Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Neil Rajesh Dhruva

Crossing the Deadline: An Automata-Based Hard Real-Time Guarantee

Semester Thesis
July 2013 to December 2013
Code: BA-2013-17

Supervisors: Pratyush Kumar, Georgia Giannopoulou
Professor: Prof. Dr. Lothar Thiele

Abstract

Real-time systems are often guaranteed in terms of schedulability, which verifies whether or not all jobs meet their deadlines. However, such a guarantee can be insufficient in certain applications. This thesis presents a method to compute a language-based guarantee, which provides a more detailed description of the deadline hit and miss patterns of an observed task. The only requirement of such a method is that the timing behavior of the real-time system be modelled by a network of timed automata. The language-based guarantee is computed by constructing an equivalent finite state automaton in an iterative manner, using a counter-example guided abstraction refinement procedure. The method is further extended to generate a guarantee for a language of unknown complexity. The method is then illustrated for two different applications: design of a networked control system, and scheduling in a mixed criticality system. In both cases, it is shown how the language-based guarantee leads to a more efficient design than the schedulability guarantee. Finally, several experimental results are presented which explore various properties of the language-based guarantee, such as scalability, and the effect of variations in certain system parameters, like the total utilization of a shared resource.

Acknowledgement

This thesis was an incredible learning experience for me. Along with an exposure to fields like formal verification, and real-time systems, I was able to get my first tangible insight into independent research.

None of this, however, would have been possible without my mentors, Pratyush Kumar, Georgia Giannopoulou and Prof. Dr. Lothar Thiele. I would like to thank them for accepting me as a thesis student at the Computer Engineering and Networks Laboratory at ETH, Zurich. I would like to thank Pratyush and Georgia for proposing a very unique and interesting topic for my thesis. However, never having worked in the fields of formal verification and real-time systems before, I was skeptical whether I will be up to the task. But, the guidance from Pratyush, Georgia and Prof. Thiele, and their continued support and advice, made it very easy for me to complete the thesis successfully.

It was indeed a great experience working with the Computer Engineering group at the TIK Laboratory at ETH, Zurich. I'd like to thank the researchers and staff for making me feel welcome at the lab. I'd especially like to thank Beat Futterknecht for helping me with my visa, stay and the administrative work required for the thesis.

Finally, I would like to thank BITS, Pilani University, India, for allowing me the opportunity to conduct my Bachelor's thesis at ETH, Zurich.

Contents

1 Introduction **6**

2 System Model and Definitions **8**
2.1 Real-Time System . . . 8
2.2 Modelling Real-Time System in UPPAAL . . . 9
2.2.1 Modelling the Real-Time Tasks . . . 9
2.2.2 Modelling Multiple Tasks per Core . . . 12
2.2.3 Modelling the Arbiters . . . 14
2.3 Guarantee Language L_G . . . 17
2.4 Representing the Language . . . 17
2.4.1 Language Definition and Properties . . . 18
2.5 Summary . . . 19

3 Computation of the Guarantee Automaton **20**
3.1 Constructing A_k for a given k . . . 20
3.1.1 The Observer Automaton . . . 20
3.1.2 Modelling the Observer in UPPAAL . . . 22
3.1.3 Iterative Construction . . . 23
3.1.4 Properties of the Computed $L_G(k)$ . . . 25
3.1.5 Minimizing the Guarantee Automaton . . . 26
3.2 Constructing the Guarantee Automaton For Unknown k . . . 27
3.2.1 Refining $L_G(k)$ by Increasing k . . . 27
3.2.2 Implementing Iterative Refinement in UPPAAL . . . 28
3.2.3 Terminating Condition . . . 30
3.3 Summary . . . 31

4 Applications **33**
4.1 Language Inclusion . . . 33
4.2 Calculating Worst-Case Deadline Miss Rate . . . 36
4.3 Summary . . . 37

5 Experiments **38**
5.1 Randomized Task-Set Generation . . . 38
5.1.1 Notations . . . 39
5.1.2 Algorithm . . . 39
5.1.3 Comments . . . 40
5.2 Experimental Results . . . 40
5.2.1 Variation in Deadline . . . 40
5.2.2 Variation in Bus (Resource) Utilization . . . 41
5.2.3 Scalability Experiments . . . 42

6 Conclusion **44**

A Appendix A **45**
A.1 Code for Inverted Pendulum Example 45
A.2 The Shell Script 47
A.3 The `verifyta` Options 47
A.4 Outline to the Python scripts 47

B Appendix B **52**
B.1 Experiment Tables 52
B.1.1 Preliminary Experiments 52
B.1.2 Test Case: Varying Deadline 69
B.1.3 Test Case: Varying Bus (Resource) Utilization 70

List of Figures

2.1 Superblock model for one task per core . . . 11
2.2 Superblock model for multiple tasks per core . . . 12
2.3 Preemptive Fixed Priority Scheduler on the core. . . . 13
2.4 FCFS with Fixed Access Latency . . . 14
2.5 FCFS with Variable Access Time . . . 15
2.6 FP with Fixed Access Latency . . . 16
2.7 TDMA with Fixed Access Latency . . . 16
2.8 Representing a Regular Language . . . 18

3.1 Observer for $k = 2$ in UPPAAL. . . . 21
3.2 Observer for $k = 2$ in UPPAAL. . . . 22
3.3 Superblock with `hit!` and `miss!` events. . . . 23
3.4 Counter-example trace generated by `verifyta`. . . . 24
3.5 Modified Observer after first counter-example. . . . 25
3.6 Observer O for $k = 2$. . . . 26
3.7 Guarantee Automaton A_2. . . . 26
3.8 The Guarantee for $k = 3$. . . . 27
3.9 Corresponding Minimized Guarantee. . . . 27
3.10 Initial Observer for $k = 3$ from A_2. . . . 28
3.11 Guarantee Automaton A_3. . . . 28
3.12 Initial Observer for $k = 1$. . . . 29
3.13 Guarantee Automaton A_1. . . . 29
3.14 Modified Observer for $k = 2$. . . . 29
3.15 Guarantee Automaton A_2. . . . 29
3.16 Observer for $k = 3$ in UPPAAL derived from A_2. . . . 30

4.1 Minimized Automaton for L_A. . . . 35
4.2 Minimized Automaton for L_G. . . . 35
4.3 Automaton representation of $L_{\texttt{diff}}$. . . . 36

5.1 WMR vs Deadline. . . . 41
5.2 Uncertainty Metric vs Deadline. . . . 41
5.3 Time vs Deadline. . . . 41
5.4 WMR vs Bus Utilization. . . . 42

List of Tables

2.1 Task model for Example 1 . . . 9
2.2 Guarantee language $L_G(2)$ for Example 1 . . . 19

3.1 Potential Indices for Stopping Condition. . . . 30
3.2 Uncertainty metric for Example 1 . . . 31

4.1 Task parameters for the control example . . . 35
4.2 Mixed criticality task-set . . . 37
4.3 Mixed criticality results . . . 37

5.1 Notations . . . 39
5.2 Scalability for 1 task per core. . . . 43
5.3 Scalability for 2 tasks per core. . . . 43
5.4 Scalability for 3 tasks per core. . . . 43

B.1 . . . 53
B.2 . . . 53
B.3 . . . 53
B.4 . . . 53
B.5 . . . 54
B.6 . . . 54
B.7 . . . 54
B.8 . . . 54
B.9 . . . 54
B.10 . . . 54
B.11 . . . 55
B.12 . . . 55
B.13 . . . 55
B.14 . . . 55
B.15 . . . 55
B.16 . . . 55
B.17 . . . 55
B.18 . . . 56
B.19 . . . 56
B.20 . . . 56
B.21 . . . 56
B.22 . . . 56
B.23 . . . 56
B.24 . . . 56
B.25 . . . 57
B.26 . . . 57
B.27 . . . 57
B.28 . . . 57
B.29 . . . 57
B.30 . . . 57
B.31 . . . 57
B.32 . . . 58

B.33 58
B.34 58
B.35 58
B.36 58
B.37 58
B.38 58
B.39 59
B.40 59
B.41 59
B.42 59
B.43 59
B.44 59
B.45 59
B.46 60
B.47 60
B.48 60
B.49 60
B.50 60
B.51 60
B.52 60
B.53 61
B.54 61
B.55 61
B.56 61
B.57 61
B.58 61
B.59 61
B.60 62
B.61 62
B.62 62
B.63 62
B.64 62
B.65 62
B.66 62
B.67 63
B.68 63
B.69 63
B.70 63
B.71 63
B.72 63
B.73 63
B.74 64
B.75 64
B.76 64
B.77 64
B.78 64
B.79 64
B.80 64
B.81 65
B.82 65
B.83 65
B.84 65
B.85 65
B.86 65
B.87 65
B.88 66
B.89 66
B.90 66

B.91 . . . 66
B.92 . . . 66
B.93 . . . 66
B.94 . . . 66
B.95 . . . 67
B.96 . . . 67
B.97 . . . 67
B.98 . . . 67
B.99 . . . 67
B.100 . . . 67
B.101 . . . 67
B.102 . . . 68
B.103 . . . 68
B.104 . . . 68
B.105 . . . 68
B.106 . . . 68
B.107 . . . 68
B.108 . . . 68
B.109 . . . 69
B.110 . . . 69
B.111 . . . 69
B.112 . . . 69
B.113 . . . 69
B.114 . . . 69
B.115 . . . 70
B.116 . . . 70
B.117 . . . 70
B.118 . . . 70
B.119 . . . 70
B.120 . . . 70
B.121 . . . 71
B.122 . . . 71
B.123 . . . 71
B.124 . . . 71
B.125 . . . 71
B.126 . . . 71
B.127 . . . 71
B.128 . . . 72
B.129 . . . 72
B.130 . . . 72
B.131 . . . 72
B.132 . . . 72
B.133 . . . 72
B.134 . . . 72
B.135 . . . 73
B.136 . . . 73

Chapter 1

Introduction

Real-time systems are designed to meet specific timing constraints in the form of deadlines. Schedulability analysis of a real-time task-set guarantees whether each task meets its deadline in every execution of the task-set. Missing the deadline often results in a complete system failure. In such cases, schedulability analysis would discard such a task-set as non-schedulable, and no further analysis would be necessary. However, in many applications, such as in control systems and mixed criticality systems, missing the deadline will not always result in a total system failure.

Consider the example of a networked control system [1]. The timing properties of the communication network may be analyzed with real-time schedulability tests which guarantee whether all jobs meet their deadlines. On the other hand, given the worst-case sensor-to-actuator delay, the physical plant can be stabilized with a controller design for time delayed feedback systems [2]. A key challenge in the composition of such sub-systems is the provision of tight and compact *guarantees*.

The real-time systems community has primarily focused on providing schedulability as a guarantee. From the perspective of the real-time system, if a task-set is schedulable, i.e., if all jobs are guaranteed to finish on or before their deadlines, then and only then, will all other sub-systems perform correctly. A large body of research has investigated methods and tools to verify such a guarantee under different assumptions of task models, scheduling policies and resource considerations.

However, schedulability can be insufficient in certain settings. For the networked control system example, the physical plant may be able to withstand a few missed deadlines. Indeed, in [3], the authors show that a well-defined class of deadline hit and miss patterns can guarantee stability of a physical plant. Similarly, consider the case of a dual-criticality system with High (HI) and Low (LO) criticality tasks. In the high criticality mode, the LO criticality tasks may produce useful results at regular intervals when the corresponding deadlines are met. Schedulability analysis of such systems, however, would conclude that the task-set is not schedulable and thus should be discarded. On the other hand, studying these patterns of deadline hits and misses could yield useful results. *Hence, it is pertinent to look for richer (more detailed) guarantees that extend beyond schedulability.*

Researchers have proposed alternate richer guarantees. In [4], the authors proposed two metrics on the deadline hit and miss patterns, which they called μ-patterns. A $\binom{n}{m}$ μ-pattern has at least n deadline hits within m consecutive jobs, and a $\left\langle{n \atop m}\right\rangle$ μ-pattern has at least n consecutive deadline hits within any m consecutive jobs. These patterns can be generalized by the notion of regular languages as proposed in [3]. A slightly different approach was followed in [5]. The authors propose a dual guarantee. A nominal guarantee is the typical schedulability test, while an exceptional guarantee specifies for how long deadlines can be missed after an exceptional event.

For the above, a major challenge is the computation or verification of the guarantees. The

μ-patterns approach has been demonstrated for fixed-priority scheduling in [4]. The use of regular languages has been shown for time-triggered architectures in [6]. In [5], the settling-time approach is applied when modeling in Real-Time Calculus [7]. It is not clear how to compute these guarantees in a general setting.

Like in [3], a regular language is used in this thesis to model the patterns of deadlines hits and misses as a guarantee. However, *the main focus lies on computing the guarantee for a broad class of scheduling algorithms and task models.* To this end, the real-time system is represented by a network of timed automata [8] in the model-checking tool UPPAAL [9]. Then, the aim is to compute a *language-based guarantee* by model-checking for different candidate languages until the one that correctly represents the observed hit and miss patterns is found. Thus, this approach is limited only by the ability to model a real-time system as a network of timed automata and by the computational cost of model-checking.

In spite of their generic applicability, a common limitation of model-checking tools is the state-space explosion and the consequent high computation cost. Being conscious of this, two specific choices are made in the model-checking process. First, a structured approach is used to identify candidate languages to model-check. To this end, an iterative procedure is employed to incrementally compute a language of a given complexity that models the observed deadline hit and miss patterns. This is similar to the counter-example guided abstraction refinement (CEGAR) [10] approach proposed for program verification.

Second, the complexity of the language that is used as the guarantee, is gradually increased. The information gathered from earlier steps is fully re-used while increasing the language complexity. Here, an uncertainty metric is defined to decide when to terminate the iterative process of increasing the language complexity. With these two steps a standard approach is proposed to compute the language-based guarantee.

The method is used for two very different applications of the language-based guarantee. First, language inclusion is used to demonstrate whether the guaranteed real-time performance matches the assumed performance by another sub-system. As an example, the controller performance of an inverted pendulum with a Linear Quadratic Regulator is presented. For chosen system parameters, the controller performance cannot be ascertained by the schedulability guarantee, but is ascertained by the language-based guarantee. In the second application, the design of mixed-criticality systems [11] is directed to 'fairly' distribute resources among low-criticality tasks in exceptional cases when high-criticality tasks require more resources than usual. This is done using a worst-case deadline miss rate metric. Such a design step is not possible with the schedulability guarantee.

Various different experiments are also presented which test different properties of the guarantee generation procedure, like the model-checking time. The task-sets for many of these experiments are constructed using an algorithm which allows users to vary different task parameters. Consequently, the effect of changes in parameters, like the total shared resource utilization, can be studied.

The rest of the thesis is organized as follows. Chapter 2 discusses the model of the real-time system and formally defines the proposed guarantee. Chapter 3 describes two approaches to compute the language-based guarantee. In Chapter 4, two applications of the language-based guarantee are presented and illustrated with numerical examples. Finally, Chapter 5 presents different experiments conducted using the guarantee generation procedure.

Chapter 2

System Model and Definitions

In order to describe the guarantee generation process, certain preliminary definitions and illustrations are necessary. The chapter begins with the definition of a model of the Real-Time System (RTS) that is used as a standard for this thesis. This is the general layout of the system that various task-sets used for experiments adhere to as well. The model definition is followed by a discussion of timed-automata [8], and illustrative examples of constructing RTS models using timed-automata in UPPAAL.

Following the UPPAAL illustrations, the chapter goes on to describe the guarantee that is generated on the deadline hit and miss patterns of a particular task in a real-time task-set. The aim is to generate the guarantee in the form of a regular language. As we will see later, a regular language admits advantages which will be useful in model-checking. Finally, various properties of the proposed language are presented.

2.1 Real-Time System

The RTS under consideration has a task-set, $T = \{\tau_0, \tau_1, ..., \tau_n\}$, which runs on a single or multiple processors with blocking access to shared resources. The language-based guarantee (which characterizes the deadline hit and miss patterns of the jobs of a task) is computed for a specific task, say $\tau_i \in T$.

The only requirement of this approach is that the timing behavior of such a system be modeled by a network of timed automata, which we denote as S (for system). Thus, this approach is restricted only by the modeling power of timed automata and the computational cost of model-checking.

The real-time system is modelled in the following manner:

- The tasks are periodic in nature.
- The tasks share a resource (e.g. memory, bus etc.), which is used to read and write data.
- Each task $\tau \in T$ has three phases: *the acquisition phase* (read data from shared resource), *the execution phase* (execute on a given core), and *the replication phase* (write data back to the resource). Such phased models of tasks have been shown to model many practical applications in the control and real-time domains [13].
- Access to the shared resource, and hence the representation of the above phases, is in one of two forms: Fixed Access Latency (FAL) or Varying Access Time (VAT).
- In case of FAL, a given multiple of the time unit, say C, is used to monitor access to the resource by a given task. After C time units, the arbiter assigns access to the resource to a different task (or again to the same task) depending on the arbitration policy, and if this or any other task is still waiting for the resource.

- For VAT, the task that gains access to the shared resource completes the entire read or write before access is granted to another task.
- Access to the shared resource is through an arbiter. The arbitration policies experimented with in this thesis are non-preemptive First Come First Serve (FCFS), Fixed Priority (FP) and Time Division Multiple Access (TDMA).
- There are either one or more tasks that execute on a given core.
- For more than one task executing per core, preemptive FP scheduling is used on each core. Additionally, access to the shared resource is asynchronous. Hence, a lower priority task can execute on the core when the higher priority task is accessing the resource.
- The deadline of the task under consideration is specified along with the access and execution ranges while modelling the real-time system.
- Resource contention and non-preemptive access to the shared resource result in the deadline hit and miss patterns that are expressed through the computed language.

Using the above layout, the real-time system task-sets are modelled in UPPAAL. A particular task, $\tau_i \in T$, is observed in order to generate the guarantee language for that task.

Example 1. *Consider a task-set T with three periodic tasks $\tau_0, \tau_1, \tau_2 \in T$ running on three separate cores and accessing a shared memory. Each task has three distinct phases: read data from memory, execute on respective cores using the read data, and write modified data back to memory. The period and the minimum and maximum time units for each phase of each task are shown in Table 2.1, and correspond to the time used by the task after access to the resource is granted. Hence, contention time is not mentioned in the table. The arbitration on the memory follows a non-preemptive first-come-first-serve policy.*

Table 2.1: Task model for Example 1

Task	Read	Execute	Write	Period
τ_0	1-2	1-3	1-2	15
τ_1	1-2	10-14	1-2	40
τ_1	1-3	12-15	1-3	50

Once such a task-set is established, it is modelled in UPPAAL using timed-automata. In order to model tasks using UPPAAL, a basic understanding of timed-automata, and features of UPPAAL, is necessary. A detailed description is provided in Section 2.3 of [17]. If one is not familiar with the concepts of timed-automata and UPPAAL, a reference to this thesis is strongly recommended before proceeding forward with the next section.

2.2 Modelling Real-Time System in UPPAAL

This section provides a detailed description of modelling real-time task-sets in UPPAAL. The description begins with the modelling of the tasks on cores, followed by modelling different arbiters that monitor access to the shared resource.

2.2.1 Modelling the Real-Time Tasks

The task-sets, as mentioned above, follow a general layout with three phases: acquisition, execution and replication. A superblock model is used to model suck task-sets in UPPAAL. Such models have been used previously, for instance, in [15] and [17]. To begin with, certain data structures are defined in UPPAAL that are necessary for modelling the superblock task structure. These are included in the *Declarations* template in UPPAAL, and are provided below:

```
typedef int[0,Procs-1] p_id;
typedef int[0,MaxDelay] time_t;
typedef int[0,MaxAccesses] acc_t;

typedef struct {
    acc_t umin_acq;   //Min. read accesses
    acc_t umax_acq;   //Max. read accesses
    acc_t umin_rep;   //Min. write accesses
    acc_t umax_rep;   //Max write accesses
    time_t exec_min;  //Min. execution time
    time_t exec_max;  //Max. execution time
    time_t total_per; //Period
} hsuperblock_t;

const hsuperblock_t hsuperblock[Procs] = {
    {17, 19, 20, 22, 167, 176, 500},
    {11, 12, 11, 13, 66, 70, 400},
    {16, 18, 19, 21, 108, 114, 400}
};

broadcast chan access[Procs];
urgent chan hurry;
broadcast chan hit, miss;     //used for superblock-Observer communication
```

Here `Procs` are the number of tasks (which equals the number of cores for a `1` task/core model), while `MaxDelay` and `MaxAccesses` represent the maximum time units and maximum resource accesses for all tasks, respectively. The `struct` *hsuperblock_t* provides the structure for the superblock. Additionally, the values for three tasks (instances of superblock) are provided using the *hsuperblock* array. Finally, different synchronization channels used in the model are mentioned.

Each task is modelled with one instance of the superblock. Hence, each task has minimum (and maximum) read (and write) accesses, as well as a range for the execution in time units. It is useful to note here that read and write ranges will be in the form of number of access for arbiters using FAL, but will be provided as time units for arbiters using VAT. Such ranges are provided when the access and execution times for a given job of a task are not fixed (which is usually the case).

Based on the structure, a model of the superblock in UPPAAL is created using the GUI. An example is shown in Figure 2.1.

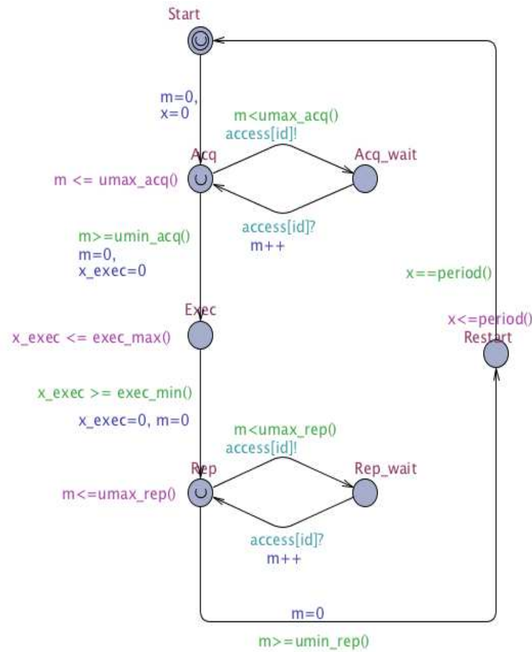

Figure 2.1: Superblock model for one task per core

The Figure 2.1 represents a system with only one task per core. The *Acq* and *Rep* locations are used as read and write locations, respectively. An access request is made to the arbiter by transitioning to the *Acq_wait* and *Rep_wait* locations. The task then waits in these locations until one read or write access is completed in case of FAL. For VAT, this transition is made only once. The *Exec* location is used when the task is in the execution phase.

A particular job of a task that is represented by the superblock above, begins in the *Start* location. When the first edge is traversed, m (the counter for number of accesses) and x (the clock) are set to 0. In UPPAAL, clock variables evaluate to real numbers, and all clocks progress synchronously. For the explanation here, it is assumed that the arbiter uses FAL, and not VAT. However, with minor changes in the model, one for VAT can easily be obtained.

After reaching the *Acq* location, the invariant $m <= umax_acq()$ is checked. This makes sure that the task has not made more read access requests than the maximum possible. A description of the function that returns the maximum read access requests value is provided below:

```
acc_t umax_acq(){
    return hsuperblock[id].umax_acq;
}
```

Once the invariant is satisfied, the task makes a resource access request to the arbiter through the `access[id]!` synchronization channel, where `id` is the task ID. This ID is used by the arbiter to distinguish among requests from different tasks. It is provided to the superblock instance as a Parameter in this case. Additionally, a *Guard* is placed on this transition that checks if $m < umax_acq()$, to make sure that the new request does not exceed the total possible number of read accesses of that task.

The task then waits in the *Acq_wait* location till it has made a successful read access to the resource. This is done using the `access[id]?` synchronization channel. When access to the resource is over, the arbiter provides the `access[id]!` synchronization and the task then moves back to the *Acq* location. Simultaneously, m is increased by 1 to keep count of the number of accesses.

Once the value of m is between the minimum and maximum read access values, the next transition can be taken. m is reset to 0 so it can be used for the same procedure while accessing

the resource for writing data. A new clock x_exec is initialized to keep track of the execution time. The task now enters the *Exec* location where it executes for a certain amount of time, between the minimum and maximum possible execution values. The clock x_exec monitors the time in this phase.

Finally, the task enters the *Rep* location for writing data back to the resource, and follows a similar procedure to that of the *Acq* location. After completing the replication phase, the job of the task is complete, and the task waits in the *Restart* location till the period of that particular job is completed.

Now, in order to initialize the task-set, the *System declarations* template of the UPPAAL model has to be modified. An example of the code for initializing the tasks as superblocks is provided below:

```
S1 = Superblock2(0);
S2 = Superblock(1);
S3 = Superblock2(2);
```

Thus, three instances (in line with the values provided in the *Declarations* template) are instantiated. These represent the three tasks of the task-set.

2.2.2 Modelling Multiple Tasks per Core

In the case of multiple tasks per core, the superblock model is modified to include an additional location, *Blocked*, as shown in Figure 2.2.

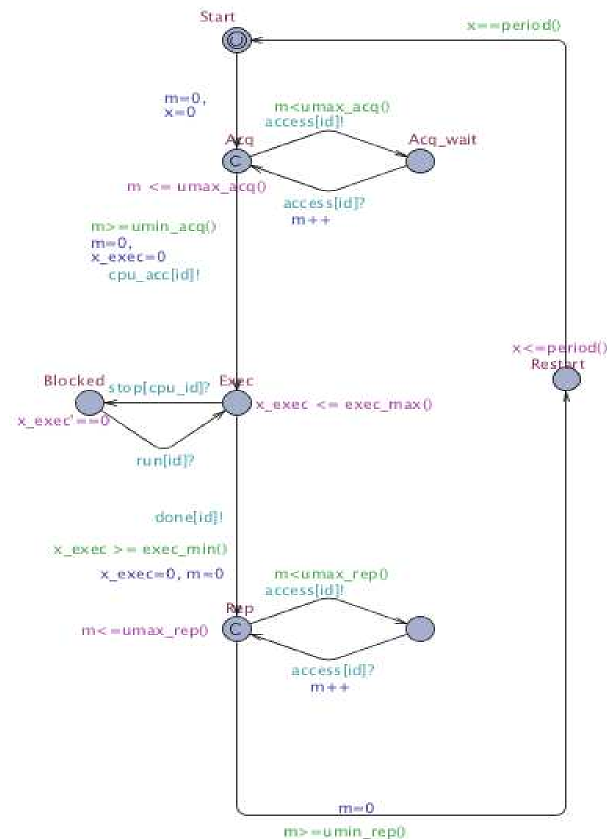

Figure 2.2: Superblock model for multiple tasks per core

Additionally, four more synchronization channels are included in the *Declarations* template as follows:

```
broadcast chan stop[CPU];
broadcast chan run[Procs];
broadcast chan done[Procs];
broadcast chan cpu_acc[Procs];
```

Here, `CPU` is the number of cores while `Procs` is the number of tasks. A feature of UPPAAL called *stopwatches* is used in order to implement preemptive FP scheduling on the core. The `x_exec'==0` invariant is used to preempt a lower priority task when a high priority task starts execution on the core. By itself, the invariant stops `x_exec` from measuring the elapsed time. The lower priority task is preempted because it received the stop signal. As a result, we use the invariant so that no time is measured while the task is blocked.

The preemptive FP scheduler for each core is modelled as shown in Figure 2.3.

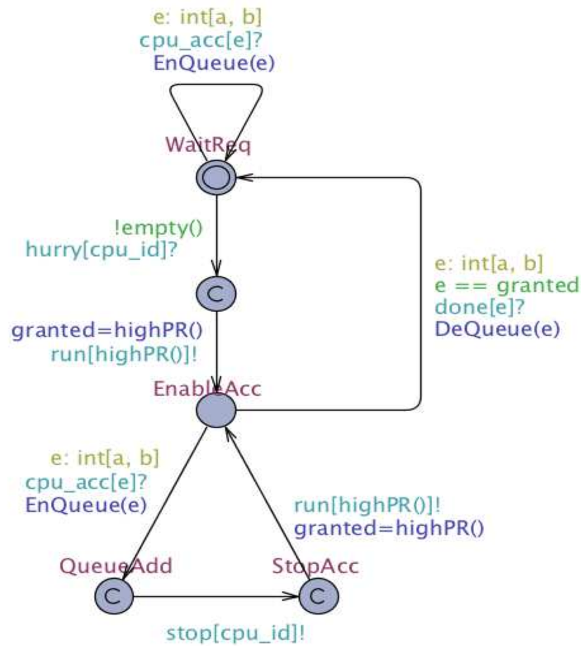

Figure 2.3: Preemptive Fixed Priority Scheduler on the core.

To explain the model of the scheduler, it would be necessary to use both Figure 2.2 and Figure 2.3. The scheduler starts in the *WaitReq* location. When a low priority task finishes read access to the resource, i.e. the acquisition phase, it is directed to the *Exec* location for execution on the core using the read data. During the transition, it sends the `cpu_acc[id]!` synchronization to the scheduler. The scheduler receives it in the *WaitReq* location and adds the task to the scheduler queue. At this point, the x_exec clock has begun measuring the elapsed execution time of that task.

`Hurry[id]?` is an urgent channel that is used to start off the scheduler whenever there is a task waiting in its queue. Hence, as soon as the queue is $!empty()$, the scheduler moves on to the `committed` location. From here, it immediately transitions to the *EnableAcc* location, and simultaneously sends the `run[highPR()]!` synchronization to the superblocks. The $highPR()$ function returns the ID of the highest priority task in the scheduler queue (which is also assigned to the $granted$ variable):

```
p_id highPR()
{
    p_id i=0;
    while (i<Procs)
    {
        if (queue[i]==1)
            return i+a;
```

```
        i++;
    }
    return 0;
}
```

However, `run` is a broadcast channel and hence, in this case as there is only one task in the queue, this send-action by the scheduler is ineffective. This is because the task, being the only one seeking access to the core, has not encountered a `stop` signal from the scheduler.

It is assumed here that the highest priority task is the one with the lowest ID. Hence a binary array monitors which task is waiting for execution on the core. Now, say a higher priority task finishes the acquisition phase and enters the *Exec* location to begin the execution phase. The scheduler at this point is in the *EnableAcc* location. The task is added to the scheduler queue by traversing to the *QueueAdd* location. The scheduler then signals both processes to stop execution through the `stop[cpu_id]!` synchronization send-action. The high priority process is effectively stopped before it can begin execution. Both the tasks are now in their respective *Blocked* locations. In this location, $x_exec' == 0$ invariant is set for both tasks. This stops the x_exec clock from recording the execution time, thus enabling the stopwatch feature on the x_exec clock.

However, *without any delay* (due to the committed *StopAcc* location), the transition back to the *EnableAcc* is taken, and through the `run[highPR()]!` synchronization send-action, the high priority task begins execution. The high priority task re-enters the *Exec* location by receiving the `run[id]?` synchronization, and begins execution. The x_exec clock at this location resumes normal functionality of recording the execution time.

When the high priority process finishes the execution phase, it sends a `done[id]!` synchronization to the scheduler, which traverses back to the *WaitReq* location. Since the queue is non-empty, it immediately signals the lower priority task to resume execution. The rest of the process (acquisition and replication phases) of all tasks are the same as those described for the 1 task/core model.

Hence, with these two superblock models, any real-time system, complying with the requirements stated at the beginning of the section, can be modeled and verified.

2.2.3 Modelling the Arbiters

The three arbitration policies used for experiments in this thesis, as mentioned previously, are FCFS, FP and TDMA. This section illustrates a few examples of implementing these arbitration policies in UPPAAL.

- FCFS with Fixed Access Latency: Figure 2.4 provides an example of implementing the FCFS arbitration policy with a fixed access latency of C time units.

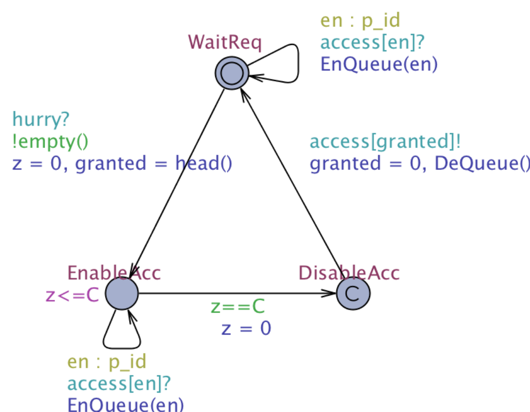

Figure 2.4: FCFS with Fixed Access Latency

For this purpose, a new line of code is added to the *Declarations* template in the UPPAAL model of the RTS:

```
const time_t C=3;  //or any other constant value
```

The arbiter starts in the *WaitReq* location, and similar to the FP scheduler, accepts an access request in this location. The task requests access to the resource, as mentioned in the previous subsection, using the `access[id]!` synchronization channel. The arbiter accepts this request using the `access[e]?` synchronization, where e is assigned the value of the ID of the task requesting access to the resource.

Next, the task is added to the arbiter queue, and like the FP scheduler, the arbiter uses the urgent channel `hurry?` to transition to the *EnableAcc* location as soon as there is a task available in its queue. The $granted$ variable is assigned the task at the head of the queue. Additionally, a clock z is initialized to 0.

At the *EnableAcc* location, the task accesses the resource for a total of C time units. Simultaneously, the arbiter tracks any other requests that other tasks may make while the current task accesses the resource. In that case, the arbiter enqueues the new tasks. When $z == C$, the resource access finishes, and the arbiter transitions to the *DisableAcc* location. Immediately, it transitions back to the *WaitReq* location, and signals the task that the resource access has completed through the `access[granted]!` synchronization send-action. The task in-turn transitions form the *Acq_wait* or *Rep_wait* location back to the *Acq* or *Rep* location, respectively. The arbiter dequeues that task from it's queue, and restarts the procedure with the new task at the head of the queue.

- FCFS with Varying Access Time: In certain implementations, FCFS needs to be implemented such that a given task reads or writes data to the shared resource for an arbitrary amount of time, at once. Such an implementation cannot use a Fixed Access Latency model since this might be longer or shorter than the exact amount of time required to access the resource for the given task. Figure 2.5 shows an FCFS arbiter with Varying Access Time.

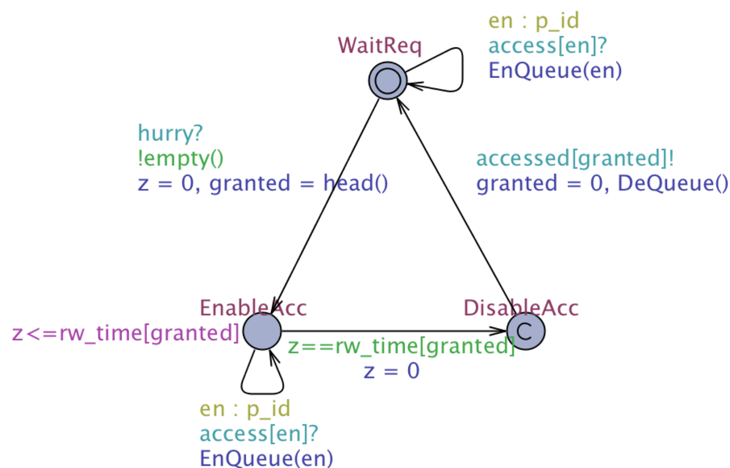

Figure 2.5: FCFS with Variable Access Time

In this case, a `rw_time` array is defined instead of the latency constant C, in the following manner:

```
time_t rw_time[Procs];
```

Each task is assigned one array element, according to task ID. When the task requests access to the resource, the corresponding element in the array is first assigned the read or write time. This is done using a *Select* statement in the superblock. Read or write time is non-deterministically selected by UPPAAL within the constraints of the *Select* statement. The arbiter in turn checks its clock z against the value of this array element. The task, thus, reads or writes continuously for a given time period when it gains access to the resource.

- FP with Fixed Access Latency: Figure 2.6 illustrates an implementation of the FP arbitration policy with a fixed access latency of C time units.

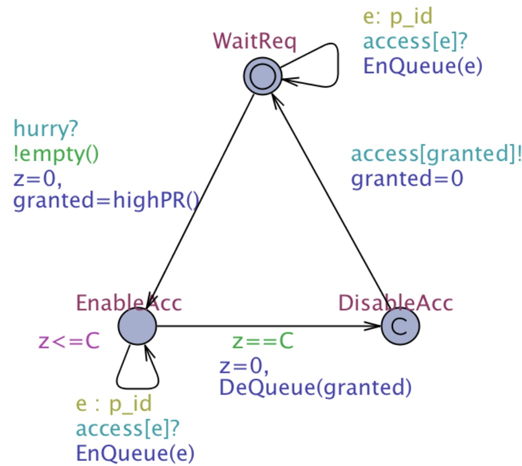

Figure 2.6: FP with Fixed Access Latency

The FP arbiter is very similar to the FCFS arbiter, except the resource access is granted to the task in the queue with the highest priority, rather than the one at the head of the queue. Additionally, instead of a queue, a binary array is used to check which task has requested access to the resource. Again, it is assumed that the highest priority task is the one with the lowest task ID.

- TDMA with Fixed Access Latency: Figure 2.6 shows an example implementation of the TDMA arbitration policy with a fixed access latency of C time units. The TDMA policy is implemented such that each task is assigned a fixed slot. The task can access the resource only during its assigned slot. Consequently, each task can be modelled separately, and this reduces the model checking time to a great extent.

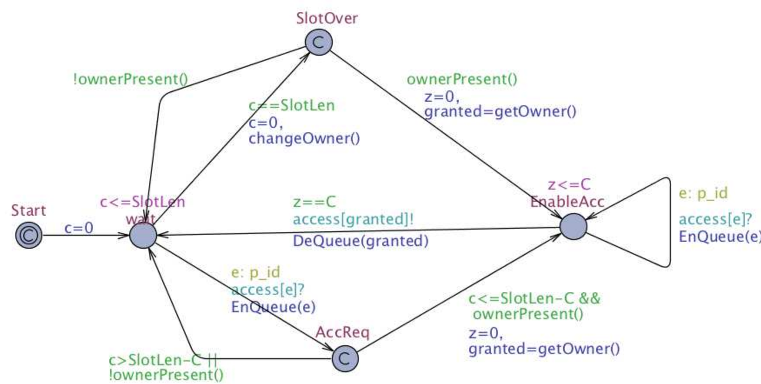

Figure 2.7: TDMA with Fixed Access Latency

Three new functions are introduced here as part of the arbiter:

```
// Assign slot to next process
void changeOwner()
{
    if(slotOwner<Procs-1)
        slotOwner++;
    else
        slotOwner=0;
}

//Return process that owns the current slot
p_id getOwner() {
    return slotOwner;
}

//Check if process to which the current slot belongs is present in the queue
bool ownerPresent(){

    return queue[slotOwner] == 1;
}
```

Additionally, along with the latency constant C, a new constant, $SlotLen$ is added to the *Declarations*:

```
const time_t SlotLen=6; //or some other constant value
```

The $SlotLen$ constant determines the length of a slot in the arbiter. The arbiter starts in the *Start* location, and immediately transitions to the *wait* location, while setting the clock c to 0. It waits at the location till an access request is made. In that case, the transition to *AccReq* is taken. Depending on whether the current task that requested the resource owns the present slot, and whether there is enough time remaining for at least one resource access, the next transition is taken to the *EnableAcc* location or back to the *wait* location. In case of the latter, the task waits till its next slot.

If a slot begins and finishes in the *wait* location, a transition is made to the *SlotOver* location, to check if the task to which the next slot belongs, is available. If so, the task is granted resource access through the *EnableAcc* location. The access to the resource is in the form of discrete accesses with a fixed access latency of C. A task can thus access the resource a maximum of $\lfloor SlotLen/C \rfloor$ times once a slot is assigned to it.

Finally, when one access to the resource finishes, the arbiter signals the superblock through the `access[granted]!` synchronization send-action. The resource can then make another resource access request, and it will be granted if the $SlotLen$ can accomodate one more access. Otherwise, the task waits for its next slot.

Through the combined use of the arbiters and the superblocks, an RTS can be modelled in UPPAAL.

2.3 Guarantee Language L_G

Having discussed the Real-Time System and modelling in UPPAAL, this section discusses the guarantee that is generated on the deadline hit and miss patterns of a particular task in the real-time task-set. The aim is to generate the guarantee in the form of a regular language. As we will see later, a regular language admits advantages which will be useful in the model-checking. The section begins by discussing the representation of such a language, and is followed by the definition and properties of the language.

2.4 Representing the Language

A regular language can be represented in various forms. Four of these are shown in Figure 2.8.

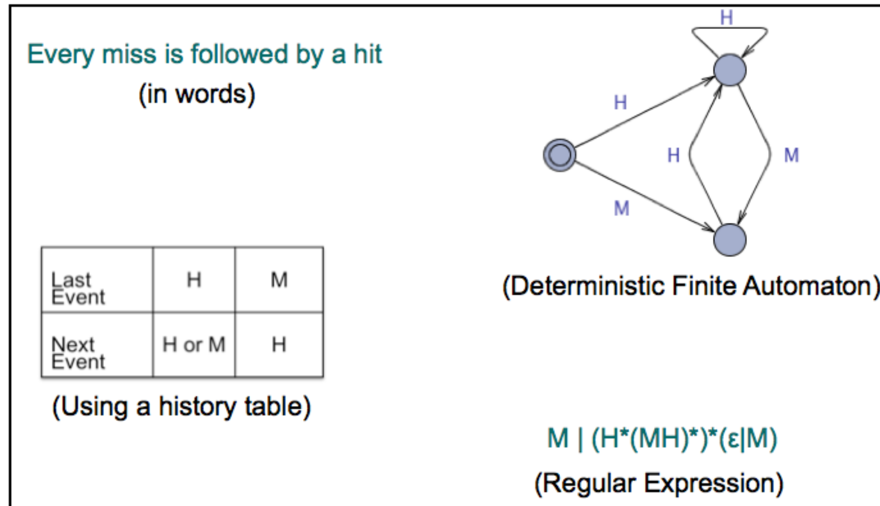

Figure 2.8: Representing a Regular Language

While all of these forms are equivalent, the one used in this thesis is the Deterministic Finite Automaton (DFA) representation of the regular language. The main reason is that it is easier to design a template for the language using a DFA. Additionally, a DFA makes it easier to generate the regular language using UPPAAL, since UPPAAL uses timed-automata to model-check real-time systems.

Additionally, it is important to note that a regular language representation is chosen over an omega-regular language [19]. Even though, in the ideal case, a deadline hit and miss pattern may be infinite in length, it is still pertinent to represent it using a regular language since a finite pattern must also be accepted by the language.

2.4.1 Language Definition and Properties

All observed deadline hit and miss patterns of the jobs of a task τ can be thought of as strings in a language, denoted as L_S (for the system language). L_S is over the alphabet $\Sigma = \{H, M\}$, where *H* implies a *hit* event (i.e., the task meets its deadline), and *M* implies a *miss* event (i.e., a task misses its deadline). An example string is *(MHH)**, which is a periodic pattern with a deadline miss followed by two deadline hits.

The aim is to identify a guarantee that closely, yet conservatively, approximates L_S. To this end, a regular language [3], denoted L_G (for the guarantee language), is computed over the alphabet $\Sigma = \{H, M\}$. We say L_G is a *correct guarantee* if

$$L_S \subseteq L_G. \tag{2.1}$$

If a property is satisfied for all strings (or no string) of a correct guarantee L_G, then it follows that it is (or is not) satisfied on L_S. An example of such a property is: “Every deadline miss is followed by at least two deadline hits". Additionally, the smaller the difference ($L_S - L_G$), the more accurate is L_S.

As shown above, a regular language can be represented by an equivalent DFA. In a DFA, every state encodes a finite observed history. Let k denote the length of the longest history encoded by any state of a DFA, then the DFA is represented as A_k. The parameter k controls the complexity of the corresponding language: a larger k corresponds to a larger automaton and possibly a more accurate guarantee.

For the A_k accepting the guarantee language, denoted $L_G(k)$, the history encoded in the states is the pattern of deadline hits and misses of the last k jobs of task τ. Then, given a hit and miss pattern of the last k jobs, A_k can be used to determine if such a pattern can be observed in L_S,

and if so, whether the next job would meet or miss its deadline. This is a Markov interpretation, where only the last k hits and misses can influence the next job's deadline hit or miss. Similar ideas have been proposed in the analysis of cache hits and branch predictions [12]. The idea of the last k deadline hits and misses influencing the next job's deadline hit or miss is similar to that of k-*testable languages* [20].

Example 2. *For the task-set in Example 1, let task τ_0 have a deadline of $D_0 = 10$. Contention and blocking accesses to shared memory lead to variable response times of jobs of τ_0 resulting in the deadline hit and miss patterns. Using the technique described in the next chapter, the language-based guarantee is derived as shown in Table 2.2. This language corresponds to a DFA A_k with $k = 2$, i.e., given the deadline hit and miss pattern of the last two jobs, the language determines the guarantee for the next job. For instance, if τ_0 encounters a deadline hit followed by a miss, i.e. HM, then the next event can be either a hit or miss. But, for two consecutive misses, i.e. MM, the next event is guaranteed to be a hit.* □

Table 2.2: Guarantee language $L_G(2)$ for Example 1

Last Two Events	HH	HM	MH	MM
Next Event	H or M	H or M	H or M	H

2.5 Summary

This chapter introduced the system model and the definitions that will be used throughout the rest of the thesis. To begin with, a real-time system model was described, which is used to model various task-sets. The task-sets have three phases: *acquisition*, *execution* and *replication*. These are modelled using either Fixed Access Latency (FAL) or Varying Access Time (VAT). The tasks access a shared resource through a resource arbiter. Resource contention and non-preemptive access to the shared resource result in the deadline hit and miss patterns that are expressed through the computed language.

The chapter then discussed the modelling of real-time tasks in UPPAAL using one or mote tasks per core through the superblock model. Additionally, the modelling of arbiters in UPPAAL was discussed, for various arbitration policies.

Finally, an illustration of the Guarantee Language, L_G, was provided. The corresponding section discussed the regular language representation, along with the definition of the language. The next chapter discusses the generation of the guarantee language.

Chapter 3

Computation of the Guarantee Automaton

The language-based guarantee on the deadline hit and miss patterns of a real-time task is generated in the form of a DFA, called the *guarantee automaton*. A guarantee generation toolkit was designed to automate the process of generating the language-based guarantee. This chapter introduces the two procedures implemented using this toolkit: generating A_k (the guarantee automaton) for a given value of k, and generating the guarantee automaton for an unknown k.

As mentioned previously, the general approach of Counterexample Guided Abstraction Refinement (CEGAR) [10] is followed for constructing DFA A_k for a given value of k. The procedure is carried out such that the equivalent regular language $L_G(k)$ conservatively approximates L_S. It involves the use of a specific template for A_k, which can be iteratively modified. At any given point in the process, the current estimate of A_k, denoted as A_k^i for the ith iteration, is modelled by an *observer* timed automaton. Modifications in A_k are mirrored by simultaneous modifications in the observer.

A *property* is then verified to check if the estimated guarantee language, i.e. the $L_G(k)$ equivalent to the current estimate of A_k, conservatively approximates L_S. If a counter-example is generated, the observer is modified, or equivalently A_k^{i+1} is constructed from A_k^i. When no more counter-examples are generated and the iterative procedure terminates, A_k is obtained and the properties of the corresponding $L_G(k)$ are shown.

For applications in which it is unclear what k should be, we need a method to choose the right k. This choice should balance between complexity and accuracy: for a large k, $L_G(k)$ can more accurately represent L_S, but can be computationally expensive to model-check. Hence, one of the solutions to this problem is an iterative method that sequentially computes $L_G(1), L_G(2), \ldots$. The procedure to fully utilize $L_G(k)$ when computing $L_G(k+1)$ is discussed later in this chapter, along with a terminating condition for such a method, and certain scalability issues are discussed as well.

3.1 Constructing A_k for a given k

This section discusses the computation of the language-based guarantee when the value of k is given. We begin with an illustrative explanation of the role of the *observer* automaton.

3.1.1 The Observer Automaton

To begin the procedure, a template for A_k is first defined. The DFA A_k is given by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, $\Sigma = \{H, M\}$ is the alphabet of accepted inputs, where *H* represents a *hit* event, and *M* represents a *miss* event. δ is the transition function

defined as $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$ is the start state, and F is the set of accepting states. The following are some properties of A_k.

- Each state $q \in Q$ corresponds to a particular string of size up to k over the alphabet Σ. Thus, there are up to $2^{k+1} - 1$ states.
- We can represent a state q equivalently by its corresponding string denoted $s(q)$. For example, when the DFA is in a state q with $s(q) = (MHH)$, the previous two jobs have met their deadlines while the one before them missed its deadline. The oldest information is the left-most. For the initial state q_0, we have $s(q_0) = \phi$.
- δ is a partial function, i.e., $\delta(q, x)$ does not have to be defined for every state $q \in Q$ and for both $x \in \Sigma$. If the DFA is in state q and the next input is x where $\delta(q, x)$ is not defined, then the input string is not accepted by A_k, and thus does not belong to $L_G(k)$.
- The transition function follows from the string representation of the states. For example, let q_i, q_j be two states with $s(q_i) = (MHH)$ and $s(q_j) = (HHM)$. If $\delta(q_i, M)$ is defined, then it is equal to q_j, for $k = 3$.
- $F = \{q \in Q : |s(q)| = k\}$.

Thus, the template for A_k defines a class of automata that accept any regular language with strings of length at least k. Any language, which can be represented as shown in Table 2.2, is accepted by a DFA belonging to this class depending on the transitions defined in the DFA.

Example 3. *For the language in Table 2.2, the corresponding A_k with the above template is shown in Fig. 3.7.* □

For the template of A_k, we design a specific observer, denoted as O, as a timed automaton [8] in UPPAAL [9], such that:

- Each state of A_k corresponds to a location of O.
- All possible transitions in A_k are modeled in O. Every location in O has two outgoing transitions for the two cases of hit and miss events. Transitions not defined in A_k have a variable update $e = 0$ (for enable) in O, while the others have a variable update $e = 1$.
- Each transition in O which corresponds to a transition in A_k that accepts an H (similarly M) has a synchronization channel receive-event `hit?` (`miss?`).

Such a structure can be represented in the form of a binary tree. However, since we deal with strings of infinite length, all the locations with a history of length k, which also are the leaves of the binary tree, have loops among them. Consequently, the maximum length of history stored by any location in the O for A_k is k. An example of the initial O for $k = 2$, corresponding to A_2^0, is shown in Figure 3.1. Each location is represented as q_y, with $y = 0$ for the start state and otherwise equal to the history corresponding to the location. The guarantee is generated using the O as discussed in the subsequent subsections.

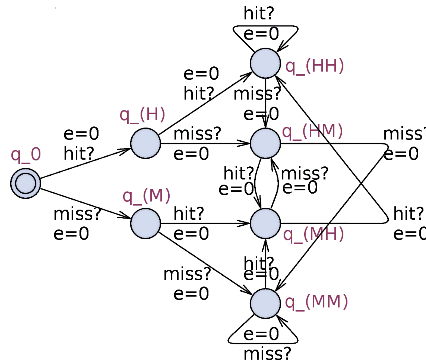

Figure 3.1: Observer for $k = 2$ in UPPAAL.

3.1.2 Modelling the Observer in UPPAAL

Before beginning the iterative construction of A_k, we need to make a few additions to the UPPAAL RTS model described in the previous chapter. So far, we have discussed the model for representing a particular real-time task from the task-set, in the form of a superblock. Additionally, the previous chapter illustrated various arbiters that are used for resource arbitration. We also discussed various declarations and methods used by the superblock and arbiter models, such as `highPR()` to return the highest priority process. Finally, the communication channel between the superblocks and the arbiter was illustrated using synchronizations in UPPAAL.

In order to generate the guarantee automaton, an observer automaton (O, as stated above) is required to model the language. This automaton is introduced into the UPPAAL model in the form of a new UPPAAL template. Instead of using the UPPAAL GUI, the process of generating O is automated using a Python script (the details of which are available in Appendix A.2. UPPAAL stores such models in an XML file. Additionally, every location of a particular timed-automaton has an ID and name (optional) associated with it. For the implementation here, a unique ID, e.g. `id0`, `id1`,..., is assigned to each location. This is particularly helpful while studying the counter-example trace generated during the iterative construction procedure (discussed in the next subsection).

The process of assigning ID's to these locations is automated. As stated above, the structure of the O is similar to that of a binary tree. Hence, every Parent location has two Child locations. Now, the ID's are assigned in the following manner:

$$Child\ ID = \begin{cases} (Parent\ ID) \times 2 + 1 & \text{for a } hit? \text{ edge} \\ (Parent\ ID) \times 2 + 2 & \text{for a } miss? \text{ edge} \end{cases}$$

An example of the O for $k = 2$, as modelled in UPPAAL, is shown in Figure 3.2. One of the changes in this figure, as compared to Figure 3.1, is the fact that the variable e is replaced by e_global. Secondly, the locations are named according to their ID numbers for the purpose of illustrating the guarantee generation procedure. There were two reasons for using e_global instead of e. First, since this variable is introduced automatically into the XML file through the code during implementation, it might be possible that a common variable like e is already present in the XML. Hence, to avoid syntactic errors in UPPAAL by introducing another variable named e, we can use the less likely e_global as a variable name. It would, however, be wise to check beforehand if this variable is present in the XML. Second, e_global signifies that this is a global variable, even though it may be updated locally in the observer.

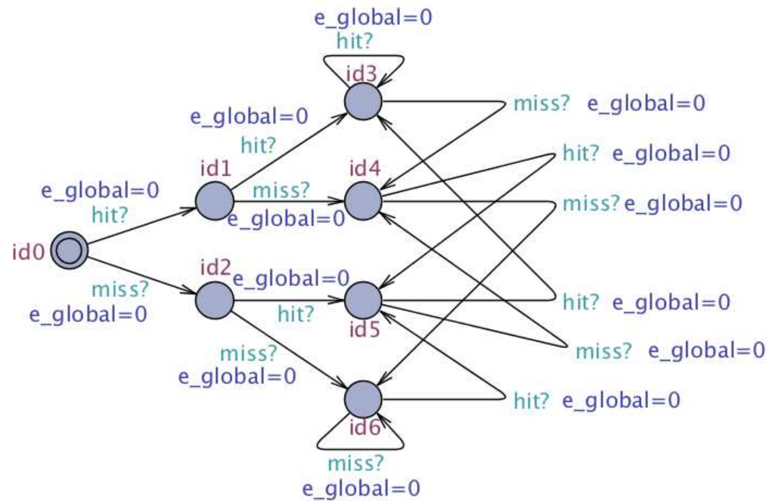

Figure 3.2: Observer for $k = 2$ in UPPAAL.

In order to synchronize O with the superblock model, synchronization channel send-events (`hit!` and `miss!`) are added to the Superblock corresponding to the observed task. These

will thus correspond to the receive-events (`hit?` and `miss?`) incorporated in O. The *Declarations* template in UPPAAL is modified to include the following code:

```
typedef int[0,1] e_poss;
e_poss e_global = 1;
```

An example of a Superblock with the send-events is shown in Figure 3.3, where *threshold* corresponds to the deadline. The threshold value is supplied to the superblock as a *Parameter* value through the *System declarations* template. The superblock instance corresponding to the observed task is modified as follows:

```
S2 = Superblock(1, 72);  //task ID=1; threshold=72
```

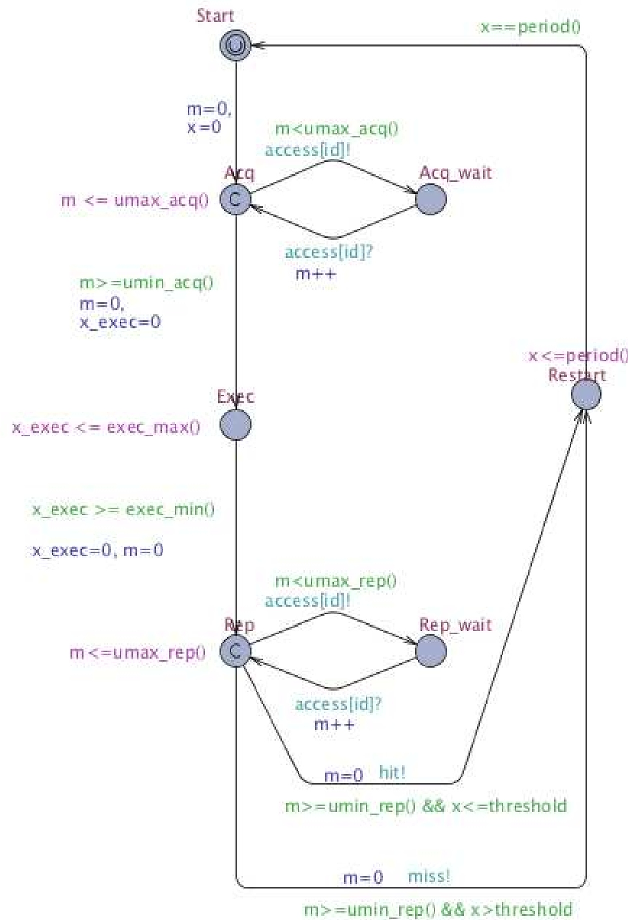

Figure 3.3: Superblock with `hit!` and `miss!` events.

The network of TA, S, thus synchronizes with O using synchronization channel send-events `hit!` and `miss!` whenever a job of the observed task τ either meets or misses its deadline, respectively. Thus, for a particular execution trace, as the jobs of τ meet or miss deadlines, S and O synchronize over corresponding channels. We have now modelled all the necessary parts in UPPAAL required for the iterative construction of the guarantee.

3.1.3 Iterative Construction

We initialize O such that the variable update rule on each transition is $e = 0$ (e and e_global are used interchangeably in the rest of the illustration). This corresponds to A_k^0 which has no defined transitions. Additionally, e is initialized to 1 at the global level. This marks the starting

point for the iterative construction procedure.

Then, (for the extended network of TA,) the following (TCTL) safety property is verified:

$$\forall \Box e = 1 \text{ (or equivalently, } A[]e == 1 \text{ in UPPAAL).} \tag{3.1}$$

This property asserts that the variable e equals 1 in all states. In terms of our usage of e, the property asserts that at all times the transitions taken by O are already defined in A_k^i. In other words, the property asserts that the current estimate of A_k, i.e. A_k^i, most closely approximates to A_k.

In order to verify the property, a `.q` (query) file is generated. The following query, with reference to (3.1) and the variable e replaced by e_global, is then placed in the file.

$$A[]e_global == 1 \tag{3.2}$$

Finally, UPPAAL's command-line tool, `verifyta`, is executed using the UPPAAL XML and query files as arguments, along with other options. `verifyta` provides some freedom to the user to configure the state space exploration procedure. After experimenting with several combinations of these options, the best one in terms of reducing the computational effort of model checking is the breadth-first search option with the 'some' trace option. This implies a BFS of the state space along with returning a random (first encountered) trace every time a counterexample is encountered. More details of the implementation are discussed in Appendix A.3. `verifyta` model-checks the RTS models provided in the UPPAAL XML file against the query. The aim is to verify if every possible execution of the model satisfies the query. If not, then there is a particular transition traversed by O that set the global variable $e_global = 0$. Consequently, `verifyta` generates a counter-example, and provides an output in the form of a trace. The last two `Transitions` of an example trace are provided in Figure 3.4.

```
State:
( S1.Restart S2._id12 Arb.DisableAcc H._id0 observer_Observer2._id0 )
S1.x=25 S1.x_exec=13 S2.x=25 S2.x_exec=11 Arb.z=0 e_global=1 S1.m=0 S2.threshold
=15 S2.m=6 Arb.arb_len=0 Arb.queue[0]=0 Arb.queue[1]=0 Arb.granted=1

Transitions:
  Arb.DisableAcc->Arb.WaitReq { 1, access[granted]!, granted := 0 }
  S2._id12->S2.Rep { 1, access[id]?, m++ }

State:
( S1.Restart S2.Rep Arb.WaitReq H._id0 observer_Observer2._id0 )
S1.x=25 S1.x_exec=13 S2.x=25 S2.x_exec=11 Arb.z=0 e_global=1 S1.m=0 S2.threshold
=15 S2.m=7 Arb.arb_len=0 Arb.queue[0]=0 Arb.queue[1]=0 Arb.granted=0

Transitions:
  S2.Rep->S2.Restart { m >= umin_rep() && x > threshold, miss!, m := 0 }
  observer_Observer2._id0->observer_Observer2._id2 { 1, miss?, e_global := 0 }

State:
( S1.Restart S2.Restart Arb.WaitReq H._id0 observer_Observer2._id2 )
S1.x=25 S1.x_exec=13 S2.x=25 S2.x_exec=11 Arb.z=0 e_global=0 S1.m=0 S2.threshold
=15 S2.m=0 Arb.arb_len=0 Arb.queue[0]=0 Arb.queue[1]=0 Arb.granted=0
```

Figure 3.4: Counter-example trace generated by `verifyta`.

The highlighted portion of the last `Transition` of the trace, in the figure, provides the ID's corresponding to the locations where the counter-example was generated. The XML containing the UPPAAL model is then modified to reflect this change, such that $e_global = 1$ is set on that transition as shown (highlighted) in Figure 3.5. Equivalently, a new transition is defined in (or added to) A_k^i to generate A_k^{i+1}.

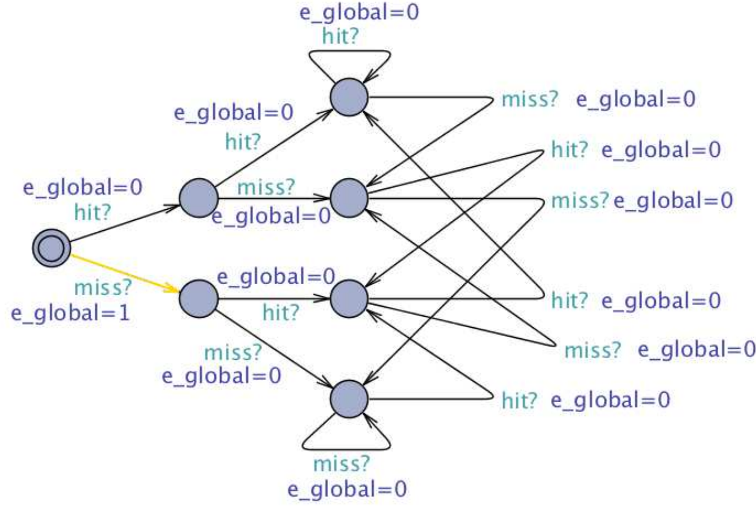

Figure 3.5: Modified Observer after first counter-example.

The process (of running verifyta, obtaining the trace, and modifying the XML) is then repeated till no more counter-examples are generated. This signals the end of the procedure, and the modified O now acts as the Guarantee, A_k, for the given value of k. All the edges with the variable update $e_global = 0$ at the end of the procedure are removed. Consequently, there are various locations in the final O that are unreachable, and are removed as well.

The entire guarantee generation procedure is automated using Python scripts. A detailed description of different functions used in these scripts is illustrated in Appendix A.4.

3.1.4 Properties of the Computed $L_G(k)$

The iterative process will terminate after a finite number of steps, as in each step a new transition is defined amongst the finite number of possible transitions of A_k. Upon termination, DFA A_k and the corresponding $L_G(k)$ satisfy the following:

Lemma 1 (Correctness)**.** *The language $L_G(k)$ corresponding to the DFA A_k satisfies $L_S \subseteq L_G(k)$.*

Lemma 2 (Tightness)**.** *If A'_k is obtained by disabling any single transition from A_k, then the corresponding language $L'_G(k)$ does* not *satisfy $L_S \subseteq L'_G(k)$.*

The correctness property follows from the fact that asserting property (3.1) asserts $L_S \subseteq L_G(k)$. The tightness follows from the initialization wherein no transition of A^0_k is defined. Thus, any defined transition in A_k is due to an observed pattern of hits and misses in S.

Example 4. *Consider the setup from Example 1. For $k = 2$, the observer O, as mentioned above, is illustrated in Fig. 3.6. The guarantee automaton, shown in Fig. 3.7, is then generated using the guarantee generation procedure. The transition M self-loop on $q_(MM)$ is undefined at the end of the procedure and is thus removed. The computed $L_G(k)$ is shown in Table 2.2.* □

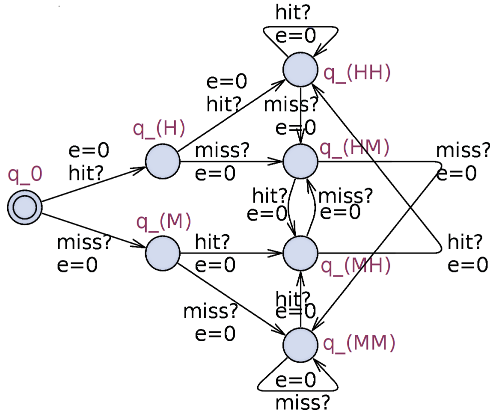

Figure 3.6: Observer O for $k = 2$.

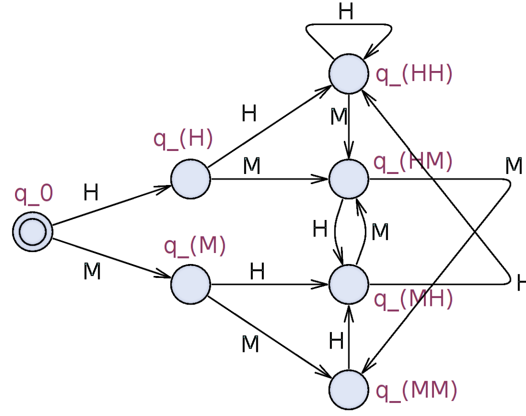

Figure 3.7: Guarantee Automaton A_2.

3.1.5 Minimizing the Guarantee Automaton

For certain applications, it is useful to convert a DFA into its minimized version. As we will see in Section 4.1, this property is useful in checking language inclusion. The DFA minimization is carried out through the use of equivalence classes, based on the Myhill-Nerode theorem [18].

For the applications considered in this thesis, it is assumed that the strings belonging to $L_G(k)$ are of infinite length. Consequently, the focus is mainly on the final states of A_k representing the corresponding $L_G(k)$. As a result, only the leaves of the corresponding final observer O are used during the minimization. Thus, every state in the minimized automaton represents a final state with a history of length k. Given the last k events, the minimized automaton can thus predict whether the next even will be a hit or a miss.

A rough outline of the algorithm is provided below (assuming that there are no unreachable locations in O, as these have all been removed):

1. Divide the leaf locations into two sets: final and non-final locations. In our case, each of these is a final location. However, we include a hypothetical non-reachable location in the non-final set to which every disabled edge from the locations in the final set points.
2. For each outgoing edge of every location in the final set, check if the target location is in the same set.
3. If 2 is true, then the procedure is over and the automaton is minimized to a single final location.
4. Otherwise, create a new set. In this new set, place every location with an outgoing edge that points to a location in another set.
5. Repeat step 4 for all the sets, including the new set, till condition 2 is satisfied.
6. Once 2 is satisfied, each set forms a new location. All the locations in that set are equivalent, and thus the set is called an equivalence class. Now, take a particular location from each set and, for each outgoing edge, determine the equivalence class (set) to which its target location belongs. Check each location of a set till all the edges (in our case the hit and miss edges) are verified. The sets to which the target locations belong will now act as new target locations for the corresponding edges from this set.

An example is shown in Figures 3.8 and 3.9. Figure 3.8 is the final modified observer corresponding to an example A_3. Here, the variable updates $e_global = 1$ are removed from the edges for a clearer illustration. Additionally, all the edges with variable updates $e_global = 0$ are removed, and the corresponding non-reachable locations are removed as well before beginning the minimization procedure. The locations corresponding to the final states of A_3 are marked as fx where x is an index number. Figure 3.9 shows the corresponding minimized guarantee

automaton for the final states of A_3. Each location is named according to the location set that it represents.

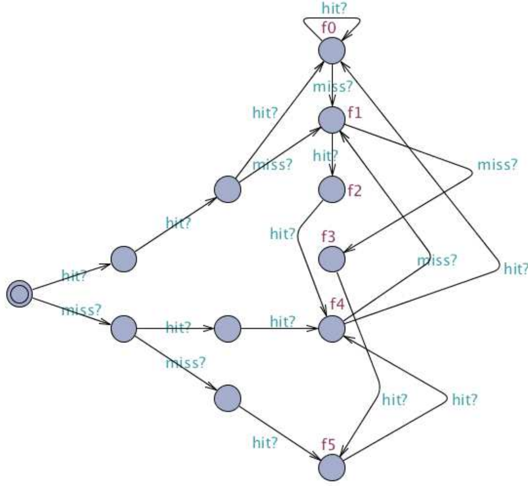

Figure 3.8: The Guarantee for $k = 3$.

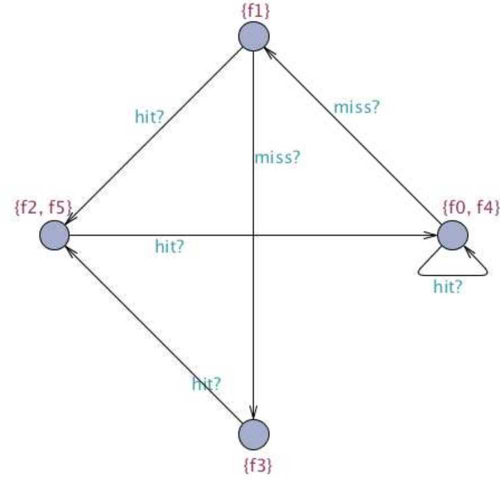

Figure 3.9: Corresponding Minimized Guarantee.

The application of the minimized version of the guarantee automaton will be discussed in the next chapter.

3.2 Constructing the Guarantee Automaton For Unknown k

Following from the introduction to this chapter, for applications in which it is unclear what k should be, we need a method to choose the right k. This section presents an iterative method that sequentially computes $L_G(1), L_G(2), \ldots$. The following lemma forms the basis of this approach. It states that the computed language for a smaller k cannot be a tighter approximation of L_S.

Lemma 3 (Inclusion). *$L_G(k+1) \subseteq L_G(k)$, $\forall k > 0$.*

The argument for the above result is presented in terms of A_k and A_{k+1}. For every state q in A_{k+1} a *parent state* $p(q)$ is defined in A_k. Two cases arise based on the string representation $s(q)$. If $|s(q)| < k+1$, then $s(p(q)) = s(q)$. If $|s(q)| = k+1$, then $s(p(q))$ is obtained by dropping the left-most (oldest) element in $s(q)$. This definition is such that, when processing a common input, if A_{k+1} is in state q, then A_k has to be in state $p(q)$. From the tightness of A_{k+1} and the correctness of A_k, we have: if either or both transition(s) (out of H and M) are disabled in A_k for state $p(q)$, then the corresponding transitions are also disabled in A_{k+1} for state q. Thus, we cannot have a string that is accepted by A_{k+1} and not by A_k.

3.2.1 Refining $L_G(k)$ by Increasing k

An iterative refinement approach is proposed for increasing k till a satisfactory $L_G(k)$ is obtained. We start with the initial observer O for $k = 1$, which is then modified using the procedure in the previous section to derive A_1. Then, we begin the process for A_2. However, we can now use information from A_1 to modify the initial O corresponding to A_2^0 before iterative verification with the model-checker. We know that for every state q in A_2^0, if the parent state $p(q)$ in A_1 has either or both transition(s) disabled, then the corresponding transitions will also be disabled in A_2. Using this, the initial O, and hence A_2^0, is modified by removing certain transitions and consequently unreachable locations. The modified observer is now used to obtain A_2. This process of constructing A_{k+1} by using information from A_k is termed *refinement*.

Example 5. *Consider guarantee automaton A_2 shown in Fig. 3.7. Using this the observer for $k = 3$ is initialized as shown in Fig. 3.10. Note that certain transitions are removed from the observer. For instance, the location q with $s(q) = (HMM)$ does not have a $\delta(q, M)$ transition. Also, location q with $s(q) = (MMM)$ is not reachable at all and is thus removed. With this observer and the procedure of the previous section, the guarantee automaton A_3 shown in Fig. 3.11 is computed.* □

This process is complete in the sense that any counter-example generated when computing A_k will not be generated when computing A_{k+1}. This clearly reduces the model-checking time.

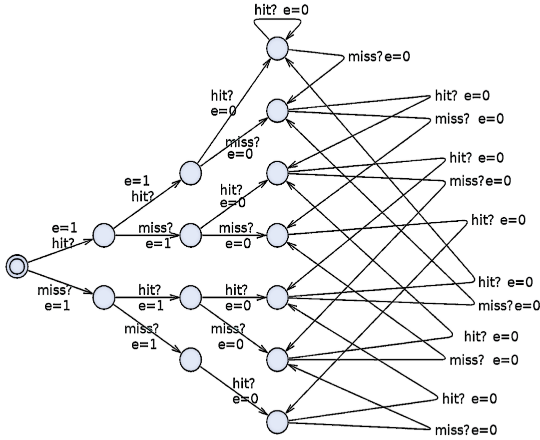

Figure 3.10: Initial Observer for $k = 3$ from A_2.

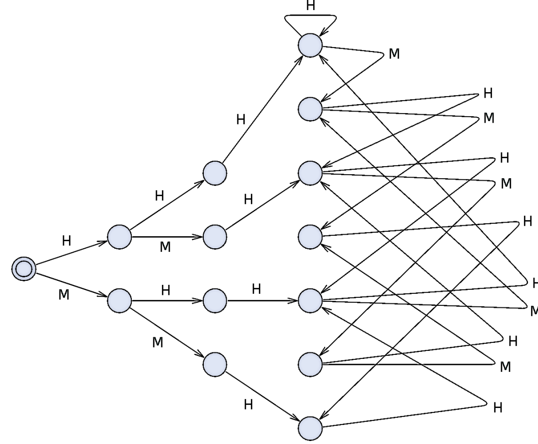

Figure 3.11: Guarantee Automaton A_3.

3.2.2 Implementing Iterative Refinement in UPPAAL

This subsection provides a step-wise guide to implement the guarantee generation procedure for an unknown k. The iterative refinement procedure starts with the observer O for $k = 1$ as shown in Figure 3.12. Location ID's are used as names for explanatory purposes. After the iterative construction procedure, an example A_1 is shown in Figure 3.13. For the purposes of illustrating the procedure, the guarantee automaton is considered interchangeable with the final O for the given k. Hence, `hit?` and H, as well as, `miss?` and M are equivalent. Additionally, ID numbers are shown in the guarantee automaton along with their e_global values.

The initial Observer for $k + 1$ is constructed from A_k in a step-wise manner.

1. Any edge that is missing in A_k, will not be added to the O for $k + 1$. For instance, the M self-loop on `id2` in A_1 was never traversed and hence e_global was never set to 1. It is thus removed from the final automaton. Consequently, when deriving O for $k = 2$ from A_1, the M edge from `id2` (Parent) to `id6` (Child) will be removed.

2. As a result of the missing M self-loop in `id2` in A_1, any string containing MM is not accepted by A_1. This implies that such strings will not be accepted by A_2 as well, following from (2). Hence, A_2 will not accept strings HMM and MMM, and any other strings that contain these strings. Consequently, the corresponding edges should not be present in O for $k = 2$. These two edges correspond to the `miss?` edges from `id4` and `id6`.

3. Any location in the new O with no incoming edges, is removed. For instance, there are no incoming edges on location `id6`, and hence the location is removed as shown in Figure 3.14.

4. The outgoing edges from all locations with a history of length $k - 1$, in the O for k, are updated with $e_global = 0$. To explain this, let's take the example of A_1 and O for $k = 2$. In A_1, these edges formed loops in the leaves of the automaton. Consequently, it is unclear whether a particular edge was traversed on the second H or M event, or any subsequent event after it. Hence, in the O for $k = 2$, these edges are checked again to verify if they are traversed during the course of the model-checking process. In A_2, each of these edges

would exist only if the corresponding event (H or M) occurs during the second job of any execution of the task under consideration.

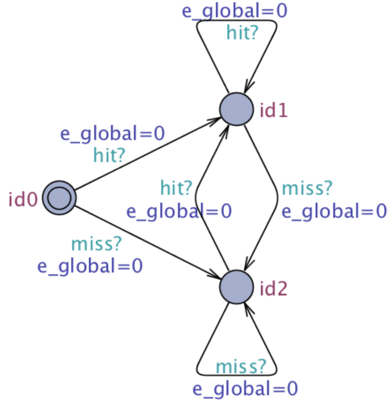

Figure 3.12: Initial Observer for $k = 1$.

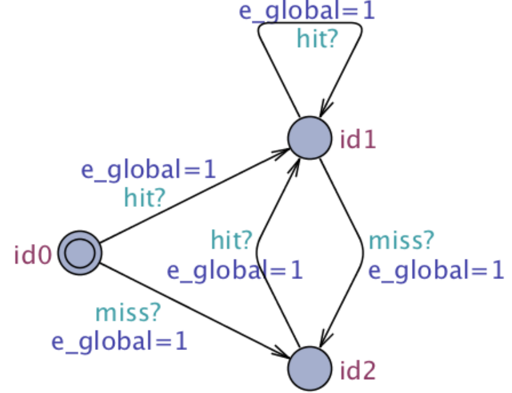

Figure 3.13: Guarantee Automaton A_1.

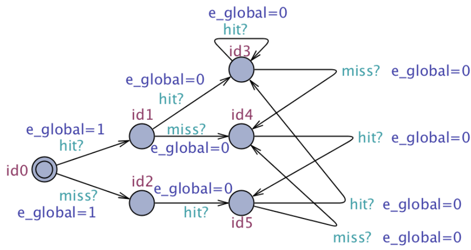

Figure 3.14: Modified Observer for $k = 2$.

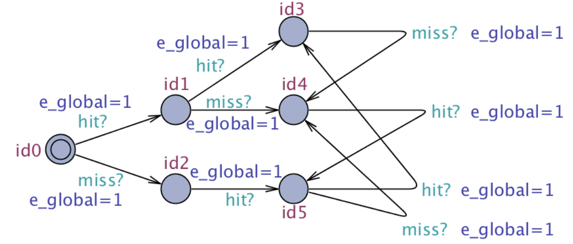

Figure 3.15: Guarantee Automaton A_2.

During the actual implementation in UPPAAL, all the potential locations are added to the new O. Now, before adding edges to the non-leaf locations of the new O (for $k+1$), they are cross-checked with the guarantee automaton (for k). If they are absent from the automaton, they are not added to O. The implementation is done using ID numbers, since the ID's of all the non-leaf locations in O for $k+1$ are the same as that in the final O corresponding to A_k.

For the next step, two ID numbers are taken as reference. The example of generating the initial O for $k = 2$ (Figure 3.14) from A_1 (Figure 3.13) is taken to illustrate this step. The first reference R_1 is the first ID (the top-most for a left-right tree structure, that corresponds to all hits) in the leaves of A_k (Figure 3.13). This corresponds to `id1`. The second reference R_2 is the first ID from the leaves of the O for $k = 2$ (Figure 3.14). Hence R_2 corresponds to `id3`.

Now, a `hit?` or H corresponds to 0, while a `miss?` or M corresponds to 1. Hence, `id3` corresponds to a history of HH or 00. The next location is HM or 01. This binary sequence (or its integer equivalent) is added to 3 which gives the next ID as `id4`. Hence, `id4` corresponds to HM if calculated the other way around.

Next, using the two reference numbers, certain edges are removed. For instance, when checking if a `miss?` edge should be present on `id4`, the corresponding location in A_1, i.e. the location with a history of M, is checked. By using R_1, it is easy to see that this location is `id2`. `id2` has the `miss?` edge missing, and hence it should not be present in `id4` either.

Another example is that of `id8` in the O for $k = 3$ ((Figure 3.16)). This locations corresponds to a history of HHM. The reference ID's in this case would be `id3` (R_1) and `id7` (R_2). The corresponding location with a history of length $k-1$ is the location with history HM. Using `id3`,

the ID of this location is derived to be `id4`. Since `id4` has a `miss?` edge missing, this edge is removed from `id8` as well. Hence, by going through each location, the outgoing edges that should be present in each location can be determined.

Finally, all the locations with no incoming edges (unreachable locations) are removed, and e_global is re-assigned the value of 0 as mentioned previously. This is also done using the ID number on the O for $k+1$ that correspond to the leaf locations of A_k. The O for $k = 3$ generated from an example A_2 (Figure 3.15) is shown in Figure 3.16.

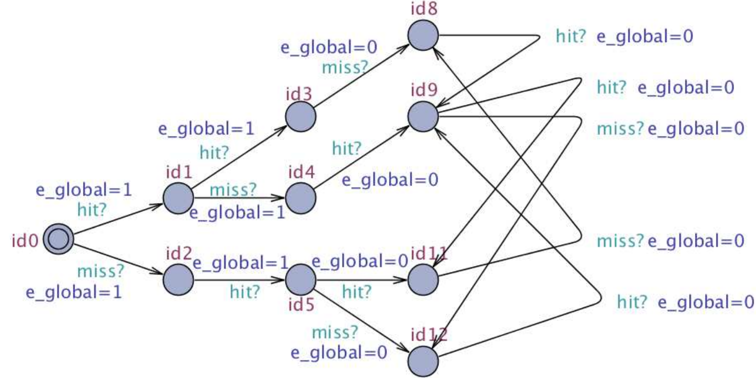

Figure 3.16: Observer for $k = 3$ in UPPAAL derived from A_2.

Once the new O is computed for $k+1$ from A_k, the iterative construction procedure described in the previous section is carried out on the new O to obtain A_{k+1}. This process of increasing k is continued till the pre-determined stopping condition, discussed in the next subsection, is met. The entire code is run automatically using Python scripts discussed in Appendix A.4.

3.2.3 Terminating Condition

Several indices were considered to act as a stopping condition for the iterative refinement procedure to find the right k. There are described in the table below:

Name	**Description**
Total Degree Index	Take every leaf state of A_k that has an out-degree equal to 2. For every such state, take the two states from A_{k+1} that have the same history for the k event before the last one. Hence, these states will have their left-most k events the same as the leaf state from A_k. Take the average of the out-degrees of the two states. If one or both of the two states are missing, then the out-degree for each missing state is 0. Now, subtract this from 2 to obtain the degree difference between the A_k state, and the average for the A_{k+1} states. Take the sum of all such differences for all the states considered from A_k. Finally, divide the sum by the maximum number of possible leaf states in A_k, which equals 2^k.
Partial Degree Index	Same as the Total Degree Index, except, in the last step, divide the sum by the number leaf states of A_k with an out-degree of 2.
Uncertainty Metric	Take the outgoing edges from all states of A_k, divide it by the total possible number of out-going edges from A_k, and subtract the result from 1.
Partial Uncertainty Metric	Same as the Uncertainty Metric, except, only out-going edges (present and possible) from the leaf states are considered.

Table 3.1: Potential Indices for Stopping Condition.

Table 3.2: Uncertainty metric for Example 1

k	1	2	3	4	5	6	7	8
$U(k)$	1.000	0.929	0.733	0.500	0.317	0.197	0.120	0.072
Iterative	0.949	2.219	4.335	6.798	9.695	13.901	19.401	27.635
Non-iterative	0.948	2.276	4.750	7.800	11.824	18.438	29.786	53.339

The ideal scenario in providing a guarantee for a system is that there is minimal uncertainty in the system. This way, a more accurate guarantee can be provided. Of all the four indices, the Uncertainty Metric provides the best insight into the amount of uncertainty in the system. The main reason is the fact that it considers all states of A_k, and hence all possible string sequences. Hence, the *Uncertainty Metric*, denoted $U(k)$, is chosen to define the terminating condition for the iterative process of increasing k.

$$U(k) = \frac{e_{pr}}{e_{po}}, \tag{3.3}$$

where e_{pr} is the number of transitions present in A_k, and $e_{po} = 2^{k+2}$, the total number of transitions possible in A_k. The metric, as mentioned above, conveys the level of uncertainty in A_k with respect to the predictions at each state, and is similar to the statistical metric of entropy.

$U(k)$ varies between 0 and 1. $U(k) = 1$ implies that every state in A_k has two outgoing transitions. Thus, irrespective of the previous sequence of hits and misses, the next event can either be a hit or miss. A lower value of $U(k)$ implies that there is lesser uncertainty, because there are states with either one or no outgoing transitions.

From Lemma 3, it is clear that A_{k+1} will have a maximum of twice the number of transitions as A_k. e_{po} increases to twice the value for each increase in k. Hence, the ratio either remains constant or decreases with an increase in k, i.e.,

Lemma 4. $U(k+1) \leq U(k)$.

Using the above property, we can decide to terminate the iterative process if $U(k)$ reaches a value close to 0. However, for some settings, uncertainty may be inevitable. For instance, if every job of τ can either hit or miss its deadline, then $U(k) = 1$ irrespective of k. Thus, in addition to the absolute terminating condition, we also have a relative condition, such as $U(k+1)/U(k)$ is a value close to 1.

Example 6. *For the running example, the uncertainty metric for different values of k is shown in Table 3.2. Row 3 shows time (in s) required to compute each A_k using the iterative refinement approach (Subsection 3.2.2). Row 4 shows time (in s) required if each A_k is computed non-iteratively, i.e., without using the iterative refinement procedure. These readings are computed on an Intel® Core™2 Quad CPU Q6600 @ 2.40GHz $\times 4$ machine with 4GiB RAM. With increase in k, the iterative approach shows a significant performance gain over the non-iterative approach.* □

3.3 Summary

The focus of this chapter lay in the illustration of the Guarantee generation procedure. The chapter was divided into two main sections: generating the Guarantee automaton A_k for a known k, and Guarantee generation for an unknown k.

The first section introduced the observer automaton O which is an integral part of the Guarantee generation procedure. The observer is undergoes iterative construction to generate A_k. The section also described the modelling of the observer in UPPAAL, followed by a detailed account of the iterative construction procedure. The guarantee generation process for a given k uses

the observer modelled for that k and verifies a query using the `verifyta` command (UPPAAL's command-line substitute). The section also presented certain properties of the computed $L_G(k)$ such as *Correctness* and *Tightness*. Finally, an algorithm to minimize the Guarantee automaton was illustrated using an example.

The second section described the Guarantee generation procedure for an unknown k. The procedure followed in this case is referred to as *refinement*. The main idea is to begin with an automaton for $k = 1$, construct the guarantee using the iterative construction procedure, and then use A_1 to generate the observer for $k = 2$. The process then continues for higher values of k. The section also explained the implementation of the refinement procedure in UPPAAL in great detail. Finally, a terminating condition was introduced to determine when to stop the refinement procedure.

Having successfully generated the guarantee, the next chapter discusses two different applications of such the guarantee language.

Chapter 4

Applications

This chapter illustrates two important applications of the language-based guarantee. First, a language inclusion test is presented, which is used to verify a controller's performance requirement. The second application compares scheduling algorithms for mixed-criticality systems by computing worst-case deadline miss rates for low-criticality tasks.

4.1 Language Inclusion

In model-based design, a standard step is to check that all assumptions are satisfied by the guarantee. Let L_A denote the language of hit and miss patterns which meet given performance constraints. Then, the real-time system S satisfies this assumption if $L_S \subseteq L_A$. It is sufficient to show that $L_G \subseteq L_A$, under the correctness property (Lemma 1).

Language inclusion for the design of a networked control system [1] is illustrated with the following example. Consider an inverted pendulum on a moving cart. Let the mass of the cart be 0.5kg, the mass, inertia and length of the pendulum be 0.2kg, 0.006 kg.m^2 and 0.3m, respectively. Let the coefficient of friction of the cart be 0.1N/m/s. The state of this control plant is given by $z = [x\ \dot{x}\ \theta\ \dot{\theta}]$, where x is the displacement of the cart, θ the angular displacement of the pendulum, and derivatives are denoted with a dot on top. With period $P = 0.6$s, the state is sensed and a feedback control is computed to change $\dot{x}$ and $\dot{\theta}$.

Assuming a sensor-to-actuator delay of exactly one period, a Linear Quadratic Regulator (LQR) [14] is now designed. Whenever the delay is larger than a period, the controller output is not actuated and the plant is in an open-loop configuration. Thus, we have a switched linear time-invariant system. As shown in [3], the plant and controller state can be collated at the nth sampling time by $X[n]$ with the following dynamics

$$
\begin{aligned}
X[n+1] &= A_{cl} X[n], \quad \text{if } d[n] \leq P, \\
X[n+1] &= A_{ol} X[n], \quad \text{otherwise},
\end{aligned}
$$

where A_{cl} and A_{ol} are the system matrices for closed-loop and open-loop, respectively, $d[n]$ is the sensor-to-actuator delay for the nth period, and P is the sampling period. For the computed LQR controller, the obtained A_{cl} and A_{ol} are shown below. We have $\|A_{cl}\| = 0.82$, and $\|A_{ol}\| = 28.2$.

$$A_{cl} = \begin{bmatrix} 1.00 & 0.56 & 1.11 & 0.16 & 0.01 & 0.06 & -3.08 & -0.55 \\ 0 & 0.82 & 6.58 & 1.11 & 0.04 & 0.27 & -13.3 & -2.39 \\ 0 & -0.2 & 14.2 & 2.54 & 0.04 & 0.29 & -14.2 & -2.56 \\ 0 & -1.2 & 78.6 & 14.2 & 0.25 & 1.73 & -84.4 & -15.2 \\ 0.42 & 0 & 2.32 & 0 & 0.59 & 0.62 & -4.29 & -0.39 \\ 0.85 & 0 & 13.1 & 0 & -0.81 & 1.1 & -19.7 & -1.29 \\ 1.72 & 0 & 28 & 0 & -1.68 & 0.1 & -28.0 & -0.02 \\ 9.6 & 0 & 155.9 & 0 & -9.35 & 0.61 & -161.7 & -1.06 \end{bmatrix}$$

$$A_{ol} = \begin{bmatrix} 1.00 & 0.56 & 1.11 & 0.16 & 0.01 & 0.06 & -3.08 & -0.55 \\ 0 & 0.82 & 6.58 & 1.11 & 0.04 & 0.27 & -13.3 & -2.39 \\ 0 & -0.2 & 14.2 & 2.54 & 0.04 & 0.29 & -14.2 & -2.56 \\ 0 & -1.2 & 78.6 & 14.2 & 0.25 & 1.73 & -84.4 & -15.2 \\ 0 & 0 & 0 & 0 & 1.01 & 0.62 & -1.97 & -0.39 \\ 0 & 0 & 0 & 0 & 0.04 & 1.1 & -6.69 & -1.29 \\ 0 & 0 & 0 & 0 & 0.04 & 0.1 & -0.02 & -0.02 \\ 0 & 0 & 0 & 0 & 0.25 & 0.61 & -5.83 & -1.06 \end{bmatrix}$$

Let the given performance requirement be: the augmented state variable X must be exponentially stable such that

$$\frac{\|X[n+6]\|}{\|X[n]\|} < 0.5, \quad \forall\, n > 0, X[n]. \tag{4.1}$$

This condition specifies that the *energy* in the vector X must at least halve in every 6 periods. This is a stronger condition than asymptotic stability. This condition may be satisfied for different deadline hit and miss patterns of 6 consecutive control signals. Let a such a pattern by represented as $\sigma = (\sigma_1, \ldots, \sigma_6)$, where $\sigma_i \in \{H, M\}$. For a given pattern, there is a corresponding matrix $A_\sigma = \Pi_{i=1,\ldots,6} A_i$, where $A_i = A_{cl}$ if $\sigma_i = H$ and $A_i = A_{ol}$ if $\sigma_i = M$. As shown in [3], condition of (4.1) is satisfied for a pattern σ if the corresponding matrix A_σ has an eigenvalue less than 0.5. All patterns satisfying this condition are specified as a regular language L_A. The MATLAB code to generate these patterns for the specifications mentioned above is given in Appendix A.1. The automaton corresponding to L_A for the computed matrices A_{cl} and A_{ol} is computed using these patterns. It is then minimized for the purpose of checking the language inclusion property and this minimized automaton is shown in Fig. 4.1. Every state in this automaton has a history of the last 5 events, since the condition specifies that the energy must at least halve in every 6 periods.

The controller is implemented in a distributed real-time system: (a) sensed data is sent via a shared bus to the controller, (b) control output is computed on a dedicated processing unit, (c) control outputs are sent via the same shared bus back to the plant. There are two other tasks which also read data via the bus, compute and write data back through the bus. The arbitration policy on the bus is non-preemptive first-come-first-serve. Thus, the bus is a shared resource which can increase the sensor-to-actuator delay. The minimum and maximum time (in s), for each phase of each task is shown in Table 4.1. Contention time before read or write is not included. τ_0 is the controller task for the inverted pendulum with a period of 0.6s.

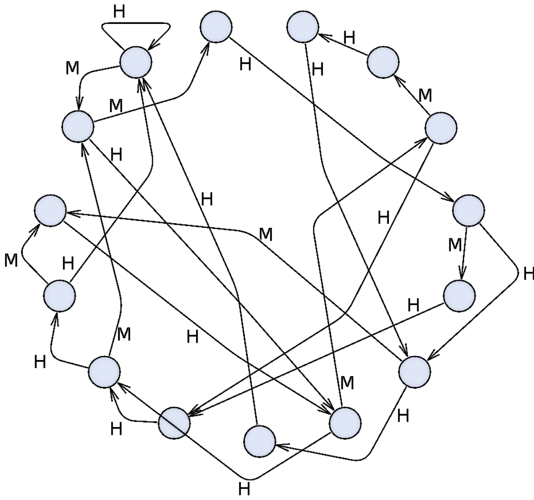

Figure 4.1: Minimized Automaton for L_A.

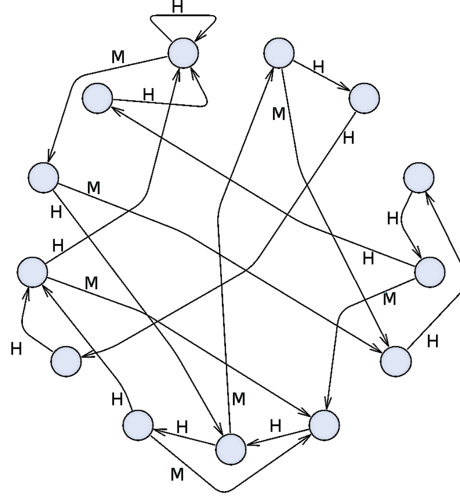

Figure 4.2: Minimized Automaton for L_G.

Table 4.1: Task parameters for the control example

Task	Read Range	Exec Range	Write Range	Period
τ_0	0.06-0.18	0.06-0.12	0.06-0.12	0.6
τ_1	0.9-1.02	0.06-0.12	0.06-0.12	2.4
τ_2	1.38-1.5	0.06-0.12	0.06-0.18	3.6

Applying the method of [15], the worst-case response time of τ_0 is 0.72s $> P$. Using the schedulability guarantee, the plant can never be guaranteed to be in the closed-loop mode, and thus is not guaranteed to meet the requirement of (4.1). Further, no amount of speed-up of the processor executing τ_0 can meet the controller requirement. One would have to speed up the bus by at least 37.5% to satisfy the requirement.

Since the given L_A accepts strings of size 6, L_G for $k = 5$ is computed. Fig. 4.2 shows the corresponding minimized automaton, where every state has a history of length k, i.e., 5. We then compute $L_{\texttt{diff}} = L_G \cap \neg L_A$, where $\neg L_A$ is the language complement. Figure 4.3 shows the automaton representing this difference. The automaton was found to have no accepting state and hence, it was verified that $L_{\texttt{diff}} = \phi$, and thus $L_G \subseteq L_A$. This shows that the real-time system can meet the controller constraint of (4.1), contrary to the conclusion from the schedulability guarantee.

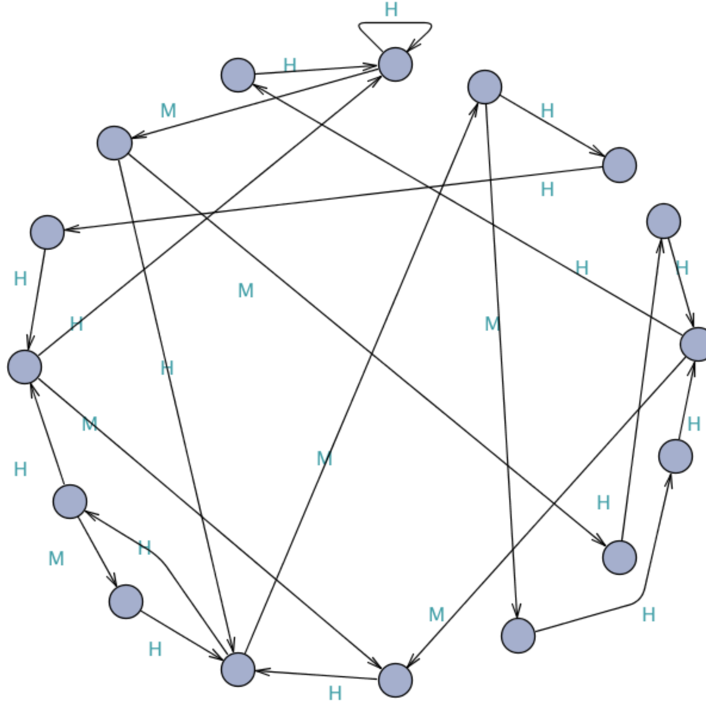

Figure 4.3: Automaton representation of $L_{\texttt{diff}}$.

4.2 Calculating Worst-Case Deadline Miss Rate

While the language-based guarantee is very detailed, it can be used to compute specific metrics. An example of this is the worst-case deadline miss rate. Let w be an infinite length string of deadline hits and misses, where the ith element $w_i \in \{H, M\}$. The worst-case deadline miss rate WMR is given as

$$\texttt{WMR} = \max_{w \in L_S} \frac{|\{i \ : \ w_i = M\}|}{|w|}. \tag{4.2}$$

From the correctness property (Lemma 1), computing the WMR with L_G, instead of L_S, is a safe over-approximation.

Given A_k corresponding to L_G, WMR is computed as follows:

- A graph $G' = (V', E', \rho)$, is constructed, where the vertices V' correspond to the states in A_k, the edges E' are the defined transitions in A_k, and weights $\rho : E \to \{-1, 1\}$ is -1 (similarly 1) on edges if the corresponding transition accepts M (H).
- The, the minimum cycle mean of G', denoted MCM, is computed with the Minimum Cycle Mean algorithm [16].
- Finally, $\texttt{WMR} = \dfrac{1 - \texttt{MCM}}{2}$.

WMR is used to compare scheduling algorithms for mixed-criticality systems [11]. In mixed-criticality systems, every task τ_i has a defined criticality level say χ_i, and at any given time, t, there is a global criticality level say $\chi_g(t)$. If $\chi_g(t) > \chi_i$ then most existing scheduling policies decide to *drop* task τ_i from t onwards. In other words, when the global criticality level rises due to certain exceptional run-time events, low criticality tasks are no more scheduled. This abrupt dropping of tasks is due to the limitation of the schedulability guarantee. Instead,the low criticality tasks may still be run under reduced performance guarantees, in particular with higher WMR.

Consider a dual-criticality (HI, LO) task-set with three tasks executing on independent processing cores, but interfering on a shared bus which follows a non-preemptive fixed-priority arbitration policy. Tasks read data via the bus, compute using the read data and write data back via the bus. Read and write is done through individual accesses, each requiring 1 time unit once access to the bus is granted. The ranges of read-write access requests and execution times (in time units) for the different tasks, when the global criticality level is HI, are shown in Table 4.2. In this case, tasks τ_1 and τ_2 may not be guaranteed to meet all their deadlines. But we can guarantee a certain WMR for each.

Table 4.2: Mixed criticality task-set

τ_i	χ_i	**Read**	**Write**	**Exec**	**Period**	**Deadline**
τ_0	HI	5-7	6-9	7-8	80	80
τ_1	LO	4-6	7-8	5-6	50	30
τ_2	LO	3-5	7-9	6-7	60	30

We consider two priority assignments, $\texttt{PA}_1 = \tau_0 > \tau_1 > \tau_2$ and $\texttt{PA}_2 = \tau_0 > \tau_2 > \tau_1$. For both, the language-based guarantee for τ_1 and τ_2 is computed. Because of the absence of a pre-defined k, the iterative refinement procedure is used to increase k with the terminating condition: $U(k) < 0.1$ or $U_{k+1}/U_k > 0.9$. WMR is then computed for each task of both priority assignments. The results in Table 4.3 are interpreted as follows:

- Fairness amongst LO-criticality tasks: The absolute value of the difference between WMR of τ_1 and τ_2 is smaller for $\texttt{PA}_2$.
- Responsiveness of LO-criticality tasks: The sum of the WMR of τ_1 and τ_2 is higher for $\texttt{PA}_2$.

Table 4.3: Mixed criticality results

	$\texttt{PA}_1$		$\texttt{PA}_2$	
	τ_1	τ_2	τ_1	τ_2
k	9	12	15	8
WMR	0.3125	0.75	0.5415	0.25

Hence, $\texttt{PA}_2$ is clearly a better priority assignment. However, when using only the schedulability guarantee, the two priority assignments are indistinguishable in the HI-criticality mode, and both τ_1 and τ_2 will be dropped in either case. We can similarly compare two different scheduling policies using the language-based guarantee.

4.3 Summary

After the introduction of the guarantee generation procedure in the last chapter, this chapter illustrated two applications of the guarantee language.

The first application explains the specific case of an inverted pendulum on a moving cart. Using the schedulability guarantee, the plant can never be guaranteed to be in the closed-loop mode, and thus is not guaranteed to meet the controller constraint. This was successfully disproved with the help of the language inclusion property of the guarantee language.

The second application demonstrated the worst-case miss-rate (WMR) property of the guarantee language. Using the example of a mixed-criticality system, it can be shown that LO-criticality tasks need not always be dropped in the HI-criticality mode. Additionally, priority assignments and scheduling policies can be compared for such tasks in the HI-criticality mode using WMR. Such a design step would not be possible with the schedulability guarantee.

In conclusion, both these applications show the superiority of the language-based guarantee over the schedulability guarantee. The next chapter discusses various experiments that were conducted to study various properties of the guarantee generation procedure.

Chapter 5

Experiments

Apart from the applications of the regular language, as discussed in the previous chapter, it is essential to study different aspects of the procedure. For instance, it would be useful to study the effects of change in different parameters such as the utilization of the shared resource, the relative deadline of the task, number of cores per task-set etc. This chapter focuses on different experiments conducted to measure such effects.

Additionally, when using model-checking, it is pertinent to check for the scalability of the method. It is important to test for the model-checking time for different test cases. The chapter begins by putting forward an algorithm used to generate random task-sets, followed by various experiments which conclude with certain remarks about the scalability of the procedure.

5.1 Randomized Task-Set Generation

To study the the effects of changes in different variables (e.g. read-write time, execution time, number of cores, etc.), the following algorithm was used to generate random task-sets. The goal of the algorithm is to facilitate changes to the values of the input variables and study the effect of such changes on certain output variables such as verification time.

5.1.1 Notations

Notation	Meaning
N	Number of cores with one task per core
T	Task-set
τ_i	Task $i \in T$
a	Time taken for each access
P_{set}	1: Same period for each task; 2: harmonic periods; 3: random periods.
P_i	Period of task τ_i
U_{bus}	Total utilization of the bus
U^i_{bus}	Utilization of the bus by τ_i such that $$U_{bus} = \sum_{i=0}^{N-1} U^i_{bus}$$
$R_{i,worst}; R_{i,best}$	Worst and best case number of resource accesses during read phase by τ_i
$W_{i,worst}; W_{i,best}$	Worst and best case number of resource accesses during write phase by τ_i
ν_i	Total access time of τ_i such that $$\nu_i = a * (R_i + W_i)$$
μ_i	Ratio of read to write access times of τ_i
$E_{i,worst}; E_{i,best}$	Worst and best case execution time of τ_i
Z_1	Ratio of Best to Worst Case Execution Time
Z_2	Ratio of Best to Worst Case Access Times
$D_{i,min}$	Minimum deadline of task τ_i which equal it's minimum finish time

Table 5.1: Notations

5.1.2 Algorithm

$U_{bus} \leftarrow random_float(0.7, 0.9)$
$N \leftarrow random_int(2, 4)$
$a \leftarrow random_int(1, 5)$
$P_{set} \leftarrow random_int(1, 3)$
$sum \leftarrow 0$
$Z_1 \leftarrow random_float(0.8, 1.0)$
$Z_2 \leftarrow random_float(0.7, 1.0)$
for $i \leftarrow 0, N-1$ **do**
  $P_i \leftarrow$ Period based on P_{set} value
  $U^i_{bus} \leftarrow random_float(0.2, 0.8)$
  $sum \leftarrow sum + U^i_{bus}$
end for
for $i \leftarrow 0, N-1$ **do**
  $U^i_{bus} \leftarrow U^i_{bus}/sum$
  $\nu_i \leftarrow U^i_{bus} * P_i$
  $E_{i,worst} \leftarrow (P_i - \nu_i) * random_float(0.2, 0.5)$
  $\mu_i \leftarrow random_float(0.45, 0.55)$
  $R_{i,worst} \leftarrow \lfloor \nu_i * \mu_i / a \rfloor$
  $W_{i,worst} \leftarrow \lfloor \nu_i / a \rfloor - R_i$
  $E_{i,best} \leftarrow \lfloor E_{i,worst} * Z_1 \rfloor$

$R_{i,best} \leftarrow \lfloor R_{i,worst} * Z_2 \rfloor$
$W_{i,best} \leftarrow \lfloor W_{i,worst} * Z_2 \rfloor$
$D_{i,min} \leftarrow \lfloor (R_{i,best} + W_{i,best}) * a + E_{i,best} \rfloor$
end for

5.1.3 Comments

- Here, the shared resource is assumed to be a bus.
- $U^i_{bus} \in [0.2, 0.8]$ so that for more processes, each process gets at least some bus access.
- $E_{i,worst} \in [0.2, 0.4]$ of the time remaining in P_i after access time is deducted from it. Values higher than 0.4 may lead to the WCRT of τ_i going beyond P_i.
- $\mu_i \in [0.45, 0.55]$, but the range can be extended.
- Similarly, ranges for Z_1 and Z_2 can be varied to generate different sets of results.

5.2 Experimental Results

The algorithm was used to generate various test cases to verify different properties of the guarantee generation procedure. However, it is difficult to find a particular pattern for different variables from this data due to the combined effect of changes in parameters such as relative deadline, number of cores etc. Nonetheless, this data, along with certain conclusions that can be drawn from it, is provided in Appendix B.1.1. Thus, only certain peculiar cases that follow from general trends seen in the data, and those which are primarily affected by variation in a single parameter, are illustrated in this chapter.

5.2.1 Variation in Deadline

The first set of experiments tested the effect of variation in the relative deadline of the task on the Worst-Case Miss Rate metric, the Uncertainty Metric and the guarantee generation time. The details of the particular test case, follow by the results, are provided below:

- The best and worst case response times were initially calculated using UPPAAL. BCRT is 162 while WCRT is 232.
- The Deadline is varied from 160 to 162 and then in steps of 14 time units.
- The effect of this Deadline variation on the verification time, WMR and U(k) is then measured.
- The task-set uses 3 cores with 1 task/core and FCFS arbitration policy.
- k varies from 1 to 10 for each deadline.

The specific task-set used for this experiment and the results are listed in Appendix B.1.2. The results are shown in Figures 5.1, 5.2 and 5.3.

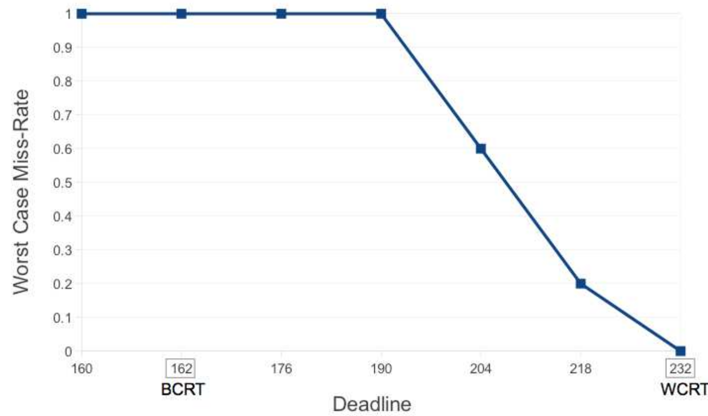

Figure 5.1: WMR vs Deadline.

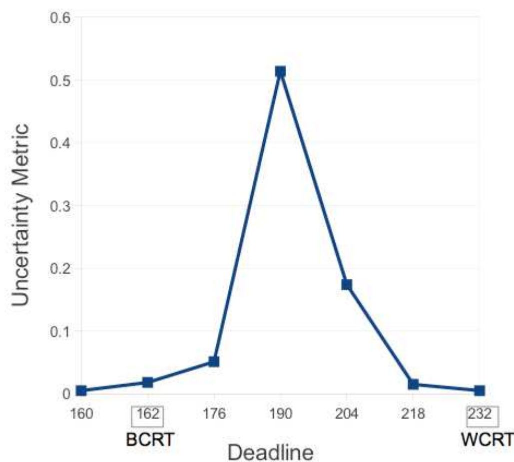

Figure 5.2: Uncertainty Metric vs Deadline.

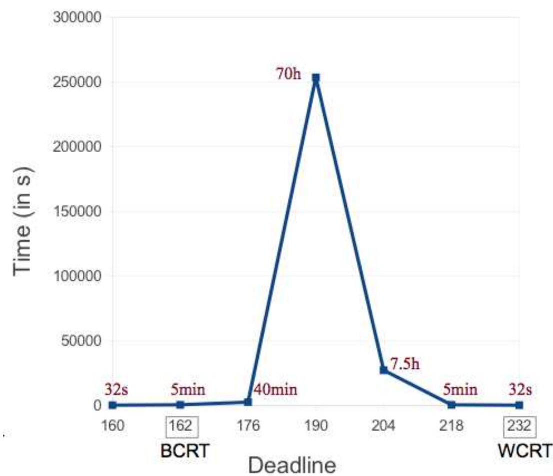

Figure 5.3: Time vs Deadline.

When the deadline is close to the best case response time, there is a high probability of the number of misses being more than the number of hits. Similarly, close to the worst case response time, it is the other way around. However, as can be seen from the graph, highest uncertainty occurs when the deadline is close to the half way mark between BCRT and WCRT. Consequently, the number of edges in the guarantee automaton are higher when the deadline is half way between BCRT and WCRT. This increases the verification time, as well as the overall uncertainty in the system. The graph provides a good picture of this intuitive explanation. Additionally, the WMR decreases as the deadline increases because the total number of misses encountered decreases as well.

However, such a trend can be seen in the data in Appendix B.1.1, and is not a one-off occurrence. Hence, in general, there is more uncertainty in the system, and consequently a higher model-checking time is observed, when the deadline is half way between BCRT and WCRT.

5.2.2 Variation in Bus (Resource) Utilization

Due to blocking access to shared resource, as the overall resource utilization and hence resource contention increases, the response time of the tasks tends to increase as well. This would result in more deadline misses. The following example verifies this property. The details of a particular test case, followed by the results, are provided below:

- The task-set uses 3 cores with 1 task/core and FCFS arbitration policy.

- k varies from 1 to 10 for each run of the procedure.
- Deadline=$(BCRT + WCRT)/2$, where BCRT and WCRT are calculated for the task-set with 50% bus utilization.
- Bus utilization changes from 0% to 100% in steps of 10%.

The specific task-sets used for this experiment and the results are listed in Appendix B.1.3.

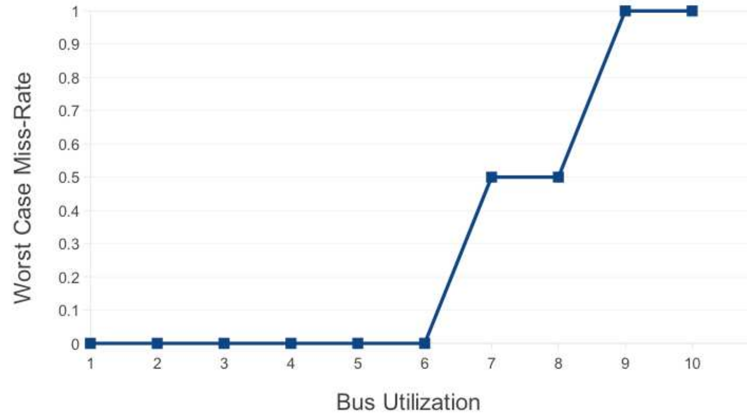

Figure 5.4: WMR vs Bus Utilization.

Hence, the behavior of the system is as predicted: with increase in bus utilization, the resource contention increases and consequently the WMR increases as well.

5.2.3 Scalability Experiments

Finally, various experiments to test the scalability of the guarantee generation procedure were conducted. These experiments were conducted for varying number of cores, with one or more tasks per core, and for different arbitration policies. For each of the experiments below, the worst-case scenario was tested. This worst-case scenario was, to a great extent, derived from the experiments conducted using the random task-set generation algorithm, and illustrated in Appendix B.1.1. Such a scenario has the following properties:

1. Deadline is half-way between best and worst case response times.
2. The access and execution ranges are large.
3. Resource utilization is relatively high.
4. The value of k is increases from 1 to 6, which is a relatively high value for the stopping condition.

As discussed previously, it was observed that when the deadline is near the half-way mark between BCRT and WCRT, there is more uncertainty in the system. With large access and execution ranges, the state space increases during the model-checking process. Additionally, a higher resource utilization causes more resource contention. And finally, a higher value of k implies a bigger observer automaton with a larger state space and more transitions. All of these significantly increase the model-checking time in UPPAAL. Hence, it provides a worst-case measure of the scalability of the guarantee generation method.

The results of these experiments are illustrated in Tables 5.2 and 5.3. The columns are the number of cores while the rows are the arbitration policies. A *computational budget of 18 hours* was set to complete the guarantee generation for the given values of k. The tables show the number of task-sets for which the guarantee generation procedure could be completed within the given time frame. (Specific task-sets for these experiments have not been mentioned or listed in this thesis.)

1. For the first set, FCFS, FP and TDMA were checked against increasing number of cores with 1 task per core.

	2	3	4	5	6
$FCFS$	$> 60\%$	$> 60\%$	$< 20\%$	$< 20\%$	$< 20\%$
FP	$> 60\%$	$> 60\%$	$20 - 60\%$	$< 20\%$	$< 20\%$
$TDMA$	$> 60\%$	$> 60\%$	$> 60\%$	$> 60\%$	$> 60\%$

Table 5.2: Scalability for 1 task per core.

2. For the second set, FCFS, FP and TDMA were checked against increasing number of cores, but with more than 1 task per core. *Stopwatches* in UPPAAL were used to model more than 1 task per core with the FP scheduling policy on each core. This further increased the guarantee generation time significantly.

   For 2 tasks per core:

	2	3
$FCFS$	$20 - 60\%$	$< 20\%$
FP	$20 - 60\%$	$< 20\%$
$TDMA$	$> 60\%$	$> 60\%$

Table 5.3: Scalability for 2 tasks per core.

   For 3 tasks per core:

	2	3
$FCFS$	$< 20\%$	$< 20\%$
FP	$< 20\%$	$< 20\%$
$TDMA$	$> 60\%$	$> 60\%$

Table 5.4: Scalability for 3 tasks per core.

To conclude, the method proposed in this thesis for generating a language-based guarantee is scalable for different arbitration policies, even in the worst-case scenario. The method performs extremely well for TDMA since the task under consideration can be isolated and checked. However, for FCFS and FP policies, such a design step is not possible. Hence, the performance in case of both these policies cannot match that achieved for TDMA. Additionally, a computational budget of 18 hours is not always sufficient when formal verification is involved.

However, given better conditions such as a deadline equal to the WCRT, or lower k values, ideally between 3 to 5, as stopping condition, there is a high possibility that the guarantee generation procedure completes within a realistic computational budget.

Chapter 6

Conclusion

The language-based guarantee details the deadline hit and miss patterns of the jobs of a task. A method to compute such a guarantee of a given complexity was proposed in this thesis. A method to solve the same problem for a language of unknown complexity was also put forward. Two applications were then presented in order to describe the applicability of such a guarantee language. The applications illustrated the superiority of the language-based guarantee over the schedulability guarantee.

With a wide scope of application, the only limitation of this method is the inevitable cost of model-checking. To address this, and to check other properties of the language, various experiments were conducted and the results have been included in this thesis. From these experiments, it could be concluded that the method scales for any number of cores with TDMA, but to a limited number of cores (a maximum of 4) with FCFS and FP, in the worst case scenario.

The inevitable model-checking costs, though, were greatly reduced by using a suitable observer template and the corresponding modification procedure, as well as the iterative refinement approach that reuses information from one iteration to the next.

In conclusion, the language-based guarantee is a viable option for the design and analysis of real-time task-sets. It provides a more advanced alternative to the schedulability guarantee, and it would be fascinating to explore other applications of this method in the future.

Appendix A

Appendix A

A.1 Code for Inverted Pendulum Example

```
%% inverted pendulum example
% with current set up:
% MMM is allowed
% MMM => H
% MMMHM => HH

clear;

VERBOSE = 'm'; % (l)ow - (m)ed - (h)igh

%% useful functions

max_eig = @(A) (max(abs(real(eig(A)))));

%% inputs

M = 0.5;
m = 0.2;
b = 0.1;
I = 0.006;
g = 9.8;
l = 0.3;

p = I*(M+m)+M*m*l^2; % denominator for the A and B matrices

%% compute state space model

% x(t + 1) = Ax(t) + Bu(t)
%     y(t) = Cx(t) + Du(t)

A = [0      1              0           0;
     0 -(I+m*l^2)*b/p  (m^2*g*l^2)/p   0;
     0      0              0           1;
     0 -(m*l*b)/p       m*g*l*(M+m)/p  0];
B = [     0;
     (I+m*l^2)/p;
          0;
        m*l/p];
C = [1 0 0 0;
     0 0 1 0];
D = [0;
     0];

%% construct matlab system model

sys = ss(A, B, C, D);

%% discretize the system model and extract state space model

Ts = .6; % was .6
```

```
sysd = c2d(sys, Ts);

A_p = sysd.a;
B_p = sysd.b;
C_p = sysd.c;

%% weight matrices

rho = 10; phi = 1;
% rho/phi controls sparsity of control
QXU = diag([rho rho rho rho phi]);

rho = .01; phi = 1;
QWV = diag([rho rho rho rho phi phi]);


%% compute the filter and regulator

KLOQ = lqg(sysd, QXU, QWV);

A_c = KLOQ.a;
B_c = KLOQ.b;
C_c = KLOQ.c;

%% compute system matrices for open and closed loop
A_cl = [A_p B_p*C_c; B_c*C_p A_c];
A_ol = [A_p B_p*C_c; zeros(4)  A_p+B_p*C_c];

%% obtained eigenvalues
fprintf('Closed loop max gain = %g\n', max_eig(A_cl));
fprintf('Open loop max gain = %g\n', max_eig(A_ol));


%% compute allowed strings

len = 6; % was 10
threshold = 0.5; % was 0.5

flag = zeros(2^len, 1);

for i = 0 : 2^len - 1
  k = i;
  A_prod = eye(8);
  seq = '';
  for j = 1 : len
    if (mod(k, 2) == 1)
      A_prod = A_prod * A_cl;
      seq = strcat(seq, 'H');
    else
      A_prod = A_prod * A_ol;
      seq = strcat(seq, 'M');
    end
    k = floor(k/2);
  end
  if (VERBOSE == 'h')
     fprintf('%g: %g\n', i, max_eig(A_prod));
  end
  if (max_eig(A_prod) < threshold)
    flag(i + 1) = 1;
    if (VERBOSE == 'm')
      disp(seq);
    end
  end
end

fprintf('Found %g strings out of %g\n', sum(flag), 2^len);
```

A.2 The Shell Script

The code to generate the guarantee language is written using Python. However, to automate the process of executing the script, a separate shell script is run. An example of this is provided below:

python path_to_main.py path_to_model.xml path_to_query.q Template_name k_1 k_2 path_to_verifyta

Here, main.py is the Python script that calls the procedures that generate the guarantee, while the Template_name is the name of the UPPAAL template where the Observer will be modelled. k_1 and k_2 are first and last k values till which the procedure will execute. However, for more complex stopping conditions, the Python script can be altered.

A.3 The `verifyta` Options

`verifyta` is UPPAAL's command-line tool used for model-checking. The general syntax of the command is as follows:

verifyta model.xml query.q -options

Here, model.xml is the XML file containing the UPPAAL models of the Real-Time System. query.q is the file containing the query. Some of the options that were tested are given below.

1. State Space reduction: S0, was used whenever enough memory was available; S1 is the conservative option, and was used when less memory was available.
2. Search order: o0 (breadth first) was faster than o1 (depth first) and o2 (random depth first).
3. Diagnostic trace: t0 (some trace) was faster than t1 (fastest trace) and t2 (shortest trace).

Hence, the ideal combination was found to be: S0, o0 and t0, along with the T option to reuse state space when possible.

A.4 Outline to the Python scripts

The entire guarantee generation process, for a given as well as unknown k, is automated using Python scripts. This section provides an outline for this code, and a brief summary of what each function in the different files does. The code is separated into different files, and the functions in each file are illustrated below.
Filename: *main.py*
It serves as the starting point for the execution of the process. It calls all the other functions involved in the guarantee generation process, and contains only the main function which takes the following arguments:

- filename – name of the XML file that contains the XML code for the model
- query_file – name of the query file that will contain the query
- template_name – base name of the template in which the user wants the observer to be placed. It might be appended based on the value of k.
- k1 – value of k required to build the first observer automaton
- k2 – value of k required to build the last observer automaton
- verifyta_path – path to the verifyta command on the local machine

Filename: *auto_generator.py*
It contains the functions used in the generation of the observer and guarantee automaton for a given value of k. The functions and corresponding arguments are described below:

1. parse_xml(filename, template_name, k): Parses the XML file (containing the RTS model) and inserts the custom node 'template_name' which acts as the observer. Arguments:
    - filename – name of the XML file that contains the XML code for the model
    - template_name – name of the template that holds the observer.
    - k – value of k required to build the observer automaton
2. change_declaration(text): Adds the global variable 'e_global' to the XML file's *Declarations* template. Arguments:
    - text – the text that is to be appended

   Returns:
    - text – the modified text
3. add_transitions(k, newNode): Adds the transitions between locations by modifying the XML file containing the RTS model. Arguments:
    - k – value of k required to build the observer automaton
    - newNode – the XML ElementTree node that acts as the 'observer' template

   Returns:
    - newNode – modified newNode
4. transition_generator(source, target, hit_miss, newNode, nail, e_global): Simplifies the repetitive addition of transitions to the XML file. Arguments:
    - src – source location of the transition
    - tar – target location of the transition
    - hit_miss – string representing hit? or miss?
    - newNode – the XML ElementTree node that acts as the 'observer' template
    - nail – location where a nail attribute (related to UPPAAL GUI) should be placed in the new transition
    - e_global – initial value of e_global on that edge

   Returns:
    - newNode – modified newNode
5. pretty_xml(newNode): Makes the XML in the XLM file look pretty. Arguments:
    - newNode – the XML ElementTree node that acts as the 'observer' template

   Returns:
    - root – root element of the modified XML tree (which is nothing but the modified newNode)

Filename: *query_generator.py*
It is used to generate the query file.

1. generate(query_file): Generates the query file and writes the query to it. Arguments:
    - query_file – name of the query file that will contain the query

Filename: *verifier.py*
It is used to run the `verifyta` command, generate and examine the counter example trace, and make changes to the observer automaton by altering the XML file.

1. verify(filename, query_file, template_name, verifyta_path): Checks which edges can possibly be traversed by the k-automaton and updates the XML to give the final k-automaton. Arguments:

- filename – name of the XML file that contains the XML code for the model
- query_file – name of the query file that will contain the query
- template_name – name of the template that holds the observer
- verifyta_path – path to the verifyta command on the local machine

2. generate_trace(filename, query_file, verifyta_path): Executes the verifyta command and produces the output and error for the given files (i.e., the XML and query files). The error contains the counter-example trace generated by verifyta. The options for verifyta are discussed in Appendix A.2. Arguments:

   - filename – name of the XML file that contains the XML code for the model
   - query_file – name of the query file that will contain the query
   - verifyta_path – path to the verifyta command on the local machine

   Returns:

   - out – output of the verifyta command
   - err – error generated by the verifyta command

3. find_ids(err, template_name): Examines the counter-example trace generated by verifyta and finds the ID's of the locations where the trace registered an error. Actually, the transition between these locations is the one that caused the counter-example trace, but it is identified using the locaiton ID's. Arguments:

   - err – error generated by the verifyta command
   - template_name – name of the template that holds the observer

   Returns:

   - id1 – source location of the error edge
   - id2 – target location of the error edge

4. correct_xml(filename, template_name, id1, id2): Corrects the e_global value on the error edge, i.e., assigns it a value 1 by modifying the XML file. The location ID's are search as source and target in the XML file to find the node corresponding to the edge (transition) responsible for the trace. Arguments:

   - filename – name of the XML file that contains the XML code for the model
   - template_name – name of the template that holds the observer
   - id1 – source location of the error edge
   - id2 – target location of the error edge

5. auto_prune(filename, template_name): Prunes the final observer automaton (for k), i.e., removes all the edges and locations not part of A_k. This is done by identifying the edges where the e_global value is 0 after no more counter-example traces are generated. Then, all the locations with no incoming transitions are removed as well. Arguments:

   - filename – name of the XML file that contains the XML code for the model
   - template_name – name of the template that the user wants the observer to be placed in

Filename: *auto_extender.py*
The functions here are used to extend the guarantee automaton (final observer automaton) for k to the initial observer for $k+1$. This step corresponds to the method discussed in Section 3.2 of the thesis.

1. extend(filename, template_name, k1, k2): Extends the observer automaton from k1 to k2 (k1<k2). This means that the function uses the observer for $k = k1$ to construct the initial observer for $k = k2$. Arguments:

- filename – name of the XML file that contains the XML code for the model
- template_name – base name of the template to be inserted. It might be appended based on the value of k.
- k1 – value of k of the existing observer automaton
- k2 – value of k for the next automaton

2. final_prune(filename, template_name): Removes the $e_global = 1$ assignment from the edges in the automaton, to make it look neater in the UPPAAL GUI. Arguments:
   - filename – name of the XML file that contains the XML code for the model
   - template_name – name of the template that is to be pruned

Filename: *minimizer.py*
It is used to minimize the entire final observer automaton using Equivalence classes.

1. minimize(filename, template_name): Minimizes the guarantee automaton (final observer). The minimizaiton procedure is divided into three parts and this function calls each one separately. Arguments:
   - filename – name of the XML file that contains the XML code for the model
   - template_name – name of the template that holds the observer to be minimized
2. generate_adjlist(filename, template_name): Generates the adjacency list from a given DFA graph (i.e., the final observer). Arguments:
   - filename – name of the XML file that contains the XML code for the model
   - template_name – name of the template that holds the observer to be minimized

   Returns:
   - adj_list – adjacency list in Python dictionary form
3. generate_classes(adj_list): Generates the equivalence classes from a given DFA. Arguments:
   - adj_list – adjacency list in Python dictionary form

   Returns:
   - final – list of equivalence classes
4. file_DFA(filename, classes, adj_list, template_name): Generates the reduced DFA using the Equivalence classes. Arguments:
   - filename – name of the XML file that contains the XML code for the model
   - classes – list of equivalence classes
   - adj_list – adjacency list in Python dictionary form
   - template_name – name of the template that holds the observer to be minimized
5. add_transitions(newNode, adj_list, i, dict_list): Adds transitions in the minimized DFA. Arguments:
   - newNode – the XML ElementTree node that acts as the observer template
   - adj_list – adjacency list in Python dictionary form
   - i – index of element in dict_list
   - dict_list – list of equivalence classes

   Returns:
   - newNode – modified newNode with transitions inserted

Filename: *last_minimizer*
The purpose of this file, including the functions are the same as that of *minimizer.py*, except in this case, only the leaf locations are used in the minimization process.
Filename: *degree_generator.py*
It is used to generate the various metrics mentioned in Section 3.2.3. Additionally, it also computes the Worst-Case Miss-Rate Metric mentioned in the Applications chapter.

1. deg_1_2(filename, template_name, k1, k2): Calculates the Total Degree Index and the Partial Degree Index for the finals observers corresponding to k and $k+1$. Arguments:
    - filename – name of the XML file that contains the XML code for the model
    - template_name – base name of the template that holds the observer. This is appended later depending on the value of k.
    - k1 – corresponds to k
    - k2 – corresponds to $k+1$
2. deg_3(filename, template_name, k1, k2): Calculates the Uncertainty Metric for all final observers from $k = k1$ to $k = k2$ $(k1 < k2)$. Arguments:
    - filename – name of the XML file that contains the XML code for the model
    - template_name – base name of the template that holds the observer. This is appended later depending on the value of k.
    - k1 – corresponds to the initial k
    - k2 – corresponds to the final k
3. deg_4(filename, template_name, k1, k2): Calculates the Worst-Case Miss-Rate Metric for all final observers from $k = k1$ to $k = k2$ $(k1 < k2)$. Arguments:
    - filename – name of the XML file that contains the XML code for the model
    - template_name – base name of the template that holds the observer. This is appended later depending on the value of k.
    - k1 – corresponds to the initial k
    - k2 – corresponds to the final k
4. deg_5(filename, template_name, k1, k2): Conveys the maximum misses before one or more guaranteed hits, for e.g., MMM=>HH for all final observers from $k = k1$ to $k = k2$ $(k1 < k2)$. Arguments:
    - filename – name of the XML file that contains the XML code for the model
    - template_name – base name of the template that holds the observer. This is appended later depending on the value of k.
    - k1 – corresponds to the initial k
    - k2 – corresponds to the final k

Appendix B

Appendix B

B.1 Experiment Tables

This sections consists of various experiments that were conducted during the thesis. The first subsection with preliminary experiments consists of task-sets generated by the random task-set generation algorithm presented in Section 5.1.2. Each of these task-sets was executed using the FCFS arbitration policy for a shared resource. This data was mainly used to derive certain generic results like the worst-case scenario described in Section 5.2.

The second and third subsections presents data for specific experiments (designed to test specific properties of the guarantee generation) discussed in Chapter 5. We begin with the preliminary experiments below.

B.1.1 Preliminary Experiments

The task-sets for this set of experiments were generated using the random task generation algorithm. The highest value of k for each experiment was set to 6. The stopping conditions used were the same as those mentioned in Section 3.2.3 of the thesis. However, even if there was no change in $U(k)$, the relative stopping condition was not applied till $k = 6$. This provided a better perspective to judge the model-checking time in cases where there was maximum uncertainty in the system.

The Access Latency of the resource (C) as well as the total bus (shared resource) utilization are mentioned at the top of every data-set. R_{min} - R_{max}, W_{min} - W_{max}, and E_{min} - E_{max} are the read, write and execution ranges, respectively. P is the period of each task while D_{calc} is the minimum deadline of the given task when there is no resource contention. U^i_{bus} provides the bus utilization of every task, while D_{min} and D_{max} are the best and worst case response times, respectively. Every task-set is followed by the k value when the process stopped, the total guarantee generation time and the $U(k)$ value when the process was terminated.

Since these task-sets were generated using a random task generation algorithm, and due to the fact that several parameters like access and execution ranges, relative deadline etc. had wide possible ranges, the data is not easy to interpret and deriving conclusions is difficult. However, several examples can be discussed which highlight certain properties of the guarantee generation procedure.

When the deadline is close to the BCRT or WCRT, the guarantee generation time is less, and the $U(k)$ values for the final k are small. This can be seen in Tables 1, 9, 18, 25, 37 and 55. However, there are a few cases where this is not true, for example, Tables 3, 4, 12, 32, 53. Thus, a deadline close to BCRT or WCRT would not always imply a smaller $U(k)$ or a smaller guarantee generation time because the system can still be highly uncertain in such cases. It is also worth noting that in most of these examples, the guarantee generation time is significantly high for higher number of cores (4).

In contrast, when the deadline is close to half-way between BCRT and WCRT, there is high uncertainty in the system which is reflected by a high $U(k)$ and a greater guarantee generation time. This can be seen in Tables 22, 33, 35, 38, 48, 50, 51, and 52. It can also be seen, again, that as the number of cores increases, the guarantee generation time increases as well. However, this is a case specific to FCFS arbitration policy, and can also be seen in FP arbitration policy, but may not apply to TDMA since in that case, each core can be isolated and checked. Additionally, there are only a few examples where $U(k)$ is low, even though the deadline is close to half-way between BCRT and WCRT. One such example can be seen in Table 21. However, the guarantee generation time is still quite large.

Apart form the number of cores and relative deadline, the width of the access and execution ranges also has an effect on the guarantee generation time. For instance, it is easy to see from Tables 10,13, 16, and 42, that even though the deadline is not close to half of BCRT and WCRT, the guarantee generation time is high.

As previously mentioned, since there are a lot of factors that simultaneously affect the guarantee generation time, $U(k)$ and other parameters, drawing conclusions from these tables is not straightforward. Additionally, such extensive experiments have only been conducted for the FCFS arbitration policy for one task per core. It would be interesting to devise techniques to 'learn' from this data, and to derive results for other arbitration policies, with more than one task per core in order to gain further insight into the performance of the guarantee generation procedure.

1. C= 3, U_{bus}= 0.895

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	5	6	5	6	10	13	100	40	0.521	52	82	82
τ_1	7	8	7	8	12	16	100	54	0.374	73	91	-

Table B.1

k	$Time(s)$	$U(k)$
5	1.027	0.0806451612903

Table B.2

2. C= 5, U_{bus}= 0.703

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	4	2	4	18	22	100	38	0.253	38	82	51
τ_1	2	4	3	5	59	72	200	84	0.45	89	145	-

Table B.3

k	$Time(s)$	$U(k)$
6	89.108	1.0

Table B.4

3. C= 1, U_{bus}= 0.773

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	34	40	39	46	105	106	400	178	0.232	210	312	-
τ_1	16	19	18	22	84	85	300	118	0.218	136	185	184
τ_2	6	8	7	9	34	35	100	47	0.139	57	94	-
τ_3	8	10	10	12	27	28	100	45	0.183	52	99	-

Table B.5

k	$Time(s)$	$U(k)$
6	295560.657	0.373015873016

Table B.6

4. C= 2, U_{bus}= 0.877

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	22	26	100	30	0.267	30	70	32
τ_1	6	8	6	8	64	74	200	88	0.148	92	180	-
τ_2	24	29	23	28	112	128	400	206	0.172	244	384	-
τ_3	46	54	44	52	252	288	800	432	0.289	576	760	-

Table B.7

k	$Time(s)$	$U(k)$
6	85814.971	0.349206349206

Table B.8

5. C= 5, U_{bus}= 0.872

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	0	1	0	1	18	22	100	18	0.188	18	60	-
τ_1	3	4	3	4	29	34	200	59	0.15	59	170	70
τ_2	9	11	9	11	95	111	400	185	0.245	200	346	-
τ_3	12	15	11	14	244	285	800	359	0.289	479	585	-

Table B.9

k	$Time(s)$	$U(k)$
6	798.886	0.222222222222

Table B.10

6. C= 4, U_{bus}= 0.865

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	11	12	10	11	37	47	300	121	0.532	141	228	-
τ_1	31	34	28	31	41	51	500	277	0.333	337	467	451

Table B.11

k	$Time(s)$	$U(k)$
6	29.136	0.190476190476

Table B.12

7. C= 1, U_{bus}= 0.833

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	21	22	24	25	20	24	100	65	0.345	80	93	84
τ_1	31	32	35	36	50	61	200	116	0.488	156	168	-

Table B.13

k	$Time(s)$	$U(k)$
6	29.897	0.222222222222

Table B.14

8. C= 4, U_{bus}= 0.717

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	21	23	100	29	0.147	29	71	-
τ_1	1	2	1	2	29	32	100	37	0.185	37	76	-
τ_2	1	2	1	2	34	37	100	42	0.169	42	76	-
τ_3	0	1	0	1	26	29	100	26	0.217	26	61	51

Table B.15

k	$Time(s)$	$U(k)$
6	4911.589	1.0

Table B.16

9. C= 2, U_{bus}= 0.852

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	7	8	8	10	18	21	100	48	0.482	62	88	66
τ_1	18	21	23	26	41	48	200	123	0.37	153	188	-

Table B.17

k	$Time(s)$	$U(k)$
6	126.411	0.634920634921

Table B.18

10. C= 4, U_{bus}= 0.78

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	0	1	16	19	100	20	0.209	20	64	-
τ_1	4	5	3	4	52	62	200	80	0.166	84	168	-
τ_2	9	10	8	9	104	124	400	172	0.204	192	280	223
τ_3	20	22	18	19	233	277	800	385	0.201	521	577	-

Table B.19

k	$Time(s)$	$U(k)$
6	1974.684	0.222222222222

Table B.20

11. C= 1, U_{bus}= 0.814

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	19	22	22	26	11	13	100	52	0.328	70	83	-
τ_1	25	29	30	35	56	64	200	111	0.486	149	175	159

Table B.21

k	$Time(s)$	$U(k)$
6	1955.142	1.0

Table B.22

12. C= 3, U_{bus}= 0.805

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	3	4	14	15	100	32	0.157	44	81	-
τ_1	3	4	2	3	27	29	100	42	0.258	54	81	80
τ_2	1	2	1	2	37	40	100	43	0.246	43	78	-
τ_3	1	2	1	2	16	18	100	22	0.144	27	54	-

Table B.23

k	$Time(s)$	$U(k)$
6	5771.843	1.0

Table B.24

13. C= 4, U_{bus}= 0.802

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	34	38	100	42	0.263	42	84	-
τ_1	7	8	6	7	33	37	200	85	0.209	91	149	-
τ_2	12	13	11	12	74	83	400	166	0.33	202	271	255

Table B.25

k	$Time(s)$	$U(k)$
6	1430.977	1.0

Table B.26

14. C= 4, U_{bus}= 0.718

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	27	30	100	43	0.244	43	88	77
τ_1	3	4	3	4	73	81	200	97	0.293	109	165	-
τ_2	9	12	8	11	60	67	400	128	0.181	160	214	-

Table B.27

k	$Time(s)$	$U(k)$
6	14.289	0.166666666667

Table B.28

15. C= 2, U_{bus}= 0.784

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	8	8	10	17	19	100	45	0.388	55	71	69
τ_1	14	17	17	21	37	41	200	99	0.396	121	153	-

Table B.29

k	$Time(s)$	$U(k)$
6	224.125	1.0

Table B.30

16. C= 2, U_{bus}= 0.888

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	51	59	41	48	96	109	500	280	0.456	296	494	-
τ_1	43	50	34	40	56	64	400	210	0.432	236	398	251

Table B.31

k	$Time(s)$	$U(k)$
6	3308.18	0.420634920635

Table B.32

17. C= 4, U_{bus}= 0.725

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	2	3	19	20	100	31	0.258	31	72	66
τ_1	4	5	5	6	74	75	200	110	0.235	122	163	-
τ_2	9	11	10	13	128	129	400	204	0.232	260	292	-

Table B.33

k	$Time(s)$	$U(k)$
6	26.851	0.277777777778

Table B.34

18. C= 1, U_{bus}= 0.713

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	19	20	16	17	12	14	100	47	0.331	63	72	63
τ_1	33	35	28	30	49	56	200	110	0.382	147	164	-

Table B.35

k	$Time(s)$	$U(k)$
6	29.251	0.222222222222

Table B.36

19. C= 5, U_{bus}= 0.712

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	0	1	29	33	100	34	0.216	34	74	-
τ_1	2	3	1	2	30	34	100	45	0.202	55	79	59
τ_2	1	2	0	1	30	34	100	35	0.294	35	74	-

Table B.37

k	$Time(s)$	$U(k)$
6	282.056	1.0

Table B.38

20. C= 3, U_{bus}= 0.726

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	3	4	12	15	100	30	0.21	30	78	42
τ_1	6	7	6	7	32	38	200	68	0.289	83	131	-
τ_2	12	14	11	13	101	119	400	170	0.227	212	290	-

Table B.39

k	$Time(s)$	$U(k)$
6	196.406	0.52380952381

Table B.40

21. C= 2, U_{bus}= 0.734

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	4	5	31	39	100	45	0.153	45	100	-
τ_1	9	10	9	10	43	53	200	79	0.198	85	164	-
τ_2	14	16	15	17	121	149	400	179	0.214	207	343	286
τ_3	27	30	28	31	142	175	800	252	0.169	344	439	-

Table B.41

k	$Time(s)$	$U(k)$
6	29436.214	0.277777777778

Table B.42

22. C= 5, U_{bus}= 0.713

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	26	29	100	46	0.36	46	79	-
τ_1	6	7	6	7	24	27	200	84	0.353	94	122	107

Table B.43

k	$Time(s)$	$U(k)$
6	112.159	1.0

Table B.44

23. C= 3, U_{bus}= 0.875

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	8	6	8	15	19	100	51	0.372	56	94	-
τ_1	9	12	8	11	36	45	200	87	0.502	108	157	121

Table B.45

k	$Time(s)$	$U(k)$
6	290.249	1.0

Table B.46

24. C= 5, U_{bus}= 0.788

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	0	1	17	20	100	22	0.381	22	60	-
τ_1	3	4	3	4	55	64	200	85	0.198	90	145	-
τ_2	12	15	11	14	103	118	400	218	0.21	258	348	295

Table B.47

k	$Time(s)$	$U(k)$
6	2575.869	1.0

Table B.48

25. C= 3, U_{bus}= 0.837

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	7	5	6	14	16	100	47	0.405	60	81	61
τ_1	12	14	10	12	29	34	200	95	0.432	125	166	-

Table B.49

k	$Time(s)$	$U(k)$
6	10.797	0.222222222222

Table B.50

26. C= 5, U_{bus}= 0.82

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	17	21	100	37	0.439	42	81	57
τ_1	3	4	3	4	11	14	100	41	0.381	51	84	-

Table B.51

k	$Time(s)$	$U(k)$
6	105.253	1.0

Table B.52

27. C= 3, U_{bus}= 0.74

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	38	41	100	44	0.238	44	85	49
τ_1	1	2	1	2	28	31	100	34	0.147	34	79	-
τ_2	2	3	2	3	27	30	100	39	0.157	48	87	-
τ_3	2	3	3	4	25	27	100	40	0.199	49	88	-

Table B.53

k	$Time(s)$	$U(k)$
6	14924.569	1.0

Table B.54

28. C= 4, U_{bus}= 0.889

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	20	22	100	28	0.339	32	62	-
τ_1	3	4	3	4	28	31	100	52	0.198	66	88	-
τ_2	3	4	3	4	14	16	100	38	0.352	50	84	57

Table B.55

k	$Time(s)$	$U(k)$
6	402.738	1.0

Table B.56

29. C= 3, U_{bus}= 0.726

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	4	5	5	6	25	26	100	52	0.393	64	92	68
τ_1	5	6	6	7	25	26	100	58	0.334	79	95	-

Table B.57

k	$Time(s)$	$U(k)$
6	165.69	1.0

Table B.58

30. C= 1, U_{bus}= 0.848

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	23	29	19	25	14	18	100	56	0.3	71	94	85
τ_1	12	16	10	13	19	24	100	41	0.548	52	82	-

Table B.59

k	$Time(s)$	$U(k)$
6	2142.46	1.0

Table B.60

31. C= 2, U_{bus}= 0.821

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	7	7	8	17	19	100	43	0.235	63	95	-
τ_1	5	6	6	7	28	32	100	50	0.316	81	97	-
τ_2	4	5	5	6	28	32	100	46	0.27	62	96	73

Table B.61

k	$Time(s)$	$U(k)$
6	3154.296	1.0

Table B.62

32. C= 2, U_{bus}= 0.871

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	7	8	8	9	27	31	100	57	0.505	69	83	-
τ_1	23	24	24	25	25	29	200	119	0.366	147	171	147

Table B.63

k	$Time(s)$	$U(k)$
6	548.336	1.0

Table B.64

33. C= 4, U_{bus}= 0.784

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	31	32	100	39	0.288	39	78	62
τ_1	2	3	2	3	33	34	100	49	0.222	57	88	-
τ_2	2	3	2	3	29	30	100	45	0.274	53	88	-

Table B.65

k	$Time(s)$	$U(k)$
6	485.631	1.0

Table B.66

34. C= 1, U_{bus}= 0.867

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	17	20	15	18	20	25	100	52	0.484	66	88	-
τ_1	43	50	39	46	35	44	200	117	0.383	151	189	170

Table B.67

k	$Time(s)$	$U(k)$
6	8054.32	1.0

Table B.68

35. C= 5, U_{bus}= 0.771

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	1	2	30	34	100	40	0.202	40	85	-
τ_1	0	1	0	1	32	37	100	32	0.23	32	70	-
τ_2	3	4	3	4	64	73	300	94	0.191	99	168	130
τ_3	6	7	7	8	140	159	400	205	0.148	215	309	-

Table B.69

k	$Time(s)$	$U(k)$
6	19926.437	1.0

Table B.70

36. C= 3, U_{bus}= 0.793

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	4	6	4	7	18	23	100	42	0.362	51	91	-
τ_1	3	5	4	6	18	22	100	39	0.431	45	88	53

Table B.71

k	$Time(s)$	$U(k)$
6	365.787	1.0

Table B.72

37. C= 3, U_{bus}= 0.746

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	23	24	19	20	74	75	300	200	0.291	251	279	268
τ_1	29	31	25	26	120	121	600	282	0.455	399	427	-

Table B.73

k	$Time(s)$	$U(k)$
6	181.463	0.746031746032

Table B.74

38. C= 3, U_{bus}= 0.863

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	4	5	18	19	100	39	0.28	51	99	-
τ_1	3	4	4	5	22	23	100	43	0.299	55	99	-
τ_2	3	4	4	5	23	24	100	44	0.284	59	98	86

Table B.75

k	$Time(s)$	$U(k)$
6	1792.222	1.0

Table B.76

39. C= 3, U_{bus}= 0.879

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	27	29	100	39	0.242	48	97	-
τ_1	2	4	2	3	31	33	100	43	0.194	52	100	-
τ_2	2	3	2	3	33	35	100	45	0.246	54	98	97
τ_3	2	4	2	3	21	22	100	33	0.198	42	97	-

Table B.77

k	$Time(s)$	$U(k)$
6	66963.204	1.0

Table B.78

40. C= 3, U_{bus}= 0.829

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	19	23	100	31	0.198	31	84	39
τ_1	4	5	4	5	45	54	200	69	0.217	75	159	-
τ_2	13	16	13	16	110	133	400	188	0.169	227	397	-
τ_3	21	26	21	26	135	162	800	261	0.245	354	515	-

Table B.79

k	$Time(s)$	$U(k)$
6	13256.895	0.5

Table B.80

41. C= 2, U_{bus}= 0.863

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	11	12	10	11	10	12	100	52	0.389	80	94	-
τ_1	8	9	8	9	19	22	100	51	0.474	77	94	83

Table B.81

k	$Time(s)$	$U(k)$
6	205.586	1.0

Table B.82

42. C= 1, U_{bus}= 0.83

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	64	78	75	91	46	56	400	185	0.404	185	358	224
τ_1	92	112	107	130	110	134	600	309	0.426	371	546	-

Table B.83

k	$Time(s)$	$U(k)$
6	4650.181	0.166666666667

Table B.84

43. C= 5, U_{bus}= 0.873

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	15	16	15	16	135	149	600	285	0.229	375	549	-
τ_1	7	8	7	8	67	74	400	137	0.272	175	330	182
τ_2	3	4	3	4	66	73	300	96	0.213	120	220	-
τ_3	5	6	5	6	44	49	300	94	0.16	146	235	-

Table B.85

k	$Time(s)$	$U(k)$
6	600.123	0.142857142857

Table B.86

44. C= 4, U_{bus}= 0.805

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	1	2	24	25	100	36	0.165	44	79	-
τ_1	2	3	1	2	26	27	100	38	0.252	46	77	-
τ_2	0	1	0	1	27	28	100	27	0.256	27	54	-
τ_3	1	2	0	1	37	38	100	41	0.132	41	78	76

Table B.87

k	$Time(s)$	$U(k)$
6	2061.485	1.0

Table B.88

45. C= 5, U_{bus}= 0.831

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	8	5	6	140	154	400	195	0.194	225	385	-
τ_1	4	5	3	4	44	49	200	79	0.182	79	195	-
τ_2	12	14	9	11	158	173	600	263	0.232	300	508	-
τ_3	8	10	6	8	135	148	500	205	0.223	205	403	323

Table B.89

k	$Time(s)$	$U(k)$
6	70479.364	0.753968253968

Table B.90

46. C= 1, U_{bus}= 0.828

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	20	22	21	23	17	18	100	58	0.374	71	90	78
τ_1	33	36	35	38	25	27	200	93	0.453	125	145	-

Table B.91

k	$Time(s)$	$U(k)$
6	412.791	1.0

Table B.92

47. C= 2, U_{bus}= 0.776

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	4	6	3	5	29	32	100	43	0.238	55	98	-
τ_1	5	7	4	6	25	28	100	43	0.257	59	98	-
τ_2	4	6	3	5	32	36	100	46	0.281	58	98	65

Table B.93

k	$Time(s)$	$U(k)$
6	4399.824	1.0

Table B.94

48. C= 4, U_{bus}= 0.744

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	4	2	4	21	27	100	37	0.406	41	91	-
τ_1	3	5	3	5	22	28	100	46	0.338	54	95	78

Table B.95

k	$Time(s)$	$U(k)$
6	219.128	1.0

Table B.96

49. C= 2, U_{bus}= 0.741

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	10	11	9	10	11	14	100	49	0.307	69	82	73
τ_1	6	7	6	7	17	20	100	41	0.433	53	76	-

Table B.97

k	$Time(s)$	$U(k)$
6	188.091	1.0

Table B.98

50. C= 4, U_{bus}= 0.833

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	1	2	0	1	14	17	100	18	0.233	18	61	-
τ_1	1	2	1	2	23	27	100	31	0.171	31	81	-
τ_2	2	3	1	2	28	33	100	40	0.208	52	85	72
τ_3	2	3	1	2	27	32	100	39	0.222	51	85	-

Table B.99

k	$Time(s)$	$U(k)$
6	5251.133	1.0

Table B.100

51. C= 2, U_{bus}= 0.759

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	5	6	4	5	14	16	100	32	0.317	48	77	-
τ_1	4	5	3	4	14	17	100	28	0.245	40	70	-
τ_2	6	8	5	7	15	18	100	37	0.197	58	79	67

Table B.101

k	$Time(s)$	$U(k)$
6	1783.613	1.0

Table B.102

52. C= 4, U_{bus}= 0.746

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	0	1	0	1	20	21	100	20	0.199	20	52	39
τ_1	4	5	4	5	74	77	200	106	0.132	106	172	-
τ_2	7	9	8	10	140	145	400	200	0.214	240	285	-
τ_3	16	19	17	20	262	270	800	394	0.201	482	572	-

Table B.103

k	$Time(s)$	$U(k)$
6	6261.409	0.738095238095

Table B.104

53. C= 4, U_{bus}= 0.707

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	3	4	26	31	100	50	0.353	50	87	-
τ_1	7	8	7	9	26	31	200	82	0.354	94	127	94

Table B.105

k	$Time(s)$	$U(k)$
6	129.371	1.0

Table B.106

54. C= 5, U_{bus}= 0.747

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	0	1	0	1	37	38	100	37	0.153	37	70	53
τ_1	2	3	2	3	52	53	200	72	0.178	72	145	-
τ_2	8	9	7	8	141	144	400	216	0.189	236	333	-
τ_3	10	12	10	11	207	211	800	307	0.227	367	451	-

Table B.107

k	$Time(s)$	$U(k)$
6	1220.487	0.968253968254

Table B.108

55. C= 1, U_{bus}= 0.774

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	12	14	13	15	24	27	100	49	0.469	60	77	76
τ_1	39	45	42	48	29	33	200	110	0.305	134	173	-

Table B.109

k	$Time(s)$	$U(k)$
6	68.609	0.222222222222

Table B.110

56. C= 5, U_{bus}= 0.754

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	15	19	100	35	0.438	35	70	41
τ_1	6	8	6	8	21	27	200	81	0.316	91	162	-

Table B.111

k	$Time(s)$	$U(k)$
6	7.865	0.222222222222

Table B.112

B.1.2 Test Case: Varying Deadline

This first test case illustrates the tables for the figures in Section 5.2.1. The experiment consists of 3 tasks, with 1 task per core and an FCFS arbiter. Table B.114 shows the results for the different deadlines for the same task-set.

C= 3, U_{bus}= 0.739

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}
τ_0	17	19	20	22	167	176	500	278	0.298	284	389
τ_1	11	12	11	13	66	70	400	132	0.248	162	232
τ_2	16	18	19	21	108	114	400	213	0.194	246	333

Table B.113

$Deadline$	k	$Time(s)$	$Degree$	$MissRate$
160	5	10.088	0.0806451612903	-1
162	7	97.342	0.0787401574803	-1
176	9	1254.913	0.073385518591	-1
190	10	253217.933	0.514173998045	-1
204	10	27040.606	0.174486803519	-0.2
218	7	94.705	0.0708661417323	0.6
232	5	9.909	0.0806451612903	1

Table B.114

B.1.3 Test Case: Varying Bus (Resource) Utilization

This experiment illustrates the results for the effects of varying bus utilization illustrated in Section 5.2.2. The U_{bus} on top of each table shows the total bus utilization for that task-set. Each task-set consists of 3 tasks with FCFS arbitration.

C= 1, U_{bus}= 0.0

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	0	0	0	0	5	6	100	5	0.0	-1	-1	-
τ_1	0	0	0	0	12	13	200	12	0.0	-1	-1	85
τ_2	0	0	0	0	24	26	400	24	0.0	-1	-1	-

Table B.115

k	$Time(s)$	$Degree$	$MissRate$
10	1.549	0.00488758553275	1

Table B.116

C= 1, U_{bus}= 0.1

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	0	1	0	1	5	6	100	5	0.033	-1	-1	-
τ_1	2	3	2	3	12	13	200	16	0.033	-1	-1	85
τ_2	5	6	5	6	24	26	400	34	0.033	-1	-1	-

Table B.117

k	$Time(s)$	$Degree$	$MissRate$
10	2.654	0.00488758553275	1

Table B.118

C= 1, U_{bus}= 0.2

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	2	3	2	3	5	6	100	9	0.067	-1	-1	-
τ_1	5	6	5	6	12	13	200	22	0.067	-1	-1	85
τ_2	11	12	12	13	24	26	400	47	0.067	-1	-1	-

Table B.119

k	$Time(s)$	$Degree$	$MissRate$
10	4.538	0.00488758553275	1

Table B.120

C= 1, U_{bus}= 0.3

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	3	4	4	5	5	6	100	12	0.1	-1	-1	-
τ_1	8	9	9	10	12	13	200	29	0.1	-1	-1	85
τ_2	18	19	19	20	24	26	400	61	0.1	-1	-1	-

Table B.121

k	$Time(s)$	$Degree$	$MissRate$
10	6.258	0.00488758553275	1

Table B.122

C= 1, U_{bus}= 0.4

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	5	6	5	6	5	6	100	15	0.133	-1	-1	-
τ_1	11	12	12	13	12	13	200	35	0.133	-1	-1	85
τ_2	23	25	25	27	24	26	400	72	0.133	-1	-1	-

Table B.123

k	$Time(s)$	$Degree$	$MissRate$
10	7.906	0.00488758553275	1

Table B.124

C= 1, U_{bus}= 0.5

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	6	7	7	8	5	6	100	18	0.167	-1	-1	-
τ_1	14	15	16	17	12	13	200	42	0.167	-1	-1	85
τ_2	29	31	32	34	24	26	400	85	0.167	-1	-1	-

Table B.125

k	$Time(s)$	$Degree$	$MissRate$
10	14.51	0.00488758553275	1

Table B.126

C= 1, U_{bus}= 0.6

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	8	9	9	10	5	6	100	22	0.2	-1	-1	-
τ_1	18	19	19	20	12	13	200	49	0.2	-1	-1	85
τ_2	36	38	38	41	24	26	400	98	0.2	-1	-1	-

Table B.127

k	$Time(s)$	$Degree$	$MissRate$
10	19.943	0.00537634408602	0.0

Table B.128

C= 1, U_{bus}= 0.7

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	10	11	11	12	5	6	100	26	0.233	-1	-1	-
τ_1	20	22	22	24	12	13	200	54	0.233	-1	-1	85
τ_2	41	44	45	48	24	26	400	110	0.233	-1	-1	-

Table B.129

k	$Time(s)$	$Degree$	$MissRate$
10	42.17	0.00537634408602	0.0

Table B.130

C= 1, U_{bus}= 0.8

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	11	12	12	13	5	6	100	28	0.267	-1	-1	-
τ_1	23	25	25	27	12	13	200	60	0.267	-1	-1	85
τ_2	48	51	52	55	24	26	400	124	0.267	-1	-1	-

Table B.131

k	$Time(s)$	$Degree$	$MissRate$
10	6277.505	0.0606060606061	-1

Table B.132

C= 1, U_{bus}= 0.9

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	13	14	14	15	5	6	100	32	0.3	-1	-1	-
τ_1	26	28	29	31	12	13	200	67	0.3	-1	-1	85
τ_2	54	57	58	62	24	26	400	136	0.3	-1	-1	-

Table B.133

k	$Time(s)$	$Degree$	$MissRate$
10	109.62	0.00488758553275	-1

Table B.134

C= 1, U_{bus}= 1.0

$Task$	R_{min}	R_{max}	W_{min}	W_{max}	E_{min}	E_{max}	P	D_{calc}	U^i_{bus}	D_{min}	D_{max}	**DL**
τ_0	14	15	16	17	5	6	100	35	0.333	-1	-1	-
τ_1	29	31	32	34	12	13	200	73	0.333	-1	-1	85
τ_2	59	63	65	69	24	26	400	148	0.333	-1	-1	-

Table B.135

k	$Time(s)$	$Degree$	$MissRate$
10	255.304	0.00488758553275	-1

Table B.136

Bibliography

[1] A. Bemporad, M. Heemels, and M. Johansson, Networked control systems. Springer, 2010, vol. 406.

[2] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu, "A survey of recent results in networked control systems," Proceedings of the IEEE, vol. 95, no. 1, pp. 138-162, 2007.

[3] R. Alur and G. Weiss, "Regular specifications of resource requirements for embedded control software," in Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE. IEEE, 2008, pp. 159-168.

[4] G. Bernat, A. Burns, and A. Llamosi, "Weakly hard real-time systems," IEEE Trans. Computers, vol. 50, no. 4, pp. 308-321, 2001.

[5] P. Kumar and L. Thiele, "Quantifying the effect of rare timing events with settling-time and overshoot," in RTSS. IEEE Computer Society, 2012, pp. 149-160.

[6] A. D'Innocenzo, G. Weiss, R. Alur, A. J. Isaksson, K. H. Johansson, and G. J. Pappas, "Scalable scheduling algorithms for wireless networked control systems," in Automation Science and Engineering, 2009. CASE 2009. IEEE International Conference on. IEEE, 2009, pp. 409-414.

[7] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on, vol. 4. IEEE, 2000, pp. 101-104.

[8] R. Alur and D. L. Dill, "A theory of timed automata," Theoretical Computer Science, vol. 126, pp. 183-235, 1994.

[9] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," International Journal on Software Tools for Technology Transfer (STTT), vol. 1, no. 1, pp. 134-152, 1997.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in Computer aided verification. Springer, 2000, pp. 154-169.

[11] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE. IEEE, 2010, pp.13-22.

[12] I. Hur and C. Lin, "Adaptive history-based memory schedulers," in Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2004, pp. 343-354.

[13] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE. IEEE, 2011, pp. 269-279.

[14] J. Hespanha, "Lecture notes on lqr/lqg controller design." [Online]. Available: http://www.uz.zgora.pl/ wpaszke/materialy/kss/lqrnotes.pdf

[15] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in Proc. International Conference on Embedded Software (EMSOFT). Tampere, Finland: ACM, Oct 2012, pp. 63-72.

[16] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," Discrete Mathematics, vol. 23, no. 3, pp. 309-311, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0012365X78900110

[17] G. Giannopoulou, "A Framework for the State-based Real-time Analysis of Resource Contention Scenarios in Multicore Architectures," Master Thesis, ETH Zurich, 2011. [Online]. Available: ftp://ftp.tik.ee.ethz.ch/pub/people/ggeorgia/MasterThesis.pdf

[18] A. Nerode. Linear automaton transformations. In Proc. of the American Mathematical Society 9, pages 541-544, 1958.

[19] Wolfgang Thomas. 1991. Automata on infinite objects. In Handbook of theoretical computer science (vol. B), Jan van Leeuwen (Ed.). MIT Press, Cambridge, MA, USA 133-191.

[20] García, P.; Vidal, E., "Inference of k-testable languages in the strict sense and application to syntactic pattern recognition," Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol.12, no.9, pp.920,925, Sep 1990