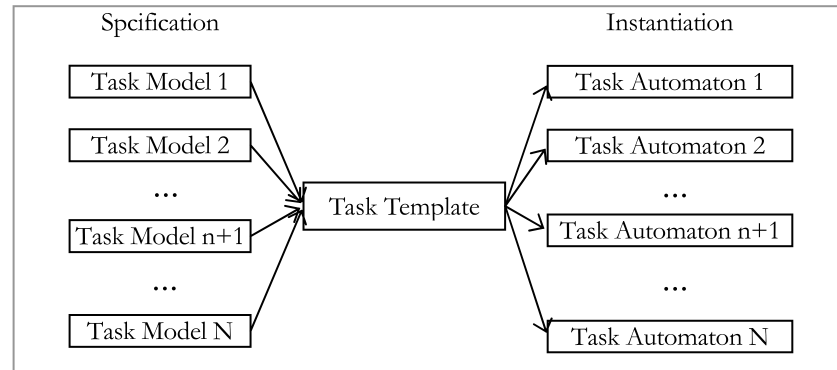

Figure 4.1 Relationships of Task Models, Task Template, and Task Automata

1.2 Generic Timed Automata

The Task Template can be defined by Generic Timed Automata [LL01]. Generic Timed Automata are an extension of Timed Automata that can be instantiated with parameters. They consist of all elements of the Timed Automata as well as a number of parameters. The elements of a Timed Automaton are

- Locations,
- Variables,
- Clocks: non-negative real numbers, always increasing, which can only be reset to 0,
- Transitions, connections between the locations,
- Invariants, constrains on the locations,
- Guards, constrains on the transitions,
- Assignments, initializing and changing variables' value and resetting the clocks,
- Channels, synchronizing the transitions among two or more automata by sending or receiving messages.

The graphical representation of Generic Timed Automata is the same as for Timed Automata. A location is represented as a circle. A double lined circle represents the initial location of an automaton. Each automaton only has one

initial location. A transition is expressed as an edge, from one location to another location. The guards, the assignments, and the messages passing via the channels are listed beside the transitions. The messages passing via a channel are expressed before a question mark or an exclamation mark. For the channel *ch*, the message sent via the *ch* is denoted by *ch!*. Correspondingly, the message received via the *ch* is denoted by *ch?*. For the channel *ch*, *ch!* (or *ch?*) is enabled if there is a corresponding *ch?* (or *ch!*) which can be executed simultaneously.

1.3 Characterizing The Task Template

In previous subsections of this chapter, we have defined the Task Model. We also find that to define the Task Template by the Generic Timed Automata and then to instantiate the Task Automata from the Task Template are more efficient than to construct automata for each task.

1.3.1 Task States and Their Relationship

Before characterizing the Task Template, we need know how the tasks work during the scheduling. From the point of view of the scheduling, a task can be in one of three possible states: ready to run, running or sleeping, as shown in Figure 4.2. The sleeping state indicates that the task has completed the execution and the period of the task is not over.

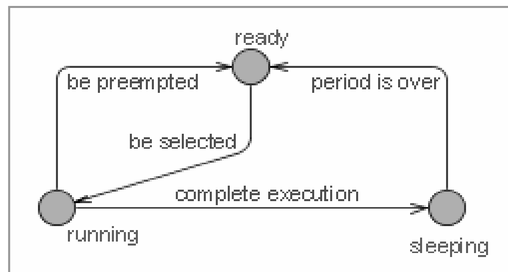

Figure 4.2 Task State Relationship Model

The three states may also be illustrated with the task parameters. See Figure 4.3

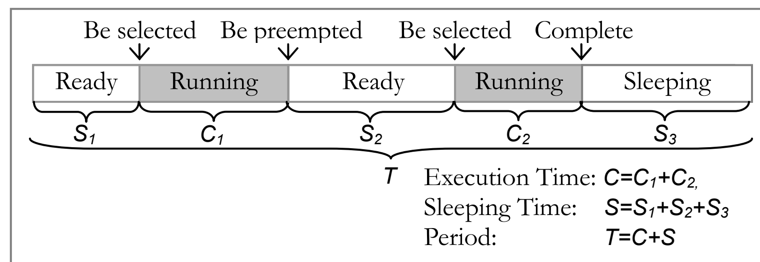

Figure 4.3 Task States and Parameters

Initially each task is in the *ready* state. The task changes from the *ready* state to the *running* state only when the CPU selects it to run. If the task is preempted by a task with higher priority, it changes from the *running* state to the *ready* state. Once the task completes its execution, it changes from the *running* state to the *sleeping* state. The task returns from the *sleeping* state to the *ready* state to execute when the period is over. A CPU can only facilitate one task at a time.

1.3.2 Task Template

According to the task state relationship, the Task Template, named *task_template*, is characterized as shown in Figure 4.4. Figure 4.4 is represented in UPPAAL format. UPPAAL is the tool we select for verification. The reasons for selecting UPPAAL will be explained later in Section 5 of this chapter.

The *task_template* has four integer parameters:

- *id*: task identity
- *pri*: task's priority,
- *exetime*: task's execution time, and
- *period*: task's period.

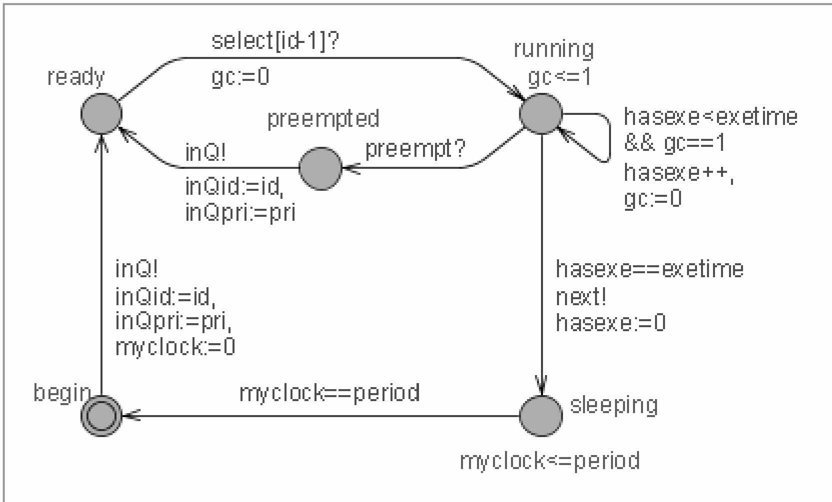

Figure 4.4 Task Template

task_template (id, pri, runtime, period)

In the *task_template*, the initial location is named *begin*. When the period of a task expires, the task becomes ready to run. The task informs the arbiter of adding it to the tail of a ready-queue by sending a message via the *inQ* channel. The ready-queue is held by the arbiter. Meanwhile, a local clock, named *myclock*, is reset to 0. The *myclock* is derived from the global clock and used to record how long the time has passed in a period. That the *task_template* is at the

ready location means it has informed the arbiter that it is ready to run. At the *ready* location, the *task_template* awaits a message sent from the *select* channel which signals granted access to the CPU. The *select* channel actually is an array of channels. Each Task Automaton instantiated from the *task_template* receives its own selected message through one channel of the array, which is corresponding to its task identity, illustrated in Figure 4.5.

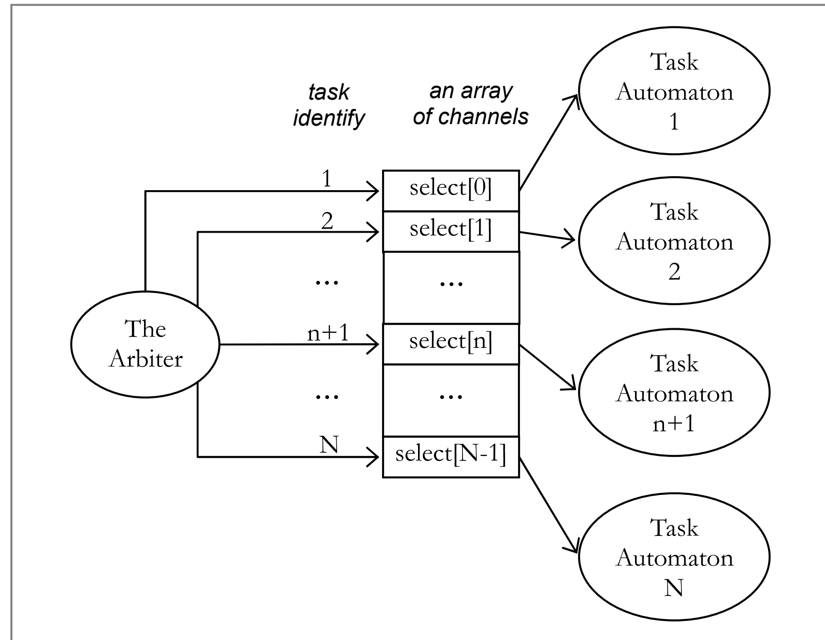

Figure 4.5 Structure of *select*

In the *task_template*, as one millisecond passes, the *hasexe* increases by one if the *task_template* is in the *running* location. The *hasexe* is an integer to record the total time that the task has executed. When the execution is over, the *task_template* goes out of the *running* location and enters into the *sleeping* location. The *task_template* may be preempted by receiving a message from the *preempt* channel. Being preempted, the *task_template* moves from the *running* location to the *preempted* location, while adding itself to the tail of the ready-queue again. Once the period is over, the *task_template* goes back to the *begin* location from the *sleeping* location.

1.4 Instantiating Task Automata from The Task Template

After defining the Task Template, we discuss how to instantiate a task automaton from the Task Template by inputting a task model. As mentioned before, a Task Model can be expressed as follows.

```
task (identity, priority, execution time, period)
```

Let's assume a Task Model is named **A**. The Task Automaton of **A** can be instantiated as follows.

```
A := task_template( A_identity, A_priority, A_execution time, A_period);
```

2. Characterizing Arbiter Automaton

In the Arbiter Automaton, the scheduling is defined by the guards and the messages via the channels of the Timed Automata. The guards and the messages govern the transitions of the Arbiter Automaton. The Arbiter Automaton based on preemptive FPS with round robin fashion algorithm, *arbiter*, is defined in Figure 4.6.

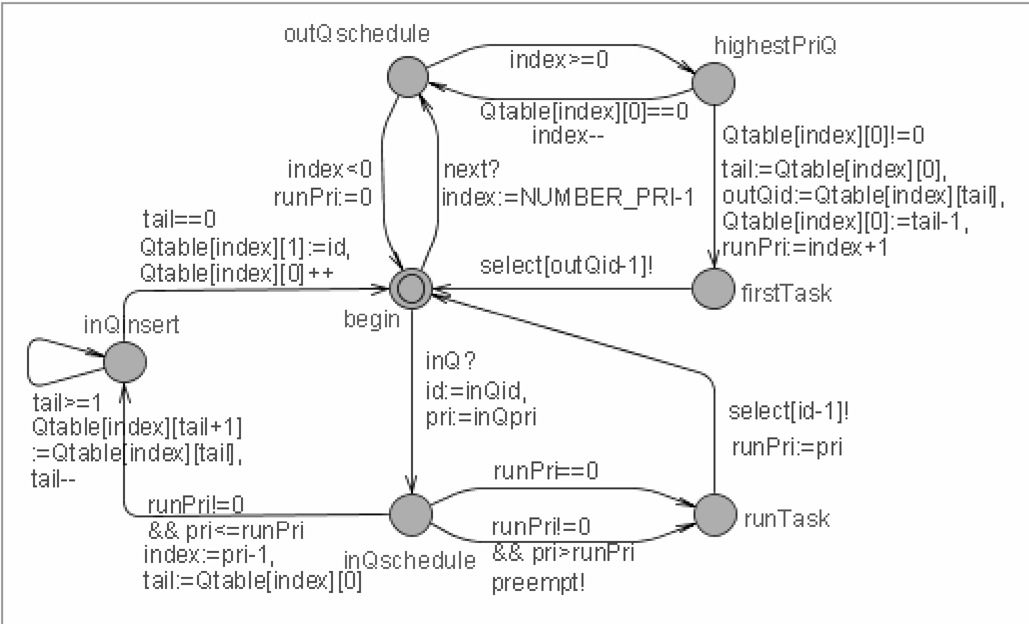

Figure 4.6 Arbiter Automaton *arbiter*

The *arbiter* can be separated into two parts.

- The first part consists of three locations, including *begin*, *inQschedule*, and *inQinsert* locations. This part is in charge of administration of the ready-queue.
- The second part consists of six locations, including *begin*, *outQschedule*, *highestPriQ*, *firstTask*, *inQschedule*, and *runTask* locations. This part is in charge of CPU assignments.

The arbiter has seven local variables. They are *runPri*, *id*, *pri*, *Qtable*, *index*, *tail*, and *outQid*.

- *runPri* stores the priority of current task that occupies the CPU.
- *id* stores the task identity of a Task Automaton that sends the *inQ* message.
- *pri* stores the priority of a Task Automaton that sends the *inQ* message.
- *Qtable* is a two-dimension array. Each row of the *Qtable* is a round robin fashion ready-queue for a certain priority. The first position of each row refers to the tail of the ready-queue. Other positions store task identities. See Figure 4.7.

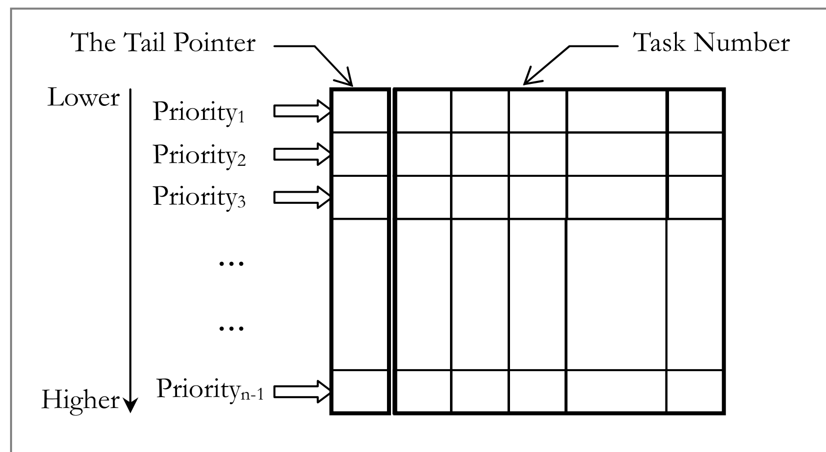

Figure 4.7 Structure of *Qtable*

- *index* refers to the row of the *Qtable*.
- *tail* refers to the tail of a row of the *Qtable*.
- *outQid* stores the identity of the task that is granted to control the CPU in next millisecond.

The first part of the *arbiter*

When the *arbiter* adds a Task Automaton, for instance **A**, into the *Qtable*, the arbiter firstly locates a row of the *Qtable* corresponding to **A**'s priority. The arbiter adds **A**'s identity to the tail position. Finally, the *arbiter* moves the tail pointer backward roundly.

The second part of the *arbiter*

A Task Automaton, for instance **A**, which enters the *ready* location, informs the *arbiter* its task identity and priority by sending a message via the *inQ* channel (*inQ!*). When the *arbiter* receives this message (*inQ?*), it stores the task identity and the priority to the *id* and *pri* respectively.

a) If *runPri* is equal to 0 that means **A** is the only Task Automaton located in the *ready* location among all Task Automata, the *arbiter* informs **A** to run by sending a message via the **A**'s *select* channel (*select[id-1]!*);
b) If *runPri* is not equal to 0 and **A**'s priority is greater than *runPri,* these mean that another Task Automaton, for instance **B**, is at the *running* location, and **A**'s priority is higher than **B**'s. The *aribiter* sends a message via the *preempt* channel (*preempt!*). This message forces **B** to move out of the *running* location. And then, the *arbiter* sends another message via the **A**'s *select* channel to let **A** enter the *running* location.
c) If *runPri* is not equal to 0 and **A**'s priority is smaller than *runPri,* these mean another Task Automaton, for instance **B**, is at the *running* location, and **A**'s priority is lower than **B**'s. The *aribiter* adds **A**'s task identify to its ready-queue.

After **A** finishes its execution, it goes to the *runsleep* location and sends a message to the *arbiter* via the *next* channel (*next!*). This message tells the *arbiter* to select another ready task to run. The *arbiter* searches the first non-empty ready-queue (row) from the higher index to the lower index, and gets the task identity from the tail position of the first non-empty ready-queue, moves the tail pointer forward, and assigns the identity and the priority to *outQid* and *runPri*, respectively. Finally, the *arbiter* informs the selected Task Automaton of entering the *running* location.

In our case study, when the tasks are scheduled in a single CPU, at most one Task Automaton can be in the *running* location. Our model guarantees this specification automatically. Before the *arbiter* grants a Task Automaton, for instance **A**, to enter the *running* location, it checks whether another Task Automaton, for instance **B**, is already at the *running* location or not. If not, the *arbiter* informs **A** to enter the *running* location. If so, the *arbiter* compares the priorities of **A** and **B** firstly. If **A**'s priority is higher than **B**'s, the *arbiter* forces **B** to move out of the *running* location, and informs **A** of entering the *running* location. Otherwise, the arbiter stores **A** into *Qtable*. Therefore, at any time, at most one Task Automaton is in the *running* location.

3. Verification Model

After characterizing the *task_template* and the *arbiter*, we discuss how to compose a verification model from them (in UPPAAL format).

Consider the system listed in Table 4.1. Firstly, we instantiate Task Automata from the *task_template*:

$$A := task_template(A_{identity}, A_{priority}, A_{execution\ time}, A_{period});$$
$$B := task_template(B_{identity}, B_{priority}, B_{execution\ time}, B_{period});$$
$$C := task_template(C_{identity}, C_{priority}, C_{execution\ time}, C_{period});$$

The verification model is defined as following

```
System A, B, C, arbiter;
```

4. Defining Properties for Verification

The implementation engineers and the architecture engineers hope that the tasks could be executed properly, which means no deadline missing occurs. In other words, each task must complete execution before the deadline expires. Because in the ElM-2 project the deadline and the period are equal, the equivalent expression is that each task must complete execution before the period expires.

In the verification model, we have defined an invariant at the *sleeping* location. The invariant is *myclock*<=*period*, which means when a Task Automaton is at the *sleeping* location, the *myclock* must be smaller than or equal to the *period*. Once the *myclock* is greater than the *period* when the task automaton finishes execution, there is no possible transition that the task automaton can fire. Namely, a deadlock occurs. Therefore, the no deadline missing property may be written in TCTL [JPK99] as follows. It means that a deadlock can never occur in the verification model.

```
A[] not deadlock
```

5. Choosing Verification Tool

As we have characterized Task Automata and the Arbiter Automaton of the verification model, we will select a verification tool to verify the property of no deadline missing. In the research report [1], we had summarized three tables for tool selection. See Table 4.2, 4.3, and 4.4.

Table 4.2 Tools Basic Comparisons

Name	**Notion**	**Temporal Logic**	**Specification language**
SPIN	Büchi Automata	PLTL	PROMELA
SMV	Kripke Structure	CTL	Extended SMV Language
UPPAAL	Timed Automata	TCTL	Non-deterministic Guarded Command Language
PMC	Timed Automata	TCTL	Extended Timed Graphs
Kronos	Timed Automata	TCTL	Kronos Input Language
HyTech	Timed Automata	TCTL	HyTech Input Language

Table 4.3 Tools Applied System Comparisons

Name	**Non-/ Real-Time**	**Non-/ Deterministic**	**Concurrency/ Sequential**
SPIN	Non-RT	Non-D	Sequential
SMV	Non-RT	Non-D	Concurrency
UPPAAL	Real-Time	Deterministic	Concurrency
PMC	Real-Time	Deterministic	Concurrency
Kronos	Real-Time	Deterministic	Concurrency
HyTech	Real-Time	Deterministic	Concurrency

Table 4.4 Tools Integrating Environment Comparisons

Name	**Graphical Editor**	**Simulator**	**Verifier**
SPIN	No	Yes	Yes
SMV	No	No	Yes
UPPAAL	Yes	Yes	Yes
PMC	No	No	Yes
Kronos	No	No	Yes
HyTech	No	No	Yes

For our case study,

- the spinning image scanner of the ELM-2 project is a real-time system [KZ95],
- our verification model is characterized in (Generic) Timed Automata, and
- the property is written in TCTL.

Therefore, we choose UPPAAL. UPPAAL supports instantiating automata from a template by inputting parameters, which is required by Generic Timed Automata. Another reason is that the integrating environment of UPPAAL not

only has a verifier, but also a graphical editor and a simulator. It is more convenient to do the verification work.

CHAPTER 5

EXPERIMENTS IN IMLEMENTATION LEVEL

1. Implementation Concerns

In the implementation level of the ELM-2 project, the four tasks are **Main**, **ArmControl**, **TransMotor**, and **DiscReader**. The implementation engineers care whether they can be scheduled in a single CPU by using preemptive FPS with round robin fashion algorithm. The implementation engineers provide their concerns in Table 5.1.

Table 5.1 Original Task Parameters in Implementation Level

Task	Priority	Execution Time	Sleeping Time	Execution Time	Sleeping Time
Main	5	1ms	10000ms		
ArmControl	9	2ms	*Yield*		
TransMotor	10	3ms	4ms	1ms	4ms
DiscReader	9	1ms	3ms		

The implementation engineers describe the tasks with three parameters, namely priority, execution time, and sleeping time. The priority of the **Main** is 5, which has the lowest priority among the four tasks. In a period, the **Main** runs for 1ms, and then sleeps for 10000ms.

The priority of the **ArmControl** is 9. In a period, the **ArmControl** runs for 2ms, and then yields the control of the CPU voluntarily. The *Yield* in the Table 4.1 indicates a special function in the Java Language. When the *yield* function is called, the calling task gives up the CPU voluntarily. [3]

The priority of the **TransMotor** is 10, the highest priority among the four tasks. The **TransMotor** is a particular task that has two pairs of execution time and sleeping time. In odd period, it runs for 3ms and sleeps for 4ms. In even period, it runs for 1ms and sleeps for 4ms, shown in Figure 5.1. The reason why the implementation engineers use two pairs of time is that the analog circuits are controlled by a microprocessor's digital outputs [8].

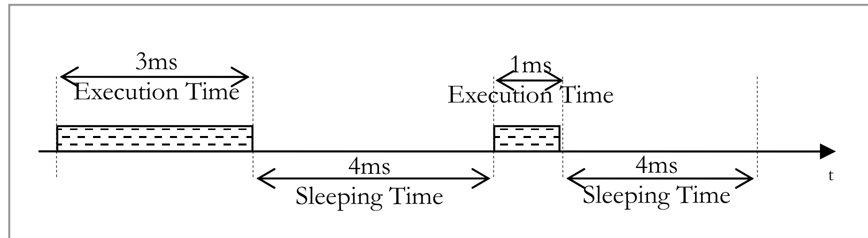

Figure 5.1 Execution Time and Sleeping Time of **TransMotor**

The priority of the **DiscReader** is also 9. The **DiscReader** runs for 1ms in a period, and then sleeps for 3ms.

Because the implementation engineers assigned values of parameters according to the programming language, we must adjust them before we do our verification work in the implementation level.

2. Verification Model in Implementation Level

In Chapter 4, we have defined the Task Model and the Task Template of the ELM-2 project. However, they were defined from a general case. We have to upgrade them so that they can meet the special features of the implementation concerns, such as two time pairs and the *Yield* function.

2.1 Implementation Task Model

As we have noticed, the structures listed in the Table 5.1 are not exactly the same. We must uniform them before we upgrade the Task Model to the Implementation Task Model.

2.1.1 Uniforming Implementation Parameters

The tasks listed in Table 5.1 are almost the same, besides two minor differences.

- The **TransMotor** contains two time pairs and other three tasks contain only one time pair.
- The sleeping time of the **ArmControl** is not a numeric value, but a *Yield* function

The first difference can be eliminated by adding a time pair to the **Main**, the **DiscReader**, and the **ArmControl**, identically to the time pair they already have.

The second difference can be eliminated by replacing *Yield* with 0. When the **ArmControl** finishes the execution, it yields the control of the CPU and the

arbiter adds it to the tail of the ready-queue immediately. This effect is equivalent to that the **ArmControl** doesn't sleep after execution, namely that the sleeping time is 0ms.

After eliminating the two minor differences, the uniformed parameters of the tasks in the implementation level are listed in Table 5.2.

Table 5.2 Uniformed Task Parameters in Implementation Level

Task	Priority	Odd Period		Even Period	
		Execution Time	Sleeping Time	Execution Time	Sleeping Time
Main	5	1ms	10000ms	1ms	10000ms
ArmControl	9	2ms	0ms	2ms	0ms
TransMotor	10	3ms	4ms	1ms	4ms
DiscReader	9	1ms	3ms	1ms	3ms

2.1.2 Upgrading Task Model to Implementation Task Model

In Table 5.2, the implementation tasks are described by the sleeping time instead of the period. However, the period can be calculated by summing up the execution time and the sleeping time. Meanwhile, we also need to adjust the task models table (Table 4.1) to the implementation task models table by adding a pair of execution time and period. See Table 5.3.

Table 5.3 Implementation Task Models Table

Task	Identity	Priority	Odd Period		Even Period	
			Execution Time	Period (Deadline)	Execution Time	Period (Deadline)
Main	1	5	1ms	10001ms	1ms	10001ms
ArmControl	2	9	2ms	2ms	2ms	2ms
TransMotor	3	10	3ms	7ms	1ms	5ms
DiscReader	4	9	1ms	4ms	1ms	4ms

2.2 Implementation Task Template

2.2.1 Upgrading Task Template to Implementation Task Template

From the view of one period, the Implementation Task Model is exactly as same as the Task Model. However, the values of parameters are different in odd and even periods. We must improve our Task Template, *task_template,* to the Implementation Task Template, *imp_task_template*, So that the *imp_task_template* is able to switch the values from period to period. See Figure 5.2.

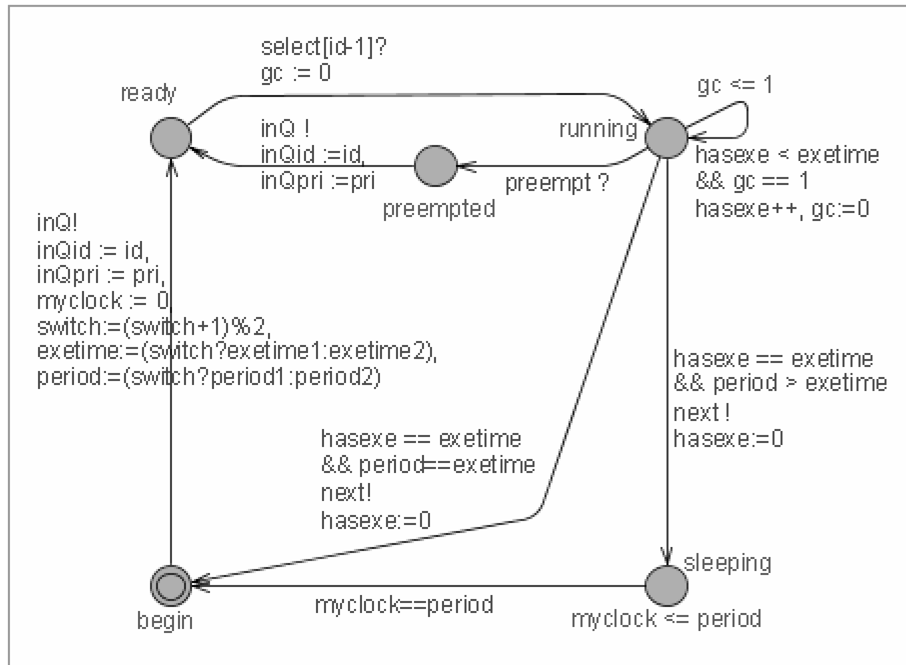

Figure 5.2 Task Template in Implementation Level
imp_task_template (*id*, *pri*, *exetime1*, *period1*, *exetime2*, *period2*)

2.2.2 Instantiating Implementation Task Automata

The four task automata in the implementation level, for short Implementation Task Automata, are instantiated from the *imp_task_template* as follows

```
Main       :=imp_task_template(1, 5, 1, 10001, 1, 10001);
ArmControl:=imp_task_template(2, 9, 2, 2, 2, 2);
TransMotor:=imp_task_template(3, 10, 3, 7, 1, 5);
DiscReader:=imp_task_template(4, 9, 1, 4, 1, 4);
```

3. Implementation Verification Model and Property

Because in the implementation level, we still use preemptive FPS with round robin fashion algorithm, it is not necessary for the *arbiter* to be modified. The verification model in the implementation level is presented as follows.

```
system Main, ArmControl, TransMotor, DiscReader, arbiter;
```

The property remains as same as devised before.

```
A[] not deadlock
```

4. Implementation Verification Result

We verify the no deadlock property based on the implementation verification model. Firstly, we use the UPPAAL simulator to simulate the scheduling. During the simulation, the four Implementation Task Automata and the *arbiter* work well, and the behaviors are correct.

And then, we use the UPPAAL verifier to check the property. However, UPPAAL didn't finish the verification within 24 hours, which means the size of the state space [JPK99] involved by the model is too large for UPPAAL. We then decreased the number of the Implementation Task Automata, and tried to find out how many Implementation Task Automata the model can handle in the implementation level. When we decrease the number from four to two (all possible combination for four automata), UPPAAL reports a satisfied result within 20 minutes. These indicate the implementation model can work and the size of the model limits the verification work.

After analyzing the implementation verification model, we notice that three factors affect the size of the state space.

a) Adding a task to the tail of the ready-queue and selecting next task from the ready-queue brings heavy work to the *arbiter*.
b) Using clock variables to count the execution time and the period involves lots of irrelevant times. Because the clock is a continuous time, every time is represented as a state in the state space. However, only the actual time at which the location changing occurs is relevant.
c) The *arbiter* is characterized from a general case that means the *arbiter* can schedule the Task Automata no matter how many they are. This way is too general and consequently not practically applicable.

CHAPTER 6

EXPERIMENTS IN ARCHITECTURE LEVEL

1. Architecture Concerns

In the architecture level of the ELM-2 project, the spinning image scanner is designed by four tasks, namely **Tm**, **Track**, **Read** and **Play**. The architecture engineers provide their concerns by using a table. See Table 6.1.

Table 6.1 Original Task Parameters in Architecture Level

Task	Execution Time	Period (Deadline)
Tm	3ms	8ms
Track	2ms	24ms
Read	1ms	12ms
Play	5ms	500ms

The architecture engineers only provide the execution time and the period of each task. We have to answer

- How many CPUs does the spinning image scanner require to accomplish these tasks?
- How to schedule these tasks using what policy?

2. Feasibility Checking

In this section, we introduce the Liu and Layland formula. According to the formula, the feasibility of scheduling tasks in a single CPU can be predicted.

2.1 Feasibility

For a given set of tasks, a scheduling is *feasible*, if the tasks can be scheduled by using a certain scheduling algorithm.

2.2 Liu and Layland Formula

The fast way to predict the feasibility of scheduling a set of task in a single CPU is to estimate the feasibility according to the Liu and Layland formula [LL73].

Suppose for a task, its period is T_i, deadline is D_i, and execution time is C_i. Its

utilization factor Ui is defined as the indication of the relative occupation of the CPU by the task, i.e.

$$Ui = \frac{Ci}{Ti}$$

For a set of tasks, the total utilization factor U may be calculated according to

$$U = \sum_{i=1}^{n} Ui = \sum_{i=1}^{n} \frac{Ci}{Ti}$$

According to the Liu and Layland formula, for a set of N independent periodic tasks, where for each task $Ui = \frac{Ci}{Ti}$, and where for each task $Di = Ti$, the tasks are guaranteed schedulable in a single CPU, the least upper bound of total utilization factor is

$$U = \sum_{i=1}^{n} \frac{Ci}{Ti} \leq n(2^{(1/n)} - 1)$$

For four tasks (*n=4*), the least upper bound of total utilization factor is 0.75683.

2.3 Checking Feasibility of Architecture Concerns

In the architecture level, the total utilization factor of four tasks is

$$U_1 = \frac{3}{8} + \frac{2}{24} + \frac{1}{12} + \frac{1}{10000} = 0.55167$$

that is smaller than 0.75683. Therefore, the scheduling of the four architecture level tasks is feasible in a single CPU.

3. Verifying Feasibility

As proven, the four architecture level tasks can be scheduled. We will use FCFS, RR, and preemptive FPS to answer the question about how to schedule these tasks.

Due to failure verification in the implementation level, we will re-characterize verification models based on FCFS, RR, and preemptive FPS scheduling. This time, the model will characterize from the exact four tasks provided by the architecture engineers. Meanwhile, we will use tick times instead of clock variables in order to avoid the irrelevant states involved in the implementation model.

The Architecture Task Model is

```
Task (identity, execution time, period)
```

And the task model table in the architecture level becomes

Table 6.2 Architecture Task Models Table

Task	Identity	Execution Time	Period (Deadline)
Tm	1	3ms	8ms
Track	2	2ms	24ms
Read	3	1ms	12ms
Play	4	5ms	500ms

3.1 Verifying FCFS Scheduling

3.1.1 Task Template and Arbiter Automaton for FCFS

FCFS scheduling also can be evaluated using a FCFS verification model, which is composed with four FCFS Task Automata and a FCFS Arbiter Automaton. The four FCFS Task Automata are instantiated from FCFS Task Template by inputting Architecture Task Models in Table 6.2. The FCFS Task Template (named *FCFS_task_template*) and the Arbiter Automaton (named *FCFS_arbiter*) are shown in Figure 6.1 and 6.2 respectively.

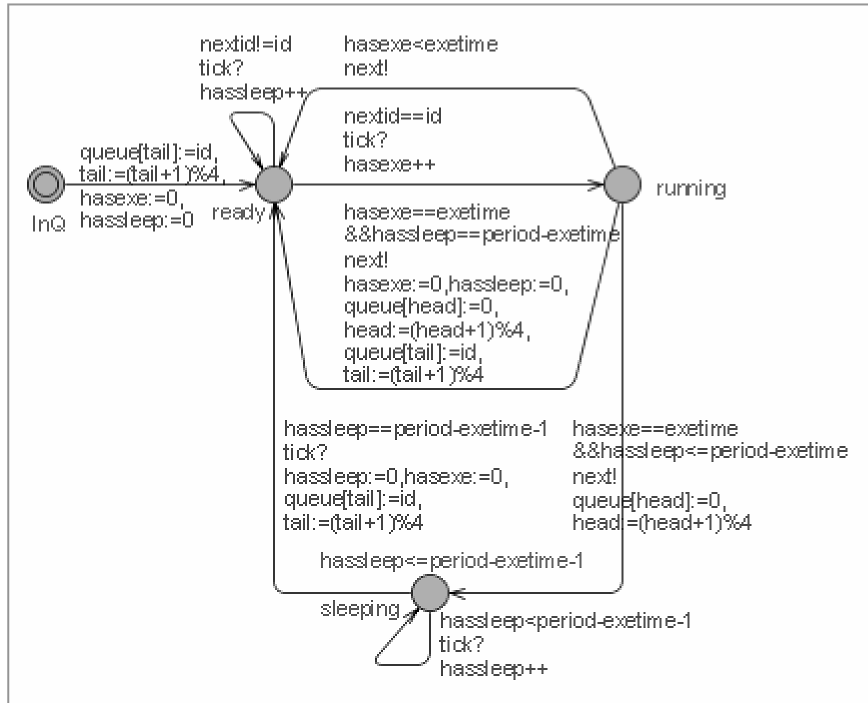

Figure 6.1 Task Template for FCFS

FCFS_task_template(id, exetime,period)

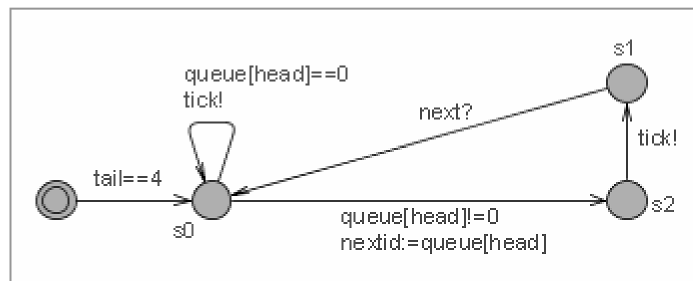

Figure 6.2 Arbiter Automaton for FCFS

FCFS_arbiter

In these two automata, they cooperate by sharing information through global variables or sending/receiving messages via channels, including

- *nextid*: task identity, indicating which task will run next.
- *queue*: First In First Out circular queue. It consists of four positions. Every position may be empty, indicated by 0, or store a task identity, ranging from 1 to 4. The sequence of the task identities storing in the queue is the arrival order of the FCFS Task Automata. The queue

also has two pointers. One is the *head*, always referring to the head of the *queue*. The other one is the *tail*, always referring to the tail of the *queue*, namely the first empty position of the queue where a new coming task identity may be stored. See Figure 6.3.

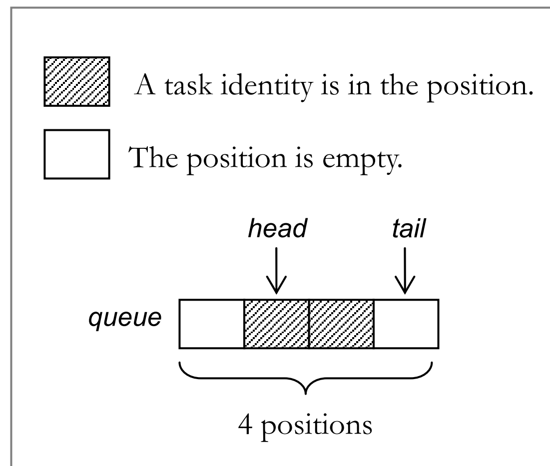

Figure 6.3 Structure of *queue* for FCFS

- *tick*: a broadcast channel. Generally, a channel connects two automata. One sends messages, and the other one receives the messages. A broadcast channel is used to connect two or more automata. One automaton sends messages, and all of the others receive the messages. If a receiving automaton is not in the location where it can receive the message, the automaton skips the message. Via the *tick*, the messages are sent from the *FCFS_arbiter*, and received by all FCFS Task Automata, which are instantiated from the *FCFS_task_template*. By this way, each FCFS Task Automaton can run or sleep for one tick time.
- *next*: a channel, via which messages are sent from a FCFS Task Automaton to the *FCFS_arbiter*. The messages indicate that the execution is over so that the *FCFS_arbiter* can select another FCFS Task Automaton to move into the *running* location.

Each FCFS Task Automaton begins with adding its task identity to the tail of the *queue*, which means that it is ready to run. After all FCFS Task Automata are at the *ready* location, the *FCFS_arbiter* begins to work. The *FCFS_arbiter* first checks whether the *queue* is empty. If not, the *FCFS_arbiter* assigns the

nextid to the task identity pointed by the *head*, and sends a broadcast message via the *tick* channel. Every FCFS Task Automaton compares its own task identity with the *nextid*, when it receives the *tick* message.

- If the task identity and the *nextid* match, the FCFS Task Automaton moves from the *ready* location to the *running* location. At the meantime, the *hasexe* increases by one. The *hasexe* is an integer variable used to record how many tick times the FCFS Task Automaton has executed.
- If the task identity and the *nextid* don't match, the FCFS Task Automaton fires a self-circle transition and increases the *hassleep* by one. The *hassleep* is also an integer variable used to record how many tick times the FCFS Task Automaton has slept.

Let a FCFS Task Automaton be in its *running* location, then

- if its execution hasn't finished, it moves out of the *running* location and sends a *next* message to inform the *FCFS_arbiter* of selecting next FCFS Task Automaton;
- if its execution finishes, but its period hasn't expired, it goes from the *running* location to the *sleeping* location to sleep. Meanwhile, it sends a *next* message and moves the *head* to the next non-empty position;
- if both its execution and period are over, it moves to the *ready* location, resets the *hasexe* and the *hassleep* to 0, moves the *head*, adds its task identity to the tail of the *queue*, and moves the *tail* to the next empty position.

Comparing the verification model characterized in Chapter 3 and 4 with the FCFS verification model, we notice that the FCFS verification model is simpler. It is characterized exactly based on the architecture concerns, which means that it is able to schedule only four Architecture Task Automata. However, by this way, the model loses lots of flexibilities. If the number of the FCFS Task Automata increases, we have to increase the capability of the *queue* and adjust the guards and the assignments of the *FCFS_task_template* and the *FCFS_arbiter*. We also discard lots of irrelevant states by applying tick times to record the time. Consequently, the size of the state space decreases. Meanwhile, we move parts of the work for administrating the *queue* to the Architecture Task Automata either. By doing this, the size of state space decreases, while the functionalities of the *FCFS_task_template* and the *FCFS_arbiter* are not as clear as before.

3.1.2 FCFS Verification Model and Property

The FCFS Task Automata are instantiated from the *FCFS_task_template*,

shown as follows.

```
FCFS_Tm     := FCFS_task_template(1, 3, 8);
FCFS_Track  := FCFS_task_template(2, 2, 24);
FCFS_Read   := FCFS_task_template(3, 1, 12);
FCFS_Play   := FCFS_task_template(4, 5, 500);
```

The FCFS verification model is

```
system FCFS_Tm, FCFS_Track, FCFS_Read, FCFS_Play,
      FCFS_scheduler;
```

The property remains the same.

```
A[] not deadlock
```

3.1.3 FCFS Verification Result

The result reported from UPPAAL is "not satisfied", which means the four tasks cannot be scheduled by using FCFS. However, the result is affected by the initial order of the ***queue***. The initial order of the ***queue*** is used to record the arrival order of the four architecture tasks. It has 24 possibilities. Based on the different orders, we verify the property one by one. The verification results are listed in Table 6.3. In this table, the initial order is indicated by the task identities. The "schedulable" means that based on the initial order, the four tasks can be scheduled by using FCFS.

Table 6.3 FCFS Verification Results Table

Initial Order				**Verification Result**
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable

3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	Schedulable
4	1	3	2	Schedulable
4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.

From Table 6.3, we know that missing deadline will never happen in some order. For instance, in the initial order of (3, 1, 2, 4), which means that the order of task arriving is the **Read**, the **Tm**, the **Track**, the **Play**, the four tasks can be scheduled by using FCFS. In other orders, missing deadline will happen. For instance, in the initial order of (2, 4, 1, 3), the **Tm** misses its deadline that means in a certain period, the **Tm** cannot finish the execution before the period expires.

3.2 Verifying RR Scheduling

3.2.1 Task Template and Arbiter Automaton for RR

RR scheduling also can be evaluated using a RR verification model, which is composed of four RR Task Automata and a RR Arbiter Automaton. The four RR Task Automata are instantiated from the RR Task Template by inputting Architecture Task Models in Table 6.2. The RR Task Template (named *RR_task_template*) and the RR Arbiter Automaton (named *RR_arbiter*) are shown in Figure 6.4 and 6.5 respectively.

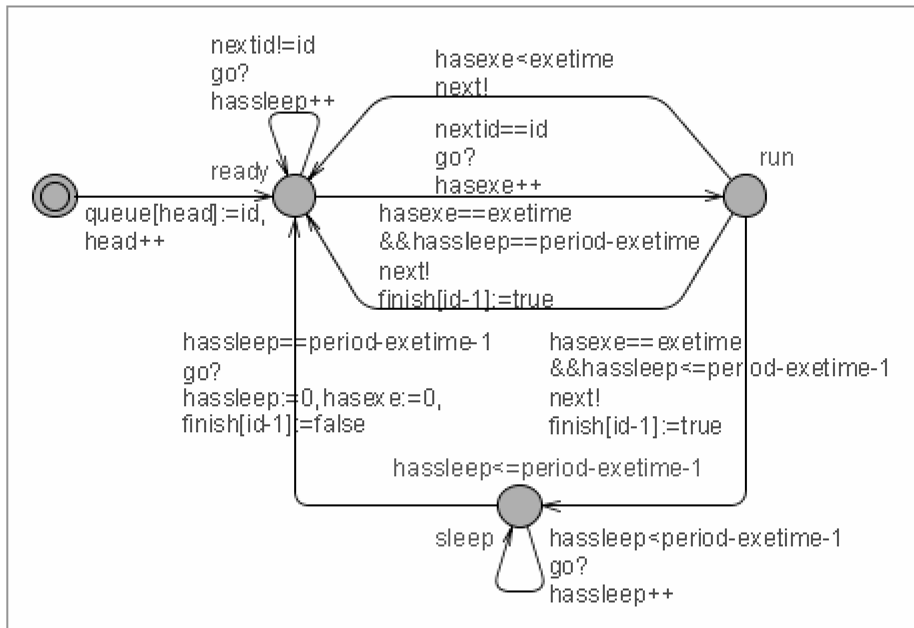

Figure 6.4 Task Template for RR
RR_template(id, exetime, period)

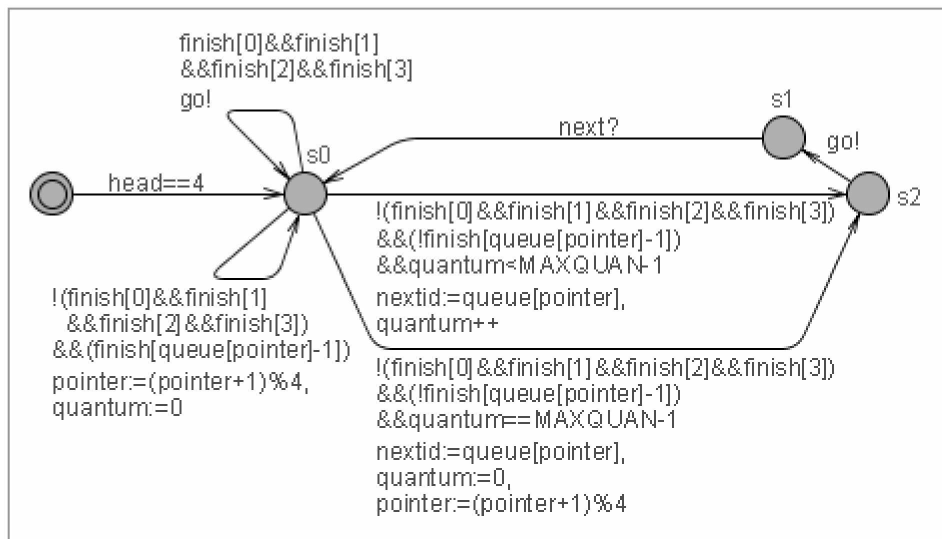

Figure 6.5 Arbiter Automaton for RR
RR_arbiter

The *RR_task_template* cooperates with the *RR_arbiter* through global variables and synchronization messages either, including the *nextid*, the *tick*, and the *next*. Different from the FCFS verification model, a task becoming ready or sleeping

is not handled by adding to or removing from a ready-queue, but by switching a flag. This flag is designed as a Boolean array for all RR Task Automata, called *ready_flag*. Each element in the *ready_flag* indicates whether the corresponding RR Task Automaton is at the *ready* location. See Figure 6.6.

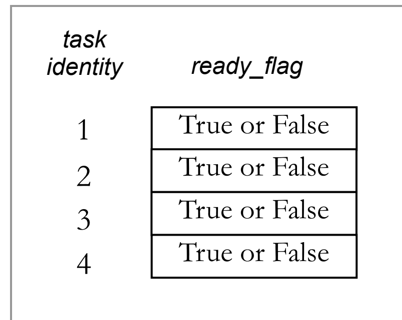

Figure 6.6 Structure of *ready_flag* for RR

The order of RR Task Automata stores in an integer array, called the *queue*, which realizes a round robin fashion queue. Once initializing the order of the *queue*, the order will never be changed during the scheduling. A pointer goes around the *queue*, named *pointer*. See Figure 6.7.

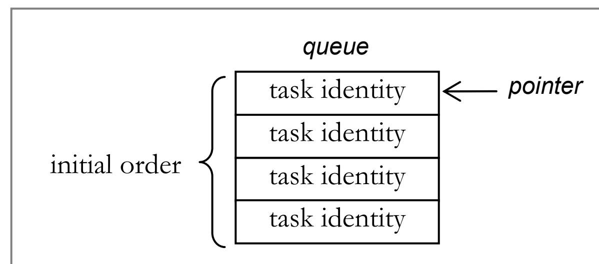

Figure 6.7 Structure of *queue* for RR

Whenever, the *RR_arboter* tries to select a RR Task Automaton to move into the *running* location, it first obtains a task identity from the *queue* referred by the *pointer*. At the same time, the *RR_arbiter* checks the *ready_flag* according to the task identity.

- That the *ready_flag* is true means the RR Task Automaton is ready. The *RR_arbiter* sets the *nextid* as the task identity.
- That the *ready_flag* is false means the RR Task Automaton is not ready-to-run, namely that the execution is finished but the period hasn't expired yet. The *RR_arbiter* moves the *pointer* forward so that it can get next task identity. The *RR_arbiter* selects a RR Task Automaton along with the *queue* until it

obtains a task identity that corresponding RR Task Automaton is ready to run.

The quantum of RR scheduling is designed by assigning different integer to the variable *MAXQUAN*. The quantum can be any positive integer.

The basic structures of the *RR_task_template* and the *RR_arbitier* are mostly like the structure of the *FCFS_task_template* and the *FCFS_arbiter*. Thus, the advantages and disadvantages of RR verification model remains the same, comparing with the verification model in the Chapter 3 and 4.

In the RR verification model, the method distinguishing the ready-to-run tasks and non-ready tasks is different from that of the FCFS verification model. In FCFS, only a ready-queue (the *queue*) is enough to record all of the information that the arbiter needs to select next task. When a FCFS Task Automaton is ready-to-run, its task identity appears in the ready-queue. In RR, for selecting next task, the arbiter requires not only a ready-array (the *ready_flag*), but also an order-queue (the *queue*) to remember the initialized order of the tasks. That is because the task order in RR is relatively fixed. For example, initially, **A** is ahead of **B**. **A** will always be ahead of **B**, whenever **A** and **B** are both ready-to-run. This fixed order doesn't exist in FCFS.

3.2.2 RR Verification Model and Property

The RR Task Automata of the **Tm**, the **Track**, the **Read** and the **Play**, are instantiated from the *RR_task_template*, shown as follows.

```
RR_Tm       := RR_task_template(1, 3, 8);
RR_Track    := RR_task_template(2, 2, 24);
RR_Read     := RR_task_template(3, 1, 12);
RR_Play     := RR_task_template(4, 5, 500);
```

The RR verification model is

```
system RR_Tm, RR_Track, RR_Read, RR_Play,
      RR_arbiter;
```

The property remains the same.

```
A[] not deadlock
```

3.2.3 RR Verification Result

The property is verified under different quantum (the *MAXQUAN*), ranging from 1ms to 8ms. For all kinds of the *MAXQUAN*, UPPAAL reports "not satisfied" results. However, the results are affected by the unchangeable order of the

queue. For the four tasks, the order has 24 possibilities. Thus, we verify the property under each order. The results are listed in the Appendix A. Here, we only take *MAXQUAN=2ms* as an example to explain what we have found. See Table 6.4.

Table 6.4 RR Verification Results Table (Quantum = 2ms)

MAXQUAN=2ms				
Initial Order				**Verification Result**
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable
3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	The **Tm** misses deadline.
4	1	3	2	The **Tm** misses deadline.
4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.

From Table 6.4, we observe that the missing deadline will never happen in some order. For instance, the initialized order is (1, 2, 3, 4), which means that the **Tm** is always ahead of the **Track**; the **Track** is always ahead of the **Read**; and the **Read** is always ahead of the **Play**. In other orders, missing deadline will appear. For instance, the initialize order is (2, 3, 4, 1), which means that the **Track** is always ahead of the **Read**; the **Read** is always ahead of the **Play**; the **Play** is always ahead of the **Tm**.

Comparing these eight results tables of different quanta, we find that when the quantum is equal to 1ms, 3ms, 4ms, and 5ms, the four tasks are impossible to be scheduled no matter what the initialized order of the *queue* is. When the quantum is equal to 2ms, 6ms, 7ms or 8ms, we must specify the initialized order of the *queue* so that the tasks can be scheduled. At the same time, most of the missing deadlines are caused by the **Tm**. Only four missing deadlines are cause by the **Read**, when the order of the *queue* is (4, 1, 2, 3).

3.3 Verifying Preemptive FPS Scheduling

3.3.1 Task Template and Arbiter Automaton for Preemptive FPS

Preemptive FPS scheduling also can be evaluated using a preemptive FPS verification model, which is composed with four preemptive FPS Task Automata and a preemptive FPS Arbiter Automaton. The four preemptive FPS Task Automata are instantiated from the preemptive FPS Task Template by inputting Architecture Task Models in Table 6.2. The preemptive FPS Task Template (named *FPS_task_template*) and the preemptive FPS Arbiter Automaton (named *FPS_arbiter*) are shown in Figure 6.8 and 6.9 respectively.

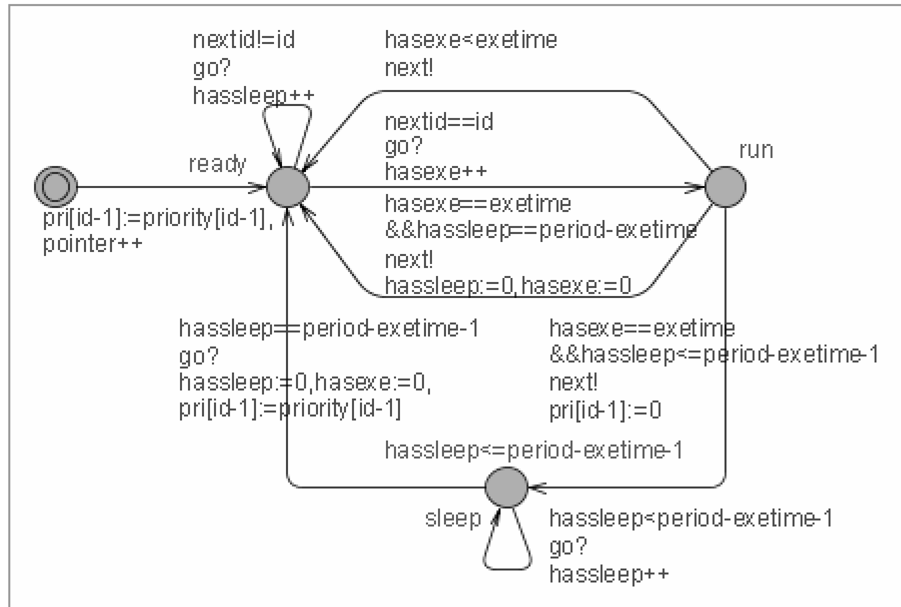

Figure 6.8 Task Template for Preemptive FPS
FPS_task_emplate(id, exetime, period)

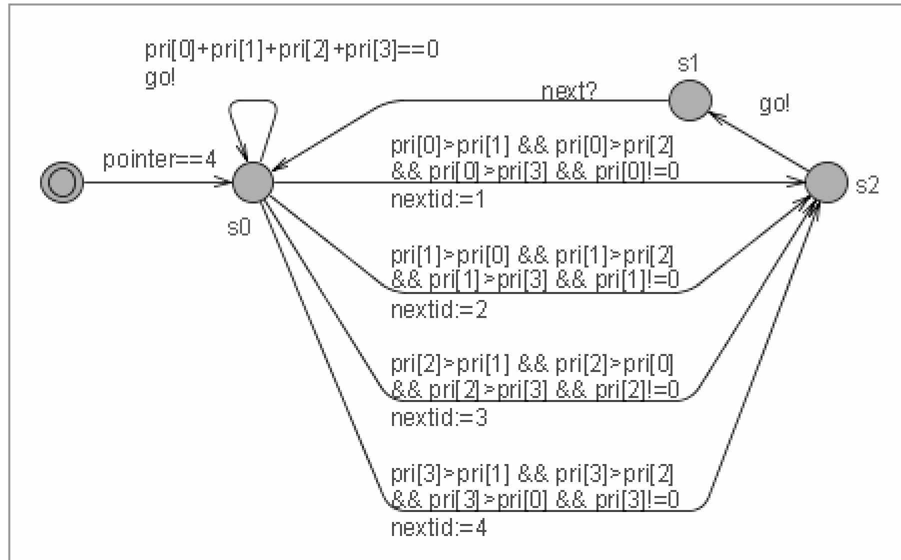

Figure 6.9 Arbiter Automata for Preemptive FPS
FPS_arbiter

In the preemptive FPS verification model, the *FPS_task_template* cooperates with the *FPS_arbiter* through global variables and synchronous messages either, including the *nextid*, the *tick*, and the *next*. There are also another two integer arrays, the *priority* and the *pri*. The *priority* array stores the initial priorities for each preemptive FPS Task Automata. The *pri* array is used to distinguish whether a preemptive FPS Task Automaton can be selected by the *FPS_arbiter* or not. If a preemptive FPS Task Automaton hasn't finished execution, its corresponding elements in the *pri* array (i.e. *pri[id-1]*) is equal to its initial priority (i.e. *priority[id-1]*). Otherwise, the *pri[id-1]* is set to 0. See Figure 6.10.

task identity	*priority*	*pri*
1	task priority	task priority or 0
2	task priority	task priority or 0
3	task priority	task priority or 0
4	task priority	task priority or 0

Figure 6.10 Structure of *priority* and *pri* for Preemptive FPS

When a preemptive FPS Task Automaton enters the *ready* location, it assigns

the *pri[id-1]* as the *priority[id-1]*. The *FPS_arbiter* selects the greatest integer from the *pri* array and sets the *nextid* as corresponding task identity. Meanwhile, the *FPS_arbiter* sends a *tick* message (*tick!*). When a preemptive FPS Task Automaton receives the *tick* message (*tick?*), it compares its own task identity with the *nexid*. If they match, the preemptive FPS Task Automaton begins to run; otherwise, it sleeps. Once the execution is finished, the preemptive FPS Task Automaton sets *pri[id-1]* to 0, so that it is impossible to be selected before the period is over.

The advantages and disadvantages of FCFS verification model remains. However, flexibilities of the FPS verification model are less than the FCFS verification model and the RR verification model. Because if the number of the Task Automata increases,

- in the FCFS model, we need to 1) increase the capability of the *queue* and 2) increase the Constance in the *FCFS_task_template*.
- in the RR model, we need to 1) increase the capability of the *ready_flag* and the *queue* , 2) increase the constance in the *RR_arbiter*, and 3) modify the assignments and/or guards of the self-circular transitions of the *s0* location and the transitions from the *s0* to the *s2* location.
- in the preemptive FPS model, we need to 1) increase the capability of the *priority* and the *pri*, 2) modify the assignment and/or guards of self-circular transitions of the *s0* location and the transitions from the *s0* to the *s2* location, and 3) add new transitions from the *s0* to the *s2* location.

The modification in the preemptive FPS model is greater and more complicit than the FCFS model and the RR model.

3.3.2 FPS Verification Model and Property

The FPS Task Automata of the **Tm**, the **Track**, the **Read** and the **Play**, are instantiated from the *FPS_task_template*, shown as follows.

```
FPS_Tm      := FPS_task_template(1, 3, 8);
FPS_Track   := FPS_task_template(2, 2, 24);
FPS_Read    := FPS_task_template(3, 1, 12);
FPS_Play    := FPS_task_template(4, 5, 500);
```

The FPS verification model is

```
system FPS_Tm, FPS_Track, FPS_Read, FPS_Play,
      FPS_arbiter;
```

The property remains the same.

```
A[] not deadlock
```

3.2.3 Preemptive FPS Verification Result

The result reported from UPPAAL is “not satisfied”, which means the four tasks cannot be scheduled by using preemptive FPS scheduling. However, the result is affected by the initial priority of each preemptive FPS Task Automaton. It has 24 possibilities. Based on every possibility, we verify the property again. The verification results are listed in Table 6.5.

Table 6.5 Preemptive FPS Verification Results Table

Original priority				
Tm	**Track**	**Read**	**Play**	***Verification Result***
1	2	3	4	The **Tm** misses deadline.
1	2	4	3	The **Tm** misses deadline.
1	3	2	4	The **Tm** misses deadline.
1	3	4	2	The **Tm** misses deadline.
1	4	2	3	The **Tm** misses deadline.
1	4	3	2	The **Tm** misses deadline.
2	1	3	4	The **Tm** misses deadline.
2	1	4	3	The **Tm** misses deadline.
2	3	1	4	The **Tm** misses deadline.
2	3	4	1	Schedulable
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	Schedulable
3	1	2	4	Schedulable
3	1	4	2	Schedulable
3	2	1	4	The **Read** misses deadline.
3	2	4	1	Schedulable
3	4	1	2	The **Read** misses deadline.
3	4	2	1	Schedulable
4	1	2	3	Schedulable
4	1	3	2	Schedulable
4	2	1	3	The **Read** misses deadline.
4	2	3	1	Schedulable
4	3	1	2	The **Read** misses deadline.
4	3	2	1	Schedulable

From Table 6.5, we know that in 10 of 24 priority assignments, the architecture tasks can be scheduled. For example, the priority assignment is (2, 3, 4, 1), which means the priority of the **Tm** is 2; the priority of the **Track** is 3; the

priority of the **Read** is 4; and the priority of the **Play** is 1. In other 14 possibilities of priority assignments, the missing deadline will happen and are caused by either the **Tm** or the **Read**.

Comparing the results, we find that when the priority of the **Tm** is equal to 1 or 2, the **Tm** mostly (i.e. 10 of 12 possibilities) causes the missing deadline. When the priority of the **Read** is 1, it mostly (i.e. 4 of 6 possibilities) causes the missing deadline. Therefore, the priority assignments for the **Tm** and the **Read** should be careful. It is better that, the priority of the **Tm** is greater than 2, and the priority of the **Read** is greater than 1.

4. Summary

In the architecture level, the verification models characterized from FCFS, RR, and preemptive FPS are applicable. UPPAAL can report the result within 2 minutes, which is acceptable. However, these three scheduling algorithm are not as good as we hope. The reason is that we must specify the four architecture tasks such as specifying the arrival order in FCFS, specifying the initial order of round robin queue in RR, and specifying the priority assignments in preemptive FPS.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we first raise a problem of software development. The problem is about the technique to guarantee system correctness. In the V-Model, Testing is used as the technique. However, Testing has some weakness. Thus, we introduce another technique, Formal Verification. When we are planning to integrate Formal Verification into the SDLC of the V-Model, we face two problems. One is about how to integrate it. The other one is about whether the integrating is practical. In order to resolve these two problems, we first construct a conceptual model, named Integrating Process. Later on, we use the ELM-2 project as the case study to check practicability and details of our solution.

In the implementation level, we haven't got any result due to the failure of verification. In the architecture level, we obtain satisfied results within acceptable time. From experiments in the architecture level, we may conclude that

- The integrating process is applicable
- By integrating Formal Verification, the verification engineers are able to give their findings to the software development engineers, if verification work succeeds.
- The concerns and the shared knowledge in the intermediate model must be understandable for both the software development engineers and the verification engineers.
- To characterize verification models from the concerns requires expert skills.
- The resulting verification models and available verification tools limit the scope of the applicability of formal verification methods.

In the future, we should answer following two questions. These questions are about the case study.

- Because in the implementation level we haven't got any result, the question is that whether it is possible to characterize a verifiable model.
- Even though we obtain some satisfied results in the architecture level, we have to characterize the model for each scheduling algorithm. For example, for FCFS we characterize the *FCFS_task_template* and the *FCFS_arbiter*; for RR we characterize the *RR_task_template* and the *RR_arbiter*. Is it possible we characterize a model that can be used for either FCFS, RR, or preemptive FPS scheduling algorithm?

BIBLIOGRAPHY

[AD94] R. Alur and D. L. Dill, "A Theory of Timed Automata", Theoretical Computer Science, 1994.

[Boe88] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," Computer, 1988.

[ER03] A. Endres and D. Rombach, "A handbook of software and systems engineering", Addison-Wesley, 2003

[CES86] E. M. Clarke, E. A. Emerson, and A.P.Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", ACM Transactions on Programming Languages and Systems, 1986.

[Fug00] A. Fuggetta, "Software process: a Roadmap", In Proceedings of the Conference on The Future of Software Engineering, ACM Press, 2000.

[HHW97] T.A. Henzinger, P. -H. Ho, and H. Wong-Toi, "HyTech: A Model Checker for Hybrid Systems", Software Tools for Technology Transfer, 1997.

[JPK99] J. P. Katoen, "Concepts, Algorithms, and Tools for Model Checking", 1999.

[LL73] C. L. Liu and J. W. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment", Journal of the ACM 20, 1973.

[LL01] C. L. Liu and J. W. Layland, "Analysis of Scheduling Behaviour using Generic Timed Automata", Elsevier Science B. V., 2001.

[KZ95] J. van. Katwijk and J. Zalewski, "Introduction to Real-time software systems (Draft Edition)", 1995.

[Pnu77] A. Pnueli, "The temporal logic of programs", 18th IEEE Symposium on Foundations of Computer Science, USA, 1977.

[Roy70] W. W. Royce, "Managing the Development of Large Software Systems", Proceedings of IEEE WESCON, 1970.

[STA98] R. F. Lutje Spelberg, W. J. Toetenel, and M. Ammerlaan, "Partition refinement in real-time model checking", In Proceedings of the 5th international Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, 1998.

[Yov97] S. Yovine, "Kronos: A Verification Tool for Real-Time Systems", International Journal of Software Tools for Technology Transfer, 1997.

[VSS95] T. Villa, G. Swamy and T. Shiple, "VIS User's Manual", University of California, Berkeley, 1995.

OTHER RESOURCES

[1] R.Wang, "Research Report: Integrating Formal Verification into the Software Development Process", 2004

[2] Experiments with Lego MindStorm, http://www.isa.ewi.tudelft.nl/~moose/restricted/index.php/Elm (May, 2004)

[3] Java Homepage, http://www.java.com (May, 2004)

[4] UPPAAL Homepage, http://www.uppaal.com (March, 2004)

[5] Kronos Homepage, http://www-verimag.imag.fr/TEMPORISE/kronos/ (March, 2004)

[6] HyTech Homepage, http://www.eecs.berkeley.edu/~tah/HyTech (March, 2004)

[7] Das V-Modell, http://www.v-modell.iabg.de (January, 2004)

[8] Embedded.com, http://www.embedded.com/story/OEG20010821S0096 (August, 2004)

APPENDIX A

VERIFICATION RESULTS
-- For RR Scheduling In the Architecture Level

In following tables, the schedulable means that based on the initial order, the four tasks in the architecture level can be scheduled by using the RR. The four tasks are the **Tm**, the **Track**, the **Read**, and the **Play**.

MAXQUAN=1	
Initial Order	***Verification Result***
Any order	The **Tm** misses deadline.

MAXQUAN=2				
Initial Order				***Verification Result***
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable
3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	The **Tm** misses deadline.
4	1	3	2	The **Tm** misses deadline.

4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.

MAXQUAN=3	
Initial Order	***Verification Result***
Any order	The **Tm** misses deadline.

MAXQUAN=4	
Initial Order	***Verification Result***
Any order	The **Tm** misses deadline.

MAXQUAN=5				
Initial Order				***Verification Result***
4	1	2	3	The **Read** misses deadline.
Other orders				The **Tm** misses deadline.

MAXQUAN=6				
Initial Order				***Verification Result***
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable

3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	The **Read** misses deadline.
4	1	3	2	Schedulable
4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.

MAXQUAN=7				
Initial Order				***Verification Result***
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable
3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	The **Read** misses deadline.
4	1	3	2	Schedulable
4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.

MAXQUAN=8				
Initial Order				***Verification Result***
1	2	3	4	Schedulable
1	2	4	3	Schedulable
1	3	2	4	Schedulable
1	3	4	2	Schedulable
1	4	2	3	Schedulable
1	4	3	2	Schedulable
2	1	3	4	Schedulable
2	1	4	3	Schedulable
2	3	1	4	Schedulable
2	3	4	1	The **Tm** misses deadline.
2	4	1	3	The **Tm** misses deadline.
2	4	3	1	The **Tm** misses deadline.
3	1	2	4	Schedulable
3	1	4	2	Schedulable
3	2	1	4	Schedulable
3	2	4	1	The **Tm** misses deadline.
3	4	1	2	The **Tm** misses deadline.
3	4	2	1	The **Tm** misses deadline.
4	1	2	3	The **Read** misses deadline.
4	1	3	2	Schedulable
4	2	1	3	The **Tm** misses deadline.
4	2	3	1	The **Tm** misses deadline.
4	3	1	2	The **Tm** misses deadline.
4	3	2	1	The **Tm** misses deadline.