Adaptive Embedded Systems

Yin Hang
Mälardalen University (MDH)
Supervisor: Hans Hansson

March 18, 2011

Abstract

Modern embedded systems are evolving in the direction of increased adaptivity and complexity. It is extremely important for a system with limited resource to be adaptive in order to maximize its efficiency of resource usage while guaranteeing a high level of fault tolerance and QoS. This report aims at exploring such a kind of system, i.e. Adaptive Embedded System (AES), which is featured by dynamic reconfiguration at runtime. Based on the investigation and analysis of a variety of case studies related with AES, we proposed the conceptual view and overall architecture of an AES by highlighting its predominant characteristics. We also made an incomplete but detailed summary of the most popular techniques that can be used to realize adaptivity. Those techniques are categorized into dynamic CPU/network resource re-allocation and adaptive fault tolerance. A majority of adaptive applications resort to one or more of those techniques. Besides, there is a separate discussion on dynamic reconfiguration and mode switch for AES. Finally, we classify adaptivity into different modeling problems at a higher abstraction level and build UPPAAL models for two different AESs, a smart phone and an object-tracking robot. Our UPPAAL models provide clear demonstration on how a typical AES works.
Keywords: adaptive,embedded systems,reconfiguration,modeling,UPPAAL

Acknowledgement

First I would like to thank my supervisor Hans Hansson, who offered me such an interesting topic as my master thesis. I procured tremendous benefit from his guidance throughout my thesis period. From the literature study at the very beginning to the final UPPAAL modeling, he gave me lots of valuable suggestions and kept enlightening me and encouraging me. With his help, I set clear subgoals and realized them one after another. I wouldn't have completed my thesis work without him.

I also would like to thank Damir Isovic who helped me a lot with administrative affairs, such as thesis registration, sign-up for presentation and many other non-academic but equally important issues.

Thank Paul Pettersson for his helpful advice on UPPAAL modeling. As one of the developers of UPPAAL and of course an expert in it, he told me quite a number of useful techniques essential to the modeling of an AES by UPPAAL. Besides, he ever recommended several other modeling tools to me as well. Although eventually UPPAAL became the only one modeling tool for my thesis work due to limited time, his recommendation undoubtedly broadened my knowledge.

Thank Eun-Young Kang who was very patient to instruct me in UPPAAL port modeling. UPPAAL port is suitable for component-based modeling and can be potentially used to model AESs. It is a pity that I did not manage to build any models by UPPAAL port, but I still believe that it is really beneficial to know UPPAAL port fundamentals.

Thank Thomas Nolte for providing me so many literatures on AESs, real-time systems and mode switch. Thank Sasikumar Punnekkat for assisting me to comprehend adaptive fault tolerance. Thank Mats Björkman for sharing his ideas on adaptive networks. Thank Moris Behnam for discussing the feedback control and AESs with me. Finally, thank Cristina Seceleanu and other staffs in IDT who ever helped me!

Contents

1 Introduction **5**

2 Existing techniques for adaptive embedded systems **10**
2.1 Dynamic CPU resource re-allocation 11
2.1.1 The Adaptive Resource Allocation (ARA) infrastructure 12
2.1.2 Imprecise computation 14
2.1.3 Elastic task model 15
2.2 Dynamic network resource re-allocation 16
2.3 Fault tolerance 18
2.3.1 Migration technique 18
2.3.2 Hardware reconfiguration in multiplex systems 18

3 Mode switch analysis **20**
3.1 Mode switch and dynamic reconfiguration 20
3.2 Mode switch problems 20
3.3 Mode switch protocol 22
3.4 Schedulability concerns during the mode switch 23
3.5 Other related issues 24

4 Application modeling **25**
4.1 Operational mode switch 25
4.2 Migration for fault tolerance and load balancing 26
4.3 Multimedia communication with stable streaming 27
4.4 Adaptive resource management and QoS degradation 28
4.5 Dynamic HW/SW module composition 29

5 The modeling of AESs and case study **30**
5.1 The modeling tool: UPPAAL 30
5.2 Case study 1: The UPPAAL modeling of a smart phone 31
5.2.1 Functionalities of the smart phone 31
5.2.2 Resource allocation mechanism and scheduling policy 32
5.2.3 The UPPAAL model of the smart phone 33
5.2.4 Property verification 41
5.2.5 Discussion 44
5.3 Case study 2: The UPPAAL modeling of an object-tracking robot . . . 45
5.3.1 Functionalities and adaptivity of the robot 45
5.3.2 The UPPAAL model of the robot 45
5.3.3 Property verification 51
5.3.4 Discussion 53

6 Related work **55**

7 Conclusion **56**

8 Appendix A: The complete UPPAAL model of the smart phone example **62**

8.1 The global variable declaration 62
8.2 The User template 63
8.3 The Application template 64
8.4 The Resource template 64
8.5 The AdmissionControl template 73
8.6 The Controller template 73
8.7 Properties and verification results 75

9 Appendix B: The complete UPPAAL model of the object-tracking robot **76**

9.1 The global variable declaration 76
9.2 The User template 77
9.3 The Robot template 77
9.4 The Sensor template 78
9.5 The Controller template 79
9.6 Properties and verification results 82

Chapter 1

Introduction

Traditional embedded systems usually work in a known and fixed environment which can be predicted and considered beforehand. However, in many cases, the operating condition is frequently changing in an unpredictable manner. Resource allocation for different applications has to be considered in worst cases for the sake of safety at design time. Consequently, most of the time resources such as CPU cycles, memory and energy cannot be efficiently utilized because their usage is overestimated. To guarantee a desired fault tolerance level, the large amount of software and hardware redundancy gives rise to excessive extra overhead of the entire system. Due to the increasing complexity and tight cost constraints of embedded systems, static approaches are no longer feasible. Instead, a system needs to be adaptive and flexible. An adaptive embedded system (AES) is supposed to reconfigure itself dynamically and automatically to deal with the varying operating environment or user requests. There is no doubt that numerous applications could benefit from the support of AESs, including avionic systems, automotive systems, robotics, multimedia, telecommunication, to name a few.

The adaptive behavior of an AES is reflected from its ability to dynamically reconfigure itself. A system may have a multitude of combinations of configurations (e.g. each combination of the worst-case computation times and periods of a task set can be regarded as one configuration), each of which corresponds to some particular operational condition. An AES should be able to switch from the current configuration to another one that is most or more suitable for the new working environment automatically at runtime, in response to an operational condition change. One challenge is that the number of configurations rises exponentially as the number of related parameters increases, while only a subset of them makes sense. For instance, many combinations of the worst-case computation times and periods of a task set are not schedulable. To guarantee that only feasible configurations appear in the dynamic reconfiguration process, some initial analysis can be done offline by extracting the schedulable subset from all possible configurations [3].

Usually, systems with dynamic reconfiguration support do not have to be designed from scratch. It is often possible to add the desired adaptive behaviors as a middleware to the existing operating system and application. The common functional modules of an operating system such as resource/energy/memory management can often be re-used and modified if necessary. Even some de facto frameworks have been explored for the development of AES such as the reflective architecture [35] [34] [24], the Hierarchical Scheduling Framework (HSF) [5], the FRESCOR framework [31] and the AUTOSAR architecture with organic middleware [46].

Generally speaking, the architecture and framework of the AES is quite dependent upon the goals as well as expected functionalities of its applications. Nonetheless, they more or less share some common characteristics. Herein we extract the abstract infor-

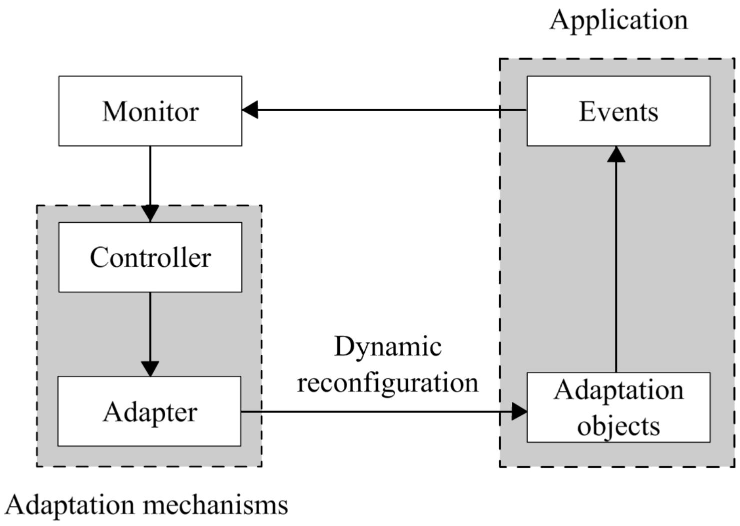

Figure 1.1: The conceptual view of an adaptive embedded system

mation of typical AESs and present a conceptual view of its functionalities in Figure 1.1. An AES is supposed to have a few key functional modules, which were also mentioned in the MCA paradigm (monitor-controller-adapter) in [15]. The system can be divided into monitor, controller and adapter. Logically speaking, the monitor is responsible for detecting the events that may trigger an adaptive behavior. Adaptation mechanisms are implemented in the controller and adapter, which are notified by the monitor and take some adaptive actions accordingly. Adaptivity will eventually be represented by the process of dynamic reconfiguration. For each running applications, any related event that requires reconfiguration will be captured by the monitor and some adaptation objects linked with those applications will be altered somehow through reconfiguration. The final goal is to satisfy the new requirements of the application. A successful reconfiguration will give rise to the abortion of those events currently generated from the application. In some other literature [25], adapter and configurator are used instead of the MCA paradigm. Although these terms are different, they still map each other well. The adapter is equivalent to the monitor and controller in the MCA paradigm whilst the configurator corresponds to the adapter in the MCA paradigm.

The conceptual view of the AES described in Figure 1.1 points out a clear direction of the AES architecture design. First, we come up with a generic overall architecture of a centralized AES in Figure 1.2. Without adaptive behaviors, the system has a normal input such as the periodic sampling of a visual sensor, and a normal output such as to display some videos or to actuate motors. The adaptive behavior must be triggered by some stimulus, which is detected by the monitoring mechanism. The triggering source can be an external environment change, or an internal event of the system itself, or a user request that is related to reconfiguration (There may be also other user interactions that are just normal inputs requiring no reconfiguration). For instance, the degradation of the network bandwidth is typically a type of external environment change, while the warning of low power supply in energy-constrained systems is an internal event from the system. Alternatively, the triggering can be periodic and time-based, which is relatively easier to handle. Different triggering sources may be detected by different monitors [23]. Once a stimulus is detected, the corresponding monitor will activate the adaptive mechanisms, which may focus on different objects with different technical

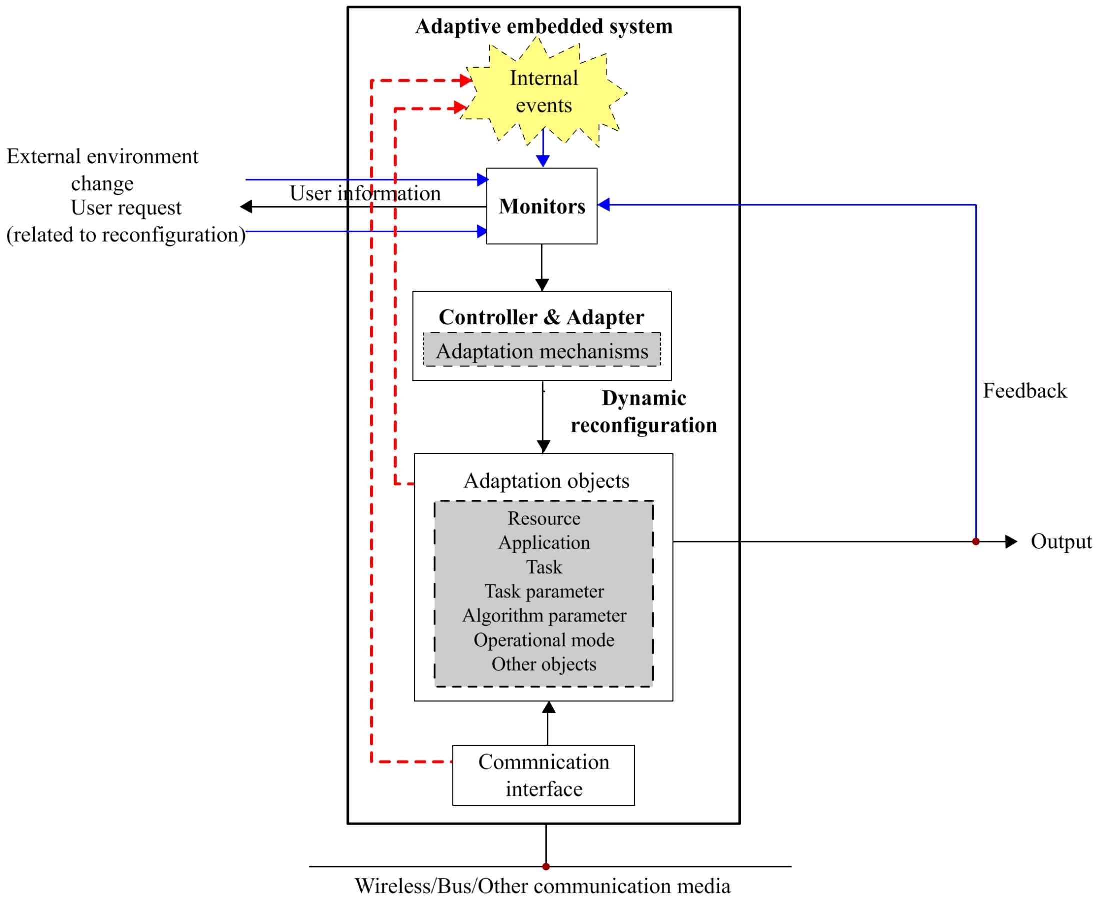

Figure 1.2: The overall architecture of a centralized adaptive embedded system

backgrounds. Then the monitor will notify the "Adaptation mechanisms" block, which is expected to take some actions timely and properly so as to adapt the new condition, such as resource re-allocation, algorithm parameter adjustment, operational mode switch, task/application migration and hardware component re-composition. Besides, in order to be able to communicate with other systems, a system should be equipped with certain communication interfaces, no matter whether it is adaptive or not. The communication interface module will be a potential source contributing to system internal events that will trigger some adaptive behavior. For instance, in multimedia applications, if the sender changes the encoding scheme and informs the receiver during the communication, the receiver could adapt its decoding scheme to match the changed encoding scheme of the sender.

By comparing Figure 1.1 and Figure 1.2, we can see that the overall architecture of a centralized AES is designed in the same pattern as the conceptual view. While the monitor is explicitly presented in both graphs, the "Adaptation mechanisms" block in Figure 1.2 functions as the controller and adapter in Figure 1.1. The controller and adapter could be middleware components. Their absence will not affect the functionality of an ordinary system, however, they are vital to an AES. Sometimes, they need more complex mechanisms such as effective QoS (Quality of Service) management in highly dynamic environments, where the assistance of feedback control is usually recommended. Actually, the feedback control mechanism is one distinctive feature of the AES because it borrows techniques from control theory to realize more advanced adaptive features.

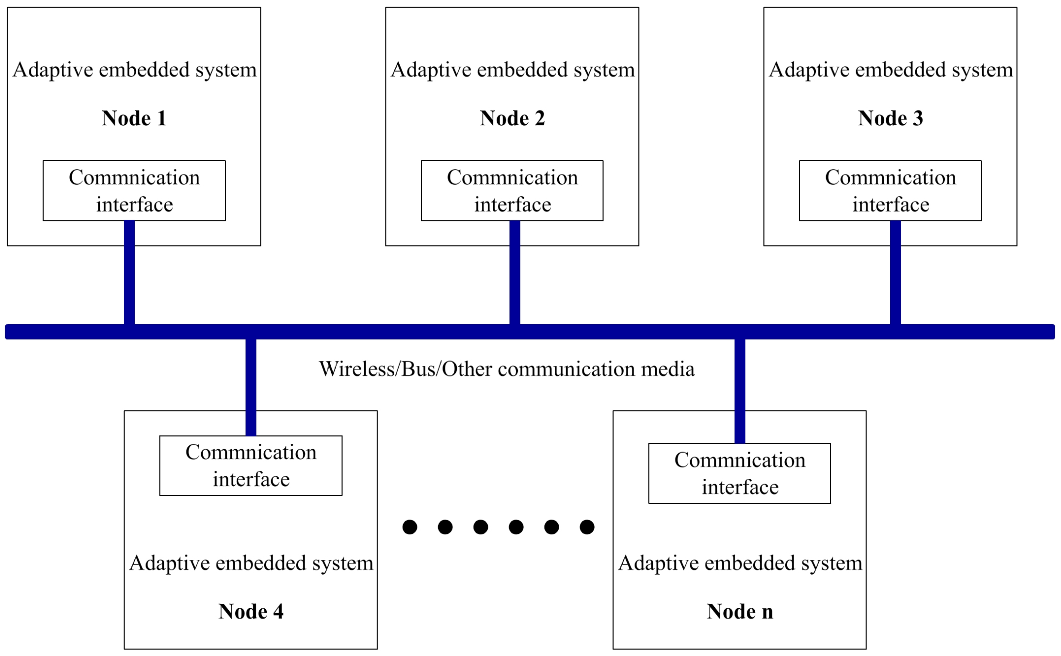

Figure 1.3: The network architecture of a distributed adaptive embedded system

The design of a feedback controller is quite flexible, as it can be integrated into the existing monitor and controller, or it can be another functional module, or it can even reside in another independent hardware component.

The conceptual view brings more flexibility for the architecture design of distributed AESs. Figure 1.3 presents a network with N nodes, which are connected through their communication interfaces and can communicate via wireless channels, buses, or other types of communication media. The distribution of different functional modules in the conceptual view becomes more principal here due to the diversity of solutions. For instance, in a peer-to-peer network where all nodes have the same behavior, each node is an AES described in Figure 1.2. Functional modules are evenly distributed among these nodes. In extreme cases, we can put all the functional modules concerning adaptivity in one node while the other nodes are normal systems without adaptive behaviors. This could be applied in a master-slave network structure. Or we can spread these functional modules among different nodes so that no single node is a complete AES and adaptivity can only be achieved by their coordination and cooperation. Actually even different monitors can be allocated to different nodes. This distribution decision should be made after thorough consideration of contributing factors such as network structure, number of nodes, hardware performance and software support of each node, desired functionalities and adaptivity.

It is not trivial to design an AES due to several reasons. First, even a simple adaptive behavior will affect quite a lot of software modules at different levels of the system. Sometimes, even the co-design of software and hardware has to be involved. In order to realize adaptivity, all the related modules need to be synchronized and cooperate with each other in a consistent manner. Second, the unpredictable working environment makes it almost unfeasible to test and verify the system in all possible cases, whereas for safety-critical applications, it is extremely significant to guarantee that any malfunction of additional adaptivity won't jeopardize the system. Finally, there is always a tradeoff between adaptivity and QoS degradation. Adaptivity makes a system more flexible, yet higher complexity and additional overhead is inevitable. However, an appropriate adaptive mechanism will try to minimize the negative influence upon the system performance while maximizing adaptivity.

The remainder of this report will be organized as follows. In Section 2, we discuss a few existing techniques for AESs in terms of dynamic resource re-allocation and adaptive fault tolerance. In Section 3, we analyze mode switch problems so as to get a better understanding of dynamic reconfiguration. We come up with five types of AESs and express them by abstract models respectively in Section 4. As two case studies, a smart phone model and an object-tracking robot model built by UPPAAL are explained in Section 5. Related work is in Section 6 and we make our conclusion in Section 7.

Chapter 2

Existing techniques for adaptive embedded systems

AESs themselves do not yield new technology. Instead, various currently existing techniques have been adopted for their realization. Adaptivity can be achieved in terms of both hardware and software. Some hardware platforms such as FPGA supporting partial dynamic reconfiguration [42] are especially suitable for the development of AES. Moreover, techniques such as Dynamic Voltage/Frequency Scaling (DVFS) and Dynamic Power Management (DPM) have also built a technical basis for AES at hardware level. Nevertheless, our focus will be software techniques that are much more versatile than hardware.

There are three main questions deserving our consideration: The reason to adapt, how to adapt (technique to adapt), and what to adapt (adaptation object). Table 2.1 lists a number of examples with possible answers to these three questions. The reason to adapt is typically related to the triggering source of the system, always detected by corresponding monitors. So far there is no unified standard specifying how to adapt. Various adaptive techniques have been proposed and implemented in different situations. For instance, techniques such as imprecise computation and the elastic task model can be used to get over an overloaded condition. The applicability of these techniques vary a lot. Some techniques are feasible for both centralized systems and distributed systems, whereas some others can only be applied in one of them. The object to adapt may be at different levels, as it can be a task parameter, or a software component, or even an application. One adaptive technique can focus on one or more objects to adapt. And one single object can be concerned by multiple adaptive techniques.

Table 2.1 cannot cover all the possible scenarios that may take place in AESs. As a matter of fact, it is barely practical to make a complete index of them by virtue of the highly unpredictable external environment and diversified adaptive techniques. However, the most common events, both internal and external, which act as the triggering source of reconfiguration, can be captured and considered beforehand at design time. In addition, the reasons to adapt in different rows are sometimes strongly correlated but not independent of each other. For example, when an application is running out of CPU resource, faults may occur if no proper adaptive actions are taken in time. This implies that some adaptive techniques can solve multiple problems even without that intention. The CPU resource re-allocation prevents critical applications from stepping into error states, meanwhile, fault tolerance is achieved as well. Table 2.1 can be regarded as an initial guidance for the AES development and can also be extended along with the evolution of AESs. In the subsequent subsections, we shall introduce some of the most representative techniques with regard to dynamic resource allocation and fault tolerance.

Reason to adapt	Technique to adapt	Object to adapt
Out of CPU resource	Imprecise computation	Task parameters (e.g. period and execution time)
	Elastic task model	
	Hardware techniques (e.g. DVFS)	Hardware parameters
Out of network resource	FTT protocol	Network bandwidth
Fault tolerance/ Unbalanced load distribution	Migration technique	Task, HW/SW component, or even application
	Redundancy+dynamic HW/SW component composition	HW/SW component
New position of mobile nodes	Routing configuration	Routing table
MCR	Mode switch protocol	User-defined parameters for each mode

- DVFS: Dynamic Voltage/Frequency Scaling
- FTT: Flexible Time/Triggered
- MCR: Mode Change Request

Table 2.1: Representative adaptive techniques

2.1 Dynamic CPU resource re-allocation

One of the most important system resources is the CPU cycle. It is the predominant obligation for the scheduler to assign CPU cycles to all the running tasks timely and properly, assuring that no hard real-time tasks miss their deadlines and that a minimal number of soft real-time tasks miss their deadlines. Any change of the working environment will alter the resource demand of a few tasks, making dynamic resource allocation necessary. Although mechanisms dealing with dynamic CPU resource allocation are quite flexible and can differ a lot from each other, most of them fall into one of the two following categories:

- The first one is to transfer the resource released by idle tasks to those tasks currently requiring extra resource. For instance, consider an automotive system, in which the ABS (Anti-lock Braking System) and cruise control subsystems cannot be active at the same time. Upon an environment change, idle tasks should release some resource which can supplement other active tasks on the verge of running out of their own resource shares.

- The second one is graceful degradation of QoS, used when the first mechanism fails to produce a satisfying result, i.e. some tasks still cannot get enough resource even after the resource re-allocation. Many applications are associated with some kind of QoS level that can be adjusted flexibly within a certain range. Therefore, a graceful degradation of QoS level without jeopardizing the entire application is indeed an efficient way to survive from running out of resources.

A technical report [12] has proposed the Adaptive Resource Allocation (ARA) infrastructure, summarizing the general and common mechanism of different resource re-allocation approaches and providing metrics to evaluate their performance. We will discuss ARA in the next subsection, and then we will introduce two popular techniques respectively from the above two categories which have been implemented in real applications: imprecise computation and elastic task model.

2.1.1 The Adaptive Resource Allocation (ARA) infrastructure

The ARA infrastructure can be used to adjust the resource allocation, whenever there is a risk of failing to satisfy the application's timing constraints. This eliminates the need for "over-sizing" real-time systems to meet worst-case application needs.

The ARA infrastructure should integrate mechanisms for:

- Collecting information about application resource usage and resource availability
- Detecting significant variations in application resource usage
- Inferring the cause of observed variations and assessing the necessity of an automatic adjustment of the resource usage
- Making decisions about automatic adjustments and resource allocation
- Notifying the application about significant changes in its resource usage
- Notifying the application and resource providers about changes in resource allocation
- Assisting them in the enactment of these changes

In order to demonstrate how ARA works in real-time applications, let's consider a radar system as an example [11]. In Figure 2.1, *Detection*,*Track Init* (Track initiation) and *Track Identif* (Track identification) are computationally intensive tasks suited for parallel implementation. Over time, their processing and communication needs vary with the number and characteristics of the input data (e.g. the number, amplitude and direction of dwells). Besides, the computation is driven by several event streams: (1) the input from the radar, (2) the input from the missile tracking device, and (3) the missile control requirements. Timing constraints concern event rates and processing latencies. For instance, the required rate of the radar input is 1500Hz, and the required missile control rate is 4Hz. Latency constraints are: a 0.2 second-bound between the detection of a potential missile (*Detect*) and activating the *Search Control* and a 0.5 second-bound on the execution of *Engage*. Given the nature of their computation, the aforementioned three tasks can adapt by changing their internal levels of parallelism. Therefore, the timing constraints can remain unaffected by an increase in the computation requirements if a new thread is started for the task on another processor. A typical example is when the system is faced with a new set of spurious tracks, the computation requirements of *Track Init* increase rapidly. However, at the same time, the requirements of *Track Identif* and *Engage* remain stable or might even decrease because no new task is produced by *Track Init* for a while. Thus as long as the load of *Track Identif* and *Engage* is low enough to avoid violation of their own timing constraints, ARA is capable of transferring their idle resources to *Track Init*, whose temporal load increase might be overcome then.

The performance of ARA is determined both by the appropriateness of its resource allocation decisions and by the delay with which it responds to unexpected changes. A set of metrics can be considered as the criteria to evaluate the performance of ARA quantitatively (See Figure 2.2):

- Reaction time: The interval between the occurrence of a critical variation and the completion of the correcting re-allocation enactment.

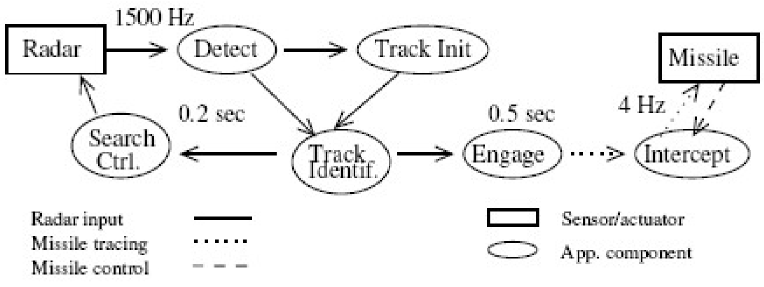

Figure 2.1: Radar application

- Recovery time: The interval between enactment completion and the restoration of an acceptable performance level.
- Performance laxity: The difference between the acceptable upper bound of the required performance and the steady state performance after re-allocation.

Recovery time and performance laxity denote the quality of resource re-allocation of ARA, while reaction time denotes how fast ARA is to respond to a change and make a decision. According to the metrics above, high performance can be implied by short reaction time, short recovery time and larger performance laxity. However, a tradeoff exists between reaction time and performance laxity. Shorter reaction time often fails to achieve the optimal solution to resource re-allocation, whereas optimality is inevitably obtained at the sacrifice of longer reaction time due to the considerable overhead. Even though optimality is a prominent goal deserving great efforts, it also increases the likelihood of failing to satisfy the application's timing constraints. It has been proved [12] that sometimes prompt reactivity is even more important than optimality. This tradeoff must be carefully balanced. For example, in the radar system mentioned above with strict timing constraints, it must be guaranteed that a successful resource re-allocation is completed in time. The optimal resource re-allocation failing to meet these timing constraints should not be expected here.

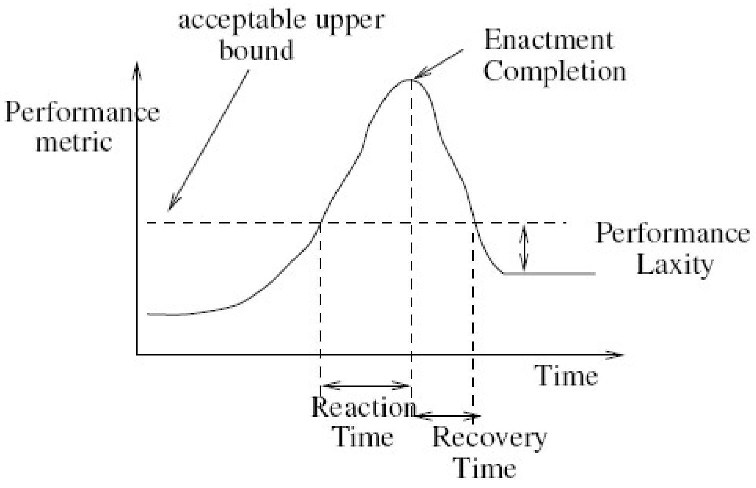

Figure 2.2: Metrics for ARA performance

2.1.2 Imprecise computation

The imprecise computation technique [27] is a way to avoid timing faults during transient overloads with graceful degradation of QoS. A system based on this technique is called an imprecise system. The key idea is to divide a task into mandatory and optional parts. The mandatory part always has to be completed. Under normal operating conditions, the optional part is completed and produces a precise result. In contrast, under overloaded conditions, the optional part is either skipped or executed partially, producing an imprecise result. In some applications such as image processing and object tracking, timely imprecise results are quite preferred compared with late precise results. After all, fuzzy images and rough estimates of target locations are more acceptable than delayed clear images and delayed accurate target locations.

Imprecise computation distinguishes three types of tasks:

- Optional task satisfying *0/1 constraint*: It is either executed to completion before its deadline or skipped entirely. This type of task offers no flexibility in scheduling.

- *Monotone* task: It produces nondecreasing intermediate results throughout its execution. Hence it can be decomposed into a mandatory task and an optional task. We have the maximum flexibility in scheduling for monotone tasks because it is possible to dynamically decide how much of each optional task is scheduled. Underlying computational algorithms enabling monotone tasks are available in many domains, including numerical computation, statistical estimation and prediction, heuristic search, sorting and database query processing [26].

- *Multiple-version* task: This type of task has at least two versions: the primary version and alternate version(s). The primary version of each task produces a precise result yet with longer processing time. An alternate version has a shorter processing time but produces an imprecise result. When multiple versions are used, it is necessary to decide which version will be executed before the task starts. During a transient overload, when the primary version of each task cannot be completed before the deadline, the system can choose to schedule the alternate versions of some tasks. As a matter of fact, if we use M and O to express the mandatory and optional part respectively, we can consider the alternate version as a mandatory task and the primary version as a mandatory task plus an optional task:

$$\begin{aligned}&\text{Primary version:}M+O\\&\text{Alternative version:}M\end{aligned} \tag{2.1}$$

  The optional task is fully scheduled in the primary version and entirely skipped in the alternate version. Therefore, algorithms for scheduling tasks with the 0/1 constraints can also be used for two-version tasks.

The imprecise computation model can be easily built. Each task τ_i is decomposed into the mandatory task M_i and the optional task O_i. If we define m_i, o_i and C_i as the processing times of M_i, O_i and τ_i respectively, then $m_i + o_i = C_i$. The classical deterministic model and the traditional soft-real-time workload model will be both special cases of this imprecise computation model, $o_i = 0$ in the former case and $m_i = 0$ in the latter case.

In a more general sense, imprecise computation is also allowed to have multiple-version tasks without mandatory or optional parts. For instance, there may be two different algorithms realizing the same functionality. They have the same input and similar outputs and only one of them is selected to run. One of these two algorithms can produce a more precise result than the other, yet with more computation time. We call them *HP* (High precision) and *LP* (Low precision) respectively. If the schedulability is known beforehand, in normal conditions, *HP* is used for better results. whereas in overloaded conditions where *HP* may fail to be completed, *LP* is used to produce an imprecise but acceptable result. If the schedulability is unknown, for the sake of safety, *LP* should always be first executed because it is uncertain whether *HP* can be completed or not. If there is still room for *HP* after the completion of *LP*, *HP* can be executed for a precise result. However, there is a high risk that *HP* will be aborted before its completion because the total execution time is the sum of the computation time of *LP* and *HP*. Yet this is still the most reasonable solution in such situation. In comparison with the multiple-version task above, *HP* and *LP* are totally independent of each other. Therefore, they don't have the mandatory part in common. Table 2.2 summarizes the different options concerning multiple-version tasks.

	Independent multiple-version tasks	Interdependent multiple-version tasks
Schedulability known	*HP* *LP*	*M+O* *M*
Schedulability unknown	*LP(+HP)*	*M+O*

Table 2.2: Imprecise computation options concerning multiple-version tasks

2.1.3 Elastic task model

The assumption of fixed computation time (C) and period (T) of each task is reasonable for most real-time systems, nevertheless, this could be too restrictive for some applications. In multimedia systems, the time for coding/decoding each frame can vary significantly. Hence the worst execution time (WCET) of a task can be much bigger than its mean execution time. This can cause a CPU resource waste if C and T are both rigid. Besides, sometimes periodic tasks are required to be executed at different rates in different operating conditions. For instance, in a flight control system, the sampling rate of the altimeter could change with the altitude. The lower altitude, the higher sampling frequency. Likewise, when a robot is approaching to an obstacle, the acquisition rate of its sensors may need to be increased.

Elastic task model [9] considers each task as flexible as a spring with a given rigidity coefficient and length constraints so that periodic tasks can be executed at different rates. Usually, elastic task model assumes fixed computation time and only adjusts the task period (flexible computation time is the focus of imprecise computation). As a result, resource re-allocation is realized every time the change of task period(s) occurs. When the utilization of a task is compressed due to increased period, it releases the CPU resource of its own share to other tasks. In a formal way, each task τ_i can be characterized by five parameters: computation time C_i, a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$, and an elastic coefficient $e_i \geq 0$ which specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible configuration. Greater e_i implies a more elastic task.

Elastic task model plays a key role in the following scenarios:

- It provides a more general admission control mechanism. When a new task arrives leading to the unschedulable status of the system, the utilizations of other tasks can be reduced (by increasing their periods) to accept this new task.
- While suffering from an overloaded condition, the system can compress the utilization of less important tasks with graceful QoS degradation by expanding their periods, as long as the periods of those tasks are still below their maximum periods.
- As the overloaded condition goes back to normal, those tasks with compressed utilization can restore their nominal periods accordingly.

- Whenever a running task terminates, other tasks can increase their utilizations if possible. In particular, tasks with compressed utilization will approach their nominal periods.

From the description above, it is not difficult to note the advantages of elastic task model:

- It allows tasks to intentionally change their execution rate to provide different QoS levels.
- It can handle overloaded situations in a flexible way.
- It provides a simple and efficient method for controlling the QoS level of the system as a function of the current load.

2.2 Dynamic network resource re-allocation

While CPU and memory resources are the key concerns in centralized systems, network resource is of equivalent importance in distributed systems. Various factors such as package collision, external disturbance, and the irregular fluctuation of bandwidth can all lead to highly dynamic network condition. This problem is particularly common in wireless communication with limited bandwidth. Therefore, some dynamic network resource allocation mechanism is indispensable to achieve efficient communication among different nodes. Here we mainly introduce a generic communication paradigm named Flexible Time-Triggered (FTT) [2] [36], which is abstracted from two popular communication protocols: FTT-CAN [4] and FTT-Ethernet [37]. FTT supports dynamic QoS management and can meet the timing constraints of message passing without losing flexibility.

The FTT paradigm uses an asymmetric synchronous architecture, comprising one master node and several slave nodes. The master node is in charge of the management and coordination of the communication activities. Communication requirements, message scheduling policy, QoS management and online admission control are all localized in the single master node. And the scheduling decisions taken in the master node are broadcast to the network using a special periodic control message called Trigger Message (TM) that controls the behavior of slave nodes.

The FTT paradigm boasts a time-triggered pattern in that its communication uses Elementary Cycles (ECs), which are consecutive time-slots with fixed duration. As is depicted in Figure 2.3, the EC starts with the reception of the TM and all slave nodes are synchronized by the reception of this message without the support of any global clock. Each EC consists of two consecutive windows: synchronous window and asynchronous window. The synchronous window conveys the time-triggered traffic specified by the TM. Its length depends on the number and size of messages scheduled for the corresponding EC. Usually it has a maximum window size in order to guarantee a minimum bandwidth share for the asynchronous window. The asynchronous window conveys event-triggered traffic that is not resolved by the master node. Instead, the asynchronous traffic is handled by a best-effort policy. A minimum bandwidth for asynchronous traffic can be guaranteed so that real-time asynchronous messages can meet their deadlines in worst-case conditions.

Any guarantee of the FTT paradigm, either concerning timeliness or safety, relies on the communication requirements, which are stored in the Communication Requirements Database (CRDB)(In some other literatures [36] it is also called System Requirements Database-SRDB) of the master node. CRDB is a data structure containing the description of the message streams currently flowing in the system. For each message

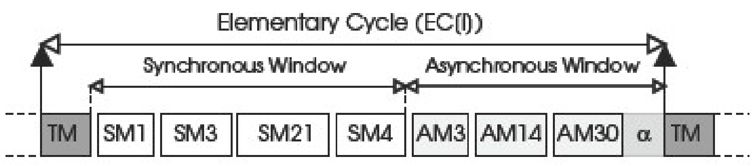

Figure 2.3: The Elementary Cycle structure

stream, the CRDB includes information such as message identification, group identification, data length, type, period or minimum inter-arrival time, relative phasing for periodic streams, timing constraints, safety constraints and a set of change attributes. CRDB supports the requirements verification upon change requests. It is dynamically scanned by a traffic scheduler (TS) so that any change in its structure can be detected at runtime. However, any change request must be handled by an online admission control before it gets accepted. If any change request would lead to an unfeasible message set, the dynamic QoS management is carried out to re-allocate the network resource. There are two typical kinds of dynamic QoS managements: using the priority-based QoS manager and using the elastic task model QoS manager. If the network resource is still insufficient after dynamic QoS management, the change has to be rejected and the CRDB remains unchanged. The overall structure of the FTT paradigm is presented in Figure 2.4, from which the relationship between the master node, slave nodes, CRDB and TS can be clearly observed.

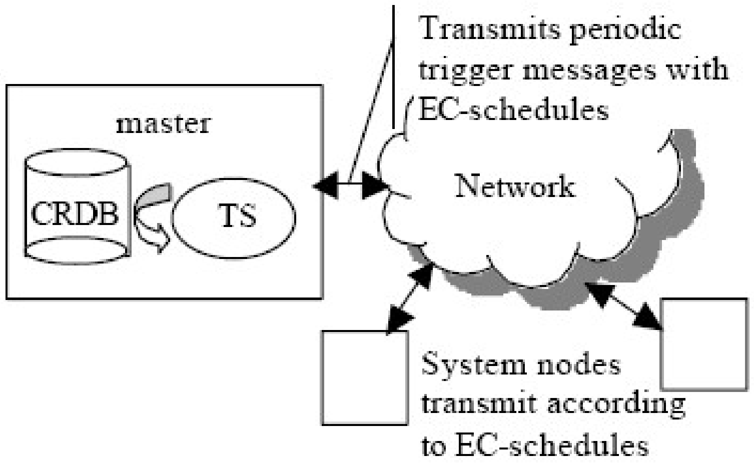
Transmits periodic trigger messages with EC-schedules
master
CRDB
TS
Network
System nodes transmit according to EC-schedules

Figure 2.4: The FTT paradigm with master-slave structure

What is potentially threatening the FTT paradigm is the fault-tolerance issue. Once the master node fails, no more TMs with EC schedules are issued and the communication will be terminated. However, this problem can be tackled by hardware redundancy, such as replication with a few master node backups.

2.3 Fault tolerance

Fault tolerance is vital for safety-critical systems. Since adaptivity makes a system even more complex, fault tolerance must be concerned even more carefully [22] in an AES. One of the most recommended solutions to fault tolerance is hardware and software redundancy. Once a HW/SW module fails to work normally, the system should switch to the backup modules immediately as if nothing wrong has happened. Hardware fault-tolerant systems tend to use a multiplex or multiplicated approach [17]. In multiplex systems, redundant components are always active and provide multiple processing paths whose results are then voted to derive a final result. In multiplicated systems, redundant components act as passive standbys that can be promoted to active status when a fault occurs. Later we will see that adaptive behaviors can be added to the multiplex approach. Unfortunately, redundancy, especially hardware redundancy, inevitably entails high cost. Hardware backup modules take up extra space and consume more energy while software backup modules raise the memory demand and make the software system more complex. Consequently, plenty of resource is wasted if no faults are found at runtime. Some adaptive techniques aim at achieving fault tolerance with dynamic reconfiguration, consuming as little resource as possible. For instance, migration and hardware reconfiguration in multiplex systems are two representative alternatives.

2.3.1 Migration technique

The major idea of migration techniques is to migrate a running object from a faulty source to another suitable location. Basically, four questions need to be answered: When is migration triggered? Which object is to be migrated? Where will it be migrated? Who makes the decision? These four questions have been well answered in [16]. The flexibility of migration technique is attributed to two properties: One is that the running object to be migrated can be a task, a process, or even an entire application. Hence the migration granularity is tunable. This granularity will impact the migration result or even the system scalability. The other is that there might exist quite a few options concerning the target location of the running object to be migrated [7]. Maybe the new location of the running object still resides in the same node in a distributed system, or it could belong to another remote node, depending on the current situation. This does not require all devices to have the same structure. A higher level abstraction will hide the hardware details so that the running object is supported by different operating systems or even different hardware platforms. A delicate migration design will enhance the overall performance, take good advantage of limited resources, and make the system more robust.

Apart from the purpose of fault tolerance, migration technique is also effective on load balancing [39]. If some nodes are temporarily in an overloaded status, some running objects on them can be migrated to other idle nodes, even if no fault occurs. Since migration is dynamically executed at runtime, each migration corresponds to a round of dynamic reconfiguration, both for the source node and the target node.

Migration technique is more suitable for permanent faults [43]. For transient faults, the migration overhead becomes non-trivial, and instead, other solutions such as checkpointing with rollback recovery might be more suitable.

2.3.2 Hardware reconfiguration in multiplex systems

In multiplex systems, redundant components work simultaneously to ensure accurate result and fault tolerance. A typical example of the multiplex approach is the sensor fusion technique where sensors are redundant hardware components. The sensor fusion technique uses a set of sensors instead of a single sensor due to the fact that multiple sensor values bring higher accuracy. Not only can the performance be improved like this, but also fault tolerance is realized because the failure of one sensor will not ruin the result. Nevertheless, each sensor contributes its own share to the final result. Faulty sensors may negatively influence the expected accuracy. An AES is supposed to isolate those faulty sensors for the sake of accuracy and energy consumption. The system should be capable of detecting faulty sensors quickly from their exceptional outputs.

More importantly, the system should have a clear overview of the availability status of each sensor. Once a faulty sensor is detected, a reconfiguration should be made to make sure only non-faulty sensors are working [28]. Of course, these sensors can be extended to any hardware components with redundancy in other similar multiplex systems.

Chapter 3

Mode switch analysis

Mode switch has a tight relationship with dynamic reconfiguration and it is absolutely one of the top concerns of an AES. For adaptive embedded real-time systems, it is quite necessary to analyze the mode switch mechanism. In this section, we shall delve into the fundamental problems associated with mode switch.

3.1 Mode switch and dynamic reconfiguration

From the introduction in the previous section, it is self-evident that dynamic reconfiguration plays an essential role in achieving adaptivity. Reconfiguration covers a wide range of possible behaviors, however, it can be mainly characterized by mode switch (or mode change), which concurrently takes place with reconfiguration in most cases. Reversely, reconfiguration must be done during a mode switch. For instance, an aircraft control system usually supports taking off mode, flight mode and landing mode [18]. The transition between different modes is realized via reconfiguration.

In the real-time systems domain, each mode is distinguished by a set of tasks, a particular scheduling policy, and many other factors. One example is a smart phone which is able to play audio/video streams as well as make and receive phone calls. If an incoming call is received while a video application is running, a mode switch request could be triggered and the priorities of different tasks may be changed, or certain tasks may be inactivated or even terminated [40].

We already know that dynamic reconfiguration must be a consequence of an event or a request that triggers the adaptive behavior. We call this triggering Mode Change Request (MCR) if mode switch is also required during reconfiguration [41]. For example, when an alarm sensor indicates an abnormal value, a related monitoring task will decide to issue an MCR so that the system transits into alarm mode from normal mode. An MCR can be either time-triggered or event-triggered and it is common that both time-triggered and event-triggered MCRs exist in a multi-mode system.

3.2 Mode switch problems

Although mode definition and mode switch are both dependent upon specific applications, any kind of mode switch can be represented by one or some of the following scenarios:

- The deletion of one or more existing tasks.
- The arrival of one or more new tasks.

- The parameter change of one or more existing tasks, such as period and worst-case execution time.
- The change of scheduling policy.

Since the change of scheduling policy is less common, we will here assume that the same scheduling policy is implemented during a mode switch, which will then boil down to the change of a given task set. In order to clearly illustrate how the task set changes during a mode switch, Pedro and Burns [38] propose five classes of tasks (see Figure 3.1):

- Old mode completed task: It is released ahead of the arrival of an MCR. If an MCR arrives during its execution, this task should continue till its completion. One typical example is a task with safety-critical functionalities. The system still delivers the old functionality during a mode switch so as to maintain a safe condition.
- Old mode aborted task: It is also released ahead of the arrival of an MCR. If an MCR arrives during its execution, it is allowed to be aborted immediately. Sometimes it may incur interference over the remaining tasks and impair the performance if it is not aborted in time. Usually this type of task is not safety-critical but related to QoS. Its abortion may lead to QoS degradation, yet without any severe consequence.
- Wholly new mode task: A task containing new added functionalities to the system. It can only be introduced after an MCR, either with or without any offset.
- Changed new mode task: This type of task represents the changed functionality of a system. It is always preceded by a corresponding old mode task whose parameters are modified during the mode switch. Sometimes an offset is added to let the old mode task run to completion.
- Unchanged new mode task: It is also preceded by a corresponding old mode task, but it is exactly the same as its preceding old mode task. An offset could be necessary for the sake of schedulability.

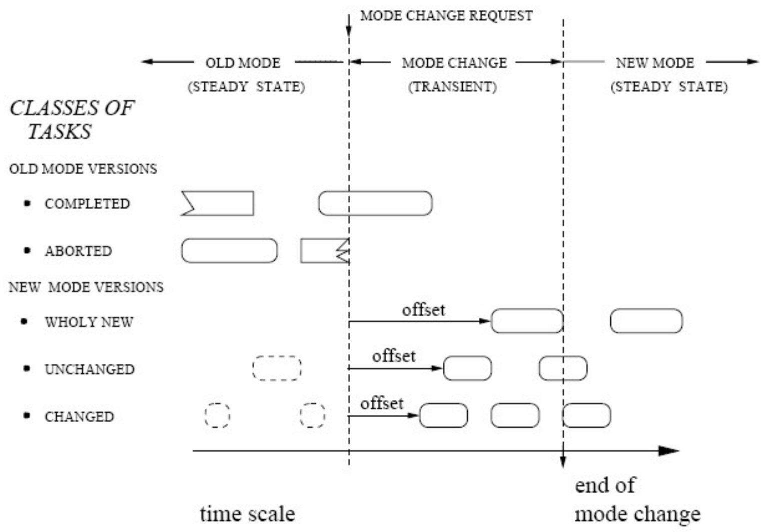

Figure 3.1: Mode switch and task classification

The above-mentioned task classification is extremely helpful for the design of mode switch protocols and schedulability analysis during mode switch.

3.3 Mode switch protocol

A mode switch protocol defines rules for the deletion or modification of existing tasks and the addition of new tasks. A number of mode switch protocols have already been proposed in existing literatures. Despite the variety of those different protocols, they mainly differ in three aspects. First, regarding unchanged tasks, there are two types of protocols [41]:

- Protocols with periodicity: Unchanged tasks preserve their activation pace and are not allowed to be delayed by any mode switch.
- Protocols without periodicity: An offset may be added to some unchanged tasks during the mode switch, leading to the loss of their periodicity. Sometimes this is necessary to guarantee schedulability and data consistency.

Apparently, the fundamental difference between the two types of protocols above lies in the offset of unchanged tasks. As a matter of fact, this offset can even be extended to all new mode tasks. The introduction of offset to new mode tasks is a very simple and effective approach to increase schedulability during mode switch. When the release of a new mode task is delayed by an offset, some old mode completed tasks will have a higher chance to be completed and the interference between old and new mode tasks can also be decreased or even eliminated in this way. However, offsets incur long latency of the mode switch, thus offsets must be chosen carefully to minimize this negative effect. It is a key design issue of the mode switch protocol to specify these offsets of the unchanged old mode tasks and different new mode tasks. More details can be found in [41].

Second, regarding both old and new mode tasks, we get the following two types of protocols [41]:

- Synchronous protocols: New mode tasks are never released until all the old mode tasks have completed their last activation. This type of protocol does not require any schedulability analysis during the mode switch because the mode switch will not result in overloaded condition. Nevertheless, the potential problem is that the mode switch process may be too long.
- Asynchronous protocols: Both old and new mode tasks are allowed to be executed during the mode switch. This protocol can shorten the time required for the completion of a mode switch, yet additional schedulability analysis is required because the workload of the system will possibly be higher than stable states during the mode switch when both old and new mode tasks are executed.

The last concern is when a mode switch is supposed to take place. An MCR is essentially a sporadic event that can only be served while the system is running in the steady state. In [41], Idle Time Protocol was introduced, specifying that mode switch can only happen at an idle instant. This idea is simple and easy to be applied. No additional schedulability is required. However, the poor promptness is a severe disadvantage for a high-utilization task set. In contrast, major period has been mentioned in [38] as the least common multiple of the task periods. Mode switch is performed at the end of this major period. Yet, long response is still a potential problem. Better solutions need to be further explored.

Based on these three factors (periodicity; synchronous or asynchronous; mode switch instant), a variety of combinations of them are allowed to make an appropriate mode switch protocol for a given application.

3.4 Schedulability concerns during the mode switch

A mode switch may increase or decrease the processor's utilization. After the mode switch, if one or more new mode tasks arrive, or the execution time of a task is increased, or the period of a task is decreased, the system schedulable in the old operational mode may become unschedulable. Furthermore, while asynchronous mode switch protocols are used, due to the co-existence of both old and new mode tasks, mode switch may lead to transient overload. Consequently, a system that is schedulable both in its old and new operational modes may not be schedulable during the mode switch. Hence additional schedulability analysis during the mode switch is required to guarantee that the timing constraints of all old and new mode tasks are met.

An exact schedulability analysis approach was described in [38], as the worst-case response time (WCRT) of each old or new mode task is calculated considering all possible interferences. Figure 3.2 depicts a scenario where a low priority old mode completed task is preempted by three different types of tasks, i.e. three sources of worst-case interference:

- Interference from higher priority old mode completed tasks
- Interference from higher priority old mode aborted tasks
- Interference from higher priority new mode tasks

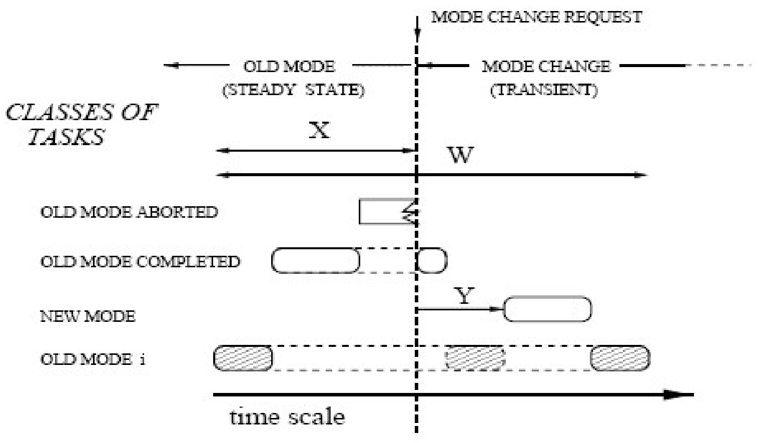

Figure 3.2: WCRT of an old mode task *i*

Similarly, Figure 3.3 depicts a scenario where a low priority new mode task is preempted by three sources of worst-case interference:

- Interference from higher priority old mode completed tasks
- Interference from higher priority new mode tasks
- The computation time of its preceding old mode completed task, which is released just upon an MCR as the worst case. If this task is associated with an offset, it should be also taken into account.

Once the WCRT of each task is calculated, considering all possible interference resources, schedulability can be determined by comparing the WCRT of each task with its own deadline. This is exactly the same as traditional response time analysis. If any task misses its deadline after the offline analysis, proper offsets can be added to one or more new mode tasks to ensure schedulability and minimize mode switch latency.

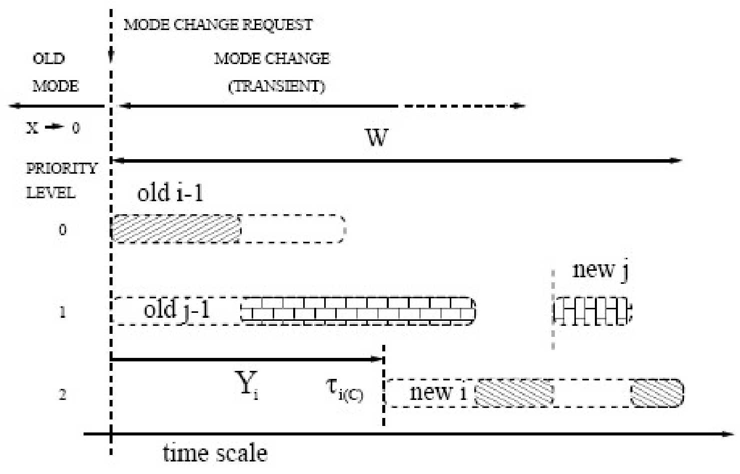

Figure 3.3: WCRT of a new mode task i

3.5 Other related issues

The analysis of mode switch becomes more complex whilst shared resource is considered. The most common way to achieve mutual exclusion is the usage of semaphore, which permits only one task to access the shared resource at a time. Semaphore brings blocking factors into the schedulability analysis in single mode systems, and mode switch can be treated in the same way for multi-mode systems. For example, in [45], the schedulability analysis during mode switch is extended by including blocking factors with the assumption that semaphores are locked and unlocked according to the priority ceiling protocol (PCP).

Besides, mode switch becomes more interesting in distributed systems due to the inter-communication problem. In order to communicate with other systems, each processor can have a "transmit" task and a "receive" task. Suppose TDMA is applied in a distributed network. The periods and computation times of the "transmit" task and the "receive" task may change after mode switch. As a consequence, probably the TDMA slots need to be reconfigured.

Another problem of mode switch in distributed systems is consistency, i.e. how to synchronize the MCRs. For instance, some functionality can only be achieved by the synchronization of two tasks residing in different processors. When the same MCR is delivered to these two tasks, they may not receive it simultaneously. Suppose the MCR occurs before the release of Task A at Processor 1 so that the new version of Task A is running. However, the same MCR occurs after the release of Task B at Processor 2 so that the old version of Task B is running. The synchronization of Task A and B will not be desired due to their inconsistent versions. One typical method to avoid this consistency problem is to introduce global time.

Chapter 4

Application modeling

There exist a vast range of applications regarding AESs. In this section, we enumerate a few exemplary scenarios of those applications and build generic models for them. Since these models are built at an abstraction level, each of them is able to present the key behaviors of numerous different applications.

4.1 Operational mode switch

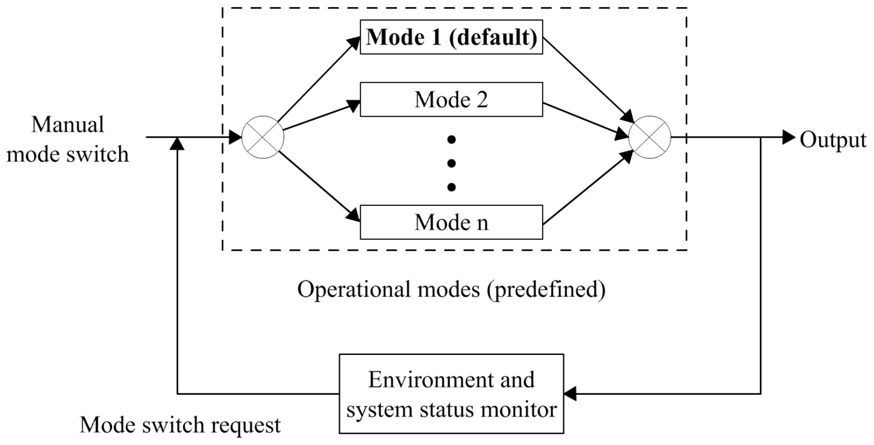

Figure 4.1: Operational mode switch

This type of adaptivity has been widely developed in modern embedded real-time systems. During system design, the most common operational conditions can be considered in advance. Since different configurations are required in different operational modes [13] [33] [8] [22], one typical straightforward way is to predefine all possible operational modes at design time. At runtime, the operational condition, i.e. the ambient environment and the system status are monitored by different sensors. If the system encounters a severe operational condition change or receives a mode change request from the operator, the current operational mode should be switched to another predefined one which is the most suitable for the new condition. This process is shown in Figure 4.1. As a matter of fact, this is still not flexible enough in that predefined modes requires too much memory and substantial offline work is involved. A future tendency is to move this offline work (predefining operational modes) to dynamic reconfiguration at runtime. That is to say, we expect that the system can reconfigure itself into a new operational mode which is not predefined but automatically generated according to the new condition.

If we compare Figure 4.1 with the overall system architecture in Figure 1.2, we may notice some commonalities. The "Environment and system status monitor" belongs to the "Monitors" component of the overall architecture. The set of predefined operational modes are actually the "Adaptation Objects". The mode switch corresponds to the "Dynamic reconfiguration".

4.2 Migration for fault tolerance and load balancing

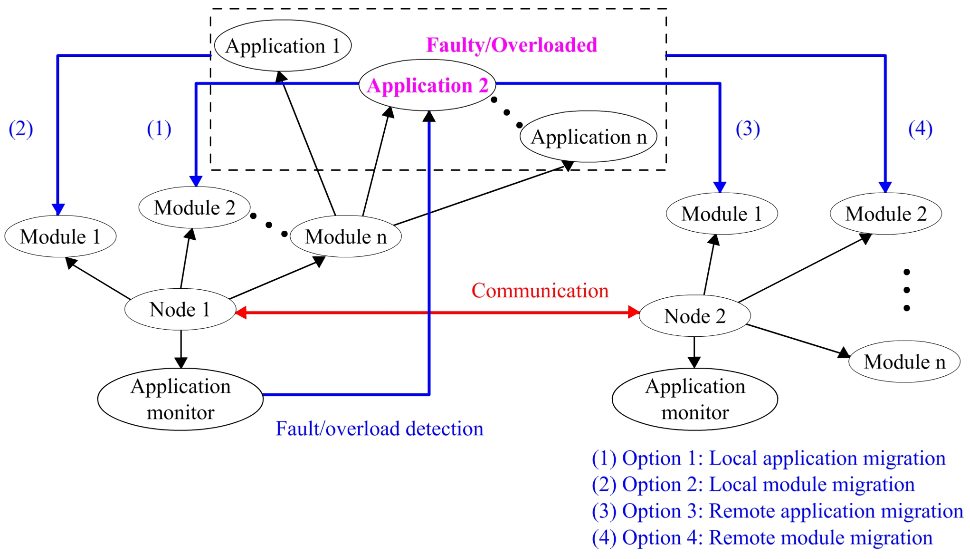

Figure 4.2: Migration for fault tolerance and load balancing in distributed systems

In Section 2.3.1, we introduced migration techniques, which can be used for fault tolerance and load balancing in an AES. Here we create a general application model demonstrating migration technique in adaptive distributed systems. As is illustrated in Figure 4.2, each single node in a distributed system consists of several functional modules. Several applications can run in each single module. All applications are monitored for the purpose of fault or overload detection. If one application is detected to be faulty or overloaded, migration techniques can be applied to solve this problem. In distributed systems, there are four options:

- This single application is migrated from the current module to another module in the same node.
- Several applications or even all the applications in the current module are migrated to another module in the same node.
- This single application is migrated from the current module to another module in another node.
- Several applications or even all the applications in the current module are migrated to another module in another node.

It is vital to note that the latter two cases will lead to more communication overhead [7]. And usually it is preferred to migrate only faulty or overloaded applications. However, in particular cases, the costly migration must be considered. For instance, when a faulty application inevitably affects other related applications, all of them should be migrated.

Moreover, The remote migration to another node must be considered when the local migration becomes unfeasible due to some reason. In addition, migration techniques are also applicable to centralized systems, however, only the first two cases out of the four options above may happen.

Actually, Figure 4.2 maps Figure 1.2 and 1.3 well while the application monitor corresponds to the "Monitors" in Figure 1.2 and the communication between Node 1 and Node 2 corresponds to the communication medium in Figure 1.3.

4.3 Multimedia communication with stable streaming

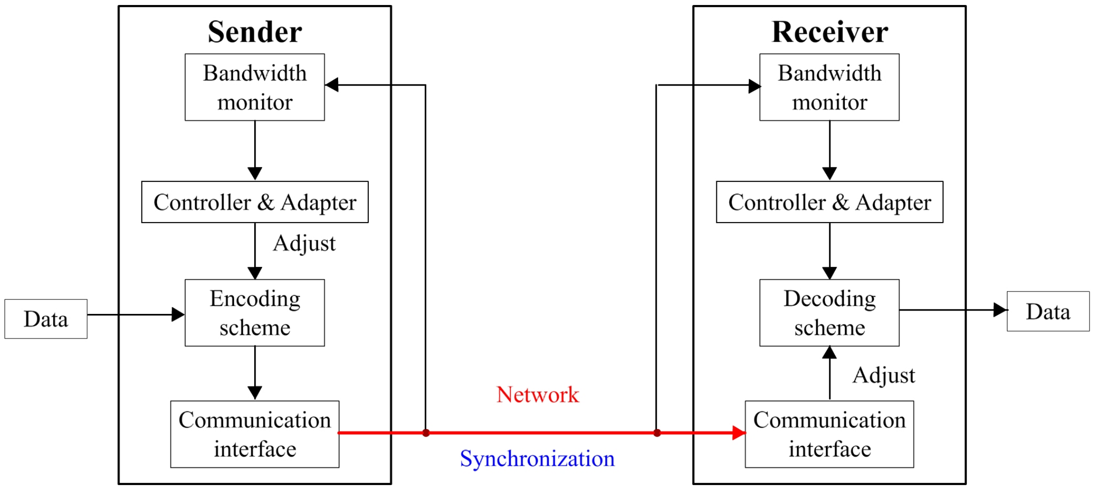

Figure 4.3: Multimedia communication with stable streaming

Multimedia communication often needs to adopt some sort of adaptive encoding/decoding schemes in variable network conditions. If only static encoding/decoding algorithms are implemented, the changing network condition will cause unstable streaming rate during the communication between the sender and the receiver. However, an adaptive system can dynamically adjust the encoding/decoding schemes according to the current network condition [14], as is demonstrated in Figure 4.3. The bandwidth of the network can be monitored by the sender. No matter what encoding/decoding scheme we use, it should be able to satisfy different QoS levels. To keep a stable streaming rate, the decreasing bandwidth can be compensated by the degraded QoS, such as the lower quality of pictures, videos or audios. This requires flexible encoding/decoding schemes. Besides, the decoding process on the receiver side should be notified and synchronized by the sender in time. Otherwise, the receiver won't be able to successfully decode the correct raw data. For the receiver, this synchronization can be notified as an internal event from the communication interface. However, for the sender, the bandwidth change is an external event.

Figure 4.3 has the same pattern as the architecture in Figure 1.2 and 1.3. The "Bandwidth monitor" plays the role of the "Monitors" component. The multimedia communication diagram also contains "Controller & adapter" and "Communication interface". The "Encoding/Decoding scheme" is the "Adaptation Object".

4.4 Adaptive resource management and QoS degradation

In resource limited systems, it is important to allocate resources to different applications appropriately. There is no doubt that in many cases fixed resource allocation will waste too many resources. For instance, the required CPU cycles and memory of a task vary from time to time. In particular, when a system becomes more complex, it will be fairly common that some applications suffer from running out of resource. This phenomenon should be monitored and some kind of adaptive resource management mechanism [12] [44] is needed to reallocate resource for related applications dynamically. We prefer to keep the desired QoS level during the resource reallocation. However, when the re-allocated resource is still not sufficient for some applications, the system has to degrade the QoS level of less important applications so that the system won't crash. This is called graceful QoS degradation. Figure 4.4 is an abstract expression of the workflow of adaptive resource management and QoS degradation. Currently, graceful QoS degradation technique has already been widely implemented in many areas [21] [1] [20], among which the most representative one is multimedia application.

Figure 4.4 matches the architecture in Figure 1.2 quite well. The "Resource monitor" functions as the "Monitors" component. The "Resource management" can be considered to be the "Controller & adapter". Resource and application are the "Adaptation Object". The dynamic reconfiguration can be realized by two actions: Reallocate resource and degrade QoS.

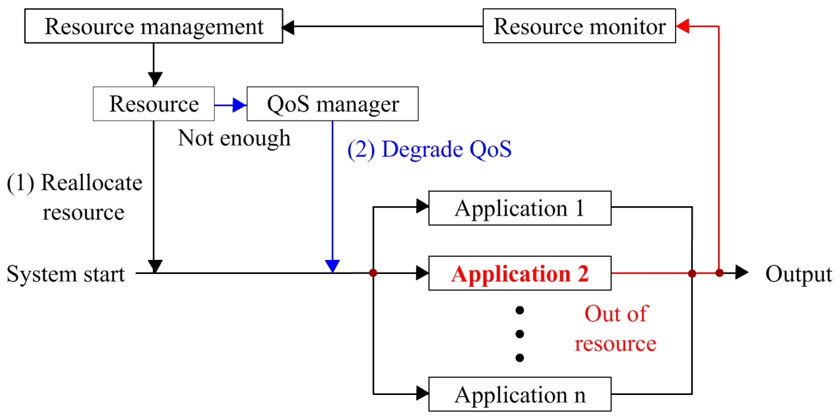

Figure 4.4: Adaptive resource management and QoS degradation

4.5 Dynamic HW/SW module composition

HW/SW redundancy is an extremely common way to realize fault tolerance. Here we mainly discuss how hardware redundancy contributes to adaptive fault tolerance. At runtime, the status of each hardware module (or component) should be monitored. If one hardware module is broken due to some unexpected reason, the system should immediately do a hardware reconfiguration by replacing the broken hardware module by backups [10]. This scenario is expressed in Figure 4.5. A special case is that in multiplex systems, multiple identical hardware modules can be used simultaneously. For instance, to achieve both high accuracy and fault tolerance, we would like to obtain the average value of ten temperature sensors using sensor fusion technique [28]. If one sensor becomes faulty, it won't jeopardize the system, yet it will lower the overall accuracy if it is not isolated from the system. We should make sure that only non-faulty sensors are in operation. Therefore, hardware module composition is taken every time a faulty sensor is detected.

Figure 4.5 is consistent with the architecture in Figure 1.2. The "HW status monitor" and "Controller & adapter" is in line with the MCA paradigm that derives from the conceptual view in Figure 1.1. The "HW module composition" together with "HW module availability change" is one type of dynamic reconfiguration. The hardware modules are "Adaptation Objects".

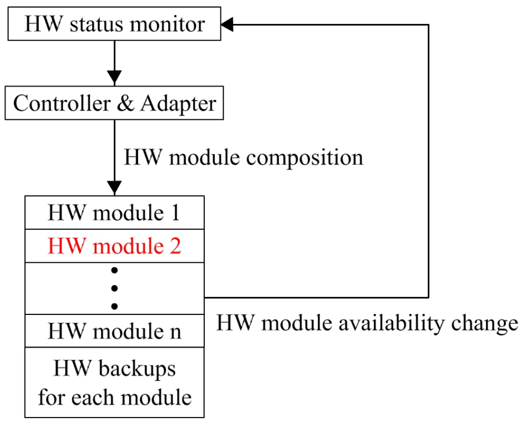

Figure 4.5: Dynamic HW/SW component composition

Chapter 5

The modeling of AESs and case study

AESs cover a wide range of application fields, such as avionics, automobile, multimedia and robotics. Despite the variety of those applications, the modeling of an AES can extract their common features and represent them at a higher level so that a cluster of applications can be described by one generic model. This chapter is about the modeling of AESs, based on two case studies: a smart phone and an object-tracking robot. The two models are designed and developed in UPPAAL, a tool for modeling, validation and verification of real-time systems. In the following sections, we first give a basic introduction of UPPAAL and then explain our two models in detail.

5.1 The modeling tool: UPPAAL

UPPAAL is a popular academic modeling tool for real-time systems. A complete UPPAAL model consists of a declaration of global variables and a few templates. Each template can be regarded as a component of the system, represented by an automaton (or a timed automaton sometimes) and a declaration of its own local variables.

A template may contain one or more input parameters, whose combination gives rise to multiple instances of the same template. For instance, in our smart phone model, the CPU and network resources have a lot of similar features, thus sharing the same Resource template with one input parameter as the resource index. Resource(0) is the CPU resource instance while Resource(1) is the network resource instance. And in our robot model, three sensors share the same Sensor template, distinguished by the input parameter *const sensorIndex s_id*.

An automaton functions as a state-machine with locations (states) and edges (transitions). A UPPAAL model can simulate a system's behavior by running all its automata concurrently. The model correctness is validated by verifying all kinds of properties. The satisfaction of a group of well-designed properties makes a model more reliable.

For real-time systems with timing constraints, UPPAAL uses timed automata featured by clocks. A clock can add a timing guard to a transition which cannot be taken until this timing guard is satisfied. A clock also allows a location to have its own invariant, which forces the automaton to change state after a specified interval. Both our two models are based on timed automata due to the existence of some real-time behaviors.

Since this report is not a manual of UPPAAL, we would like to skip other details and save more words for our two models. The thorough introduction of UPPAAL can be found in [6].

5.2 Case study 1: The UPPAAL modeling of a smart phone

In this section, we are going to present the UPPAAL modeling of an example of an AES, a smart phone. Its main adaptivity is dynamic resource re-allocation and graceful QoS degradation, which has been mentioned in Section 4.4. Next we shall focus on the UPPAAL model design of the smart phone, its main functionalities, resource allocation mechanisms and other related issues. The model fits the smart phone quite well, moreover, it is generic enough to simulate the behavior of other similar AESs.

5.2.1 Functionalities of the smart phone

Modern cellphones are becoming more and more versatile with respect to their functionalities. To simplify the model, we only consider three typical functions which may be associated with adaptive behaviors. Each function corresponds to an application, whose detail information is listed in Table 5.1. Each application requires a particular mix of resources which can be categorized in many types, e.g. CPU cycles, network bandwidth, memory and power. Here we will mainly consider the CPU and network resources, i.e. CPU cycles and network bandwidth. The smart phone supports three typical functions which are illustrated in Table 5.1:

- Phone call: The smart phone should be able to make and receive phone calls. The CPU and bandwidth consumption of a phone call is fixed. Since the phone call is the fundamental function of a smart phone, it should be assigned the highest priority and it is non-stoppable, meaning that it cannot be interrupted by other applications.
- Video chatting (Online): The smart phone has a camera which can be used for online video chatting or recording. Its resource consumption is also fixed. The video chatting is less urgent than the phone call, thus it has lower priority and it is stoppable, i.e. it can be interrupted by other applications.
- Multimedia online: This function is completely for entertainment. The user is able to watch online video and audio by launching the multimedia application. The multimedia application consumes much resource mainly due to the large amount of video and audio streams. However, its resource consumption is adjustable as it has three QoS levels regarding CPU consumption and two QoS levels regarding bandwidth consumption. In Table 5.1, "Level A|B" means Level A for CPU consumption and Level B for bandwidth consumption and the default levels are "Level 3|2". The higher level, the better video and audio quality. When the system is overloaded, QoS degradation is allowed.

Application	CPU consumption	Bandwidth consumption	Priority	Stoppable
Phone call	2	1	1	No
Video chatting (Online)	2	1	2	Yes
Multimedia online-Level 3|2	4	3	3	Yes
Multimedia online-Level 2|1	3	2		
Multimedia online-Level 1|1	2	2		

Table 5.1: The application description of the smart phone example

Resource	CPU	Bandwidth
Level 1	5	3
Level 2	7	4
Level 3	9	5

Table 5.2: The available CPU and network resources at different QoS levels of the smart phone example

One typical feature of an AES is that some types of resources may change dynamically. For example, the bandwidth of a wireless communication may be unstable. The CPU resource is relatively more stable, however, the CPU resource could have different levels for different operating modes. In particular, some advanced hardware is able to adjust voltage and frequency, leading to the changing availability of CPU resource. In this smart phone example, we define three levels for the CPU and network resources respectively. Table 5.2 lists the total available CPU and network resources at different levels, with Level 2 as the default level for both resources. Please note that the values in both Table 5.1 and 5.2 are conceptual. They are not absolute values but relative to each other. Maybe they seem to make no sense in a real smart phone, yet these values are properly defined to demonstrate all kinds of interesting scenarios. We could have specified that there is sufficient CPU and network resource even when all three applications are active, with the multimedia application running at the top QoS level. This is not what we are interested in because the resource allocation mechanism to deal with tradeoffs becomes trivial if the resource is always sufficient.

5.2.2 Resource allocation mechanism and scheduling policy

Any one or more applications of this smart phone could run at any time. The multiple multimedia QoS levels and different CPU and network resource levels bring much flexibility to the system, yet giving rise to much unpredictability at the same time. An appropriate resource allocation mechanism and scheduling policy is required to bring the maximal benefit for the system. This is independent of the modeling of AES, because it can be designed separately. In our smart phone model, the resource allocation mechanism and scheduling policy is guided by the following principles:

- The phone call application has the highest priority and it shouldn't be interrupted by any other applications.
- When the system load is not high and there are sufficient CPU and network resources, the multimedia application should run at the top QoS level. If an overloaded condition occurs while the multimedia application is still running, its QoS degradation is first considered before the termination of any application by force. Likewise, when the system restores the normal condition from an overloaded condition, the QoS level of the multimedia application should be raised accordingly to make full use of the resources.
- The admission control of a new application is based on the sufficiency of both CPU and network resources. The new application is only accepted directly if both resources are sufficient. Otherwise, even if one type of resource is insufficient, the new application cannot be accepted without affecting other running applications. To handle this issue, first the possibility of QoS degradation is checked. If the currently running applications are not associated with QoS levels, or the resources are still insufficient even after the maximal possible QoS degradation, we must

terminate applications by force. The termination sequence starts from low priority and stoppable running applications.

- The CPU level change is requested by the user. If the CPU level is raised from a lower level, more CPU resource will be provided. This request is surely accepted and what can be done further is that the QoS level can also be raised if possible. Conversely, if the CPU level is lowered from a higher level, a potential problem is that the currently occupied CPU resource by running applications may be even more than the total available CPU resource at the new level. This problem might be solved by QoS degradation, however, if there is no chance for QoS degradation or QoS degradation fails to release enough resource, the CPU level change request by the user has to be rejected.
- The network condition cannot be decided by the user, therefore the bandwidth may change at any time. It may automatically fall into a lower level or reach a higher level. Different from the CPU change request, the bandwidth change request must be accepted because it is out of the user's control. When the bandwidth level is raised, QoS upgrade possibility will be checked. When the bandwidth level is lowered, leading to the overloaded condition, QoS degradation possibility is checked first. If QoS degradation is not feasible or fails to release enough resource, some applications may need to be terminated. In the worst case, even non-stoppable applications should be forced to terminate to adapt the deteriorated network condition.

5.2.3 The UPPAAL model of the smart phone

In this subsection, we shall delve into the modeling of the smart phone example by UPPAAL. The resource allocation mechanism and scheduling policy explained in the previous subsection will be implemented in the model. In our smart phone model, five templates are involved to simulate different parts of the system: user, application, resource, admission control and the main controller. First we give a basic introduction of the architecture of our UPPAAL model. Next, the key global variables and each template will be explained in sequence.

The model architecture

Our model consists of five components, user, application, resource, admission control and the main controller, with each component as one template. There are frequent interactions and tight relationship between these components. Figure 5.1 depicts the model architecture, showing how different components are connected with each other. "CPU resource" and "Network resource" in Figure 5.1 belong to the same "Resource" component. There are two main scenarios initiated by the user. The user can either start an application (marked in blue) or stop an application (marked in red). If an application is started, it consumes CPU and network resources, whose availability will be checked and then reported to the admission control. Then based on the availability reports, the admission control may accept the application directly, or refuse the application temporarily and let the main controller make the final decision. Particular scheduling policies are implemented in the main controller. In our model, it is allowed to terminate other running applications of less importance to increase the possibility to accept a new application. And the main controller is responsible to reject the new application if no more available resources can be released. If an application is stopped, both the CPU and network resources previously occupied by it will be released. The admission control does not have to know this event. However, the main controller should be informed

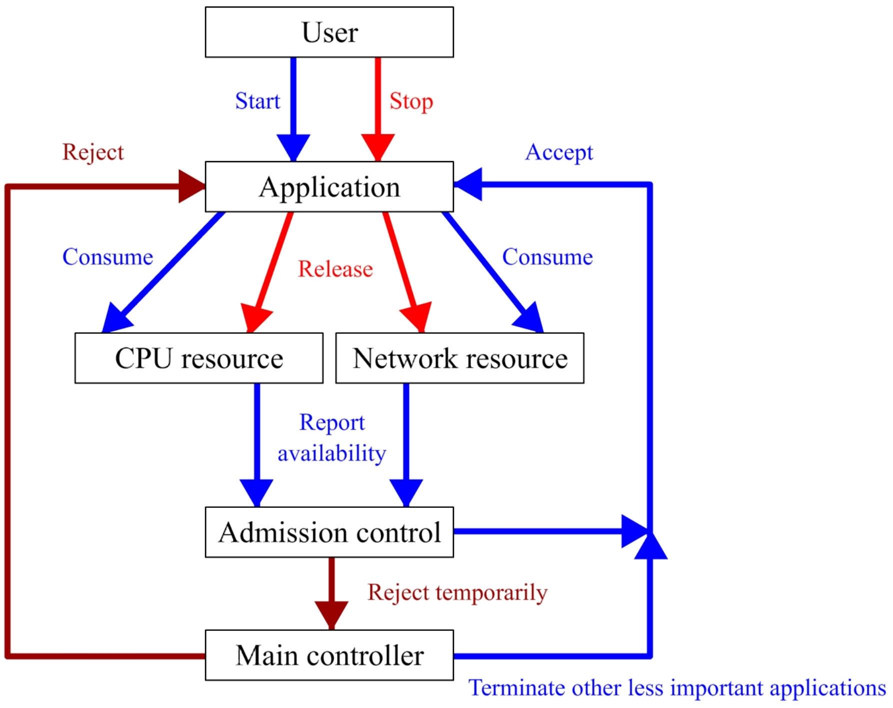

Figure 5.1: The architecture of the smart phone model

because it needs to know how many applications are running. The detail relationship between the application and the main controller will be explained in later subsections.

Key global variables

The global variables will be shared and accessed by the entire model. Table 5.3 enumerates the most important global variables of the smart phone model together with corresponding short explanations. More details and all the global variables can be found in the appendix and the source code.

A few more comments deserving to be mentioned for Table 5.3:

- The programming language supported by UPPAAL resembles the C language to a large extent. However, UPPAAL has its own syntax, such as *int[1,3]* that defines an integer type within the range of [1,3].

- The *appNo* type is a user-defined type, equivalent to *int[0,2]* here. Since it is used as the index of one element in an array related with applications, its range is from 0 to 2 instead of from 1 to 3. Another type not mentioned in Table 5.3 is *R_type*, equivalent to *int[0,1]*. *R_type* functions in the same way as *appNo* except that it is used to distinguish CPU and network resource types.

- In the arrays with two elements such as *totalResource[2]*, *enough[2]* and *videoLevel[2]*, the first element corresponds to the CPU resource while the second element corresponds to the network resource.

- For *CPU_occupy[3]* and *BW_occupy[3]*, the last element in the array corresponds to the last application, i.e. the multimedia application. Since the multimedia application consumes different resources at different QoS levels, the value of the

Global variable	Type	Additional notes
OnApp[3]	bool	The on/off status of each application
appID	appNo	To indicate the most recent application examined by the admission control
priorityApp[3]	const int	The priority of each application
stoppableApp[3]	const bool	To tell whether each application is stoppable or not
CPU_Level	int[1,3]	To indicate the current CPU resource level
CPU_R[3]	int	The total CPU resource at each level
BW_Level	int[1,3]	To indicate the current bandwidth level
BW_R[3]	int	The total network resource at each level
totalResource[2]	int	The currently available CPU and network resource
enough[2]	bool	To indicate if the CPU or network resource is sufficient or not currently
CPU_occupy[3]	int	The CPU resource consumption of each application
BW_occupy[3]	int	The bandwidth consumption of each application
videoLevel[2]	int[1,3]	To indicate the QoS level of the multimedia application. 3 levels regarding CPU consumption and 2 levels regarding bandwidth consumption

Table 5.3: The key global variables of the smart phone model

last element represents the resource consumption at the current QoS level. Thus the two elements *CPU_occupy[3]* and *BW_occupy[3]* should be updated during the QoS level change.

- *priorityApp[3]* and *stoppableApp[3]* are constant type variables because they should never be updated but remain their initial value all the time.

The User template

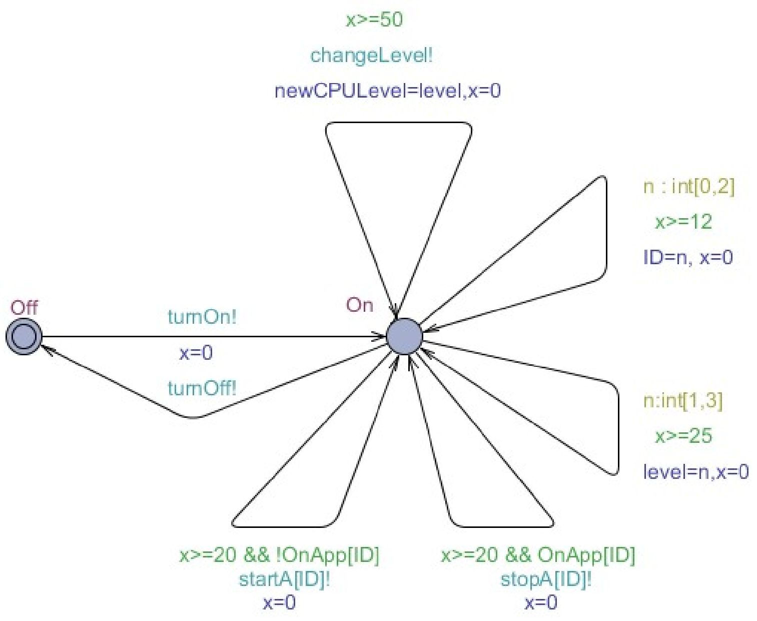

Figure 5.2: The User automaton of the smart phone model

The User template simulates the smart phone user's behavior, with Figure 5.2 illustrating its automaton. A smart phone user toggles between two states: **On** and **Off**. The initial state is **Off**. As the user turns on the smart phone, both the user and the phone step into state **On**, which can go back to **Off** when the user turns off the smart phone. turnOn! and turnOff! are two channels in UPPAAL for the synchronization between different templates. Clock x is used as timing guards. For instance, *x>=20 && OnApp[ID]* guarantees that the associated transition will not be taken until clock x reaches 20 time units and *OnApp[ID]* should be true simultaneously. The main purpose of setting these timing guards is to prevent the user's actions from interrupting the system scheduling. After the user launches a new application, it takes some time for the system to either accept or reject this application. We can assume that this interval is short enough to make it atomic. Two reasons are backing up the validity of this assumption. First, a user with normal behavior seldom turns on or off different applications so frequently. Second, even if a user with lunatic mind has the inclination to switch the running status of each application as frequently as possible, the CPU is still much faster as it has enough time to process a new application and finish the resource allocation between two consecutive actions of the user.

The user's operation on the smart phone becomes a sporadic event due to the timing guards. Each time the user randomly selects one of those three applications and changes its running status. That is to say, if the selected application is running, the user stops it and if the selected application is not running, the user starts it. Different from standard C language, UPPAAL does not support random number generation. Instead, we use the *select* syntax to simulate the user's random behavior. A local variable *ID* is defined as the index of each application. It is assigned a value, which could be 0, 1 or 2, when the clock x is bigger than 12. startA[ID]! and stopA[ID]! are two channels to start or stop the selected application.

Moreover, sometimes the user may also change the CPU resource level. A local variable *level* is assigned 1, 2 or 3 in the same way as *ID*. When the user tries to change the CPU resource level, the value of *level* is assigned to a temporal global variable *newCPULevel*, which is temporal because the user's CPU resource level change request may be denied and its value is finally assigned to another global variable *CPU_Level* introduced in Table 5.3 if the request is accepted.

Notice that the two channels, turnOn! and turnOff!, are not limited by any timing guards. The reason is that they are top priority actions in most systems. Therefore, any process can be interrupted by turning off the smart phone. This problem must be always considered in all the automata except for committed locations.

The Application template

The Application template is a generic template with an input parameter, simulating different applications of the smart phone or other similar systems. The automaton in Figure 5.3 shows that each application has three states: **Off**, **Wait** and **On**. **Off** is the initial state. It goes to state **Wait** when it is started by the user. startA[e]? in Figure 5.3 is synchronized by startA[ID]! of the user, or the main controller which will be introduced later. *e* is an *appNo* type variable to specify the current application to be processed. Before an application goes to state **Wait** from state **Off**, it updates the global variable *appID* to its own ID so that all the other parts of the system know which application is in process. Then the application will wait for the feedback from the admission control, which will be discussed later. The application could be either accepted or refused due to limited resource, and it is able to reach state **On** only when it is accepted. An application cannot transit from **On** to **Wait**. Instead, only **On->Off** is possible when the application is stopped or the smart phone is turned off. The running

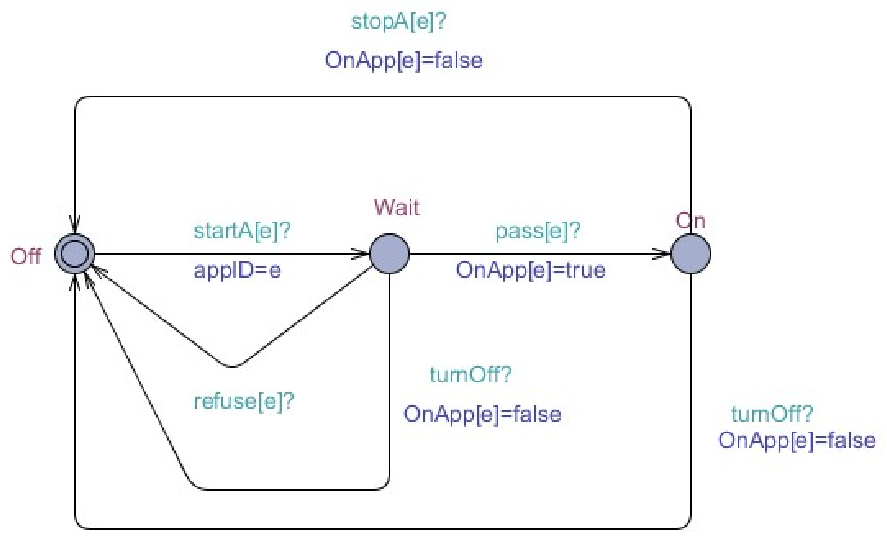

Figure 5.3: The Application automaton of the smart phone model

status of each application is stored in the global variable *onApp[3]*, where "true" means the corresponding application is on and "false" means the corresponding application is off.

The Resource template

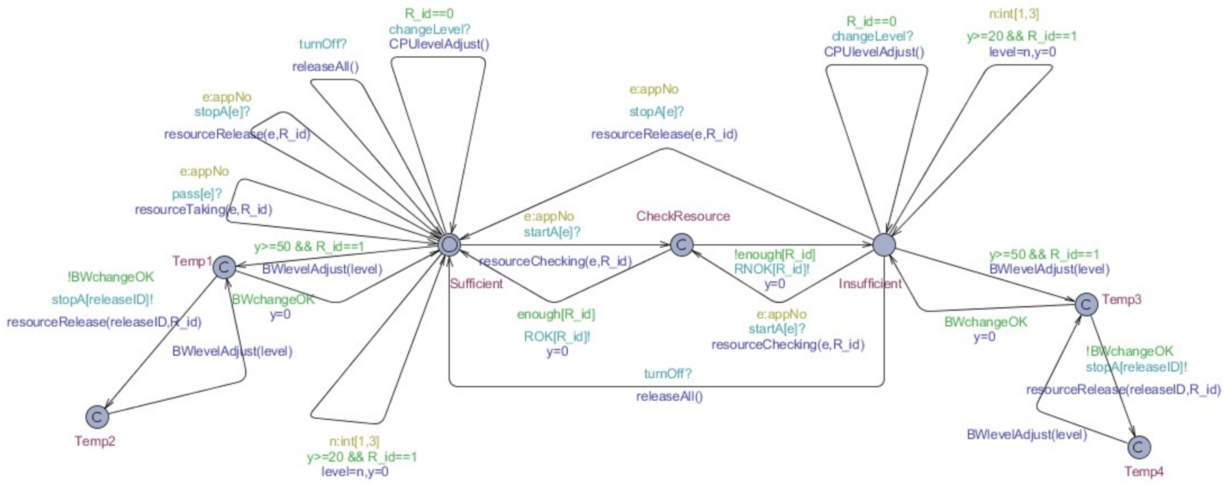

Figure 5.4: The Resource automaton of the smart phone model

The Resource template is designed in a generic way in order to fit all types of resources. It is the core of the entire UPPAAL model of the smart phone example because the major part of the resource allocation mechanism is integrated in the Resource template.

Figure 5.4 depicts the Resource automaton, which is the most complex one among all the automata of this smart phone model. As was mentioned before, the input parameter *R_id* distinguishes the CPU and network resources (0 for CPU resource and 1 for network resource). Despite the complexity, each resource switches only between two stable states: **Sufficient** and **Insufficient**. **Sufficient** is the initial state for no applications are occupying any resource before the system starts. When a new application starts running, each type of resource will be informed and checks its availability, stepping into the committed location **CheckResource**. The function *resourceChecking(e,R_id)* in Figure 5.4 considers three different cases:

- If the multimedia is neither running nor the new application, the feasibility of QoS degradation becomes out of consideration. For each type of resource, the new application is accepted if it consumes less than its total amount of available resource.

- If the multimedia is the new application and its current QoS level of CPU or bandwidth is higher than the minimum, there will be a higher chance for it to be accepted. If one type of resource is not sufficient to accept the multimedia application, we check the resource for another round by applying QoS degradation. The multimedia application is refused only when the resource is still insufficient and no further QoS degradation can take place.

- If the new application is not the multimedia, which yet is a currently running application, the QoS degradation of the multimedia application increases the likelihood of accepting the new application.

After checking the resource availability, each type of resource will report its own status to the Admission Control (AC) which will then make a decision to either accept or temporarily reject this new application. In Figure 5.4, ROK![R_id] means the resource indexed by *R_id* is sufficient for this new application while RNOK![R_id] implies insufficiency. Once a type of resource reports its insufficient status, it falls into state **Insufficient**. If the resource is sufficient for a new application, it may receive the "pass" signal from the AC soon. Then the new application is accepted, consumes resources (realized by the function *resourceTaking(e,R_id)* in Figure 5.4) of different types and starts running.

In previous subsections, we have mentioned that both the CPU and network resources have three levels. The CPU resource level is changed by the user and the network resource level changes automatically in an unpredictable manner. The channel changeLevel? in Figure 5.4 is synchronized together with the user. The guard *R_id==0* makes sure that only the CPU resource responds to this synchronization. The function *CPUlevelAdjust()*, which is line with the resource allocation mechanism described in Section 5.2.2, examines four cases:

- If the CPU level is raised, the change request is undoubtedly accepted. Meanwhile, the possibility of QoS upgrade of the multimedia application is checked due to the increased CPU resource.

- The CPU level is lowered, but the currently occupied CPU resource by running applications are no more than the total available CPU resource at the new level. Then the change request is also accepted without affecting any running applications.

- The CPU level is lowered and the currently occupied CPU resource by running applications are more than the total available CPU resource at the new level. However, the multimedia application is running at a high QoS level. The change request is only rejected when the QoS degradation still fails to release enough CPU resource.

- If none of the above three cases are encountered, the change request is rejected at once.

The bandwidth change is simulated as a sporadic event with a minimal interval of 50 time units. It is realized by assigning a local variable *level* (an integer ranging from 1 to 3) to the global variable BW_Level. The function *BWlevelAdjust(level)* implements the resource allocation mechanism to adapt the changing bandwidth and it is very similar to *CPUlevelAdjust()* as also four cases are considered. Nevertheless, the bandwidth change request can never be rejected. If the system becomes devoid of network resource after the bandwidth level is lowered, in the worst case, no resource allocation mechanism being able to overcome this situation, one or even more running applications must be terminated by force. This corresponds to the loop linked with two committed locations **Temp1** (or **Temp3** in the other loop) and **Temp2** (or **Temp4** in the other loop). The network resource is checked again every time a running application is terminated until $BWchangeOK$ is true, indicating enough network resource is released.

The AdmissionControl template

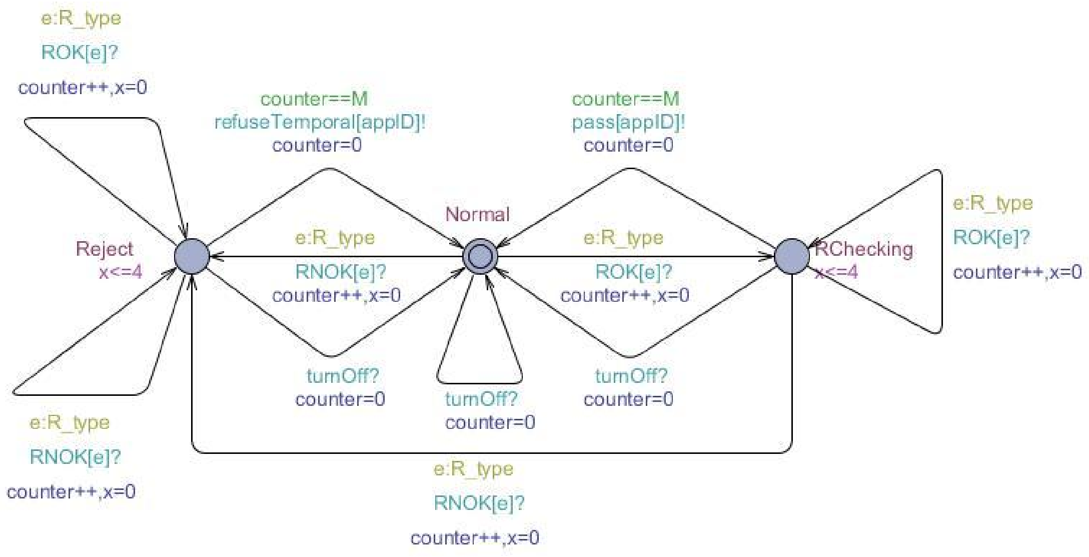

Figure 5.5: The AdmissionControl automaton of the smart phone model

The AdmissionControl template is in charge of the admission of a new application. An AC may adopt different policies to decide whether a new application should be accepted or not. In our model, the policy is fairly simple. The AC awaits the reports from all types of resources. The new application is accepted only if all these reports indicate the resource sufficiency. Otherwise, the new application will be temporarily refused. To implement this strategy, only three states suffice to consider all possible cases. These three states are **Normal**, **RChecking** and **Reject** in Figure 5.5. As a matter of fact, only **Normal** is the stable state while the other two are intermediate states. Once a new application arrives, a local counter of the AC will keep track of the number of reports to determine if all the resources have reported their own statuses. In our model, we assumes that the reports from all resources to the AC can never get lost. **Reject** is a state which indicates that at least one type of resource is insufficient after the arrival of the new application. **RChecking** is a state which indicates that all the reports already received are positive. However, state **RChecking** does not guarantee the acceptance of the new application. Before receiving the reports from all the resources, **RChecking** may transit to state **Reject** as the AC receives the first negative report. When the counter reaches the number of resource types, the AC makes its decision by either accepting or temporarily refusing the new application, based on which temporal state it belongs to. Only the transition **RChecking->Normal** will lead to the acceptance of the new application. Nevertheless, the AC does not have the final saying of the rejection of a new application. If the resource is still insufficient even after QoS degradation, the new application is refused temporarily, indicated by the channel refuseTemporal[appID]! in Figure 5.5. Later on we shall see that a new application that has been temporarily rejected may still be accepted eventually.

The Controller template

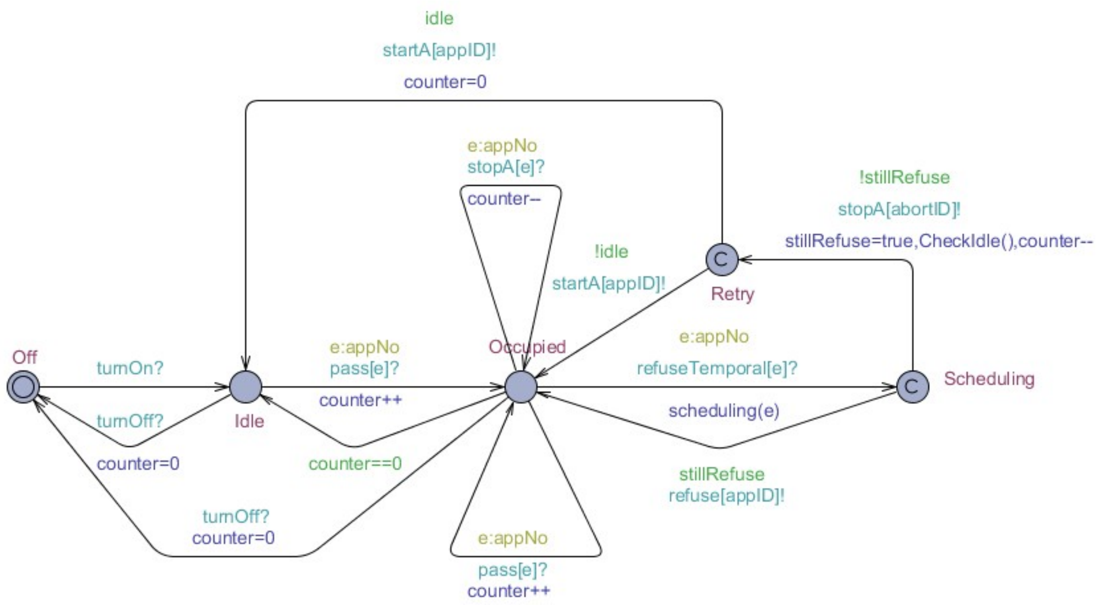

Figure 5.6: The Controller automaton of the smart phone model

The Controller template simulates the scheduling policy of the main controller. In Figure 5.6, we distinguish three main states: **Off**, **Idle** and **Occupied**. The main controller is in state **Off** when the smart phone is off. State **Idle** implies that the smart phone is on but none of the three main applications is running. If at least one of these three applications is running, the controller must be in state **Occupied**. A local variable *counter* keeps tracking the number of running applications so that the controller state can switch between **Idle** and **Occupied**. The consistency between *counter* and the controller state should be guaranteed. Every time a new application passes the admission control or an old application terminates, the counter is updated accordingly. Obviously, *counter* should be 0 while the controller is in state **Idle**.

A special case is that when the AC refuses a new application temporarily, the main controller explores further chances to accept it. This corresponds to the function *scheduling(e)* in Figure 5.6. *scheduling(e)* implements a priority-based and non-preemptive scheduling policy. After a new application is temporarily rejected, the main controller first tries to find whether a stoppable application with lower priority is running. Simultaneously, a local boolean variable *stillRefuse* is updated by the function *scheduling(e)*. If such an application does not exist, the new application must be rejected at once. If such an application exists, that application is terminated due to less importance and *stillRefuse* is set to false. This case is presented by the switch from state **Scheduling** to state **Retry**. While an application is terminated in this way, the main controller checks if there is still any running application via the function *CheckIdle()*. This information is required to know the next state, either **Idle** or **Occupied**. Actually the state loop formed by **Occupied**, **Scheduling** and **Retry** contains an iterative procedure. The resource availability is examined after a low-priority stoppable running application is terminated until the new application is accepted or rejected. The channel startA[appID]! initiated by the main controller enables the next round of scheduling. We should notice that startA[appID]! can be initiated by both the user and the main controller. They have different purposes, yet for each application and each type of resource, the origin of startA[appID]! does not matter.

5.2.4 Property verification

A model is built to satisfy the requirements on the object that is being modeled. Therefore, the correctness of a model must be checked during its design. UPPAAL uses a simplified version of CTL (Computation Tree Logic) [6] to describe model properties which will be verified by the built-in model checker. The most common properties can be classified into reachability, safety, and liveness.

Our smart phone model has been validated through the verification of quite a number of properties. Due to the model complexity, there is no guarantee that the model is completely free of any potential problem. However, most desired functionalities of the smart phone are correctly verified in our model according to the following properties.

Property 1: A[] not deadlock *Satisfied*
There is no deadlock in this UPPAAL model. It is an essential property that should be verified before all the other properties. This property must be satisfied without any compromise.

Property 2: E<> Resource(1).Temp2 *Satisfied*
This is a typical reachability property. We have proved that the state **Temp2** in Figure 5.4 is reachable for the network resource. As we already mentioned, Resource(0) corresponds to the CPU resource and Resource(1) corresponds to the network resource. Let's consider one possible scenario leading to the state **Temp2** of Resource(1). The bandwidth level degrades from 2 to 1 while all the three applications are running. As a consequence, the network resource becomes insufficient and the multimedia application must be terminated to adapt the new bandwidth level. As the multimedia application is terminated, Resource(1) reaches state **Temp2**.

Property 3: E<> Resource(1).Temp3 *Satisfied*
Similar to Property 2, this property tests whether the state **Temp3** of Resource(1) can be reached. Its satisfaction is evident from Figure 5.4.

Property 4: E<> Resource(1).Temp4 *Not satisfied*
This property is interesting because the state **Temp4** of Resource(1) turns out to be unreachable. However, this does not imply any model design error. Instead, the reason derives from the parameters of the smart phone model, listed in Table 5.1 and 5.2. The network resource, i.e. Resource(1), only goes to state **Insufficient** while the bandwidth level is 1. Hence, the bandwidth level cannot decrease further as long as Resource(1) is in state **Insufficient**. And the increase of the bandwidth level will not lead to the termination of any running application. As a result, Resource(1) will never reach **Temp4**. Nevertheless, if the values in Table 5.1 and 5.2 are modified with thoughtful consideration, **Temp4** may be reachable for the network resource.

Property 5: A[] totalResource[0]<=9 *Satisfied*

The available CPU resource is always no more than 9. From Table 5.2, we notice that the upper bound of the available CPU resource is 9, which can never be exceeded.

Property 6: A[] totalResource[1]<=5 *Satisfied*
The available network resource is always no more than 5. From Table 5.2, we notice that the upper bound of the available network resource is 5, which can never be exceeded.

Property 7: E<> (OnApp[0] & OnApp[1] & OnApp[2])==true *Satisfied*
It is possible that all the three applications run simultaneously. By referring Table 5.1 and 5.2, we know that all the three applications can run at the same time as long as the CPU level and bandwidth level are both higher than 1.

Property 8: A[] forall (i:appNo) Application(i).Wait imply appID==i *Satisfied*
Whilst an application is waiting for the answer from the AC, the global variable *appID* indicates its own application ID. For the AC, only one application is in process at a time.

Property 9: A[] Controller.Idle imply (OnApp[0] | OnApp[1] | OnApp[2])==false *Satisfied*
While the main controller is in state **Idle**, no application should be running. Otherwise, if at least one application is running, the main controller will be in state **Occupied**.

Property 10: A[] Controller.Idle imply Controller.counter==0 *Satisfied*
This property checks the consistency between the state and the local counter of the main controller. The satisfaction of Property 9 indicates that no application is running if the main controller is in state **Idle**. Property 10 offers a supplement, claiming that the local counter of the main controller is zero in state **Idle**.

Property 11: A[] AdmissionControl.counter<=2 *Satisfied*
The local counter of the AC should never exceed 2. This counter expresses how many types of resources have reported their availability for the new arriving application. In this smart phone example, there are only two types of resources, the CPU resource and network resource. Therefore, the local counter of the AC is always no more than 2.

Property 12: E<> Controller.counter==3 *Satisfied*
As a matter of fact, this property is equivalent to Property 7. The local counter of the main controller expresses how many applications are running. Since it is possible that all the three applications can run at the same time as per Property 7, the local counter of the main controller can be 3 as well.

Property 13: Application(0).Wait –> Application(0).On *Not satisfied*
This is a liveness property. If an application is in state **Wait**, will it eventually reach state **On**? In other words, if a new application is waiting for the answer from the AC, will it be accepted sooner or later? This property is not satisfied due to the "Turn off" command of the user. In Figure 5.3, the channel turnOff? from **Wait** to **Off** can prevent an application from reaching state **On**.

Property 14: A[] forall (i:R_type) Resource(i).Sufficient imply enough[i]==true *Satisfied*
For either the CPU resource or the network resource, state **Sufficient** implies the resource sufficiency of that type. The global variable *enough[2]* has been introduced in Table 5.3.

Property 15: A[] forall (i:R_type) Resource(i).Insufficient imply enough[i]==false *Satisfied*
Similar to Property 14, for each type of resource, state **Insufficient** implies resource insufficiency.

Property 16: A[] Controller.Scheduling imply (enough[0] & enough[1])==false *Satisfied*
If the main controller is in state **Scheduling**, at least one type of resource is insufficient for the new arriving application. Apparently, if at least one of *enough[0]* and *enough[1]* is false, *(enough[0] & enough[1])* will be false. It is not hard to predict the verification result of this property. The main controller goes to state **Scheduling** because a new application is temporarily refused by the AC. And this temporal rejection is due to the insufficient resource.

Property 17: E<> Resource(1).Insufficient and (forall (i:appNo) OnApp[i]==false) *Not satisfied*
Is it possible that the network resource is in state **Insufficient** while no application is running? This property is not satisfied because any type of resource is in state **Sufficient** if no application is running. We will get the same verification result even if "Resource(1)" is replaced by "Resource(0)".

Property 18: AdmissionControl.Reject –> AdmissionControl.Normal *Satisfied*
If the AC is in state **Reject**, it will eventually reach state **Normal**. From Figure 5.5, we notice that all outgoing edges from **Reject** lead to either **Reject** itself or **Normal**. Once the AC receives the reports from both the CPU and network resources, it will eventually go to state **Normal** from state **Reject**.

Property 19: AdmissionControl.RChecking –> AdmissionControl.Normal *Satisfied*
If the AC is in state **RChecking**, it will eventually reach state **Normal**. From Figure 5.5, we notice that all outgoing edges from **RChecking** lead to **RChecking** itself, **Reject** or **Normal**. If both reports from the CPU and network resources are positive, the AC will go to state **Normal** from **RChecking** eventually. If the AC transits to state **Reject** due to a negative resource report, the AC will eventually go to **Normal**, too. This has been proved in Property 18.

Property 20: E<> (forall (i:appNo) OnApp[i]==true) and videoLevel[0]==3 and videoLevel[1]==2 *Satisfied*
It is possible that all the three applications are active with the multimedia application running at highest QoS level, i.e. Level 3 for the CPU resource and Level 2 for the network resource. After the analysis of Table 5.1 and 5.2, we come to know that this scenario can happen while both the CPU resource level and the network resource level are upgraded up to 3.

Property 21: A[] ((forall (i:appNo) OnApp[i]==true) and videoLevel[0]==3 and videoLevel[1]==2) imply (CPU_Level==3 and BW_Level==3) *Satisfied*
This property follows Property 20 directly. If the scenario described in Property 20 really exists, does it mean that both the CPU and network resource levels must be 3? The verification result tells us that this is true. Only top resource levels will satisfy Property 20.

5.2.5 Discussion

We have modeled a smart phone in UPPAAL, aiming at demonstrating how an AES typically behaves. The smart phone model presents techniques such as dynamic resource re-allocation and graceful QoS degradation. We should remember that the smart phone is just a case study in our model, whose generality enables it to describe many other similar systems with arbitrary number of applications and resource types. The resource allocation mechanism and scheduling policy can be easily altered to generate various versions of this model. The model complexity is not sensitive to the number of applications and resource types because all the automata remain almost unchanged. In contrast, only global variables and the values in Table 5.1 and 5.2 need to be modified. However, the property verification time grows exponentially as the application number and resource type rise. Any kind of possible model simplification will be desired.

5.3 Case study 2: The UPPAAL modeling of an object-tracking robot

In this section, we turn to another UPPAAL model: an object-tracking robot. As another typical AES, the robot model demonstrates how hardware redundancy contributes to adaptive fault tolerance as well as other issues concerning adaptivity. Section 4.5 has already described this problem, but the robot model will be much more concrete. Next we shall focus on the main functionalities of the robot and its UPPAAL model design. Just like the smart phone model, the robot model possesses certain generality so that it is able to simulate other similar AESs.

5.3.1 Functionalities and adaptivity of the robot

The robot may be capable of fulfilling various missions, but our focus here is its object tracking function. The robot is equipped with a group of localization sensors in order to keep track of moving objects. These localization sensors are identical and work simultaneously. We typically call this sensor fusion technique. The purpose of this type of hardware redundancy is to achieve both fault tolerance and high accuracy. The final sensor value is the average of all sensor readings. On the one hand, this improves accuracy. On the other hand, if one sensor is broken due to some reason, the other sensors can still provide a decent result. Since the reading of the broken sensor may deviate a lot from the readings of other normal sensors, it will negatively affect the final result. As an AES, the robot should immediately notice this anomaly and isolate it from the system by deactivating the faulty sensor. Once we turn off a faulty sensor, its wrong reading will not jeopardize the final result, and another benefit is that the power consumption will be reduced.

The robot works in different modes. What we concern most is the object tracking mode. We assume that none of those localization sensors are used in any other mode rather than the object tracking mode and they should be turned off automatically to save power. The adaptivity is reflected from the dynamic switch of hardware component availability due to the malfunction of a single sensor or mode switch.

Furthermore, the sampling rate of each localization sensor can also be adaptive. While the moving object is far away, low sampling rate is preferred to save power. In contrast, as the robot is approaching to the moving object, higher sampling rate is expected because tracking the object becomes more urgent at a closer distance.

5.3.2 The UPPAAL model of the robot

Our robot model is composed of four templates: user, robot, sensor, and controller. First it is necessary to present the architecture of our robot model. The architecture denotes how these four templates relate to each other. Then we introduce the key global variables, each template, and the property verification, respectively.

The model architecture

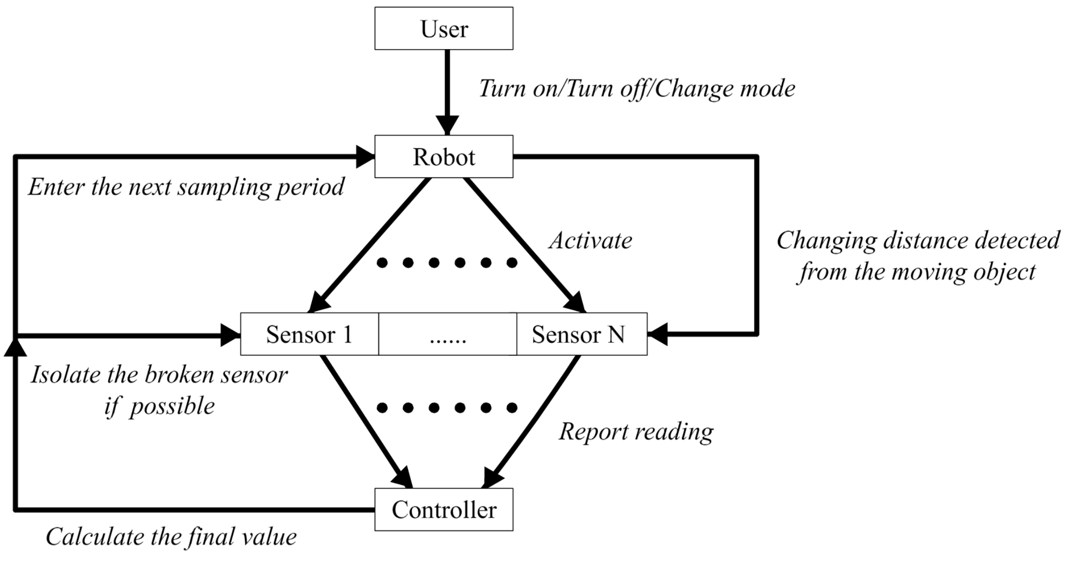
User
Turn on/Turn off/Change mode
Robot
Enter the next sampling period
Activate
Changing distance detected from the moving object
Sensor 1
......
Sensor N
Isolate the broken sensor if possible
Report reading
Controller
Calculate the final value

Figure 5.7: The architecture of the object-tracking robot model

Our model consists of four components, user, robot, sensor and controller, with each component as one template. There are frequent interactions and tight relationship between these components. Figure 5.7 illustrates the model architecture, showing how different components are connected with each other. There are altogether N sensors (N=*3* here) and they all belong to the same "Sensor" component. The user can turn on or turn off the robot and change its operational mode. Since our focus is the object tracking mode, other modes are invisible in Figure 5.7. In the object tracking mode, each non-broken sensor reports its own reading to the controller periodically. Then the controller will calculate the final value, i.e. the average of all available sensor readings in our model, and tell the robot to enter the next sampling period. The average obtained from the controller is the criterion to find out the faulty sensor with too much deviation. In our model, we can pick up at most one faulty sensor each time and this faulty sensor must be isolated from the rest of the system.

Key global variables

The global variables will be shared and accessed by the entire model. Table 5.4 enumerates the most important global variables of the robot model together with some short explanations. More details and all the global variables can be found in the appendix and the source code.

There are a few more comments deserving to be mentioned for Table 5.4:

Global variable	Type	Additional notes
initSV	int[1,3]	This global variable will be explained in the following paragraph.
Svalues[3]	sensorValue	The readings of all localization sensors.
sensorStatus[3]	int[0,2]	The status of each sensor. 0: Off; 1: On; 2: Broken
sampling[3]	bool	To specify if a sensor can report its reading during each sampling period.
samplingRate[3]	int[0,12]	The sampling rate of each sensor, depending on how far the moving object is.
distance	int[8,10]	The distance between the robot and the moving object.

Table 5.4: The key global variables of the object-tracking robot model

- As the UPPAAL syntax denotes, *int[1,3]* defines an integer type within the range of [1,3]. This has also been explained in the smart phone model.
- The *sensorValue* type is a user-defined type, equivalent to *int[0,5]* here, specifying the range of the sensor readings.
- In our model, only three localization sensors are used to simplify the model complexity. Therefore, in the arrays with three elements such as *Svalues[3]*, *sensorStatus[3]* and *samplingRate[3]*, each element corresponds to one sensor. There could be more sensors in real applications, but the principle is the same as in this simple model.
- Due to the fact that there exist slight differences among those sensor readings, we should simulate those similar but different values in our model. This is realized by a common base value and a variable offset for each sensor. All sensors have the same base value but different offsets. Both the base value and offset can change within their own ranges during each sampling period. *initSV* is the base value, from 1 to 3. Although in reality this range is definitely too small, it is preferred for a model where complexity is not favored. In later sections, we shall see that the offset for each sensor ranges from -1 to 2. By combining *initSV* and the offset, we figure out that the range of each sensor reading falls between 0 and 5.
- The actual range of *samplingRate[3]* is from 10 to 12 for a normal sensor. However, once a sensor is broken, its sampling rate becomes 0. Thus the possible sampling rates are 0, 10,11 and 12.
- Just like the smart phone model, all the values of these variables are conceptual as they are only used to demonstrate adaptive scenarios.

The User template

The User template simulates how the user manipulates the robot, with Figure 5.8 illustrating its automaton. The user can start or stop running the robot by turning it on or off. Besides, the user can also manually change the operational modes of the robot. The robot may have lots of different operational modes, however, in our model we only consider those activities taking place in the object tracking mode. That is why the User automaton only has four states. Apart from states **Off**, **Idle**, and **TrackingMode**, all the other modes belong to **OtherModes**. The activities in other modes and the mode switch among them will not be considered here. The clock x and the timing guard

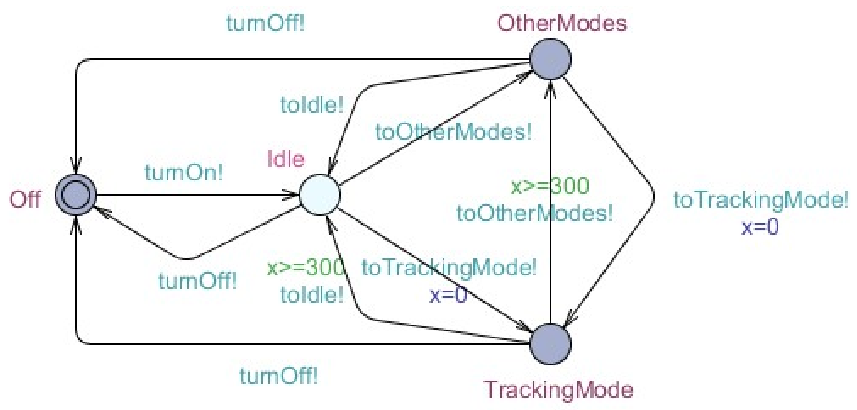

Figure 5.8: The User automaton of the object-tracking robot model

x>=300 are used to make sure that the duration of the object tracking mode for the robot is long enough. Now that we only concern this mode, we don't expect this mode to be interrupted by other modes too frequently.

The Robot template

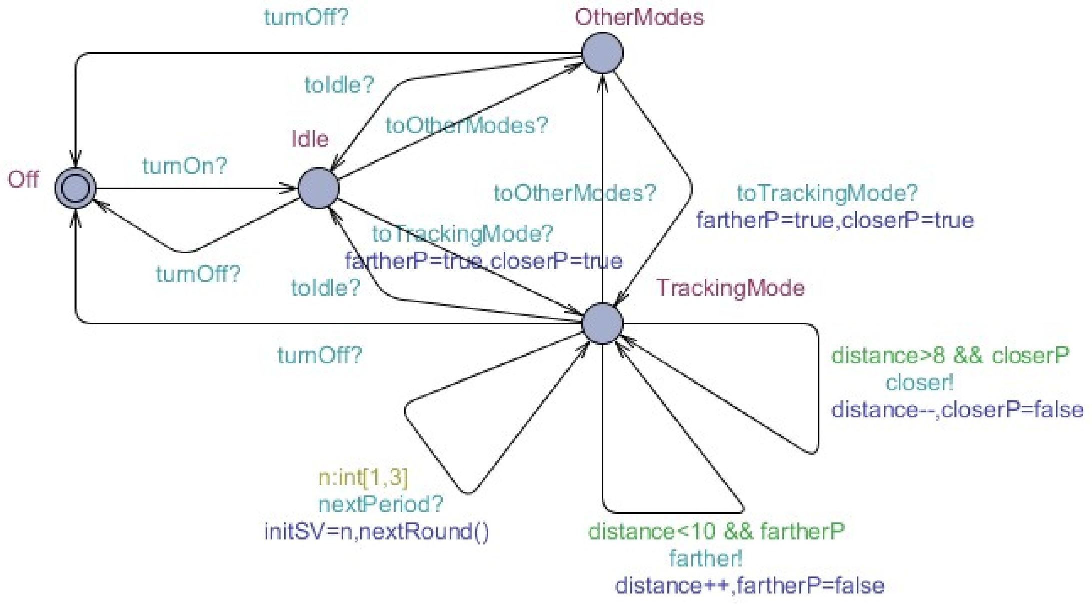

Figure 5.9: The Robot automaton of the object-tracking robot model

The robot has direct interactions with the user. Especially, it has exactly the same states as the user, as is indicated by the Robot automaton in Figure 5.9. *n:int[1,3]* generates an integer from 1 to 3 at random and assigns this integer to the global variable *initSV*. The value of *initSV* is updated in this way after each sampling period. The sampling period is defined as the interval during which all non-broken sensors report their readings to the controller. We shall see that nextPeriod? is a signal issued from the controller when we explain the Controller template later. This signal forces the robot and all non-broken sensors to enter the next sampling period. Furthermore, the robot is responsible for telling the sensors to adapt their sampling rates to the changing distance between itself and the object being tracked. The closer distance, the higher sampling rate and the other way round. When the distance is changed, the robot will notify the sensors through the channels closer! and farther!. The frequency of the distance change is unpredictable in reality, nonetheless, we specify that the distance can increase or decrease at most once during each sampling period. This is realized by the boolean variables *closerP* and *fartherP*. The guards in Figure 5.9 also guarantee that the distance won't exceed the range [8,10].

The Sensor template

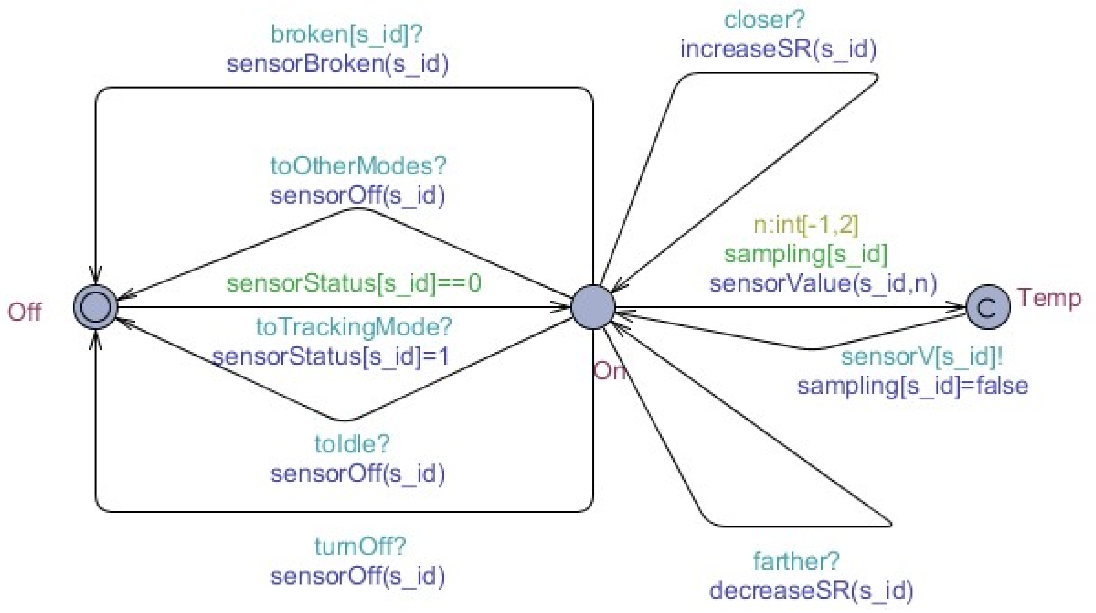

Figure 5.10: The Sensor automaton of the object-tracking robot model

The Sensor template simulates a group of localization sensors of the robot. It has one input parameter *const sensorIndex s_id*, which distinguishes each sensor from the other sensors. A Sensor automaton typically has three states, **Off**, **On** and **Temp**, shown in Figure 5.10. State **Temp**, which is also a committed location, expresses no substantial meaning but is only used for UPPAAL model design. The global variable *sensorStatus[3]* in Table 5.4 has clarified that a sensor has three statuses: Off, On and Broken. This may imply inconsistency because no **Broken** state exists in the Sensor automaton. However, the Broken status is actually included in state **Off**. We do it in this way for the sake of model simplicity as each extra state may make the verification time grow exponentially.

The switch from **Off** to **On** is quite straightforward. The initial sate is **Off**. While the sensor is informed that the robot enters the object tracking mode, it switches to state **On** as long as it is not broken. The guard *sensorStatus[s_id]==0* guarantees that a broken sensor will not be activated. In the object tracking mode, all non-broken sensors work according to the uniform sampling period. During each sampling period, we assume that a non-broken sensor must report its reading to the controller once. The guard *sampling[s_id]* makes use of the global boolean variable *sampling[3]* explained in Table 5.4. The purpose is to avoid the case that a sensor reports its reading to the controller more than once during each sampling period. Once a sensor has reported its reading to the controller during the current sampling period, *sampling[s_id]* will be set to false. In fact, *sampling[3]* functions in the same way as *fartherP* and *closerP* in the Robot automaton.

The function *sensorValue(s_id,n)* generates a sensor reading randomly within a specified range. A sensor reading is the sum of the common base value *initSV* (ranging from 1 to 3) and the offset (ranging from -1 to 2). Since the offset of each sensor may differ, all non-broken sensors will usually report similar but slightly different readings. This is how our model simulates the real sensor readings.

Besides reporting the reading to the controller, a sensor may also change its own sampling rate when the robot informs it that the distance to the moving object is changing. The functions *increaseSR(s_id)* and *decreaseSR(s_id)* are in charge of adjusting the sampling rate of a sensor according to the updated distance. We have a different assumption of the sampling rate adjustment compared with the sensor reading report. During each sampling period, while a sensor reading report is compulsory for a non-broken sensor, the sensor sampling rate can be updated once due to the changing distance or not updated at all.

The switch from **On** to **Off** can be triggered by different conditions. When the entire system is deactivated, or the robot starts working in another operational mode, all the localization sensors will be turned off to save power. In addition, if a sensor is detected to be faulty, it also enters state **Off**. The key difference is that its status will be Broken and its sampling rate is reset to 0. Once a sensor is broken, it will never be used again in our model, which yet can be extended by introducing some fault recovery policy so that a faulty sensor can be reused.

The Controller template

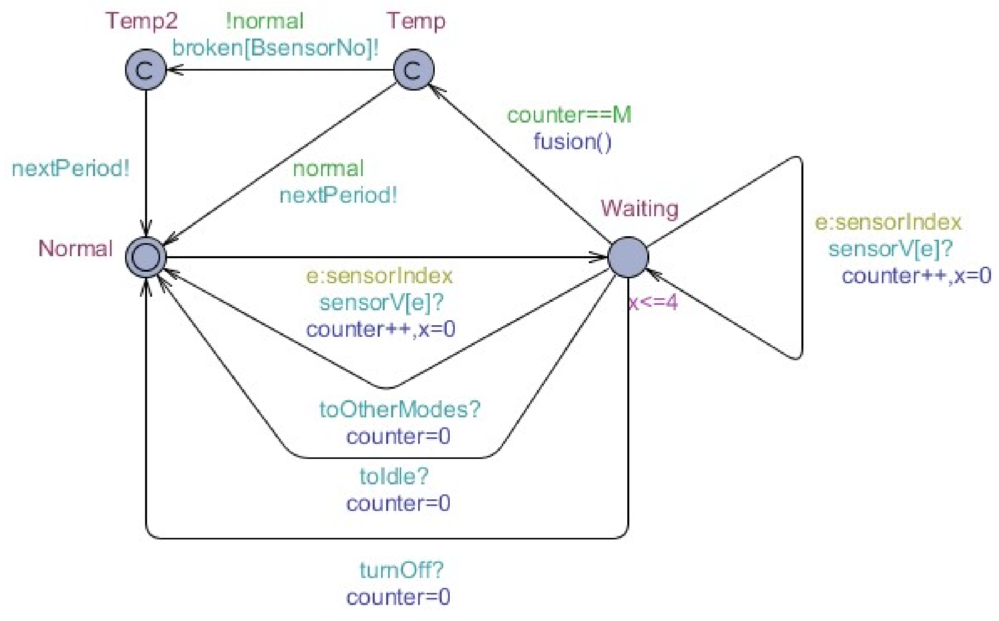

Figure 5.11: The Controller automaton of the object-tracking robot model

The Controller template is the most intelligent part of the robot. It collects the readings from all non-broken sensors, calculates the final sensor value and detects faulty sensors in time. Figure 5.11 presents the Controller automaton. **Normal** is the initial and steady state. The controller should always be in this state at the beginning of each sampling period. Upon receiving the first reading report from a localization sensor, the controller changes its state to **Waiting** and waits for the reading reports from the other non-broken sensors. A local counter records the number of reading reports that have been received. In our model, we assume that the reading report from each sensor to the controller can never get lost. When the local counter denotes that all the expected reading reports have been received, it is time for the controller to calculate the average value and detect the potential faulty sensor. The function *fusion()* decides what policy is adopted to get the final sensor value based on these readings. The current policy here is quite straightforward. The final sensor value is the average of all readings during each sampling period. Then each reading is compared with this average, and their difference is stored as an offset in an array. The maximal offset will be selected, however, since at most one faulty sensor can be detected each time, we ignore the case where two or more equivalent maximal offsets are found. If we indeed discover the single maximal offset, we realize that the corresponding sensor has reported the least accurate reading during this sampling period. Nevertheless, this does not necessarily imply that this sensor is faulty. We still need to check if this maximal offset is above a pre-defined threshold, which is 1 in our model. If the offset being checked is bigger than the threshold, the reading error will be beyond the tolerance of the system and this sensor will be considered to be faulty.

In Figure 5.11, there are two outgoing branches from state **Temp**. The local boolean variable *normal* decides which branch the controller should take after processing *fusion()*. On most occasions, no faulty sensor is found and *normal* is true. Thereafter the controller informs the robot to enter the next sampling period and all non-broken sensors will be aware of this at once. In case that a faulty sensor is detected, *normal* will be false and the signal broken[BsensorNo]! in Figure 5.11 will be sent from the controller to the faulty sensor. After this faulty sensor is shut down, the robot enters the next sampling period.

5.3.3 Property verification

Compared with the smart phone model, the robot model is relatively less complex. We have verified 13 properties and except for the last one, all the other 12 properties are satisfied.

Property 1: A[] not deadlock *Satisfied*
There is no deadlock in this UPPAAL model. This property must be satisfied.

Property 2: E<> Controller.Temp2 *Satisfied*
This is a typical reachability property. State **Temp2** of the Controller automaton implies a faulty sensor is detected. If this state is reachable, it will be certain that some scenarios can lead to the discovery of a faulty sensor. The satisfaction of this property is desired because it is impossible to demonstrate the major adaptivity of the robot without detecting a faulty sensor.

Property 3: A[] M>=2 && M<=3 *Satisfied*
This property is based on our assumption: When there are only two non-broken sensors left, we would never be able to know which one is faulty. Hence neither of them should be turned off. Instead, both of them are considered to be in good condition and their average is the final sensor value. The global variable M here is the number of non-broken sensors. In our model, we define three localization sensors. Therefore, M should be either 3 or 2. In other words, at most one sensor can be faulty.

Property 4: A[] (sensorStatus[0]+sensorStatus[1]+sensorStatus[2])<5 *Satisfied*
This property essentially has the same purpose as Property 3. Out of the three sensors, at most one can become faulty. The global variable *sensorStatus[3]* has been shortly explained in Table 5.3. For a sensor indexed by i, *sensorStatus[i]==0* means off, *sen-*

sorStatus[i]==1 means on and *sensorStatus[i]==2* means broken. Then we are sure that the maximal value of *sensorStatus[i]* is 2. Property 4 verifies the upper bound of *(sensorStatus[0]+sensorStatus[1]+sensorStatus[2])*. The verification result tells us their sum will never reach 5. When two sensors are working and the third sensor is broken, the sum is 4. Yet 5 requires that at least two sensors are broken. According to Property 3, this is impossible in our model. Therefore, Property 4 does not break the consistency with Property 3.

Property 5: A[] Controller.counter>=0 && Controller.counter<=3 *Satisfied*
The local counter of the controller always falls into the interval [0,3]. In Section 5.3.2, we ever mentioned that the local counter of the controller increases by 1 as one sensor reading report is received. When the number of received reading reports reaches the number of non-broken sensors, i.e. *M*, the controller will no longer wait for any reading report during the current sampling period. As a matter of fact, what the controller will do next is to calculate the final sensor value and set the local counter to 0. We can deduce that the upper bound of *Controller.counter* is *M*. Property 3 has proved that the upper bound of *M* is 3, thus *Controller.counter<=3* should always hold. Besides, the increment of *Controller.counter* starts from 0. Our analysis is proved via the verification of Property 5.

Property 6: A[] Controller.Normal imply Controller.counter==0 *Satisfied*
This property directly follows Property 5. The local counter should always be 0 whilst the controller is in state **Normal**. The reason is that once the counter becomes 1 from 0, the controller must go to state **Waiting** upon receiving the first reading report.

Property 7: Controller.Waiting –> Controller.Normal *Satisfied*
This is the only one liveness property in our model. When the controller is in state **Waiting**, it will eventually go to state **Normal**. This can be analyzed by observing the Controller automaton in Figure 5.11. Starting from state **Waiting**, three outgoing edges directly lead to state **Normal**. The invariant *x<=4* ensures that the controller will not stay in state **Waiting** forever. When all the reading reports are received, the controller will be forced to go to state **Temp**, which is a committed location and will go back to state **Normal** sooner or later.

Property 8: A[] (distance==8 && sensorStatus[0]==1) imply (samplingRate[0]==12) *Satisfied*
In our model, there is a clear mapping between the distance towards the moving object and the sampling rate of each non-broken sensor. The initial distance is 10 and the sampling rate associated with this distance of an active sensor is 10. When the distance is decreased by 1, the sampling rate will be increased by 1 accordingly. To simplify the model, the minimal distance is set to 8 here. From the given mapping, the corresponding sampling rate should be 12. If this property were denied somehow, there would be some potential mapping problem. Fortunately, we did not encounter such a problem. Here we only tested the sensor indexed by 0. The verification result will be exactly the same for the other two sensors.

Property 9: E<> distance==8 *Satisfied*
This property is complementary to Property 8. Property 8 assumes that *distance==8* can happen, but is it really possible? Property 9 confirms this assumption. The desired behavior is that *distance* can fluctuate between 8 and 10.

Property 10: A[] (Robot.Off||Robot.Idle||Robot.OtherModes) imply (forall (i:sensorIndex)

sensorStatus[i]!=1) *Satisfied*
This property intends to demonstrate one type of adaptivity. While the robot is not in the object tracking mode, no localization sensors should be active because they only work to track objects. The robot is supposed to be adaptive enough to switch the operational status of its localization sensors automatically during a mode switch, contributing to much less power consumption without compromising any functionality.

Property 11: A[] User.TrackingMode imply Robot.TrackingMode *Satisfied*
The User automaton (Figure 5.8) and the Robot automaton (Figure 5.9) exactly share the same states. That is to say, the robot is always synchronized to the state where the user stays. Property 11 examines their consistency in state **TrackingMode**, which represents the object tracking mode and deserves most attention. The synchronization between the user and the robot is rather simple, i.e. by UPPAAL channels.

Property 12: A[] sensorStatus[0]==2 imply Sensor(0).Off *Satisfied*
When a sensor is faulty, it will stay in state **Off** forever. This property checks the consistency between the global variable *sensorStatus[3]* and state **Off** of the Sensor automaton. The same is true of the other two sensors.

Property 13: A[] sensorStatus[0]==2 imply (Controller.average==(Svalues[1]+Svalues[2])/2) *Not satisfied*
This is the only one property that is not satisfied. Yet the reason does not lie in the model design problem. If a sensor is faulty, the final sensor value will be the average of the other two non-broken sensors. This seems to make sense, however, there is still one exception. *Controller.average*, *Svalues[1]* and *Svalues[2]* are all variables which are updated at different moments. Since the update of different variables must follow some sequence, there is always an intermediate moment when en equation with variables on both sides does not hold. For instance, when the variables on the left side are updated but the variables on the right side are not, the balance of an equation will be temporarily broken due to inconsistency problems. *Controller.average* is calculated at the end of each sampling period. When a sensor is found to be faulty, its status is changed but *Controller.average* will not be updated until the next sampling period. Anyway, our purpose is to ensure that a faulty sensor will not contribute to the final sensor value any more. Property 12 has verified this indirectly. Now that a faulty sensor will always stay in state **Off**, it will not report its reading to the controller, thus the final sensor value will only depend on the other two sensors.

5.3.4 Discussion

We have modeled an object-tracking robot by UPPAAL, demonstrating another type of AES. This model is generic enough to describe many other similar systems with hardware redundancy and multi-operational modes. The localization sensors can be replaced by any other type of sensors or even other hardware components for similar usage. And the model allows arbitrary number (the minimum is 3) of sensors or other substitutes. However, although three sensors work fine in our model, even one extra sensor will give rise to a state explosion as a consequence. We have tested four sensors in the Windows version of Uppaal-4.0.11 whose Hash table is set to maximum, yet we still ran out of memory during the verification of the "No deadlock" property. Theoretically speaking, the model structure remains when the number of sensors increases, whereas much more memory is required.

In our model, we calculate the average of multiple sensor readings. This can be extended to any other more complex algorithms and policies without affecting the model structure. Many different criteria can be taken into consideration to find out a faulty sensor.

Moreover, there is still some leeway for the robot to become more intelligent. For example, in order to be robust against faults and errors, some kind of fault recovery mechanism can be implemented so that a faulty sensor still has the chance to serve the system.

Chapter 6

Related work

Besides the topics discussed in this report, quite a lot of other issues of AES have been concerned in other publications. de Oliveira et al. [13] consider each reconfiguration option as an optimization problem whose objective is to maximize the overall system benefit. Two different models, the Integer Programming (IP) and the Linear Programming (LP), are formulated and analyzed. Haase et al. [19] use SDVM (the scalable dataflow-driven virtual machine) as a virtualization layer for multicore-FPGAs with support of dynamic reconfiguration. Noguera and Badia [32] present a dynamic scheduling algorithm for multi-context platforms based on DRL (Dynamically Reconfigurable Logic) architectures.

There is a growing trend for AESs to borrow techniques from control theory: Lu et al. [30] present a Feedback Control real-time Scheduling (FCS) framework for adaptive real-time systems. Lu et al. [29] also propose a framework based on control theory for the design of adaptive, real-time software systems, whose performance is evaluated by a few metrics borrowed from control systems. As an improvement of FC-EDF, a new scheduling algorithm FC-EDF2 is designed by integrating two PID feedback controllers with an EDF scheduler. Simões et al. [33] implement a Generic Algorithm, which is also from control theory, to gradually adapt the system to environmental changes. Persson et al. [39] analyze dynamic load balancing as a control problem while both feedforward and feedback controllers are considered as load balancers. Shankaran et al. [44] explore an adaptive resource management architecture with two feedback loops for both fine-grained and coarse-grained adaptation levels.

Chapter 7

Conclusion

We have carried out a research on Adaptive Embedded Systems (AESs) in this report. After studying numerous adaptive applications, we concluded that an AES is characterized by dynamic reconfiguration. Even though AESs can be demarcated into miscellaneous types which differ a lot from each other, they are more or less related. We proposed a conceptual view pointing out the key functional modules of a typical AES, which must include monitor, controller and adapter. Following the same pattern as the conceptual view, we also presented the overall architecture of both centralized and distributed AESs.

we looked into a few existing techniques that made AESs possible. Our focus is dynamic resource re-allocation and adaptive fault tolerance, which probably could be implemented in a majority of AESs. A concurrent event that happens together with dynamic reconfiguration is the mode switch. We additionally investigated the mode switch problem at task levels, summarized the most common mode switch protocols and analyzed schedulability during mode switch.

We have distinguished five types of AESs and come up with a general model for each of them. The five types are:

- Operational mode switch
- Migration for fault tolerance and load balancing
- Multimedia communication with stable streaming
- Adaptive resource management and QoS degradation
- Dynamic HW/SW module composition

Actually, each type above is an instantiation of our conceptual architecture of AESs. All five types share the same structure but represent different adaptive problems.

At the final stage of our research, we built concrete models for the last two types of AESs, i.e. dynamic resource re-allocation and graceful QoS degradation, and dynamic hardware component composition. The two models were both built by UPPAAL and based on real case studies.

The first model simulates the adaptive behavior of a smart phone. Its dynamic resource allocation mechanism makes the system adaptive to its unstable resources that can be utilized most efficiently. The graceful QoS degradation of its multimedia application turns out to be an effective way of dealing with limited resources. The model is built in a generic fashion so that it is not just dedicated to the smart phone example. Instead, the model is able to represent a cluster of AESs where resources may change dynamically online.

The second model is an object-tracking robot. Its adaptivity lies in three aspects. First, its localization sensors only work in the object tracking mode and are deactivated automatically in other modes to save power. Second, the sampling rate of each localization sensor is adjusted properly as the distance between the robot and the object changes to some extent. The last type of adaptivity, also the most important one, is that a faulty sensor can be detected and shut down immediately. For a system where multiple redundant sensors are working simultaneously, this adaptive behavior can save power and provide better results in comparison with ordinary systems. By virtue of the model generality, the model is not just dedicated to the object-tracking robot. Instead, it can also represent other similar AESs with homogeneous hardware redundancy.

In a word, AES is a new researching area and for sure it will be a future trend deserving our great concern.

Bibliography

[1] T. Abdelzaher, E. Atkins, and K. Shin. Qos negotiation in real-time systems and its application to automated flight control. *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 228–238, 1997.

[2] L. Almeida. A word for operational flexibility in distributed safety-critical systems. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, page 177, 2003.

[3] L. Almeida, S. Fischmeister, M. Anand, and I. Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 67–74, 2007.

[4] L. Almeida, P. Pedreiras, and J. Fonseca. The ftt-can protocol: why and how. *Industrial Electronics, IEEE Transactions on*, 49(6):1189–1201, 2002.

[5] M. Behnam, T. Nolte, and I. Shin. A hierarchical approach for reconfigurable and adaptive embedded systems. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 51–54, 2008.

[6] G. Behrmann, R. David, and K. G. Larsen. A tutorial on uppaal. pages 200–236, 2004.

[7] P. Bieber et al. Preliminary design of future reconfigurable ima platforms. In *APRES '09: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 21–24, 2009.

[8] E. Borde, P. Feiler, G. Haïk, and L. Pautet. A new design approach for complex, adaptive, and critical embedded systems. In *APRES '09: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 33–36, 2009.

[9] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 286, Washington, DC, USA, 1998.

[10] M. Chu and J. Liu. Enabling extensibility of sensing systems through automatic composition over physical location. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 74–77, 2008.

[11] R. Daniela. Dynamic resource allocation for adaptive real-time applications. Technical Report GIT-CC-99-22, Georgia Institue of Technology, 1999. http://hdl.handle.net/1853/6622.

[12] R. Daniela, S. Karsten, Y. Sudhakar, and J. Rakesh. On adaptive resource allocation for complex real-time applications. Technical Report GIT-CC-97-26, Georgia Institue of Technology, 1997. http://hdl.handle.net/1853/6799.

[13] A. B. de Oliveira, E. Camponogara, and G. Lima. Dynamic reconfiguration in reservation-based scheduling: An optimization approach. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, pages 173–182, 2009.

[14] O. Derin and A. Ferrante. Enabling self-adaptivity in component-based streaming applications. In *APRES '09: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 51–54, 2009.

[15] O. Derin, A. Ferrante, and A. V. Taddeo. Coordinated management of hardware and software self-adaptivity. *J. Syst. Archit.*, 55(3):170–179, 2009.

[16] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21:757–785, 1991.

[17] S. M. Ellis. Dynamic software reconfiguration for fault-tolerant real-time avionic systems. *Microprocessors and Microsystems*, 21(1):29 – 39, 1997. Proceedings of the 1996 Avionics Conference and Exhibition.

[18] G. Fohler. Changing operational modes in the context of pre run-time scheduling. 1995.

[19] J. Haase, A. Hofmann, and K. Waldschmidt. A self distributing virtual machine for adaptive multicore environments. *Int J Parallel Prog*, pages 19–37, 2010.

[20] H. Kaneko, J. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 206–217, 1996.

[21] H. Karimi, M. Kargahi, and N. Yazdani. Energy-efficient cluster-based scheme for handling node failure in real-time sensor networks. *2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 143–148, 2009.

[22] K. H. Kim and T. F. Lawrence. Adaptive fault tolerance: issues and approaches. In *Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 38–46, Cairo, Egypt, 1990.

[23] F. Kluge, J. Mische, S. Uhrig, and T. Ungerer. Building adaptive embedded systems by monitoring and dynamic loading of application modules. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 23–26, 2008.

[24] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications ACM.*, 45(6):33–38, 2002.

[25] B. Li and K. Nahrstedt. Dynamic reconfiguration for complex multimedia applications. In *ICMCS '99: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 9165, Washington, DC, USA, 1999.

[26] J. W. Liu et al. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[27] J. W. S. Liu et al. Imprecise computations. In *Proceedings of the IEEE*, volume 82, pages 83–94, January 1994.

[28] Y. Liu, E. Collins, and M. Selekwa. Parity relation based fault detection, isolation and reconfiguration for autonomous ground vehicle localization. In *24th Army Science Conference*, October 2004.

[29] C. Lu et al. Performance specifications and metrics for adaptive real-time systems. In *IN REAL-TIME SYSTEMS SYMPOSIUM*, 2000.

[30] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23(1/2):85–126, 2002.

[31] R. Marau et al. Integration of a flexible time triggered network in the frescor resource contracting framework. In *2007 IEEE Conference on Emerging Technologies&Factory Automation (EFTA 2007)*, pages 1481–1488, 2007.

[32] J. Noguera and R. M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 205–210, New York, NY, USA, 2002.

[33] M. A. C. S. oes, G. Lima, and E. Camponogara. A ga-based approach to dynamic reconfiguration of real-time systems. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 43–46, 2008.

[34] A. Patil and N. Audsley. An application adaptive generic module-based reflective framework for real-time operating systems. In *In Proceedings of the 25th IEEE Work in Progress session of Real-time Systems Symposium*, December 2004.

[35] A. Patil and N. Audsley. Adaptive framework for efficient resource management in rtos. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 12–15, 2008.

[36] P. Pedreiras and L. Almeida. The flexible time-triggered (ftt) paradigm: an approach to qos management in distributed real-time systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 123.1, Washington, DC, USA, 2003.

[37] P. Pedreiras, L. Almeida, and P. Gai. The ftt-ethernet protocol: merging flexibility, timeliness and efficiency. *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 134–142, 2002.

[38] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 172–179, 1998.

[39] M. Persson, T. N. Quresshi, and M. Törngren. Suitability of dynamic load balancing in resource-constrained embedded systems: An overview of challenges and limitations. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 55–58, 2008.

[40] L. T. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. *2009 30th IEEE Real-Time Systems Symposium*, pages 271–280, 2009.

[41] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[42] S. Samara, D. Orfanus, and P. Janacik. Toward biologically inspired decentralized self-adaptive os services for distributed reconfigurable system on chip (rsoc). In *APRES '09: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 7–10, 2009.

[43] P. K. Saraswat, P. Pop, and J. Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. In *APRES '09: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 15–18, 2009.

[44] N. Shankaran et al. Towards an integrated planning and adaptive resource management architecture for distributed real-time embedded(dre) systems. In *APRES '08: Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 65–68, 2008.

[45] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority preemptively scheduled systems. *Real-Time Systems Symposium, 1992*, pages 100–109, 1992.

[46] W. Trumler et al. Self-configuration and self-healing in autosar. In *14th Asia Pacific Automotive Engineering Conference (APAC-14)*, August 2007.

Chapter 8

Appendix A: The complete UPPAAL model of the smart phone example

8.1 The global variable declaration

```
//This is a model of a smart phone. Three types of functions
    are analyzed: the phone call, the camera to record video and
     the multimedia. Usually the camera is used off-line and
    does not consume any network resource, but we assume it is
    used online for the sake of our analysis. The multimedia
    consists of online video and audio functions. The detail of
    these applications is summarized in the table below:
//                              |CPU|BW |priority|stoppable|
//Call                          | 2 | 1 |    1    |    no    |
//Camera(record video)          | 2 | 1 |    2    |   yes    |
//Multimedia--Level 3|2         | 4 | 3 |    3    |   yes    |
//The default level for the video application
//Multimedia--Level 2|1         | 3 | 2 |
//Multimedia--Level 1|1         | 2 | 2 |
//Total                         | 9 | 5 |----Level 3
//The total available resource in terms of CPU and network
    bandwidth
//Total                         | 7 | 4 |----Level 2
//Total                         | 5 | 3 |----Level 1
const int M = 2;      //resource types
const int N=3;     //number of applications
typedef int[0,M-1] R_type;
typedef int[0,N-1] appNo;
bool OnApp[N]={false,false,false};     //the on/off status of
    each application
appNo appID=0;
const bool stoppableApp[N]={false,true,true};      //The phone
    call application is non-stoppable so that it cannot be
    interrupted by other applications
const int priorityApp[N]={1,2,3};  //1 has the highest priority
    , for the phone call
```

```
int[1,3] CPU_Level=2;    //The CPU resource is associated with
    three levels and the default level is 2. The CPU level can
    be manually switched by the user, but the user request could
     be denied if too much CPU resource is occupied.
int[1,3] newCPULevel;    //Only for programming purpose. It is
    updated during the synchronization between the user and
    resource(0).
int CPU_R[N]={5,7,9};    //The available CPU resource at each
    level
int[1,3] BW_Level=2;     //Similar to the CPU resource, the
    bandwidth resource also has three levels and the default
    level is 2.
int BW_R[N]={3,4,5};     //The available bandwidth resource at
    each level
const int CYCLE=7;      //The amount of available CPU resource
    at the default level
const int BANDWIDTH=4;   //The amount of available network
    resource, i.e. bandwidth at the default level
int totalResource[M]={CYCLE,BANDWIDTH};  //This array is used
    to store the currently available resources of each type
bool enough[M]={true,true};   //To indicate if the CPU or
    network resource is sufficient or not currently
int CPU_occupy[N]={2,2,4};   //CPU resource ID=0. Each element
    corresponds to the consumption of each application.
int BW_occupy[N]={1,1,3};    //Bandwidth resource ID=1. Each
    element corresponds to the consumption of each application.
const int CPULMAX=3;        //The maximal video level regarding
     CPU consumption
const int BWLMAX=2;         //The maximal viedo level regarding
     bandwidth consumption
int[1,3] videoLevel[M]={CPULMAX,BWLMAX};    //for M==0--Level
    1: consumes 1 CPU unit; Level 2: consumes 2 CPU units; Level
     3: consumes 3 CPU units
//for M==1--Level 1: consumes 1 bandwidth unit; Level 2:
    consumes 2 bandwidth units
//The audio application always takes 1 CPU unit and 1 bandwidth
     unit. No QoS level is related.
int[1,3] newVideoLevel[M]={CPULMAX,BWLMAX};      //Only for
    programming purpose

chan turnOn,ROK[M],RNOK[M],refuseTemporal[N],refuse[N],
    changeLevel;
broadcast chan turnOff,startA[N],stopA[N],pass[N];
```

8.2 The User template

```
clock x;
appNo ID=0;  //The user randomly selects one application and
    turns it on/off, depending on its status.
int[1,3] level=2;   //The user randomly switches the CPU level
```

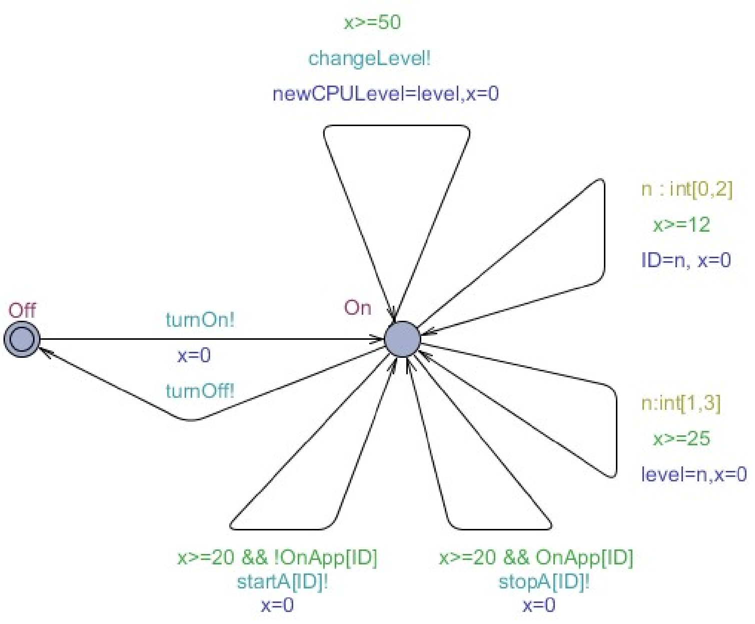

Figure 8.1: The User automaton of the smart phone model

```
    from 1 to 3. The initial level is 2. This is just a temporal
      variable and only the global variable CPU_Level presents
    the current CPU level.
```

8.3 The Application template

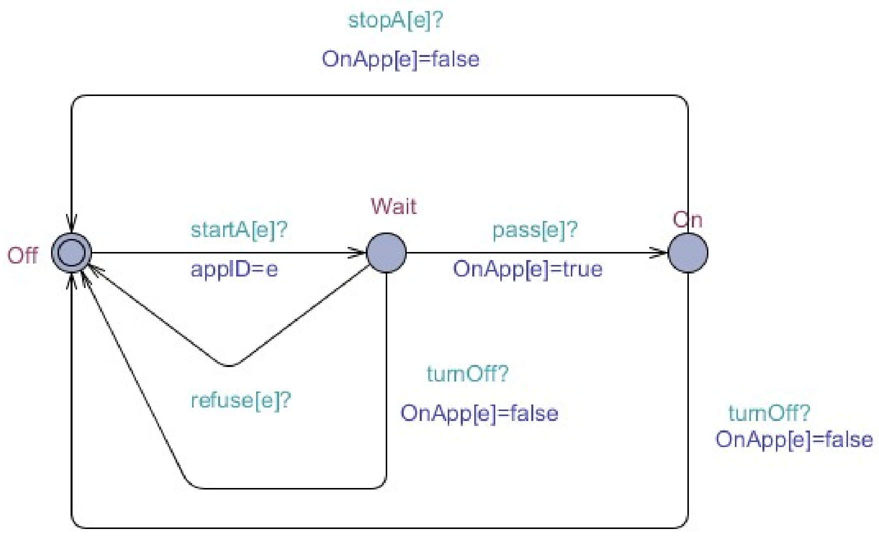

Figure 8.2: The Application automaton of the smart phone model

There is no local declaration for the Application template.

8.4 The Resource template

```
clock y;  //The bandwidth may change sometimes. We simulate
    this as a sporadic event, whose minimal arriving internal is
```

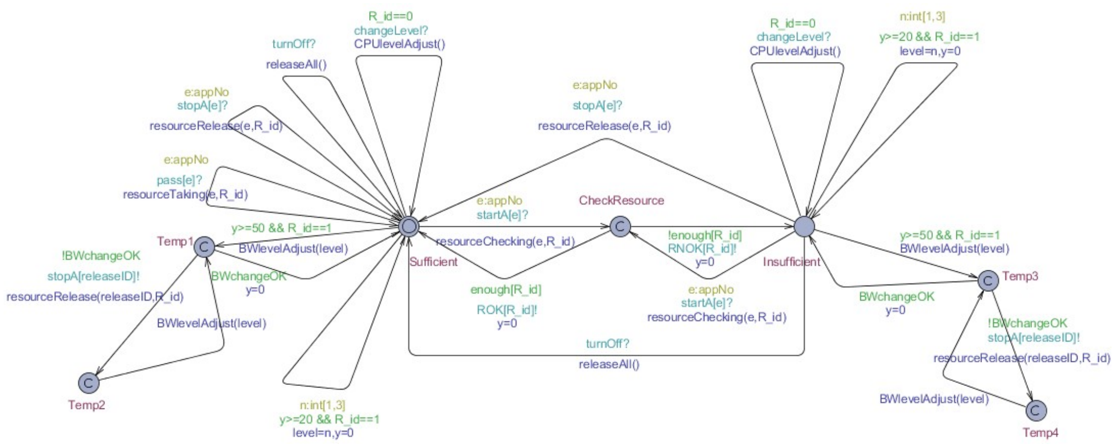

Figure 8.3: The Resource automaton of the smart phone model

```
    limited by the clock.
int level=2;    //This is just a temporal variable and only the
   global variable BW_Level presents the current bandwidth
   level.
bool BWchangeOK=true;  //To present if the bandwidth change is
   OK without affecting currently running applications
appNo releaseID=2;      //Since a bandwidth change request must
   be accepted, some running applications may be terminated by
   force to release some bandwidth resource to accommodate this
    degradation. This variable specifies which running
   application is going to be terminated.

bool degradeOK(R_type R_id)  //If the new application is
   multimedia and the current video level is higher than the
   minimum 1, call this function
{
     int i=newVideoLevel[R_id];
     int temp;
     if (R_id==0)
     {
             temp=CPU_occupy[2];
     }
     else
     {
             temp=BW_occupy[2];
     }
     while (i>0)
     {
             if (totalResource[R_id]>=temp)         //There is
                enough resource left for the currently pending
                application
             {
                 return true;
             }
             else          //Degrade the video level until the
                resource is enough or the video level falls to 1.
```

```
            {
                if (newVideoLevel[R_id]>1)
                {
                        newVideoLevel[R_id]--;
                    temp--;
                }
            }
            i--;
    }
    return false;
}

bool degradeOK2(appNo A_id,R_type R_id)     //Used if the new
   application is not multimedia and the multimedia is running
   at levels higher than the bottom levels for different
   resources.
{
    int i;
    int temp;
    int temp2=totalResource[R_id];
    if (R_id==0)
    {
            temp=CPU_occupy[A_id];
    }
    else
    {
            temp=BW_occupy[A_id];
    }
    for (i=newVideoLevel[R_id];i>0;i--)
    {
            if (temp2>=temp)
            {
                return true;
            }
            else
            {
                if (newVideoLevel[R_id]>1)
                {
                    newVideoLevel[R_id]--;
                    temp2++;           //Is there enough resource
                       if the video level is lowered?
                }
            }
    }
    return false;
}

void resourceChecking(appNo A_id,R_type R_id)     //To check if
     the resource is enough for a new application. It updates
   enough[R_id].
{
    int temp;
    if (R_id==0)
    {
            temp=CPU_occupy[A_id];
    }
    else
    {
            temp=BW_occupy[A_id];
    }
    newVideoLevel[R_id]=videoLevel[R_id];
    if (A_id==2 && newVideoLevel[R_id]>1)      //If the new
        application is multimedia and the current video level is
         higher than the minimum 1
    {
            enough[R_id]=degradeOK(R_id);      //Degradation can
                be applied if necessary
    }
    else if (A_id!=2 && OnApp[2])     //If the new application is
         not multimedia but the multimedia application is
        running
    {
            enough[R_id]=degradeOK2(A_id, R_id);
    }
    else        //If no multimedia application is involved, no
        QoS degradation is considered.
    {
            if (totalResource[R_id]>=temp)
            {
                enough[R_id]=true;
            }
            else
            {
                enough[R_id]=false;
            }
    }
}

void resourceTaking(appNo A_id,R_type R_id)       //If all types
    of resources are enough for the new application after the
    checking process, this new application is accepted and the
    corresponding resources are consumed.
{
    int levelChange=videoLevel[R_id]-newVideoLevel[R_id];      //
        If the levelChange is not 0, video level must be lowered
        .
    if (R_id==0)    //CPU resource
    {
            if (A_id==2 && levelChange!=0)
            {
                videoLevel[R_id]=newVideoLevel[R_id];
                CPU_occupy[2]-=levelChange;
                totalResource[R_id]-=CPU_occupy[2];
            }
            else if (A_id!=2 && OnApp[2])     //By default we
                assume the video level is lowered, but if it is
                not, levelChange==0. This branch still holds.
            {
                videoLevel[R_id]=newVideoLevel[R_id];
                CPU_occupy[2]-=levelChange;
                totalResource[R_id]+=levelChange;
                totalResource[R_id]-=CPU_occupy[A_id];
            }
            else
            {
                totalResource[R_id]-=CPU_occupy[A_id];
            }
    }
    else    //Network resource    R_id==1    Similar steps are
        taken compared with CPU resource
    {
            if (A_id==2 && levelChange!=0)
            {
                videoLevel[R_id]=newVideoLevel[R_id];
                BW_occupy[2]-=levelChange;
                totalResource[R_id]-=BW_occupy[2];
            }
            else if (A_id!=2 && OnApp[2])
            {
                videoLevel[R_id]=newVideoLevel[R_id];
                BW_occupy[2]-=levelChange;
                totalResource[R_id]+=levelChange;
                totalResource[R_id]-=BW_occupy[A_id];
            }
            else
            {
                totalResource[R_id]-=BW_occupy[A_id];
            }
    }
    y=0;
}

void resourceRelease(appNo A_id,R_type R_id)
{
    int temp;
    if (R_id==0)    //CPU resource
    {
        totalResource[R_id]+=CPU_occupy[A_id];
            enough[R_id]=true;
            if (A_id!=2 && OnApp[2]) //When an application rather
                than the multimedia is terminated, the
                multimedia tries to restore the highest video
                quality
            {
                temp=CPUL_MAX-videoLevel[R_id];
                temp=((temp<=totalResource[R_id])?temp:
                    totalResource[R_id]);
                videoLevel[R_id]+=temp;
                CPU_occupy[2]+=temp;
                totalResource[R_id]-=temp;
            }
    }
    else     //Network resource     R_id==1
    {
            totalResource[R_id]+=BW_occupy[A_id];
            enough[R_id]=true;
            if (A_id!=2 && OnApp[2])
            {
                temp=BWL_MAX-videoLevel[R_id];
                temp=((temp<=totalResource[R_id])?temp:
                    totalResource[R_id]);
                videoLevel[R_id]+=temp;
                BW_occupy[2]+=temp;
                totalResource[R_id]-=temp;
            }
    }
    y=0;
}

void releaseAll()    //This is a "reset" function applied when
    the user turns off the smart phone. Some variables should be
     set to the initial value.
{
    totalResource[0]=CYCLE;
    totalResource[1]=BANDWIDTH;
    CPU_Level=2;
    BW_Level=2;
    enough[0]=true;
    enough[1]=true;
    videoLevel[0]=CPUL_MAX;
    videoLevel[1]=BWL_MAX;
    CPU_occupy[2]=4;
    BW_occupy[2]=3;
    level=2;
    //counter=0;
    y=0;
}

void CPUlevelAdjust()    //The user can manually adjust CPU
    level
{
    int occupiedCPU;    //To indicate how much CPU resource has
         been occupied
    int temp,temp2,temp3;
    int change=newCPULevel−CPU_Level;
    occupiedCPU=CPU_R[CPU_Level−1]−totalResource[0];
    y=0;
    if (change>0)   //If the CPU level is raised and multimedia
         is running, we can raise its QoS level
    {
            CPU_Level=newCPULevel;    //The level change request
                is accepted
            totalResource[0]=CPU_R[newCPULevel−1]−occupiedCPU;
                    //The currently available CPU resource is
                updated due to the level change
            if (OnApp[2] && videoLevel[0]<CPUL_MAX)
            {
                temp=CPUL_MAX−videoLevel[0];
                temp=((temp<=totalResource[0])?temp:
                    totalResource[0]);
                videoLevel[0]+=temp;
                CPU_occupy[2]+=temp;
                totalResource[0]−=temp;
            }
    }
    else if (occupiedCPU<=CPU_R[newCPULevel−1])      //If CPU
        level is lowered and less CPU resource has been occupied
         than the total amount of CPU resource at the new level
    {
            CPU_Level=newCPULevel;    //The level change request
                is accepted
            totalResource[0]=CPU_R[newCPULevel−1]−occupiedCPU;
                    //The currently available CPU resource is
                updated due to the level change
    }
    else if (OnApp[2] && videoLevel[0]>1)
    //If CPU level is lowered and more CPU resource has been
        occupied than the total amount of CPU resource at the
        new level and it is possible to degrade QoS level
    {
            temp=videoLevel[0];
            temp2=occupiedCPU;
            temp3=CPU_occupy[2];
            while (temp>1)
            {
                temp−−;
                temp2−−;
                temp3−−;
                if (temp2<=CPU_R[newCPULevel−1])
                {
                        videoLevel[0]=temp;
                        occupiedCPU=temp2;
                        CPU_Level=newCPULevel;
                        totalResource[0]=CPU_R[newCPULevel−1]−
                            occupiedCPU;
```

```
                        CPU_occupy[2]=temp3;
                    }
                }
        }
        else   //The CPU level request is refused
        {
                newCPULevel=CPU_Level;    //The CPU level remains the
                    same. The level change request is denied
        }
}

void BWlevelAdjust(int newBWLevel)    //The bandwidth may change
     its levels automatically due to the unstable network
    condition
{
        int levelChange=newBWLevel-BW_Level;    //It is positive if
            the bandwidth condition is getting better, i.e. the
            level is raised
        int occupiedBW=BW_R[BW_Level-1]-totalResource[1];      //To
            indicate how much bandwidth resource has been occupied
        int i=2;
        int temp,temp2,temp3;
        if (levelChange>=0)
        //If the network condition is getting better or less
            bandwidth resource is occupied than the total amount of
            bandwidth resource at the new level
        {
                BWchangeOK=true;        //The bandwidth change is
                    directly accepted, no running applications are
                    affected
                BW_Level=newBWLevel;
                totalResource[1]=BW_R[newBWLevel-1]-occupiedBW;    //
                    The currently available bandwidth resource is
                    updated due to the level change
                if (OnApp[2]&& videoLevel[1]<BWLMAX)
                {
                     temp=BWLMAX-videoLevel[1];
                     temp=((temp<=totalResource[1])?temp:
                         totalResource[1]);
                     videoLevel[1]+=temp;
                     BW_occupy[2]+=temp;
                     totalResource[1]-=temp;
                }
                return;
        }
        else if (occupiedBW<=BW_R[newBWLevel-1])
        {
                BWchangeOK=true;
                BW_Level=newBWLevel;
                totalResource[1]=BW_R[newBWLevel-1]-occupiedBW;
                return;
```

```
}
if (OnApp[2] && videoLevel[1]>1)  //If the degraded
    bandwidth must affect the running applications, check if
     the QoS level can be lowered first
{
     temp=videoLevel[1];
          temp2=occupiedBW;
          temp3=BW_occupy[2];
          while (temp>1)
          {
               temp--;
               temp2--;
               temp3--;
               if (temp2<=BW_R[newBWLevel-1])
               {
                         videoLevel[1]=temp;
                         occupiedBW=temp2;
                         BW_Level=newBWLevel;
                         totalResource[1]=BW_R[newBWLevel-1]-
                             occupiedBW;
                         BW_occupy[2]=temp3;
                         BWchangeOK=true;
                         return;
               }
          }
}
 //The network condition change cannot be denied, and some
     running applications must be terminated to adap it if
     necessary
BWchangeOK=false;
while(i>=0)
{
     if (OnApp[i] && stoppableApp[i])   //low prioroty
          stoppable running applications are preferred to be
          terminated
          {
               releaseID=i;
               i=-1;
               return;
          }
          i--;
}
if (i!=-2)  //If no stoppable running applications are
    found, even stoppable running applications should be
    terminated
{
          i=2;
          while(i>=0)
          {
               if (OnApp[i])
               {
```

```
                              releaseID=i;
                              return;
                        }
                     i--;
                  }
        }
}
```

8.5 The AdmissionControl template

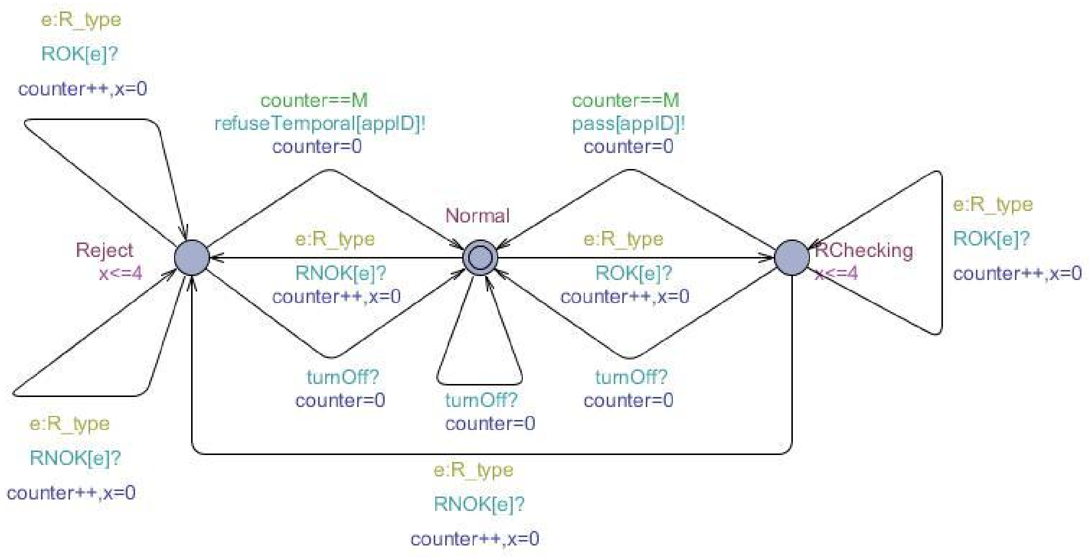

Figure 8.4: The AdmissionControl automaton of the smart phone model

```
int counter=0;   //The counter is increased by one each time it
    receives the OK/NOK report from a resource
clock x;
```

8.6 The Controller template

```
//If there is no way of releasing enough resource to accept the
     new application according to the scheduling policy, this
   application must be rejected.
bool stillRefuse=true;
appNo abortID;
int[1,3] i=N;
int counter=0;
bool mark=true;
bool idle=false; //From location "Scheduling" to "Retry", an
   application is terminated. Then we should check if no
   application is running. This is not necessary mostly because
    it only happens if the system is overloaded by just two
   applications.
//The scheduling policy implemented in this model: It is
   priority-based non-preemptive scheduling. If there is no
```

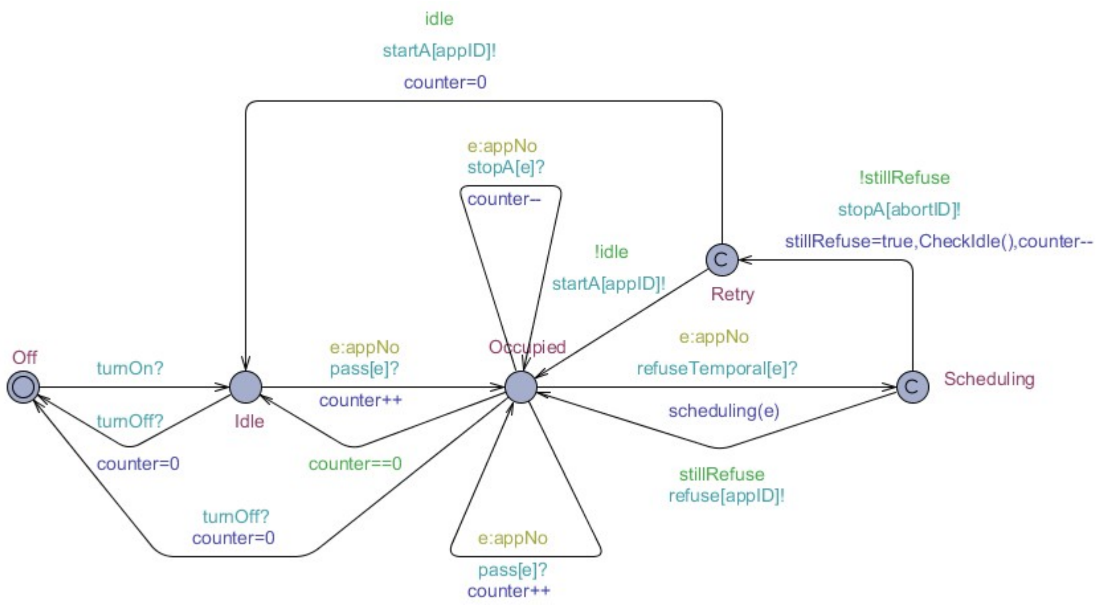

Figure 8.5: The Controller automaton of the smart phone model

```
    enough resource for a new application, we first check if
    there are any running applications with lower priorities,
    and then check if they are stoppable. If the checking result
     is not empty, we can stop those applications to release
    resource. Each time only one application is allowed to be
    stopped and the lowest-priority stoppable running
    application is considered first.
//This scheduling policy is not optimal. There is a risk that
    there is still no enough resource for a new application even
     after many other applications are stopped. However, this
    policy is straightforward and simple to be implemented.

void scheduling(appNo ID)
{
    stillRefuse=true;
    mark=true;
    i=N;
    abortID=appID;
    if (priorityApp[ID]<N)
    {
            while (i>priorityApp[ID] && mark)
            {
                if (OnApp[i-1] && stoppableApp[i-1])
                {
                        abortID=i-1;
                        stillRefuse=false;
                    mark=false;   //break the iteration while "
                        break" syntax is not supported yet
                }
             i--;
            }
    }
}

void CheckIdle()
{
    if (OnApp[0] | OnApp[1] | OnApp[2])
    {
            idle=false;
    }
    else
    {
            idle=true;
    }
}
```

8.7 Properties and verification results

Index	Property	Verification result
1	A[] not deadlock	Satisfied
2	E<> Resource(1).Temp2	Satisfied
3	E<> Resource(1).Temp3	Satisfied
4	E<> Resource(1).Temp4	Not satisfied
5	A[] totalResource[0]<=9	Satisfied
6	A[] totalResource[1]<=5	Satisfied
7	E<> (OnApp[0] & OnApp[1] & OnApp[2])==true	Satisfied
8	A[] forall (i:appNo) Application(i).Wait imply appID==i	Satisfied
9	A[] Controller.Idle imply (OnApp[0] | OnApp[1] | OnApp[2])==false	Satisfied
10	A[] Controller.Idle imply Controller.counter==0	Satisfied
11	A[] AdmissionControl.counter<=2	Satisfied
12	E<> Controller.counter==3	Satisfied
13	Application(0).Wait –> Application(0).On	Not satisfied
14	A[] forall (i:R_type) Resource(i).Sufficient imply enough[i]==true	Satisfied
15	A[] forall (i:R_type) Resource(i).Insufficient imply enough[i]==false	Satisfied
16	A[] Controller.Scheduling imply (enough[0] & enough[1])==false	Satisfied
17	E<> Resource(1).Insufficient and (forall (i:appNo) OnApp[i]==false)	Not satisfied
18	AdmissionControl.Reject –> AdmissionControl.Normal	Satisfied
19	AdmissionControl.RChecking –> AdmissionControl.Normal	Satisfied
20	E<> (forall (i:appNo) OnApp[i]==true) and videoLevel[0]==3 and videoLevel[1]==2	Satisfied
21	A[] ((forall (i:appNo) OnApp[i]==true) and videoLevel[0]==3 and videoLevel[1]==2) imply (CPU_Level==3 and BW_Level==3)	Satisfied

Table 8.1: The properties and verification results of the smart phone model

Chapter 9

Appendix B: The complete UPPAAL model of the object-tracking robot

9.1 The global variable declaration

```
//This is a UPPAAL model for an object-tracking robot. The
    purpose is to demonstrate a typical Adaptive Embedded System
     (AES). The robot is equipped with a group of localization
    sensors which work together to track the moving object in
    the distance. The adaptivity of this robot lies in three
    aspects.
//First, the localization sensors only work in the object
    tracking mode and are deactivated automatically in other
    modes to save power. Second, the sampling rate of each
    localization sensor is adjusted properly as the distance
    between the robot and the object changes to some extent.
    Third, a faulty sensor can be detected automatically and
    shut down immediately to save power and mainain accuracy.

int M = 3;//Number of localization sensors that are not broken
const int N=3;    //Number of total localization sensors
//We shall call sensors for short from now on, instead of
    localization sensors.
typedef int[0,N-1] sensorIndex;//The index of each sensor: 0-2
typedef int[0,5] sensorValue;  //The value detected by each
    sensor falls between 0 and 5
int[1,3] initSV=2; //We assume the initial value of each
    sensor is 2 and it will be updated as a common base value
    for all sensors
sensorValue Svalues[N]={0,0,0};    //The initial value of each
    sensor, all 0
int[0,2] sensorStatus[N]={0,0,0};    //Sensor status     0: Off;
      1: On;    2: Broken
bool sampling[N]={true,true,true};
int[0,12] samplingRate[N]={10,10,10};  //Sensor sampling rate;
    can change with the distance from the object       Normal:
```

```
    10-12       Broken: 0
int[8,10] distance=10;  //We assume the distance between the
    robot and the moving object is from 8 to 10. 10 is the
    initial distance. If the distance is farther than 10, the
    robot will not be able to detect the object.
chan turnOn, sensorV[N], broken[N], nextPeriod;
broadcast chan turnOff, toIdle, toOtherModes, toTrackingMode,
    closer, farther;
```

9.2 The User template

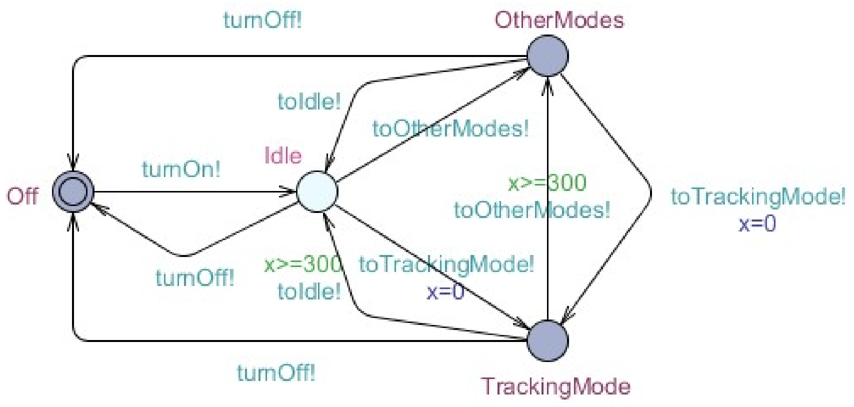

Figure 9.1: The User automaton of the object-tracking robot model

```
clock x;
```

9.3 The Robot template

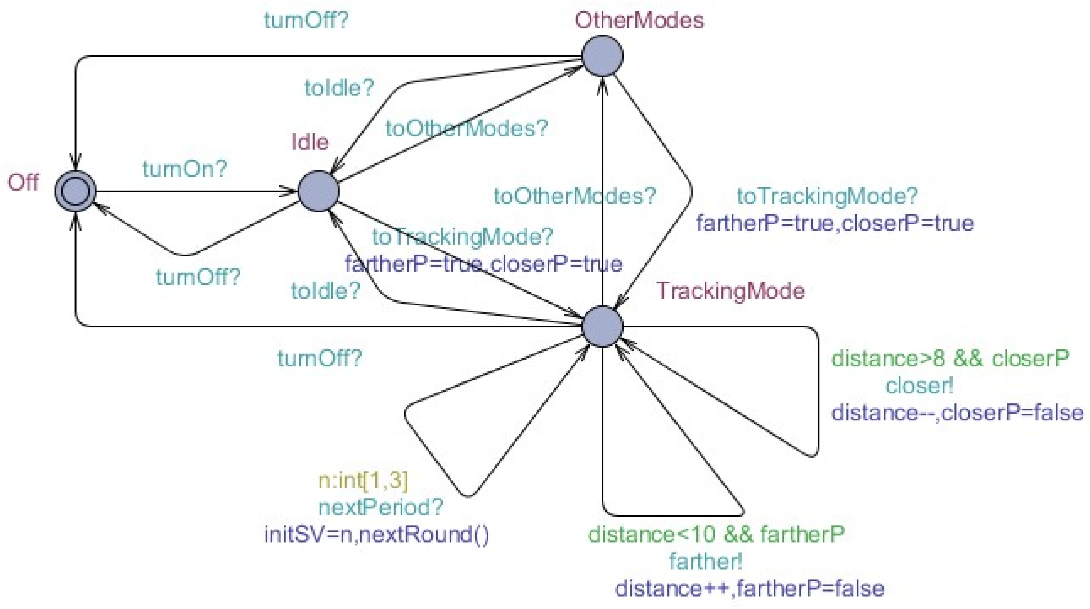

Figure 9.2: The Robot automaton of the object-tracking robot model

```
bool closerP=true;    //To make sure that distance from the
    object can get frather or closer at most once during each
    sampling period.
bool fartherP=true;

void nextRound()      //To enter the next sampling period
{
          int i;
          for(i=0;i<N;i++)
          {
                    sampling[i]=true;
          }
          closerP=true;
          fartherP=true;
}
```

9.4 The Sensor template

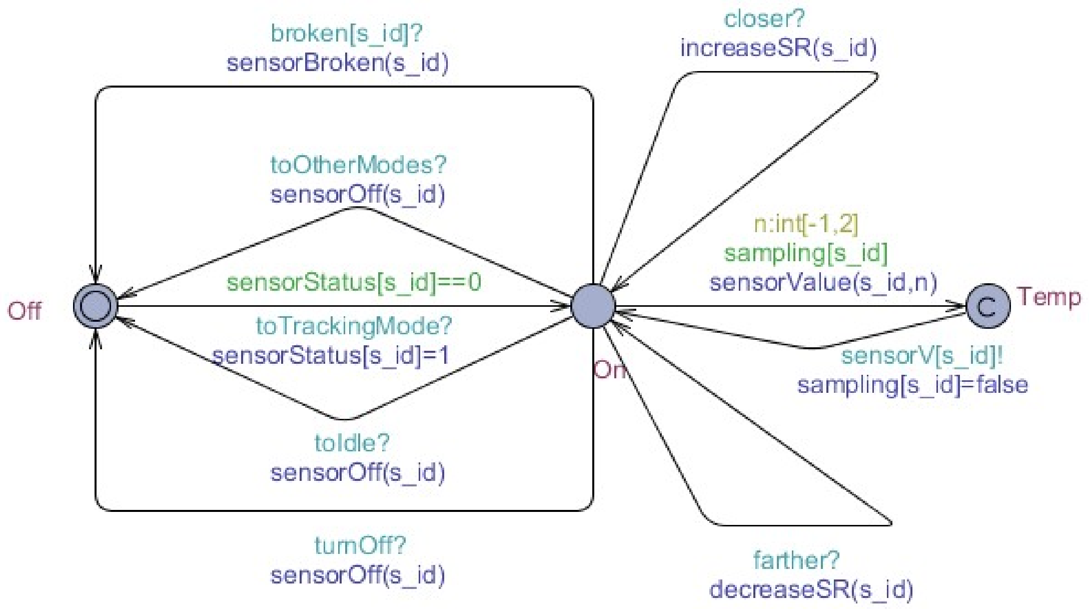

Figure 9.3: The Sensor automaton of the object-tracking robot model

```
void sensorValue(sensorIndex s_id,int offset)  //The reading of
     each sensor is the combination of the common base value and
     its own offset
{
          Svalues[s_id]=initSV+offset;
}

void sensorBroken(sensorIndex s_id) //When a sensor is found to
     be faulty
{
          sensorStatus[s_id]=2;
         M--;    //The number of available sensors is decreased
```

```
        Svalues[s_id]=0;
        sampling[s_id]=false;
        samplingRate[s_id]=0;
}

void sensorOff(sensorIndex s_id)  //When a sensor is
    deactivated due to mode switch or power off
{
        sensorStatus[s_id]=0;
        Svalues[s_id]=0;
        sampling[s_id]=true;
}

void increaseSR(sensorIndex s_id)   //Increase the sampling
    rate of the corresponding sensor
{
        samplingRate[s_id]++;
}

void decreaseSR(sensorIndex s_id)   //Decrease the sampling
    rate of the corresponding sensor
{
        samplingRate[s_id]--;
}
```

9.5 The Controller template

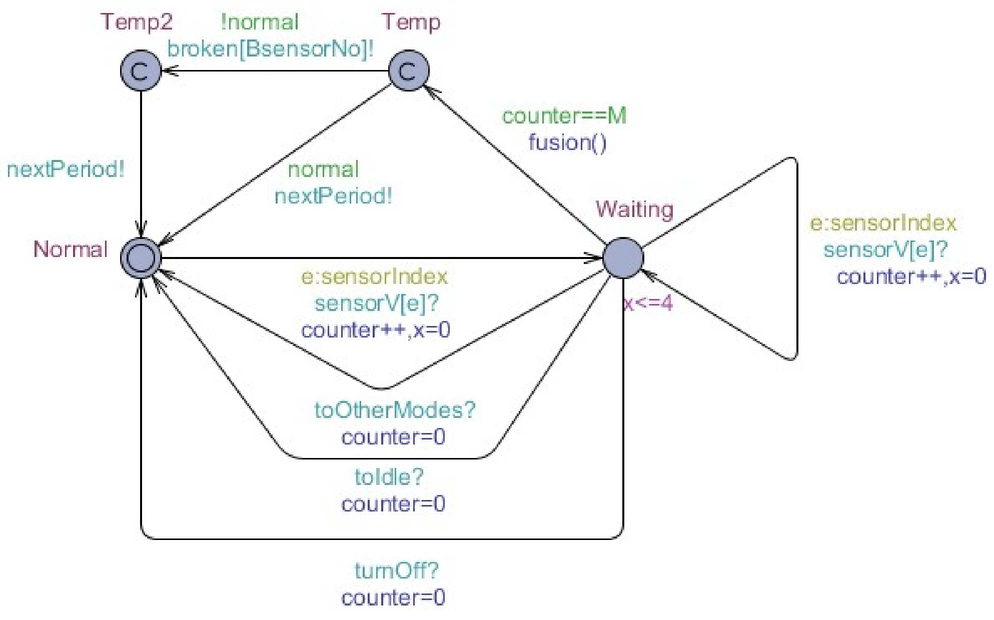

Figure 9.4: The Controller automaton of the object-tracking robot model

```
clock x;
int counter=0;   //The local counter to count how many reading
    reports the controller has received
```

```
bool normal=true;   //If no faulty sensor is found after each
   sampling period, it is true. Otherwise, it is false.
sensorIndex BsensorNo=0;    //The index of the faulty sensor

int brokenSensorIndex;   //The index of the sensor whose
   reading deviates most from the average. It is not
   necessarily the index of a faulty sensor
sensorValue average=0;   //The average of all readings
int sum=0;  //The sum of all readings
int offsets[N]={0,0,0};   //This array stores the offsets of
   those sensors from the average value

sensorValue abs(int a)   //A function to calculate the absolute
    value of an integer
{
        sensorValue new=a>=0?a:-a;
        return new;
}
//A function to calculate the maximal absolute value of a given
    array. If two or more equivalent maximal items are found,
   it returns -1.
int max(sensorValue a[N])
{
        int i;
        int j;
        sensorValue maximal=a[0];
        bool unique=true;    //If there is only one maximal
           item, it is true.
        j=0;
        for(i=1;i<N;i++)
        {
                if(maximal<a[i])
                {
                        maximal=a[i];
                        j=i;
                        unique=true;
                }
                else
                {
                        if(maximal==a[i])
                        {
                                unique=false;
                        }
                }
        }
        if(unique)
        {
                return j;
        }
        else
        {
                return -1;
        }
}

void fusion()
{
        int i;
        sum=0;

        if(M==2)    //If there are only two non-broken sensors
            left, we no longer detect faulty sensors but just
            calculate the average.
        {
                normal=true;
                counter=0;
                for(i=0;i<N;i++)
                {
                        sum+=Svalues[i];    //Please note that
                            the the readings of faulty sensors
                            are 0.
                }
                average=sum/2;
                return;
        }

        for(i=0;i<N;i++)
        {
                sum+=Svalues[i];
        }
        average=sum/M;        //To calculate the average of all
            readings
        for(i=0;i<N;i++)
        {
                if(sensorStatus[i]==1)
                {
                        offsets[i]=abs(Svalues[i]-average); //
                            To calculate the offset of a non-
                            broken sensor from the average
                }
                else
                {
                        offsets[i]=0;      //The offset of the
                            faulty sensor is 0.
                }
        }
        brokenSensorIndex=max(offsets);   //To find out the
            single sensor with the maximal offset
        if(brokenSensorIndex==-1)    //If no single sensor with
            the maximal offset is found
        {
                normal=true;
        }
        else    //If the single sensor with the maximal offset
           is indeed found
        {
                if(offsets[brokenSensorIndex]>1)    //To check
                   if this maximal offset is above a pre-
                   defined threshold
                {
                        normal=false;
                        BsensorNo=brokenSensorIndex;    //This
                           sensor is faulty.
                }
                else
                {
                        normal=true;    //This sensor is still
                           not faulty despite its maximal
                           offset.
                }
        }
        counter=0;
}
```

9.6 Properties and verification results

Index	Property	Verification result
1	A[] not deadlock	Satisfied
2	E<> Controller.Temp2	Satisfied
3	A[] M>=2 && M<=3	Satisfied
4	A[] (sensorStatus[0]+sensorStatus[1]+sensorStatus[2])<5	Satisfied
5	A[] Controller.counter>=0 && Controller.counter<=3	Satisfied
6	A[] Controller.Normal imply Controller.counter==0	Satisfied
7	Controller.Waiting –> Controller.Normal	Satisfied
8	A[] (distance==8 && sensorStatus[0]==1) imply (samplingRate[0]==12)	Satisfied
9	E<> distance==8	Satisfied
10	A[] (Robot.Off||Robot.Idle||Robot.OtherModes) imply (forall (i:sensorIndex) sensorStatus[i]!=1)	Satisfied
11	A[] User.TrackingMode imply Robot.TrackingMode	Satisfied
12	A[] sensorStatus[0]==2 imply Sensor(0).Off	Satisfied
13	A[] sensorStatus[0]==2 imply (Controller.average==(Svalues[1]+Svalues[2])/2)	Not satisfied

Table 9.1: The properties and verification results of the object-tracking robot model